

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



Network Attack Injection

João Alexandre Simões Antunes

DOUTORAMENTO EM INFORMÁTICA
ESPECIALIDADE CIÊNCIA DA COMPUTAÇÃO

2012

UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



Network Attack Injection

João Alexandre Simões Antunes

Tese orientada pelo Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
especialmente elaborada para a obtenção do grau de
doutor em Informática, especialidade de Ciência da Computação.

2012

Abstract

The increasing reliance on networked computer systems demands for high levels of dependability. Unfortunately, new threats and forms of attack are constantly emerging to exploit vulnerabilities in systems, compromising their correctness. An intrusion in a network server may affect its users and have serious repercussions in other services, possibly leading to new security breaches that can be exploited by further attacks. Software testing is the first line of defense in opposing attacks because it can support the discovery and removal of weaknesses in the systems. However, searching for flaws is a difficult and error-prone task, which has invariably overlooked vulnerabilities.

The thesis proposes a novel methodology for vulnerability discovery that systematically generates and injects attacks, while monitoring and analyzing the target system. An attack that triggers an unexpected behavior provides a strong indication of the presence of a flaw. This attack can then be given to the developers as a test case to reproduce the anomaly and to assist in the correction of the problem.

The main focus of the investigation is to provide a theoretical and experimental framework for the implementation and execution of attack injection on network servers. Several innovative solutions related to this approach are covered, including ways to infer a specification of the protocol implemented by the server, the generation of a comprehensive set of attacks, the injection and monitoring of the target system, and the automatic analysis of results.

Furthermore, we apply some of the developed techniques to other areas of network security, namely to intrusion tolerance and detection. In particular, a new method is proposed to assist on the evaluation of the compliance of diverse replicas in intrusion-tolerant systems.

Keywords: Network Security, Network Servers, Vulnerabilities, Attack Injection, Monitoring, Protocol Reverse Engineering, Test Case Generation, Behavioral Inference

Resumo

O aumento da dependência e confiança depositada nos sistemas de rede, exige níveis de confiabilidade cada vez mais elevados. Infelizmente, novas ameaças e formas de ataque estão constantemente a surgir, explorando vulnerabilidades nos sistemas e comprometendo a sua correta operação. Uma intrusão num servidor de rede pode afetar os seus utilizadores e ter graves repercussões noutros serviços, eventualmente abrindo novas brechas de segurança que podem ser exploradas por outros ataques. O teste de software é a primeira linha de defesa na oposição a ataques porque pode apoiar a descoberta e remoção de fraquezas dos sistemas. No entanto, a procura de falhas é uma tarefa difícil e propensa a erros, e que tem invariavelmente deixado escapar vulnerabilidades.

A tese propõe uma nova metodologia para a descoberta da vulnerabilidades que permite a sistemática geração e injeção de ataques, e a simultânea monitorização e análise do sistema-alvo. Um ataque que desencadeie um comportamento inesperado é uma forte indicação da presença de uma falha. Este ataque pode então ser dado à equipa de desenvolvimento como um caso de teste para reproduzir a anomalia e para auxiliar na correção do problema.

O foco principal da investigação é fornecer um quadro teórico e experimental para a concretização e execução da injeção de ataques em servidores de rede. Diversas soluções inovadoras relacionadas

com esta abordagem são estudadas, incluindo a inferência da especificação do protocolo concretizado pelo servidor, a geração de um conjunto abrangente de ataques, a injeção e monitorização do sistema-alvo e a análise automática dos resultados.

Além disso, aplicamos algumas das técnicas aqui desenvolvidas noutras áreas de segurança de redes, nomeadamente, para a tolerância e deteção de intrusões. Em particular, é proposto um novo método para a avaliação da conformidade de réplicas em sistemas tolerantes a intrusões com diversidade.

Palavras-Chave: Segurança em Redes, Servidores de Rede, Vulnerabilidades, Injecção de Ataques, Engenharia de Reversão de Protocolos, Geração de Casos de Teste, Inferência Comportamental

Resumo Alargado

Ao longo do último século assistiu-se a um crescente número de inovações que foram decisivas para o avanço do conhecimento e da tecnologia. A Internet foi uma dessas inovações que teve um impacto profundo na sociedade e revolucionou o modo como comunicamos e vivemos.

Hoje dependemos da Internet e dos seus variados serviços para muito mais que a simples busca e partilha de informação. Estima-se que em 2011 o número de utilizadores da Internet tenha ultrapassado os 2,4 mil milhões, mais do dobro do que em 2006 ([ITU, 2011](#)). Não só a sua popularidade tem vindo a aumentar, como é também um grande impulsionador económico, movimentando quase 6 biliões de Euros por ano no mundo inteiro. Calcula-se mesmo que a Internet represente em média cerca de 3% do PIB dos países desenvolvidos ou em forte crescimento económico, ultrapassando mesmo os setores da agricultura e da energia ([Rausas et al., 2011](#)).

A Internet é deste ponto de vista um serviço essencial à sociedade e à economia, criando emprego e conhecimento, alterando estilos de vida e até afetando a política internacional, evidenciado nas recentes quedas de velhos regimes ([Coz, 2011](#)). Esta dependência é ainda agravada pelo facto da Internet servir de suporte a algumas infraestruturas críticas, sem as quais a sociedade como a conhecemos não poderia subsistir. Os transportes, a comunicação, a distribuição de energia elétrica e atividades bancárias e financeiras são alguns dos serviços dos quais dependemos e que são controlados e operados através da Internet. A interrupção de um destes serviços pode ter graves consequências ([US-Canada Power System Outage Task Force, 2006](#); [Castle, 2006](#)), tornando a Internet num ponto de dependência único e extremamente sensível, quer a ataques físicos, quer a ataques

informáticos.

A exploração maliciosa de um servidor na Internet pode afetar não apenas os seus utilizadores, como também despoletar um efeito em cascata com graves repercussões noutros serviços (Sharma, 2010). Vários estudos e inquéritos revelaram que o número de ataques informáticos tem vindo a aumentar, sendo cada vez mais eficientes e sofisticados (Dobbins and Morales, 2011; McAfee, 2011; Symantec, 2012). Estes ataques já não são simples e inofensivas demonstrações de poder entre grupos de piratas informáticos, mas são esforços bem concertados, organizados e financiados. As suas motivações são variadas, podendo ir do lucro pessoal a razões políticas ou comerciais, envolvendo por vezes a sabotagem e espionagem industrial ou governamental (Garg et al., 2003; Anderson and Nagaraja, 2009; Krekel, 2009; Kanich et al., 2011; Rashid, 2011).

Defender os servidores de rede de ataques e evitar que sejam explorados maliciosamente torna-se assim essencial ao correto funcionamento da Internet. Porém, as tarefas relacionadas com a segurança são muitas vezes delegadas para segundo plano, em parte porque são vistas como menos fundamentais quando comparadas com a inclusão de novas funções. De facto, a maior parte do esforço no desenvolvimento de software vai tipicamente para a introdução de novas funcionalidades, uma vez que estas são comercialmente apelativas. Isto torna, por exemplo, as operações de teste cada vez mais complexas e dispendiosas, visto haver um maior número de requisitos funcionais a ser verificado, bem como requisitos não-funcionais, tais como garantir níveis aceitáveis de desempenho e de confiabilidade.

O teste de software é a primeira linha de defesa na prevenção e resistência a ataques pois suporta a descoberta e remoção de vulnerabilidades. Todavia, procurar por erros em aplicações e servidores é um processo meticuloso e moroso, sendo tradicionalmente realizado por seres humanos. Por conseguinte, os méto-

dos clássicos de teste têm invariavelmente deixado escapar erros. Por outro lado, os métodos levados a cabo pelos atacantes na detecção e exploração de vulnerabilidades têm-se revelado bastante eficazes, o que é bem visível no crescente número de vulnerabilidades reportadas ([Fossi et al., 2011](#)).

Como resultado, há a necessidade de se criarem novos mecanismos para a análise dos sistemas, em particular no que diz respeito a vulnerabilidades de segurança. Este trabalho de doutoramento visa contribuir com uma nova abordagem para a descoberta de vulnerabilidades que assenta na ideia de que uma vulnerabilidade existente no sistema só se manifesta, através de um erro ou de uma falha, quando ativada por um ataque específico. A metodologia de injeção de ataques aqui proposta procura solucionar o problema da descoberta de vulnerabilidades de uma forma automatizada, através da geração e injeção de um conjunto de ataques (ou casos de teste) com o máximo de cobertura. A automação do processo de teste permite potencialmente obter ganhos significativos, não só na diminuição dos custos, que em alguns casos podem chegar a 50% dos custos totais de desenvolvimento ([Beizer, 1990](#)), mas também na obtenção de resultados mais completos e confiáveis.

No entanto, este tipo de solução traz consigo vários desafios, requerendo soluções complexas que abordam vários domínios de investigação, desde a indução de gramáticas à monitorização de aplicações, passando pela geração e execução de casos de teste. A solução proposta na tese engloba por isso diversas fases, concretizadas por diferentes componentes, responsáveis por solucionar necessidades específicas da metodologia.

Numa primeira instância, pretende-se obter uma especificação do sistema-alvo que modele o seu comportamento em relação ao modo como fornece o seu serviço. No contexto da injeção de ataques de rede, o sistema-alvo é um servidor que troca mensagens de rede com os seus clientes de maneira a oferecer

o serviço. Portanto, a modelização do servidor pode ser aproximada através da especificação do protocolo de comunicação que concretiza. A obtenção desta especificação é então um componente fundamental da metodologia, uma vez que os ataques utilizam como veículo de transporte o envio de mensagens em conformidade com aquele protocolo.

Em alguns casos, a especificação do protocolo pode não estar acessível, como por exemplo se o protocolo for proprietário (ou fechado). Mesmo em protocolos abertos, traduzir a documentação que descreve a especificação num modelo formal é um processo relativamente moroso e propício a erros. A tese propõe uma solução para a obtenção da especificação de um protocolo de uma forma automatizada. A solução seguida assenta numa técnica de engenharia de reversão que utiliza apenas as mensagens do protocolo trocadas entre clientes e servidores. A abordagem baseia-se no problema de indução de gramáticas e é aplicada para inferir a linguagem do protocolo (i.e., quais as mensagens válidas e os seus respetivos formatos) e a sua máquina de estados (i.e., quais as relações estabelecidas entre os diferentes tipos de mensagens).

A especificação do protocolo do sistema-alvo é depois utilizada pela metodologia no processo de geração de ataques. Tal como na perspetiva de um adversário, o conjunto de ataques deverá ser exaustivo, de modo a obter uma boa cobertura de teste, mas não demasiado numeroso ao ponto de tornar a sua injeção impraticável. Duas aproximações para a geração de ataques são estudadas na tese. A primeira define quatro algoritmos de geração de casos de teste que procuram experimentar diferentes aspetos da concretização, desde mensagens com erros de sintaxe a mensagens com conteúdo potencialmente malicioso. A segunda abordagem procura reaproveitar casos de teste existentes (e.g., produzidos por ferramentas de segurança) para gerar ataques específicos para o sistema-alvo. Os casos de teste reciclados podem até ter sido definidos para outros protocolos.

Logo, esta solução pode permitir testar sistemas-alvo que concretizem novos protocolos (ou extensões) e que por isso ainda não são suportados pelas ferramentas existentes.

Uma vez definidos os casos de tese, é conduzida uma campanha de injeção de ataques num ambiente controlado em que os casos de teste são executados no sistema-alvo. A injeção é realizada a par da monitorização do servidor, de modo a detetar qualquer comportamento anómalo que possa indiciar a ativação de uma vulnerabilidade. A monitorização é um aspeto essencial na injeção de ataques e por isso foram definidas diferentes maneiras de a concretizar. Dois aspetos essenciais tiveram de ser tomados em consideração. Por um lado, o tipo de monitor utilizado pode determinar as classes de vulnerabilidades que se podem detetar. Por outro lado, quanto maior a capacidade do monitor, mais requisitos são exigidos ao seu ambiente de execução, o que pode tornar a sua operação demasiado complexa e/ou lenta (i.e., pode afetar o correto funcionamento do sistema-alvo e a respetiva avaliação).

O passo final na injeção de ataques é procurar por indícios de anomalias no comportamento do sistema-alvo. Visto que a execução do sistema-alvo é ditada pela forma como este processa os ataques, qualquer comportamento fora do esperado indica que uma potencial vulnerabilidade foi despoletada. O ataque que causou o erro (tipicamente o último) poderá então ser utilizado como caso de teste que permitirá reproduzir a anomalia e localizar a vulnerabilidade (e.g., erro no código ou na configuração). Dois tipos de abordagens foram aqui seguidos. No primeiro são definidas várias classes de comportamento anómalo, típicas de vulnerabilidades conhecidas, que depois são procuradas nas respostas e na monitorização do sistema-alvo. Esta solução permite descobrir rápida e automaticamente vários tipos de vulnerabilidades cujos efeitos são bem conhecidos (e.g. *crash*).

A segunda solução, estende a especificação do protocolo com dados de monitorização interna, de modo a definir o comportamento correto do sistema-alvo para todo o espaço do protocolo. Esta especificação estendida, chamada de perfil comportamental, é depois comparada com a execução do sistema-alvo enquanto processa os ataques. Qualquer comportamento desviante é então assinalado como causado por uma potencial vulnerabilidade. Uma análise experimental utilizando esta abordagem permitiu concluir que a definição do perfil comportamental permite detetar os diferentes tipos de execução faltosa.

Algumas das ideias aqui apresentadas foram também aplicadas na criação de outros tipos de mecanismos de segurança. Nomeadamente, assistindo na criação de um sistema tolerante a intrusões utilizando diversidade, e no desenvolvimento de sistemas de deteção de intrusões mais capazes e adequados a sistemas críticos.

Acknowledgements

First and foremost, I want to acknowledge my advisor, Professor Nuno Ferreira Neves. His support and guidance was crucial to the success of this doctoral work, and I am appreciative of the confidence and freedom he has indulged me throughout this journey. He has been an inspiration for pursuing perfection and for becoming a better academic researcher. Thank you!

I would also like to thank Professor Paulo Veríssimo. He is a cornerstone in the research status of the group and his strong leadership and insights are evident throughout the accomplishments of both the Navigators team and of its current and past members.

I would like to acknowledge the University of Lisboa. These walls have shaped and formed many men and women since 1911 and it vows to lead its students *ad lucem* (to the light) of knowledge. This has been my home for more than a decade and is not without a misty eye that I now leave. I would like to thank a few notable individuals from this community, in particular, from the Faculty of Sciences, for their invaluable assistance, expertise, and friendship: Professors Miguel Correia, António Casimiro, Alysson Bessani, Ana Respício, Luís Correia from the Department of Informatics (DI) and Professor João Gomes from the Department of Statistics and Operational Research (DEIO).

A very special and kind thanks goes to all my past and present colleagues at the Large-Scale Informatics Systems Laboratory (LaSIGE) research laboratory, for their friendship and tireless support. In particular, thank you very much Giuliana, Mônica, Vinicius, Letícia, Henrique, André, Simão, Bruno, Tiago, and Miguel.

Finally, I gratefully acknowledge the funding sources that made this doctoral work possible. I was supported by the Fundação para a Ciência e Tecnologia (FCT) through the Doctoral Degree Grant SFRH/BD/-44336/2008, through the projects POSC/EIA/61643/2004 (AJECT) and PTDC/EIA-EIA/100894/2008 (DIVERSE), and by the Multi-annual and CMU-Portugal Programmes. In addition, this work was also supported by the European Commission through projects IST-2004-27513 (CRUTIAL) and FP7-257475 (MASSIF).

Ao meu pai.

CONTENTS

List of Figures	xix
List of Tables	xxiii
List of Algorithms and Listings	xxv
List of Acronyms	xxvii
List of Publications	xxxi
1 Introduction	1
1.1 Objective and Main Areas of Work	4
1.2 Summary of Contributions	9
1.3 Structure of the Thesis	11
2 Related Work	13
2.1 Detection of Faults and Vulnerabilities	14
2.1.1 Fault injection	14
2.1.2 Manual analysis and test oracle	17
2.1.3 Model checking	19
2.1.4 Static analysis	22

2.1.5	Robustness testing and fuzzing	27
2.1.6	Vulnerability scanners	29
2.1.7	Run-time detection	30
2.1.8	Resource usage detection	33
2.2	Protocol Specification	35
2.2.1	Manually defined	36
2.2.2	Dynamic analysis	38
2.2.3	Grammatical induction	40
2.3	Generation of Test Cases	43
2.3.1	Combinatorial test data generation	44
2.3.2	Random data generation and fuzzing	45
2.3.3	Dynamic analysis test data generation	47
2.3.4	Symbolic execution test data generation	47
2.3.5	Specification-based test data generation	49
3	Network Attack Injection Framework	53
3.1	General Methodology	54
3.2	Overview of the Framework	59
3.3	Protocol Specification	61
3.4	Attack Generation	63
3.5	Injection & Monitoring	64
3.6	Attack Analysis	68
3.7	Conclusions	69
4	Protocol Specification	71
4.1	Communication Protocol	72
4.1.1	Protocol language	74
4.1.2	Protocol state machine	75
4.2	Manual Protocol Specification	76
4.3	Protocol Reverse Engineering	78
4.3.1	Inferring the language	81
4.3.2	Inferring the protocol state machine	90
4.3.3	Input versus input/output state machine	95
4.3.4	Experimental evaluation	98

4.4	Automatically Complementing a Specification	111
4.4.1	Overview of the approach	113
4.4.2	Experimental evaluation	120
4.5	Conclusions	123
5	Attack Generation	125
5.1	Combinatorial Test Case Generation	126
5.1.1	Delimiter test definition	128
5.1.2	Syntax test definition	131
5.1.3	Value test definition	132
5.1.4	Privileged access violation test definition	135
5.1.5	Experimental evaluation	136
5.2	Recycling-based Test Case Generation	137
5.2.1	Overview of the approach	139
5.2.2	Tool implementation	142
5.2.3	Experimental evaluation	150
5.3	Conclusions	163
6	Injection, Monitoring, and Analysis	165
6.1	Injection	166
6.1.1	Single injection campaign with restart	166
6.1.2	Single injection campaign without restart	167
6.1.3	Repeated injection campaign with restart	168
6.2	Monitoring	169
6.2.1	External monitor	170
6.2.2	Generic internal monitor	171
6.2.3	Specialized internal monitor	172
6.3	Analysis of the Attacks	174
6.3.1	Analysis through fault pattern detection	175
6.3.2	Analysis with a resource usage profile	176
6.3.3	Analysis with a behavioral profile	183
6.4	Conclusions	192

7	Attack Injection Tools	195
7.1	AJECT	196
7.1.1	Architecture and implementation	197
7.1.2	Experimental evaluation	199
7.2	PREDATOR	215
7.2.1	Architecture and implementation	216
7.2.2	Experimental evaluation	219
7.3	REVEAL	230
7.3.1	Architecture and implementation	230
7.3.2	Experimental evaluation	232
7.4	Conclusions	241
8	Applications of the Developed Techniques	245
8.1	DIVEINTO: Supporting Diversity in IT Systems	246
8.1.1	Overview of an IT system	248
8.1.2	Classification of violations	250
8.1.3	Methodology	254
8.1.4	Tool implementation	262
8.1.5	Experimental evaluation	263
8.1.6	Case-study	272
8.2	Adaptive IDS for Critical Servers	276
8.2.1	Methodology	277
8.3	Conclusions	281
9	Conclusions and Future Work	283
9.1	Conclusions	284
9.2	Future Work	287
	Bibliography	291
	Index	311

LIST OF FIGURES

3.1	The attack process on a faulty (or vulnerable) component.	55
3.2	The attack injection methodology.	57
3.3	Framework of network attack injection.	60
4.1	Protocol input language of the FTP.	75
4.2	Protocol state machine of the FTP.	76
4.3	Screenshot of the AJECT protocol specification.	77
4.4	ReverX overview for inferring an input/output specification.	79
4.5	Example of an FTP network trace.	83
4.6	Inference of the protocol language.	84
4.7	Inferred output language.	90
4.8	Input/output state machine inference.	93
4.9	Input and input/output state machine inference.	97
4.10	Additional session in the network trace.	98
4.11	Impact of the sample size on the language inference.	105
4.12	Inferred protocol language versus RFC 959.	107

4.13	Impact of the sample size on the state machine inference.	109
4.14	Inferred protocol state machine versus RFC 959.	110
4.15	ReverX average execution time.	111
4.16	FTP input state machine complemented.	122
5.1	Recycling-based test case generation approach.	139
5.2	Architecture of the implemented tool.	142
5.3	Reverse engineering the protocol specification.	145
5.4	Analysis of the true vulnerability coverage of experiment 3.	161
6.1	Range of monitoring solutions.	170
6.2	Specialized internal monitor.	172
6.3	Subset of the FTP specification.	185
6.4	Subset of the FTP specification extended with monitoring data. . .	187
6.5	Obtaining a behavioral profile (learning phase).	189
6.6	Using the behavioral profile (testing phase).	191
7.1	Architecture of the AJECT tool.	198
7.2	Input state machine of the POP3 protocol.	200
7.3	Input state machine of the IMAP4Rev1 protocol.	201
7.4	Architecture of the PREDATOR tool.	217
7.5	DNS message format.	219
7.6	Memory usage profiles for the D ₁ synthetic server (with thread leak).226	
7.7	Memory consumption in MaraDNS.	229
7.8	Architecture of the REVEAL tool.	231
7.9	Vulnerability detection of test cases (Experiment 1).	237
7.10	Vulnerability detection of test cases (Experiment 2).	238
7.11	Vulnerability detection of test cases (Experiment 3).	238
7.12	Vulnerability detection of test cases (Experiment 4).	239

7.13 Vulnerability detection of test cases (Experiment 5).	240
8.1 IT system architecture.	249
8.2 Methodology overview.	255
8.3 Input/output state machine of an example protocol.	256
8.4 Behavior model for FTP server IIS6 with conformance violations from IIS5	268
8.5 Inferring the behavioral profile (learning phase)	278
8.6 Using the behavioral profile (operational phase)	280

LIST OF TABLES

4.1	Characterization of the training sets.	100
4.2	Evaluation of the inferred input and output languages.	104
4.3	Discovered message formats and respective RFC extensions. . . .	121
5.1	Example of a set of test cases and the respective extracted payloads.	141
5.2	Target server transition table and prelude generation.	147
5.3	Test case generation example.	149
5.4	Test case generation tools for FTP protocol.	151
5.5	Coverage of the FTP protocol space.	152
5.6	Taxonomy of payloads from exploits of 131 FTP vulnerabilities. . .	156
5.7	Potential vulnerability coverage of FTP test cases (part 1).	156
5.8	Potential vulnerability coverage of FTP test cases (part 2).	157
5.9	Test case generation tools for non-FTP protocols.	158
5.10	Potential vulnerability coverage of non-FTP test cases (part 1). . . .	159
5.11	Potential vulnerability coverage of non-FTP test cases (part 2). . . .	160

7.1	Target POP and IMAP e-mail servers.	204
7.2	E-mail servers with newly discovered vulnerabilities.	209
7.3	Synthetic leak servers with resource leaks.	221
7.4	Projections for a disk and memory leak created from n injections. .	222
7.5	Resource usage projections for the synthetic leak servers (with $p = 2$).224	
7.6	R_a^2 and MSE for the resource usage profiles for the synthetic leak servers.	225
7.7	Resource usage profiles for the DNS servers.	228
7.8	Reported known FTP vulnerabilities.	233
8.1	Reference and testing traces.	258
8.2	Correlation table.	259
8.3	OS and server configuration of the three IT scenarios.	264
8.4	Violations detected on the FTP replication scenario.	267
8.5	Violations detected on the SMTP replication scenario.	270
8.6	Violations detected on the POP replication scenario.	271

LIST OF ALGORITHMS AND LISTINGS

4.1	Generalization of the protocol language.	87
4.2	Merging process to produce the protocol input/output state machine.	94
4.3	Complementing the language of an existing protocol specification.	115
4.4	Complementing the state machine of an existing protocol specification.	119
5.1	Algorithm for the Delimiter Test generation.	130
5.2	Algorithm for the Syntax Test generation.	131
5.3	Algorithm for the Value Test generation.	133
5.4	Algorithm for the generation of malicious strings.	134
7.1	File with malicious tokens for POP protocol.	206
7.2	File with malicious tokens for IMAP protocol.	206
8.1	Detected violations between IIS6 and IIS5.	269
8.2	Excerpt of the normalizer (Python).	274

LIST OF ACRONYMS

ABNF Augmented Backus-Naur Form

AJECT Attack InJEction Tool

API Application Programming Interface

ARP Address Resolution Protocol

ASN.1 Abstract Syntax Notation One

AST Abstract Syntax Tree

BNF Backus-Naur Form

COTS Commercial Off-The-Shelf

CR Carriage Return

CT Command names Threshold

CVE Common Vulnerabilities and Exposures

DIVEINTO DIVERse INtrusion TOLerant systems

DNS Domain Name System

DoS Denial-of-Service

DS Distinguishing Sequence

FSM Finite-State Machine

FTP File Transfer Protocol

GUI Graphical User Interface

ID identifier

IDS Intrusion Detection System

IEEE Institute of Electrical and Electronics Engineers

IETF Internet Engineering Task Force

IMAP Internet Message Access Protocol

ISO International Organization for Standardization

IT Intrusion Tolerance

ITU-T International Telecommunication Union Telecommunication Standardization Sector

LBL Lawrence Berkeley National Laboratory

LF Line Feed

LTL Linear Temporal Logic

IP Internet Protocol

MSE Mean Square Error

NP Nondeterministic Polynomial time

OS Operating System

POP Post Office Protocol

PREDATOR PREDicting ATtacks On Resources

PTA Prefix Tree Acceptor

REVEAL REvealing Vulnerabilities with bEhAvioral profiLe

RFC Request For Comments

SDL Specification and Description Language

SIP Session Initiation Protocol

SMTP Simple Mail Transfer Protocol

SQL Structured Query Language

SWIFI Software-Implemented Fault Injection

TCP Transmission Control Protocol

UIO Unique Input/Output

VAT Vulnerability Assessment Tool

VDM-SL Vienna Development Method Specification Language

VM Virtual Machine

VT Variability Threshold

W3C World Wide Web Consortium

XSS cross-site scripting

LIST OF PUBLICATIONS

International Conferences and Journals

- [P1] Antunes, J. and Neves, N., *Inferring a Protocol Specification from Network Traces*. Submitted for publication in a journal.
- [P2] Antunes, J. and Neves, N., *Recycling Test Cases to Detect Security Vulnerabilities*. Submitted for publication in a conference.
- [P3] Antunes, J. and Neves, N., *Using Behavioral Profiles to Detect Software Flaws in Network Servers*. In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), Hiroshima, Japan, November 2011.
- [P4] Antunes, J., Neves, N., and Verissimo, P., *ReverX: Reverse Engineering of Protocols*. In Proceedings of the Working Conference On Reverse Engineering (WCRE), Limerick, Ireland, October 2011.
- [P5] Antunes, J. and Neves, N., *DiveInto: Supporting Diversity in Intrusion-Tolerant Systems*. In Proceedings of the International Symposium on Reliable Distributed Systems (SRDS), Madrid, Spain, October 2011.
- [P6] Antunes, J. and Neves, N., *Automatically Complementing Protocol Specifications From Network Traces*. In Proceedings of the European Workshop on Dependable Computing (EWDC), Pisa, Italy, May 2011.

- [P7] Antunes, J., Neves, N., Correia, M., Verissimo, P., and Rui Neves. *Vulnerability Discovery With Attack Injection*. IEEE Transactions on Software Engineering, 36:357-370, May/June 2010.
- [P8] Antunes, J., Neves, N., and Verissimo, P., *Detection and Prediction of Resource-Exhaustion Vulnerabilities*. In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE), Seattle, WA, USA, November 2008.

National Conferences, Fast Abstracts, Student Papers

- [P9] Antunes, J. and Neves, N., *Adaptive Monitoring to Detect Intrusions in Critical Servers*. In Fast Abstract in Supplement of the International Conference on Dependable Systems and Networks (DSN), Boston, MA, USA, June 2012.
- [P10] Antunes, J., Neves, N., and Verissimo, P., *Using Attack Injection on Closed Protocols*. In Fast Abstract in Supplement of the International Conference on Dependable Systems and Networks (DSN), Chicago, USA, June 2010.
- [P11] Antunes, J. and Neves, N., *Building an Automaton Towards Protocol Reverse Engineering*. In Proceedings of the Simpósio de Informática Inforum, Lisboa, Portugal, September 2009.
- [P12] Antunes, J., Neves, N., and Verissimo, P., *Finding Local Resource Exhaustion Vulnerabilities*. In Student paper at the International Symposium on Software Reliability Engineering (ISSRE), Trollhättan, Sweden, November 2007.
- [P13] Teixeira, E., Antunes, J., and Neves, N., *Avaliação de Ferramentas de Análise Estática de Código para Detecção de Vulnerabilidades*. In Proceedings of the Segurança Informática nas Organizações (SINO), Lisboa, Portugal, November 2007.

Technical Reports

- [P14] Antunes, J., Neves, N., and Verissimo, P., *ReverX: Reverse Engineering of Protocols*. Technical Report TR-2011-01, Faculdade de Ciências da Universidade de Lisboa, January 2011.
- [P15] Franceschinis, G., Alata, E., Antunes, J., Beitollah, H., Bessani, A., Correia, M., Dantas, W., Deconinck, G., Kaâniche, M., Neves, N., Nicomette, V.,

Sousa, P., and Verissimo, P., *Experimental Validation of Architectural Solutions (CRUTIAL Deliverable D20)*, Technical Report TR-2009-008, Faculdade de Ciências da Universidade de Lisboa, March 2009.

CHAPTER I

INTRODUCTION

In recent history, we have witnessed a growing number of scientific and technological innovations that are critical to the advancement of knowledge, the propagation of culture, and the generation of wealth. The Internet was one of those innovations and it had a profound impact on society, revolutionizing the way we communicate and live.

It is estimated that the number of Internet users exceeded 2.4 billion in 2011, more than the double of 2006 ([ITU, 2011](#)). In parallel with its increasing popularity, the Internet has become a major economic driver, representing a trading volume of almost 6 trillion Euros worldwide per year. It is estimated that the Web represents on average about 3% of GDP in the developed countries or in countries with strong economic growth, surpassing the agriculture and energy sectors¹

¹This study included the G8 countries, the emerging economies of China, India and Brazil,

([Rausas et al., 2011](#)).

The Internet is therefore seen as an essential service to society and economy, generating jobs and knowledge, changing lifestyles and even having a political impact, such as in the recent uprisings in the Middle East and North Africa supported by social networking ([Coz, 2011](#)). This dependence is further aggravated by the Internet's role in the management of some critical infrastructures, without which society, as we know it, could not survive. Transport, communication, energy, and banking, are some of the services that we depend on and are controlled and operated through the Internet. The disruption of these services can have serious and devastating consequences ([US-Canada Power System Outage Task Force, 2006](#); [Castle, 2006](#)), making the Internet an extremely sensitive point of failure, both to physical and cyber attacks.

The security compromise of a server on the Internet may affect not only its users, but could also trigger a cascade effect with serious repercussions on other services ([Sharma, 2010](#)). Several recent studies and surveys reveal that the number of attacks is escalating and they are becoming increasingly more efficient and sophisticated ([Dobbins and Morales, 2011](#); [McAfee, 2011](#); [Symantec, 2012](#)). These attacks are no longer plain and harmless demonstrations of power between hackers, but are ill intended and perpetrated by carefully organized and well-financed groups. Their motives range from personal profit and activism, to industrial and inter-governmental sabotage, and even terrorism ([Garg et al., 2003](#); [Anderson and Nagaraja, 2009](#); [Krekel, 2009](#); [Kanich et al., 2011](#); [Rashid, 2011](#)).

Consequently, protecting the network servers from being exploited becomes essential to the correct and safe operation of the Internet. Sometimes, however, and two countries with high broadband penetration, Sweden and South Korea.

security related tasks are delegated to the background, in part because they are not perceived as fundamental as the inclusion of new functionality. In fact, most of the effort in development typically goes to the creation of new features, since they are commercially more appealing. This makes, for example, testing operations increasingly complex and expensive, as the correctness of more functional requirements needs to be checked, together with non-functional aspects, such as ensuring acceptable performance levels and the safety and reliability of systems.

Software testing is the first line of defense in preventing attacks because it can support the discovery and removal of vulnerabilities. However, searching for errors or flaws in applications and servers is a painstaking and error-prone process, traditionally performed by humans. Consequently, the classical methods of testing have invariably overlooked errors. On the other hand, the methods carried out by attackers to find software flaws have been proved quite effective, which is clearly visible in the growing number of reported vulnerabilities ([Fossi et al., 2011](#)).

As a result, newer mechanisms to increase the correctness of systems are needed, in particular with regard to security flaws. This doctoral work contributes with a new methodology for vulnerability discovery, inspired by a technique used by some attackers and computer security analysts, in which they persistently experiment different forms of attack, looking for evidence that a vulnerability was activated. Our attack injection methodology follows a similar approach by systematically generating and injecting test cases (or attacks), while monitoring and analyzing the target system ([Neves et al., 2006](#)). An attack that triggers some unexpected behavior provides a strong indication of the presence of a flaw that could be exploited by that attack or by some variation of it. This attack can then be

given to the developers as a test case that reproduces the anomaly, to assist in the removal of the flaw.

The attack injection methodology can be applied to most types of components that one may wish to search for vulnerabilities. In the thesis, we narrowed our study to network servers because, from a security point of view, they are probably the most interesting class of target systems. First, these large and often complex applications are designed to sustain long periods of uninterrupted operation and are usually accessible through the Internet. Second, an intrusion in a network server usually has a significant impact, since a corruption on the server may compromise the security of all clients (e.g., if the adversary gets a root shell). Consequently, network servers are a highly coveted target by malicious hackers.

1.1 Objective and Main Areas of Work

The main focus of this work is to provide a theoretical and experimental framework for the implementation and execution of attack injection on network servers, with the aim of finding vulnerabilities in an automated and systematic way. The thesis covers several aspects related to this approach, including the inference of a specification of the protocol implemented by the server, the generation of test cases and their injection, and the monitoring of the system to detect failures. Naturally, such a broad topic comes into contact with various fields of expertise, each with its own wealth of open problems and techniques. Therefore, we had to limit the scope of the work and concentrate on a particular approach of attack injection, unveiling some of its challenges and benefits, and offering original solutions to address each step of the methodology. Future works can then take these ideas

and further explore them to develop tools and methodologies that can support the development of more secure and dependable computer systems.

In the rest of this section, we give an overview of the main areas of investigation that are covered in the thesis.

Attack injection

The fundamental objective of this research is the design of a methodology that covers all the steps of automated attack injection. We define this overall process in the context of an attack injection framework that is able to accommodate different techniques that can be used during the discovery of security vulnerabilities.

The complete framework was implemented in three attack injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#). [AJECT](#) is the archetypal attack injection tool that adapts and extends the traditional techniques of fault injection to look for security vulnerabilities. This tool was used in several experiments with network servers, demonstrating that it can discover new vulnerabilities. [PREDATOR](#) is a tool that evolved from [AJECT](#), but is focused on accelerating the software aging effects of network servers and in detecting vulnerabilities related to the inappropriate use of resources. This kind of vulnerabilities is typically difficult to detect because its effects are not immediately perceived. In fact, virtually any network server connected to the Internet is susceptible to the exhaustion of resources when faced with an overwhelming number of requests (e.g., caused by a Slashdot effect ([Adler, 1999](#)) or by a concerted distributed network attack ([Lau et al., 2000](#))), and therefore, it is important to understand the extent that a server is vulnerable to this problem. [REVEAL](#) is another specialization of [AJECT](#), which implements an automated analysis approach for detecting abnormal behaviors. Flaws can then

be discovered during a testing phase if an attack causes the server to execute in an way that violates a previously inferred behavioral profile that models the correct execution of the target system.

Automatic inference of a protocol specification

A network server provides a service by exchanging messages with the clients, and therefore it is appropriate to model a server by deriving a specification of the communication protocol that it implements. For instance, a File Transfer Protocol (FTP) server is modeled after the protocol specified by the Internet Engineering Task Force (IETF) in the Request For Comments (RFC) 959 (Postel and Reynolds, 1985). This specification can then be used in several components of the framework, such as in the generation of more effective test cases or to support the thorough monitoring of the server.

However, public specifications, such as those published by the IETF and other standardization organizations, are not always available, such as when the server implements a proprietary (or closed) protocol. Furthermore, even when the protocol specification is accessible, manually translating the documentation into a formal model is a time consuming and error-prone process. The thesis presents a novel protocol reverse engineering approach that uses network messages to and from the server to automatically derive a specification of the protocol, capturing the syntax of the messages and the protocol state machine. This solution is based on the problem of inducing grammars, and it consists in deriving a formal model from a finite subset of examples.

The applications of this work are not limited to attack injection. Once inferred, the specification may also be used in other testing approaches, such as conformity

testing ([Dahbura et al., 1990](#)), or to create more effective defense mechanisms, namely firewalls ([Ingham and Forrest, 2006](#); [Peng et al., 2007](#)) or Intrusion Detection Systems (IDSs) ([Roesch, 1999](#); [Paxson, 1999](#); [Hoagland and Staniford, 2009](#)). Later in the document, we explain how the protocol reverse engineering mechanism can help on the deployment of diverse replicated systems and monitoring of critical servers.

Automatic generation of attacks

The attacks generated by a tool should exercise the entire functionality provided by the target system and cover the most relevant input domain. In practice, it is unfeasible to create test cases for all possible input values. Although it has been proven that there exists a finite set of test cases that can reliably determine whether a given program is correct, obtaining such a set is an undecidable problem ([Howden, 1976](#)).

The thesis proposes two solutions for obtaining a finite set of test cases that are both exhaustive (covering the whole specification) and comprehensive (searching for several classes of vulnerability). Both solutions resort to malicious payloads, which have been observed in most of the reported network attacks, in order to determine if it is viable to circumvent the protection mechanisms implemented in the target system.

One solution is based on a combinatorial generation of test cases, such as the pairwise testing approach, where different values consisting of legal and malicious payloads are combined. Another approach consists in recycling existing test cases, such as those generated by specialized security tools, to produce a manageable number of attacks with a high coverage of different classes of vul-

nerabilities.

Monitoring and behavior inference

Analyzing the results of the injection of attacks to seek evidence of the activation of vulnerabilities, such as errors or system failures, can be a challenging problem. The thesis proposes two approaches to accomplish this objective. One solution involves explicitly looking for characteristics of known flaws, such as memory access violations or crashes. Depending on the type of error (or failure), its identification can be quite trivial or require a careful observation of the target system to identify more subtle signs of anomalies.

A more sophisticated solution consists in inferring a description of the correct behavior of the target system. For this purpose, we devised a technique that builds a behavioral profile by extending the protocol specification of the server with information about its internal execution. This mechanism is thus able to draw a complete and detailed picture of the expected behavior of the target system, allowing it to determine when a vulnerability is activated.

The information about the server's behavior while processing the attacks is provided by a monitor component, which can be implemented through different methods with varying abilities and restrictions. Three distinct types of monitors were developed and studied: a thorough monitor, with detailed and accurate monitoring capabilities, but restricted to Unix-based systems; a universal and simple monitor, but with no support for resource usage or execution tracing; and a remote monitor that infers the status of the server through an additional network connection.

Applications to other domains

Finally, some of the solutions that resulted from this work have also been applied to other areas. One research path that was followed uses the protocol reverse engineering techniques as an effective way to compare the behavior of diverse replicas in Intrusion Tolerance (IT) systems. This approach supports the evaluation of the compatibility of different implementations and configurations of replicas. A tool implementing this approach, [DIVEINTO](#), was developed and applied to three replication scenarios. The experiments demonstrate that various sorts of violations, including problems related to nondeterministic execution, can be revealed by this approach.

Another possible application that was briefly addressed is the use of the behavioral profile analysis to implement a more detailed [IDS](#) for critical servers. This novel approach differs from previous [IDS](#) solutions by providing a means of selectively inspecting the execution of the most sensible protocol tasks. Such an [IDS](#) can thus attain a greater level of security and protection, while still maintaining a good overall throughput and performance.

1.2 Summary of Contributions

This section summarizes the most important contributions that resulted from this work.

Network attack injection framework

The main contribution of this thesis is an attack injection framework that encompasses the various aspects related to the generation and injection of attacks, the monitoring of the target system, and the analysis of results. Three attack injection tools were developed, [AJECT](#), [PREDATOR](#), and [REVEAL](#), which implement alternative injection and monitoring approaches, test case generation algorithms, and provide a basis for the automated analysis of the results. These tools and techniques have been integrated and recently released in the open source project [AJECT](#)². Most documentation and experimental results pertaining to the attack injection framework were published in P3, P7, P8, P12, and P15, and the recently submitted P2.

Protocol reverse engineering

Communication protocols determine how network components interact with each other. The ability to derive a specification of a protocol can be useful in many contexts, such as to support deeper black-box testing techniques or the development of effective network defense mechanisms. Unfortunately, since it is very hard to obtain these specifications in an automatic way, little work has been done in this area in the past. The thesis provides a solution based on grammar induction to infer a specification of a protocol from samples of network traces. This approach was implemented in an open source tool called ReverX³ and it is used in several instances of the thesis. The results of using ReverX show that it is able to infer protocol specifications with high levels of precision and recall. The most

²<http://code.google.com/p/aject/>

³<http://code.google.com/p/reverx/>

relevant publications are: P4, P6, P11, and the recently submitted P1.

Diversity for IT systems

The final goal of attack injection is to prevent intrusions by removing vulnerabilities. Interestingly, the thesis also contributed to the areas of intrusion tolerance and intrusion detection. In particular, a new methodology was created to assist the introduction of diversity in [IT](#) systems and to evaluate the compliance of diverse replicated components. A tool implementing this methodology, [DIVEINTO](#), was used to identify several incompatibilities between diverse implementations and configurations of replicas. The findings were published in P5.

1.3 Structure of the Thesis

This thesis is organized as follows:

Chapter [2](#) provides the context in which the thesis appears and presents related work in the literature.

Chapter [3](#) describes the overall attack injection framework and its components, which are detailed in the rest of the thesis.

Chapter [4](#) deals with the problem of obtaining a specification of the communication protocol of the target system, which is the means for the injection of attacks and also to assist in the identification of anomalous behavior.

Chapter [5](#) addresses the generation of attacks by using a specification of the communication protocol and by resorting to malicious payloads present in known exploits and test cases created by security tools.

Chapter [6](#) presents different injection strategies and monitoring solutions, and

discusses the problem identifying anomalies in the behavior of the target system.

Chapter 7 introduces three attack injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#), that implement and evaluate many of the techniques described in the thesis.

Chapter 8 offers a few applications of some of the techniques that resulted from this work. One application is a methodology and a tool to evaluate the compatibility of diverse replicas in [IT](#) systems. Another application is based on the inference of the correct behavior of the target system to identify intrusions in real-time.

The thesis ends with conclusions and future work in Chapter 9.

CHAPTER 2

RELATED WORK

The thesis investigates and explores a new approach for the discovery of vulnerabilities in network servers. At its core, it resorts to the injection of previously generated attacks and to their respective analysis. There are, however, numerous challenges revolving around this approach that must be addressed, such as the construction of specifications to assist the generation of test cases, the injection and monitoring of the attacks, and the automatic identification of anomalies. Consequently, this work touches many research areas and has been influenced by a large body of work from the scientific literature.

We organized the related work in three main areas of interest. The first section deals with the underlying and fundamental problem of finding faults, and in particular, security vulnerabilities, and it discusses several testing approaches,

including the verification and validation of software and the monitoring of applications. Then, we review solutions concerning the specification of protocols, which is an essential component in network attack injection. Finally, we look in more detail into the problem of generating test cases that are aimed at discovering software flaws.

2.1 *Detection of Faults and Vulnerabilities*

In the next subsection, we provide a brief overview of the techniques used in the injection of faults in software, which has strongly influenced attack injection. After that, we review some of the methods used to detect errors in software, such as manually inspecting the code or automating this process with a test oracle, model checking, and static analysis. Then, the remaining subsections are devoted to the problem of finding vulnerabilities, namely fuzzing, scanners, and mechanisms that allow the detection of security flaws at run-time and by monitoring the resource utilization.

2.1.1 *Fault injection*

One of the core concepts in this thesis is the injection of faults, which in general terms consists in purposely introducing one or more faults in the system for evaluation and testing purposes.

Fault injection is a classical experimental approach for the verification of fault handling mechanisms (fault removal) and for the estimation of various parameters that characterize an operational system (fault forecasting), such as fault coverage and error latency ([Avizienis and Rennels, 1972](#); [Arlat et al., 1993](#)). Traditionally,

fault injection has been utilized to emulate several kinds of hardware problems, ranging from transient memory corruption to permanent stuck-at faults. Different methods and tools have been proposed, initially injecting faults at the hardware level (logical or electrical faults) (Arlat et al., 1989; Gunneflo et al., 1989) and, more recently, at the software level (code or data corruption) (Segall et al., 1988; Kanawati et al., 1992; Hsueh et al., 1997).

The idea behind fault injection is to evaluate the system's execution in the presence of artificially inserted faults. These injected faults try to mimic realistic hardware or software faults, and therefore, it evaluates the system's ability to cope with real faults. However, physical injection techniques become impractical in more complex hardware systems. To address this difficulty, some works use hardware description models and functional simulation tools, instead of the actual hardware, to simulate the activation of hardware faults (Choi and Iyer, 1992; Jenn et al., 1994; Goswami et al., 1997; Kaâniche et al., 2001). The main advantage of resorting to functional simulation is that it can be used in the early stages of design to predict the behavior of hardware and software architectures, without any target prototype or special evaluation hardware. This approach provides great controllability to inject faults in components of the simulation model of the target system. However, the accuracy of the results depends on the level of abstraction of the model, which can never be as realistic as the actual target system.

Software-Implemented Fault Injection (SWIFI) is a low-cost and easily controllable alternative to physical fault injection in real target systems. SWIFI is less expensive because it does not require special hardware instrumentation to inject faults. This is usually achieved by changing the memory or register contents based

on specified fault models that emulate the effects of hardware faults (Arlat et al., 1989; Arlat et al., 2003) or by mimicking regular high-level programming errors (Durães and Madeira, 2003). Even though *SWIFI* can emulate the error behavior caused by most faults, it can only inject faults into locations that are accessible to software (e.g., by altering bits in program instructions, address pointers or data, or by changing entire sequences of instructions). Moreover, modifying the target system (e.g., by adding special instructions, software traps) can disturb the observed results, something that should be minimized as much as possible because it might affect the evaluation. However, given the greater sophistication of software-implemented faults, *SWIFI* can be used to test more complex systems, such as applications or even Operating Systems (OSs). Over the years, several *SWIFI* tools were developed, and few examples are: Xception (Carreira et al., 1998), FERRARI (Kanawati et al., 1995), FTAPE (Tsai and Iyer, 1995) and GOOFI (Aidemark et al., 2001).

Fault injection was also applied to insert software bugs in applications. G-SWIFIT is a sophisticated tool that resorts to a library of mutation operators to determine how and where to insert software faults in binary code (Durães and Madeira, 2002, 2006). The library of mutations was obtained in a field data study, whose aim was to find the most frequent high-level programming errors and their translation to low-level code (Durães and Madeira, 2003). The tool scans the binary for specific low-level instruction patterns and then performs mutations on those patterns to emulate high-level problems. This approach has the advantage of not requiring access to the source code. However, it relies on the correctness and completeness of an up-to-date library that supports the set of emulated faults.

An evolution from the previous work consists in injecting vulnerabilities, i.e.,

programming bugs that can be exploited by an adversary. One approach builds on this idea to evaluate the security mechanisms of IDS and vulnerability scanners, by injecting vulnerabilities in Web applications, and the respective attacks that should trigger them (Fonseca et al., 2009). Based on a field data study of the most important real world Web security flaws, two types of vulnerabilities are emulated, cross-site scripting (XSS) and Structured Query Language (SQL) injection (Fonseca and Vieira, 2008). Both kinds of faults rely on the exploitation of variables, and therefore the source code is searched for all input variables that affect SQL queries. Vulnerabilities are then inserted through specific code mutations at determined locations, each one resulting in a new vulnerable version of the Web application. For each vulnerable version, a collection of attacks is generated to activate the injected flaw. An attack is considered successful if it can modify the structure of the SQL query, which would not happen if the vulnerability had not been injected and if the security mechanism under evaluation had been able to deter the attack. Albeit this approach does not try to uncover unknown vulnerabilities, it can be used to evaluate the effectiveness and correctness of different security protection mechanisms.

2.1.2 Manual analysis and test oracle

The most basic approach to testing, and in particular to the analysis of the outputs from the execution of the test cases, is to perform manual inspection. Automation is usually limited to the execution of the test cases, such as when tests are run overnight to present the results in the morning. Some testing tools then present large amounts of data that have to be interpreted. However, the intricate and complex nature of the test results requires a detailed knowledge about the

system, which is usually absent in automated test tools. Therefore, the final assessment of the results has been largely delegated to human testers. In fact, some empirical results show that most companies do little automated testing ([Anderson and Runeson, 2002](#)) and some writers even advocate the moderate use of test automation given its cost/value ratio ([Fewster and Graham, 1999](#)).

The analysis of the test case execution consists in observing its output to determine the correctness of the program or system. An *oracle* is a mechanism that can reliably determine whether the output of a test case is valid ([Howden, 1978](#)). To ascertain if a test passes or fails, the oracle compares the outcome of the test case with the correct and expected outputs. Finding the correct outputs is known in the scientific community as the oracle problem ([Ammann and Offutt, 2008](#)).

A *complete oracle* is able to provide correct outputs for any test case. One possible solution to build a complete oracle is to resort to the program specification to extract the expected results into a database and manually perform lookups ([Ammann and Offutt, 2008](#)). However, the number of expected outputs can be very large and render this approach unfeasible.

Some authors propose writing a high-level functional *specification*, with logical constraints that must be satisfied, as an implicit complete oracle ([Bousquet et al., 1998](#)). This specification can be constructed from formal descriptions, such as software environment constraints, functional and safety-oriented properties, software operational profiles and software behavior patterns. Oracles can also be derived from specification languages ([Hall, 1991](#); [Stocks and Carrington, 1996](#)). However, the main challenge is to provide a reliable and automated means of generating expected output.

Complete oracles can be expensive and even impossible, and manual oracles

are costly and unreliable. So, some effort has been put to automate the process of producing *incomplete oracles*. Researchers have suggested that automated oracles require a simulated model of system. One simple solution employs at least one additional version of the program that behaves correctly (Weyuker, 1982; Manolache and Kourie, 2001). This golden version, which must implement the same functionality as the program under test, is then run with the same inputs and the outputs are compared.

Another approach models the expected behavior of the program using a *decision table* (Di Lucca et al., 2002). A decision table is a suitable way of presenting the combination of conditions that affect the program execution and the respective output. Other solutions resort to heuristic approaches, such as AI planning (Memon et al., 2000) or neural networks (Vanmali et al., 2002).

2.1.3 Model checking

Model checking can be used to verify if a high-level requirement is violated or to prove that the system satisfies some property. A formal model that represents the system under evaluation is defined by an abstract specification, which simplifies many details of the actual implementation. Then, the reachable state space of the formal model is exhaustively explored, searching for violations of previously defined properties (Clarke et al., 1994, 2000). The thoroughness at exploring the state space of the system makes model checking a good method for finding errors in unusual system states. However, it requires the explicit definition of the properties being checked. Some example model checking tools are MOPS (Chen and Wagner, 2002), CMC (Musuvathi et al., 2002), and FiSC (Yang et al., 2006).

MOPS is a static analysis tool that checks if a program can violate specified se-

curity properties ([Chen and Wagner, 2002](#); [Chen et al., 2004](#)). Security properties are modeled by finite state automata and supplied to the tool. A security property might define, for instance, that a *setuid-root* program must drop root privileges before executing an untrusted program. Other examples of such properties include: creating *chroot* jails securely, avoiding race conditions when accessing the file system or when creating temporary files, and preventing attacks on standard file descriptors (e.g., standard input). MOPS exhaustively searches the control-flow graph of the program to check if any path may violate a safety property. Any violation is reported along with the execution path that caused it. In some experiments, MOPS was able to find errors in known network servers, such as Apache, Bind, OpenSSH, PostFix, Samba, and SendMail.

CMC is a model checker for C and C++ that executes the code directly, and therefore, it does not require a separate high-level specification of the system ([Musuvathi et al., 2002](#)). CMC emulates a virtual machine or an OS. The system is modeled as a collection of interacting concurrent processes, where each process runs unmodified C or C++ code from the actual implementation, which is scheduled and executed by CMC. Different scheduling decisions and other nondeterministic events give the ability to search the possible system states for violations to the correctness properties (e.g., the program must not access illegal memory or a particular function should not return an invalid object). One of the drawbacks of this approach is that the user is required to specify the correctness properties. In addition, the user is also responsible for building a test environment that adequately models the behavior of the system where the program is executed—this test environment fakes an OS and anything outside the system under evaluation.

Later on, CMC was used to create a model checking infrastructure for file systems called FiSC (Yang et al., 2006). This tool actually runs a Linux kernel in CMC. The model checker starts with a formatted disk, and recursively generates and checks successive states by executing state transitions based on actions induced by a file system test driver. Examples of these actions are: creating, removing, or renaming files, directories, and hard links; writing to and truncating files; or mounting and unmounting the file system. As each new state is generated, FiSC intercepts all disk writes, checking if the disk has reached a state that cannot be repaired (i.e., invalid file system).

High-level software requirements can be translated into Linear Temporal Logic (LTL) properties, which are then used as a specification of the formal model (Tan et al., 2004; Whalen et al., 2006). A model checker then looks in all states of the model for possible executions that violate the properties, generating the respective test cases, i.e., counterexamples that illustrate how the violation of the property can take place. The test cases are then used in the actual program to verify if there is actually a problem.

Model checking has an inherent limitation when security vulnerabilities are concerned. Since the causes for a vulnerability are not known in advance, it is hard to explicitly specify all security properties in the software requirements, making the resulting functional test cases unsuitable to detect them. Furthermore, model checking requires a formal model to be manually created or derived from the source code. Either approach is prone to generate incomplete specifications, thus resulting in formal models that might not be useful for software verification.

2.1.4 Static analysis

Static vulnerability analyzers search the source code of the applications for well-known dangerous constructs and report their findings (Chess and McGraw, 2004). The programmer then goes through the parts of the code for which there were warnings to determine whether the problem actually exists. This idea has also been extended to the analysis of binary code (Durães and Madeira, 2005).

Most of the work in this area has focused on finding buffer overflow vulnerabilities, although it has also been applied to other kinds of flaws, such as race conditions during the access of (temporary) files (Bishop and Dilger, 1996). Some of the tools examine the source code for a fixed set of unsafe patterns, or rules, and then provide a listing of their locations (Wagner et al., 2000; Viega et al., 2000; Larochelle and Evans, 2001; Haugh and Bishop, 2003).

Lexical analysis is one of the simplest forms of static code checking. Lexical analyzers usually look for unsafe library functions or system calls, such as glibc's functions *gets()* or *strcpy()*. The source code is fed to the analyzer, which parses the code into tokens that are then matched against a database of dangerous constructs (Viega et al., 2000; Wheeler, 2007; Fortify Software, Inc. 2009). This methodology is quite effective at locating programming errors, however, the tool will never find particular problems if the corresponding pattern has not been written. Additionally, these tools have the limitation of producing many false positives, i.e., warnings that should not be raised, because the way in which the suspicious pattern is used is not actually introducing a vulnerability.

As an example, ITS4 is a lexical analysis tool for scanning C and C++ code for vulnerabilities (Viega et al., 2000). This simple tool tries to automate a lot of

the manual source code inspection when performing security audits. ITS4 uses a database of dangerous patterns, such as function calls susceptible to buffer overflows, and parses the source code into lexical tokens. These tokens are then compared against the patterns in the database. Anything in the database is flagged, possibly resulting in a large number of false positives/warnings.

FindBugs is a lexical analyzer tool that looks for usual coding mistakes in Java programs ([Ayewah et al., 2008](#)). FindBugs is able to recognize numerous programming errors and dubious coding idioms by using simple analysis techniques. Moreover, it also helps to identify other difficult types of flaws, such as null pointer dereferencing, which require more specific solutions.

Static analysis can be improved by mimicking some of the features that make some programming languages more secure. One of these features is strong *type checking* (used in Java, for instance), which unfortunately some programming languages lack. The C language, for instance, was designed with space and performance considerations in mind, in an epoch where security was not a big concern. A C programmer can directly manipulate memory pointers, but he is entrusted with the responsibility of performing the boundary checks, which is sometimes neglected. Static analysis can impose stronger type checking by employing special annotations (e.g., type qualifiers) that are written as comments in the source code ([Evans et al., 1994](#); [Foster et al., 1999](#); [Wagner et al., 2000](#)). These additional keywords are ignored by the compiler, but are recognized by the static analyzer to give an indication about the data type and domain of the variables. This information can be used to enforce the correct usage of the variables according to their specified type, such as preventing positive integer variables to overflow to negative values.

A related approach resorts to an additional abstract data type to complement the buffer definition in C in order to detect buffer overflows. Memory allocations are characterized as pairs of integer ranges (specifying their reserved space) regardless of their contents (Wagner et al., 2000). This idea was incorporated in BOON, which formulates the buffer overrun detection problem as an integer constraint problem. The tool uses simple graph theoretic techniques to construct an efficient algorithm for solving integer constraints. Using this approach, detecting buffer overruns is a question of tracking the integer ranges of the abstract data type. First, a constraint language is used to model string operations, and then, integer range analysis solves the constraint system. Any constraint violation indicates a possible vulnerability.

CQual is a framework for adding type qualifiers to a programming language (Foster et al., 1999). The use of type qualifiers (such as the construct `dynamic nonzero int`) supports type checking and the inference of qualified types and their relationships. This framework can thus provide more sophisticated paradigms, such as polymorphism. In addition, CQual also supports *data-flow analysis* by adding tag information to some particular types of data (e.g., external input data) and tracking their execution paths (Shankar et al., 2001). Type inference rules are then constructed to detect inconsistencies on how the data is actually used. CQual uses this analysis to detect data coming from the outside that is interpreted as a format string in a function call, thus warning for potential format string vulnerabilities. The implementation of data-flow analysis is based on Perl’s taint mode (Birznies, 1998)—types are marked as *tainted* if originating or modified from the outside. The tool then tracks the propagation of the tainted data. If there is an execution path in which tainted data is passed to a function that expects

untainted data or if tainted data is interpreted as a format string, CQual raises an error.

LCLint started as an annotation-assisted static checking tool for finding buffer overflow vulnerabilities (Evans et al., 1994). Later on, more expressive annotations were added, allowing programmers to explicitly state function pre- and post-conditions (Larochelle and Evans, 2001). The annotations describe some assumptions about the buffers that are passed to functions and their expected state as functions return. For instance, the programmer can annotate the function to restrict the maximum size of a particular buffer to a global variable used in the code, or to specify the minimum and maximum buffer indices that can be read. LCLint combines traditional data-flow analysis with constraint generation and resolution. The tool generates constraints for the C statements by building an Abstract Syntax Tree (AST) and storing each constraint in the corresponding node. Then, as the tree is traversed, constrain-based analysis techniques are employed to resolve and check those constraints. LCLint takes into account the value of the predicates on different code paths in order to detect vulnerabilities. There are, however, certain programming constructs that LCLint is unable to interpret, and in addition, many spurious warnings might end up being generated.

Another solution to provide a more elaborate analysis is to examine the application's flow of control. This approach generally consists in parsing the source code and building an AST in order to study the various relations among the different modules and functions of the program (Bush et al., 2000; Grammatech, 2012). The analyzer traverses the entire tree to simulate different control-flow paths and to identify any inconsistencies (i.e., potential vulnerabilities). This analysis may detect problems like invalid pointer references, the use of uninitialized memory,

or improper operations on system resources (e.g., trying to close an already closed file descriptor).

PREfix is an error detection tool for C and C++ that performs control-flow analysis by simulating the execution of individual functions ([Bush et al., 2000](#)). The tool automatically generates models of the functions that describe their behavior as a set of conditionals, consistency rules, and expression evaluations. It then traces individual execution paths, and whenever a function call is encountered, the corresponding model is used to simulate the action of each operation. By tracking the state of the memory during the path execution, and applying consistency rules of the language to each operation, inconsistencies can be detected and reported. Additionally, the detailed tracking information of the paths and values can shed some light on the conditions in which such inconsistencies have manifested. The programming flaws that PREfix can warn are related to memory operations, such as the use of uninitialized memory, dereferencing uninitialized, null, or invalid pointers, and memory leaks.

More recently, other static analysis approaches have emerged that combine more than one type of analysis. BEST resorts to control- and data-flow analysis to detect security problems in binaries ([Wang, 2010](#)). The flow graphs are retrieved by third-party tools, such as IDA Pro ([Hex-Rays, 2012](#)) or ObjDump ([GNU Binutils, 2012](#)). The graphs are combined with a translation of the binary to create a representation of the program resembling C with an embedded assembler syntax, which is much easier to understand and to assess. Security analysts can check the binary for security flaws by evaluating the flow graphs (such as to show nesting relationships among the control structures) and the generated program representation.

Another solution produces a control-flow security model instead, which can be applied to source code or binaries, to identify execution paths that violate previously defined security properties ([Chunlei et al., 2009](#)). This approach is able to detect flaws that allow the memory to be overwritten or the execution path of the binary to be hijacked. However, vulnerabilities originating from dynamic linked libraries cannot be detected because these libraries are not supported in the analysis.

2.1.5 Robustness testing and fuzzing

Robustness testing studies the behavior of the software components in the presence of erroneous input conditions. In this approach, faults are inserted at the interaction between the component and the outside. Typically, the Application Programming Interface ([API](#)) of the component under test is successively called with a combination of correct and erroneous (e.g., out of bounds) parameters. Then, the behavior of the component is observed to determine if it can cope with bad data, for example without halting. Over the years, this approach has been applied to various areas, such as POSIX and device driver [APIs](#) ([Koopman and DeVale, 1999](#); [Albinet et al., 2004](#); [Mendonça and Neves, 2007](#)).

Fuzzing is a special case of robustness testing. This technique was inspired by the effect of noisy dial-up lines that sometimes scrambled command line characters and crashed applications. Researchers were surprised to find that these spurious characters alone were causing such problems in a significant number of basic [OS](#) utilities—simple and mature programs, subject to some years of usage and testing, and therefore regarded as robust applications. Researchers were able to reproduce the same behavior with Fuzz by feeding several Unix command line

utilities with random characters ([Miller et al., 1990](#)).

In its simplest form, fuzzing makes few or no assumptions about the system under test besides the type of interface (e.g., file format, protocol message, or an environment variable), and it can thus be applied to a wide range of systems ([Oehlert, 2005](#); [Sutton et al., 2007](#)). Typically, a fuzzer produces a very large number of input sequences that it presents to the interface of the target system. Since it normally lacks a monitoring mechanism to detect the faults, a test case is considered to fail if the system crashes. Therefore, fuzzing is quite successful at detecting vulnerabilities that cause fatal errors (e.g., buffer overflows or Denial-of-Service (DoS) vulnerabilities), but is more limited in security problems that do not result in such evident behavior (e.g., privileged access violation, [SQL](#) injection, or software aging problems).

Fuzz testing gained significant interest due to events like “Month of the Browser Bugs”, where new exploits were produced each day for the most popular Internet browsers ([Moore, 2006](#)). A new generation of fuzzers resorted to specialized knowledge to go beyond simple random testing and to perform more complex interactions with the target, such as sending HTTP messages. These messages still comply with the format accepted by the parsing mechanisms of the target system but also contain irregular fuzzing elements that, if not properly handled, can cause erroneous behaviors ([Sutton, 2005](#); [Biege, 2002–2011](#); [Betouin, 2006](#); [Infigo Information Security, 2012](#); [AutoSec Tools, 2012](#); [Navarrete and Hernandez, 2012](#); [Codenomicon, 2012](#)).

Fuzzing frameworks have also been developed to ease the process of tailoring fuzzing tools to specific systems ([Roning, J. et al. 1999–2003](#); [Greene, 2005](#); [Rapid7, 2012](#); [AutoSec Tools, 2012](#); [Navarrete and Hernandez, 2012](#)). They allow

the customization of the fuzzing process for a particular target, using for example the Backus-Naur Form (BNF) dialect for protocol specification (Roning, J. et al. 1999–2003) and scripting languages for file format generation (Greene, 2005) or the creation of new testing modules (Rapid7, 2012). Metasploit is one of these fuzzing frameworks that supports the definition of the complete testing process, from the payload generation to the actual test case execution.

Fuzzing, however, can only generate negative test cases, i.e., tests that explore situations not defined in the standard specifications, and normally they only provide a random sample of the system's behavior. Additionally, their analysis is usually too superficial to allow the detection of other than simple crash failures. Nevertheless, bugs detected by this approach are often found to be exploitable vulnerabilities, so it is particularly useful for security assessment.

2.1.6 Vulnerability scanners

Vulnerability Assessment Tools (VATs) allow system administrators to check applications, computer systems, or even entire networks against a database of known vulnerabilities and misconfigurations (IBM Internet Security Systems, 2006; McAfee, Inc. 2003–2012; Saint Corp. 2012; Qualys, Inc. 2008–2012; Tenable Network Security, 2002–12a; Greenbone Networks GMBH, 2012; Rapid7, 2012). The scanner first interacts with the target to obtain information about its execution environment (e.g., OS and available services). This information is then correlated with the data stored in the database to determine the potential problems that this type of system known to contain. In addition, for each potential vulnerability, the database offers a test case to detect it. The test case can be as thorough as a real exploit or as simple as a check on the welcome banner of the server. To

complete the procedure, the scanner carries out the relevant test cases for that target system and presents the results.

Some scanners also attempt to be less intrusive. Tenable Passive Vulnerability Scanner ([Tenable Network Security, 2002–12b](#)), for instance, passively scans the network for vulnerable systems, watching for potential application compromises, client and server trust relationships, and network protocols in use. This type of VAT continuously listens to network traffic, identifying OSs and services by fingerprinting packets and cross-referencing them with port numbers.

Even though VATs are extremely useful to improve the security of systems in production, they have the limitation of being unable to uncover new vulnerabilities—any vulnerability not previously reported and incorporated in the database will remain undetected.

2.1.7 *Run-time detection*

Run-time prevention mechanisms change the program's execution environment with the objective of discovering and thwarting the ongoing exploitation of vulnerabilities. The idea here is that removing all bugs from a program is infeasible, and therefore it is preferable to *contain* the damage caused by their exploitation. Instead of solving the problem at the source (i.e., by correcting the program), they try to mend it at the end (e.g., by preventing the stack from being overflowed). Although these mechanisms are usually implemented at compile-time, the detection and prevention is achieved at the time of execution.

Most of these mechanisms were developed to protect systems from buffer overflows. Attackers exploit these vulnerabilities by supplying specially crafted data to hijack the program's flow of control. For example, by overflowing a func-

tion's return address with one provided by the attacker, the program will execute the instructions located at that address, which can, for instance, spawn a root shell. In the literature, there are several examples of tools designed to protect the integrity of the stack, or at least to detect any violation.

StackGuard and ProPolice are compiler extensions that provide the means to detect invalid changes to the function's return address and to prevent those changes from corrupting the program's execution (Cowan et al., 1998; Wagle and Cowan, 2003; Etoh and Yoda, 2002). Stack smashing attacks typically explore the fact that the return address is located near a local variable with weak bounds checking. Therefore, the attacker only has to overwrite the memory from that variable's address to the return address. By placing a *canary word* (i.e., a pre-selected random value) next to the return address of a function, StackGuard can detect buffer overflows on the stack. First, it checks if the canary word is intact before jumping to the address pointed by the return address, i.e., *before* the function returns. If a change is found, this probably means that the return address is also modified, and the execution of the program is aborted (i.e., fail-safe stop).

Other authors extend the idea of using canaries to protect not only the return address, but also other stack areas susceptible to overflow, this way aiming at preventing all types of stack overflow (Zúquete, 2004). Each local variable is preceded by a canary word. These boundary canaries are stored in the form of a linked list, where each entry is protected using the previous XOR canary. This way an attacker causing an overflow of a stack variable cannot easily guess the valid values of the boundary canaries of the neighboring variables. In addition, two modes of run-time validation are provided for either development or production scenarios.

These tools can also be configured to prevent return address modifications from occurring with the aim of avoiding the interruption of the program's execution. By resorting to fine grain memory protection, a tool can restrict access to individual memory addresses via a special [API](#), as provided by MemGuard ([Cowan et al., 1997](#)). The return address is set as read-only memory when the function is called, and the protection is only raised when the function returns. Attacks can thus be prevented because the only way to overwrite the return address would be through the MemGuard [API](#).

More sophisticated techniques mitigate pointer corruption exploits. PointGuard ([Cowan et al., 2003](#)) adds special code to the program that prevents an attacker from producing predictable pointer values. A key generated at run-time is used to encrypt pointers while they are in memory. When a pointer is dereferenced, its value is decrypted from memory and loaded into the CPU register. If an attacker overwrites the pointer value, the program will jump into an unpredictable memory address, thus making the attack impracticable.

A few studies compare the effectiveness of some of these techniques, showing that they are useful to prevent only a subset of the attacks ([Wilander and Kamkar, 2003](#); [Zhivich et al., 2005](#)). In particular, the main limitation with PointGuard is that it can only defended against pointer corruption attacks. Standard library functions and data structures, such as *malloc*, are not supported.

Other approaches employ some sort of memory page protection. Windows XP SP 2 introduced Data Execution Prevention (DEP), which stops any application from executing code from a non-executable region ([Microsoft, Corp. 2006](#)). However, some attacks (such as return-to-libc attacks ([Designer, 1997](#))) are still possible because some parts of the address space remain executable. PaX was

developed for the Linux kernel, and it does program memory randomization to thwart the previous type of attacks and to prevent pointer prediction (PaX Team, 2009). Another approach, Libsafe, intercepts library calls and executes modified versions of the dangerous functions to foil attackers from overwriting the return address and hijacking the control-flow of applications (Tsai and Singh, 2001).

2.1.8 Resource usage detection

Resource usage monitoring has been employed mainly for performance analysis, but it has also been used to detect some types of vulnerabilities, such as memory leaks. Various timing facilities are available in Linux systems to support the former, such as `/proc/stat`, `getrusage`, `getpinfo`, or even more portable solutions such as LibGTop (Baulig and Kacar, 2012). However, given the small time granularity of many OS activities (Hines et al., 2000) or due to processor fluctuations (Wiebalck et al., 2003) the measured time may yield inaccurate values. In order to improve the timing resolution, hardware performance counters present in modern processors can be utilized (London et al., 2001; Innovative Computing Laboratory, 2012).

Besides the CPU time, memory is also an important resource. Memory leak detectors such as Valgrind (Nethercote and Seward, 2007) or memprof (Taylor, 1999-2007) trace the memory allocation and de-allocation during the program execution. For example, Valgrind is a memory debugging and profiling tool that emulates the execution of programs on a virtual x86 processor (Nethercote and Seward, 2007). First, Valgrind converts the original program into a temporary intermediate format and performs the necessary code modifications, namely it adds extra instrumentation code to allow fine-grained track of the memory usage

and inserts memory guards around any allocated chunk of memory. Then, it recompiles the modified program back to binary code to run on the host machine. If the program crashes or encounters a memory violation, the error is reported with context information about the memory layout. Even though the errors are detected in real-time, this procedure considerably impacts the performance of the program. Moreover, it is up to the developer to provide the different execution paths, or test cases, in order to attain a reasonable coverage.

Some faults may remain dormant or unnoticed until their effects accumulate over time, reducing the performance of the system and eventually causing a failure. The exhaustion of OS resources, fragmentation and accumulation of errors that build up through continuous operation, is known as *software aging* (Parnas, 1994; Garg et al., 1998). *Software rejuvenation* is meant to mitigate its effects by periodically restarting the program to a previous checkpoint (Huang et al., 1995; Vaidyanathan and Trivedi, 2005). This method is very successful at proactively removing the effects of software aging. However, since it is not concerned with the actual causes of the phenomenon (e.g., a memory leak or an unreleased file lock), it can only regularly delay its impact. After the system is rejuvenated, the vulnerabilities that caused the faulty behavior are typically not eliminated, and thus the problem will eventually arise again. In a security context, this can be a concern because an adversary may systematically force the conditions that age the systems.

2.2 Protocol Specification

In the context of network attack injection, a target system is an application server that provides a service over the network. One of the ideas suggested in this work is the utilization of the specification of communication protocols in the attack injection framework to assist in the automatic generation of network attacks and in the evaluation of the results of the tests by deriving the correct behavior of the system. In this section, we review some of the solutions that address the problem of obtaining a formal specification of a communication protocol.

A protocol specification defines the set of rules that dictate the communication between the parties, i.e., the clients and the servers. It determines the syntax of the messages (protocol language) and the conditions for exchanging them (protocol state machine). The specification can thus be seen as a formal language modeled by grammars or Finite-State Machine (FSM) automata, which describe how symbols of an alphabet (i.e., message fields or protocol states) can be combined to form valid words (i.e., an entire protocol message or a protocol state transition).

We organized this section in three main types of solutions. We begin by addressing the use of formal languages to manually define specifications. Then, we present works that reverse engineer protocol implementations with dynamic analysis techniques. Finally, we describe another type of approach that is based on the grammatical induction problem, which aims to infer a specification from a trace with protocol messages.

2.2.1 *Manually defined*

A protocol can be defined through a formal specification, which is a mathematical description of the syntax and semantics of the protocol, such that every word and symbol has a well-defined meaning and its use must follow exact rules. Since a specification describes the protocol in a precise and unambiguous way, it can be standardized and used as a guide for future implementations. Moreover, given a formal specification, it is possible to use formal verification techniques to check the correctness and to test the implementation of the protocol.

Specifications can be written in formal specification languages ([Plat and Larsen, 1992](#); [Spivey, 1992](#); [ITU-T, 1997](#); [Crocker and Overell, 2008](#)). Some of these languages are appropriate to represent the system as a mathematical model. These are called model-based specifications, and they describe the states and operations of the system through well-defined logical expressions, and arithmetic and algebraic relations ([Spivey, 1992](#); [Plat and Larsen, 1992](#)).

The Z specification language is a typed first-order set theory language that can define different types of formal specifications (besides model-based), in natural language and complemented with formal descriptions, called schemas ([Spivey, 1992](#)). A schema describes the states and properties of the system in standard mathematical notation. Z uses a mix of formal and informal language to relate the mathematics to objects in the system and to produce more readable specifications. In addition, since all expressions in Z notation are typed, it allows type-checking tools to verify the type of every object in a specification. Even though the natural language may be a good means to explain complex parts of a specification succinctly and in a more straightforward way, most specification

languages prohibit it, as it is a source of potential ambiguity.

Vienna Development Method Specification Language ([VDM-SL](#)) is one of the most popular formal specification languages ([Plat and Larsen, 1992](#)). In [VDM-SL](#), a system is modeled as an abstract machine with a finite number of states and operations on those states, where an operation is a function that maps an input and a state to a new state value. [VDM-SL](#) can also resort to pre- and post-conditions to characterize the conditions and properties that are assumed to hold before and after the operation, thus producing a specification of the system that can be tested and validated. Moreover, explicit function definitions, which determine how the function should compute the output, can also be used to generate program code automatically from a validated model.

Another popular notation is the Augmented Backus-Naur Form ([ABNF](#)) that is currently being used by the [IETF](#) for the syntax specification of its communication protocols ([Crocker and Overell, 2008](#)). This notation is based on the [BNF](#) ([Backus, 1959](#)) and was created with the goal of describing a system as a bidirectional communication protocol. An [ABNF](#) specification is written as a set of derivation rules that portray the syntax of the language, i.e., the format of the data or messages.

Other formal notations and languages that are used to define the syntax of communication protocols are the Specification and Description Language ([SDL](#)), created and used by the International Telecommunication Union Telecommunication Standardization Sector ([ITU-T](#)) ([ITU, 1999a](#)), and the Abstract Syntax Notation One ([ASN.1](#)), which is an International Organization for Standardization ([ISO](#)) standard ([ITU-T, 1997](#)). [SDL](#) is a formal language created for specifying telecommunication systems ([ITU, 1999a](#)). In [SDL](#), a system is defined as a

collection of [FSMs](#) that interact and communicate with each other and with the outside by means of signals/messages. [SDL](#) defines a formal notation with well-understood semantics, thus it is able to describe the system in an unambiguous and precise way.

[ASN.1](#) is a formal specification language with a more flexible notation that can be used to define the format of the data and the rules for its transmission in a machine-independent way ([ITU-T, 1997](#)). Therefore, protocol designers are free from dealing with architecture-dependent details, such as byte representation and ordering. Its abstract syntax notation is a formal notation, similar to [BNF](#), that is also able to depict a communication system unambiguously. ASN compilers can even take [ASN.1](#) specifications to generate data structures and encoding/decoding routines automatically in common programming languages. In addition, [ASN.1](#) can also be encapsulated in other specification languages, describing the overall behavior of the system in [SDL](#), but keeping the data, signals/messages, and the encoding/decoding schemes defined in [ASN.1](#) ([ITU, 1999b](#)).

2.2.2 *Dynamic analysis*

It was only recently that the field protocol specification inference has seen some developments through a few reverse engineering approaches. One of these approaches uses dynamic analysis on an implementation of the protocol to assist in the inference of its syntax language, i.e., the formats of messages accepted by the protocol.

Dynamic analysis tools closely monitor the program's execution while processing a message. They inspect each program statement and conditional branch by actually running the program with some input data, such as a network mes-

sage. Taint analysis is normally employed to track the data-flow throughout the program's code, and therefore to identify the statements that parse the packets (Birznies, 1998). This information is then correlated with the portions of the protocol message that were processed, and the resulting execution trace is examined to locate the message fields and their content type (e.g., length fields) (Caballero et al., 2007; Lin et al., 2008; Cui et al., 2008; Wondracek et al., 2008).

Typically, solutions that resort to a program's execution are limited to derive only the language of the protocol. The state machine of the protocol, i.e., the rules that govern the interaction between the parties (clients and servers), cannot be derived by looking only at the execution of one of the parties. There is, however, one tool that employs dynamic analysis to obtain execution traces that are then used to infer the state machine of the protocol using a grammatical induction approach (Comparetti et al., 2009). This tool is called Prospex and is discussed in the next subsection.

Even though these tools have shown interesting practical results, they have some limitations. For instance, if the server employs non-standard libraries or if its parsing mechanisms deviate from what is expected, dynamic analysis tools may be unable to make any sense of the fields or even the entire message. Also, software piracy prevention techniques, such as code obfuscation (Naumovich and Memon, 2003), can distort or even preclude the understanding of the logic of the program. These tools are also depend on the system and programming language, due to the taint analysis engine, which limits the number of programs that can be analyzed.

2.2.3 Grammatical induction

Another type of approach resorts to examples of the protocol usage (i.e., network traces) in order to reverse engineer its specification. Manual protocol reverse engineering has been traditionally a laborious task, with a few tools to ease the process of capturing and analyzing individual network packets ([Jacobson and McCanne, 1987–2012](#); [Rauch, 2006](#); [Beardsley, 2009](#); [Combs, G. et al. 2012](#)). In alternative, a more systematic approach is based on grammatical inference solutions ([Fu and Booth, 1986](#)) to derive the underlying grammar, or the equivalent **FSM** automaton.

The problem of *automata inference* has been tackled in different research areas in the past, from natural languages to biology and to software component behavior ([Higuera, 2010](#); [Biermann and Feldman, 1972](#); [Sakakibara, 2005](#)). Typically, a Prefix Tree Acceptor (**PTA**) is first built from the training set, accepting all events. Then, similar states are merged according to their local behavior (e.g., states with the same transitions or states that accept the same k consecutive events) ([Biermann and Feldman, 1972](#); [Lo et al., 2009](#)). Nevertheless, learning a **FSM** in an efficient way is a challenging research problem because it is known to be **NP**-complete even for finite state grammars ([Gold, 1978](#)). Hence, some solutions resort to specific rules or heuristics to aid the inference process ([Mariani and Pastore, 2008](#)). Although these techniques can produce useful models, their precision can be affected when applied to larger and complex models, and some works have tried to address this limitation ([Lo and Khoo, 2006](#)).

In general, most research in protocol reverse engineering focused on inferring only the language of the protocol. To derive the formats of the messages using

only network traces requires some sort of identification of the similar parts of the messages. One solution employs bioinformatics sequence alignment algorithms to reveal similarities in messages. Then, consensus sequences are produced and analyzed to find the location and lengths of some message fields ([Beddoe, 2005](#)).

Discoverer resorts to a different approach to obtain more information about the messages ([Cui et al., 2007](#)). It uses an initial clustering to group messages with similar sequences of text or binary tokens, and then, recursive clustering and sequence alignment techniques refine each cluster and produce more detailed message formats. Experiments have shown that Discoverer could generate an approximate specification of the language for some protocols. However, it was not able to correctly infer about 10% of the message formats, in part due to some inaccurate parsing.

Besides the language, a typical protocol specification (e.g., [IETF](#) standards) also defines its state machine, i.e., the rules for exchanging messages between the parties. We are aware of only three approaches to derive the state machine of a protocol. Prospex combines dynamic analysis and grammatical inference to get the language and the state machine of the protocol. It employs taint analysis to obtain execution traces for each session of communication, which are then used to build an acceptor machine ([Comparetti et al., 2009](#)). The state machine is generated by building an augmented [PTA](#) from the sequences of message types of the sessions, and then by transforming the tree into the smallest automaton that is consistent with the training data. As explained in the previous section, one of the difficulties in applying this technique is the use of taint analysis, which requires a restricted controlled environment to run and to collect the program execution data.

PEXT utilizes network traces to infer an approximate state machine (Shevertalov and Mancoridis, 2007). First, it clusters messages based on a distance metric using the length of the longest common substring and labels each message with the corresponding cluster identifier (ID). Then, it translates each session into a sequence of cluster IDs. States and transitions are generated from similarities between the sequences of IDs in the sessions and the order in which they appear in the traces. This approach is useful to evidence patterns of sequences of messages that arise from using specific protocol features. However, it cannot derive the message formats, creating a semantic gap between the final automaton and the observed data. The clustering method can be error prone because the use of the longest common substring metric might induce incorrect clustering of different message types that share long common parameters (e.g., path name).

Trifilò et al. (2009) describe a protocol reverse engineering solution that resorts to the statistical analysis of network traces. This approach, however, assumes a single message format for the protocol, which allows all messages to be aligned and compared. The distributions of the variance of the bytes over different messages are compared in order to identify the most relevant field, i.e., the field that is most likely to dictate the logic of the protocol. The protocol state machine is then obtained from the order of messages in the traces and the values of this relevant field. While this may provide good results for some binary protocols (e.g., Address Resolution Protocol (ARP) or Domain Name System (DNS)), it is not suitable for the majority of application protocols because they have different message formats (e.g., Simple Mail Transfer Protocol (SMTP) or Session Initiation Protocol (SIP)). In addition, it may be insufficient to use the variance of distributions as a means to detect the most relevant field(s), because the results are

significantly dependent on how uniform the various kinds of messages appear in the traces.

2.3 *Generation of Test Cases*

In our approach, attacks can be seen as test cases that are targeted at the interface of the network server. Over the years, a significant amount of investigation has been devoted to the subject of creating test cases that are representative of the entire testing space. Although it has been proven that there exists a finite set of test cases that can reliably determine whether a given program is correct, obtaining such a set is an undecidable problem ([Howden, 1976](#)).

In this section, we present an overview of many techniques that have been devised to address the problem of test case generation. Most of these approaches are dedicated to search for general programming errors. Security vulnerabilities are particular kinds of errors because they are triggered only through specific attacks (or exploits), and therefore, they can be quite difficult to discover.

We begin by illustrating some classical approaches that resort to the combinatorial generation of input data for testing. Then, we describe a few solutions that employ random data generation and fuzzing techniques as a way to explore the input space domain, although not exhaustively. Then, two types of approaches that make use of the actual system implementation are addressed, dynamic analysis and symbolic execution. The last subsection is devoted to solutions that generate test cases based on a formal specification of the system.

2.3.1 Combinatorial test data generation

The number of the test cases is usually proportional to the size and complexity of the input space, i.e., the number of variables and the range of possible values that each variable can hold. However, testing all possible combinations is in most cases prohibitive, and therefore, some sort of combinatorial test generation or heuristic has to be employed (Beizer, 1990). One classical solution to reduce the number of test cases is to limit the values tested in each variable in what is called equivalence partitioning (Hamlet and Taylor, 1990; Gutjahr, 1999; Myers et al., 2011). Thus, instead of experimenting all possible values, only a subset of those values is tested, such as boundary values and some chosen values in between. This effectively reduces the testing effort, however, there is no evidence that the ignored values are represented by the tested values.

Another approach consists instead in restricting the variables being tested. One common practice is called *default testing*, in which one variable at a time is tried with several values, while the other variables hold some default value (Cohen et al., 1994b; Burr and Young, 1998). While this approach reduces the number of test cases considerably, it does not test the interactions and dependencies that may exist between the various variables.

Other methods were created in order to test potential dependencies among the variables. One of these approaches is *pairwise testing* (also referred as *all-pairs testing* or *2-way testing*), in which a minimum number of test cases covers all possible combinations of values for each pair of variables (Cohen et al., 1996). This technique can be further generalized for *n-way testing*, although the size of the higher order test sets grows very rapidly.

Creating pairwise test sets is not a trivial task. The most common way is achieved through *orthogonal arrays* (Phadke, 1989; Hedayat et al., 1999), a combinatorial arrangements technique that produces vectors of testing values where each pair of inputs occurs exactly the same number of times. However, there is no unified approach for obtaining orthogonal arrays and in some cases, it is even impossible (e.g., there does not exist an orthogonal array for six variables, each with seven different values) (Cohen et al., 1994a). In addition, orthogonal arrays may produce redundant pairs of input values, which in the context of software testing can be wasteful (i.e., duplicated tests will yield identical results) (Singpurwalla and Wilson, 1999).

2.3.2 Random data generation and fuzzing

A more simple approach is to test a random sample of the input domain (Myers, 1979; Bird and Munoz, 1983). *Random testing* does not require partitioning the input domain to find equivalent classes of values, but instead, values are chosen at random. The expectation is that the combinations of values that trigger faults (i.e., test cases that fail) are eventually generated. This approach can produce very quick results (in particular in the early stages of software development) and, in theory, given the necessary amount of time, it would asymptotically approach exhaustive testing.

Random testing also provides *negative testing* (as opposed to functional testing), in which test cases that contain data outside the input domain evaluate the system's response to undefined stimulus. This approach has been used in several testing tools, such as the aforementioned fuzzers (Miller et al., 1990), because it offers a high benefit-to-cost ratio. However, random testing may require an

excessive amount of time to achieve a good coverage and since it relies on probabilities, the chances in finding faults that can only be revealed by a small subset of input values are quite low ([Offutt and Hayes, 1996](#)).

Fuzzing can also find odd oversights and defects that other test case generation approaches often fail to discover. This happens because many of the bugs found through fuzzing are a result of some strange and unexpected state that is reached by very unusual input data. Test cases with this characteristic are hard to be conceived by human test designers and prohibitive to be reached through exhaustive testing. By automating testing with fuzzing, millions of iterations can cover a significant number of interesting permutations that would be difficult to be conceived by a human tester ([Oehlert, 2005](#)). Fuzzing can provide such test cases with prior little knowledge because no preconception about the system's behavior is required—the test cases are considered to fail when the program crashes.

Modern fuzzers, however, resort to predefined sets of irregular data, known as fuzzing vectors, in their test case generation (e.g., strings with several A's or characters in Unicode representation). They also have hard-coded (or pre-configured) a basic specification of the program's interface, such as the format of the files ([Sutton, 2005](#); [Greene, 2005](#)) or messages ([Roning, J. et al. 1999–2003](#); [Rapid7, 2012](#); [Infigo Information Security, 2012](#); [Codonomicon, 2012](#)) that it supports. This specification is quite simplistic and only provides the syntax rules for generating test cases, allowing the tools to systematically create test cases for each input format of the program's interface. In addition, the state machine of the protocol is usually hard-coded in a predefined sequence of messages that can take the protocol to the desired state (e.g., a valid login at the start of each test case).

2.3.3 *Dynamic analysis test data generation*

Automatic test case generation can resort to the program's source code and/or its execution to choose adequate input test data. These approaches can use generation strategies based on different adequacy criteria, such as the coverage of the executed program statements or conditional branches. Most test data generation solutions that use *dynamic analysis* fall in either path-oriented or goal-oriented methods. *Path-oriented* methods reduce the problem of test data creation to a path traversal problem (Boyer et al., 1975; Clarke, 1976; Korel, 1990; Ramamoorthy et al., 1976). This method is used to derive input data that forces the execution of a selected program path. Test cases can be systematically generated in this way until all paths are covered or found unreachable.

Goal-oriented methods produce input test data with the objective of executing a chosen program statement, regardless of the execution path (Korel, 1990, 1992; Ferguson and Korel, 1996; Korel and Al-Yami, 1996). They usually resort to function minimization search algorithms to automatically find new input that will change the flow of the program in order to concentrate only on branches that affect the execution of the goal statement. This approach is also known to obtain an arbitrary execution path (provided that it reaches the goal statement), thus it is hard to predict the overall code coverage given a set of statement goals.

2.3.4 *Symbolic execution test data generation*

Some test data generation approaches try to find input data by solving constraints such as path- or branch-predicates. *Symbolic execution* has been used to predict the program's behavior under specific inputs and to identify abnormal execution

paths within the program (Boyer et al., 1975; Clarke, 1976; King, 1976). In this approach, the program is analyzed statically and its execution is emulated as an algebraic expression over symbolic input values. Symbolic execution then generates path constraints that consist in conditional expressions on the input variables. Thus, to generate test data that traverses a given path is a matter of finding the symbolic input that satisfies the respective path constraints.

EXE, for instance, is a tool that produces input test data for real code (Cadar et al., 2006). Instead of running the program with real input, it emulates the execution with symbolic inputs and tracks the generated constraints on each symbolic memory location. The original code is recompiled to include the necessary instrumentation code and to operate on the symbolic expressions. The code is then run with the symbolic input, which is initially set to *anything*. When the program conditionally checks a symbolic expression, EXE forks the execution, constraining the expression to be true on one branch and false on the other. Whenever a path terminates or hits a bug, a test case is automatically constructed to reach this execution path. One of the drawbacks of this approach is that additional symbolic information has to be added to the program in order to generate the instrumentation code.

SAGE is an instruction-level tracing and emulation tool that performs white-box fuzzing of file-reading applications (Codefroid et al., 2008). It uses a seed file of valid input, which it feeds to the application, and a combination of static and dynamic analysis in order to guide the execution path the program. Static analysis is employed to collect information about the structure of the program and to identify interesting paths of execution. Whenever a conditional control transfer is encountered, a data-flow analysis is performed to determine if the con-

ditional statement depends on the input. A constraint solver then returns new input data that will produce different and unexplored execution paths, which are then returned as test cases.

SAGE relies on a coverage-maximizing heuristic in the dynamic test generation. However, full program path coverage does not imply a complete test case coverage because the whole input domain is too large to be thoroughly tested. The same instruction can be executed with different input data complying with the same conditional branch, and yet produce dissimilar results (e.g., buffer overflows). The tool can also suffer from path explosion because the systematic execution of all possible paths is not a very scalable solution. As conditional branches become increasingly complex (e.g., string operations) so does the test generation process. Additionally, relying on conditional branches alone to generate the input data may not be sufficient to create test cases that can discover vulnerabilities caused by external conditions, such as some particular option in a configuration file or some previously created user data that the application reads and uses. Finally, symbolic execution is an incomplete emulation, which uses and keeps track of symbolic values rather than actual executing the code. This can greatly increase the complexity as it would have to support all machine-code operations and carefully solve the conditional constraints. In fact, SAGE cannot handle pointer dereferences, for instance. Therefore, some faults, in particular security vulnerabilities, are going to be missed by the test cases.

2.3.5 *Specification-based test data generation*

In specification-based test data generation, a well-defined formal representation that describes the intended behavior of the program guides the generation of input

test data. The program is depicted as a black-box (without any knowledge about its internal execution) that changes state according to a given input. One major advantage is that no access is required to either the source code or the binary of the program. In fact, specifications can be created prior any implementation to assist in the software development and to ensure interoperability among different implementations (e.g., [IETF](#) standard specifications). Consequently, a battery of conformance test cases can be created even before the program exists.

As mentioned earlier, specifications can be written in formal languages, such as [VDM-SL](#) ([Plat and Larsen, 1992](#)), [Z](#) ([Spivey, 1992](#)), or [AsmL](#) ([Microsoft, Corp. 2001](#)). However, these specifications describe arbitrarily general systems with a potentially infinite space state. A more suitable approach is to define the system through state transitions over a finite set of values, usually resorting to [FSMs](#) ([Lee and Yannakakis, 1996](#)). These mathematical models, conceived as abstract machines with a finite number of states and transitions, have become popular in software testing because they are a very practical and suitable way of modeling communication protocols.

Most testing approaches that make use of finite machines consist in finding input data that take the program to the required state for the test, and checking if the ensuing state is correct. In general, these solutions resort to traversal algorithms to produce input sequences that test the entire [FSM](#), i.e., they calculate the paths within the automaton to maximize coverage with the minimum number of testing sequences ([Chow, 1978](#); [Lee and Yannakakis, 1996](#); [Offutt et al., 2003](#); [Derderian et al., 2006](#)). However, finding appropriate paths has fundamental limitations: a chosen path might not be feasible and it is undecidable whether a path is feasible or not ([Davis, 1973](#)). Therefore, several methods resort to special conditions

in the automata to obtain input data sequences, such as a strongly connected automaton or the existence of an input sequence that can verify its current state.

T-method is a relatively simple solution that uses transition tours with random input until all the transitions in the automaton are traversed (Naito and Tsunoyama, 1981). Redundant inputs are then removed using a reduction procedure. A single transition tour takes the automaton from the initial state, executes every transition in the automaton, and returns the automaton to the same state. This approach, however, requires a strongly connected FSM, which may be impractical in a real specification.

The D-Method requires a special Distinguishing Sequence (DS) that produces unique output for each initial state in the automaton (Gonenc, 1970). Test cases are produced by creating input sequences that traverse the automaton from the initial state to every other state. The new state is then checked by means of the DS. However, very few FSMs actually possess a DS, also making this solution unfeasible in practice.

A similar approach, the U-method, resorts to a more common type of input sequences present in most FSMs, Unique Input/Output (UIO) sequences (Sabnani and Dahbura, 1988). A UIO sequence for a particular initial state produces an output sequence that can only be produced by that state. The input test data is obtained by deriving the UIO sequences, which will serve as both test case and oracle, although they cannot be used to deduce the (wrong) state if the test caused an error.

Another approach resorts to yet a different type of input sequences to determine if the automaton reached the expected state after a test. The W-method derives a set of input sequences that can distinguish the behavior of every pair of

states, called the characterization set ([Chow, 1978](#)). Input sequences are generated to traverse the automaton and to check the expected state using the characterization set for that state. This approach can also be applied to [FSMs](#) that do not possess a [DS](#), however, it usually yields longer test sequences.

Most test case generation approaches that use [FSMs](#) or similar automata to create input test data, resort to one or a combination of the above solutions ([Koufareva and Dorofeeva, 2002](#); [Dorofeeva et al., 2005](#); [Simão and Petrenko, 2009](#)).

CHAPTER 3

NETWORK ATTACK INJECTION FRAMEWORK

The main focus of the work is to provide a theoretical and experimental framework for executing attack injection, with the aim of discovering security vulnerabilities in network servers. Despite this objective, we will attempt to keep the framework as general as possible, so that it can be applied to other types of target components (e.g., local programs that read files or Web browsers that render HTML pages). We decided to focus on network servers because, from a security point of view, they are probably the most interesting class of target systems for an adversary. First, these are large and often complex applications designed to sustain long periods of uninterrupted operation, and usually remotely accessible through the Internet. Second, an intrusion in a network server can have a significant impact, since it compromises the security of all clients and frequently

opens a door into the internal systems of an organization. Consequently, network servers are a highly coveted target by black-hat hackers and organized criminal groups.

This chapter describes the general framework for network attack injection, indicating how each component fit into the whole process, from the specification of the system and generation of test cases¹, to the injection and analysis of the attacks.

3.1 General Methodology

Vulnerabilities are often a result of subtle and inconspicuous anomalies, which may only emerge in such unusual circumstances that are not contemplated in test design, since conventional test cases do not cover all the obscure and unexpected usage scenarios. Hence, vulnerabilities are typically found either by accident, or by attackers or special tiger teams (also called penetration testers) who perform thorough security audits. Typically, searching for new vulnerabilities is a slow and tedious manual process. Specifically, the source code is carefully scrutinized for security flaws or the application is exhaustively experimented with several kinds of input (e.g., unusual and random data, or more elaborate input based on previously known exploits) looking for problems during its execution.

Figure 3.1 shows a model of a component or a system with existing vulnerabilities. The same rationale can be applied recursively to any abstraction level of a component, from the smallest subcomponent to more complex and larger systems, so we will use the terms component and system interchangeably. Boxes

¹In the context of software testing, the generated attacks are deemed as test cases, therefore, we use both terms interchangeably.

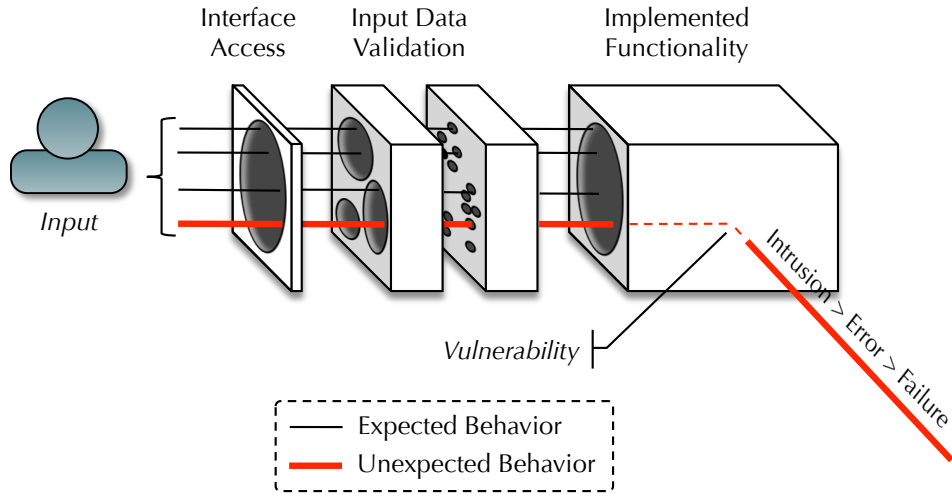


Figure 3.1: The attack process on a faulty (or vulnerable) component.

in the figure represent the different modules or software layers that compose the software component, with the holes symbolizing access (either deliberate by the developers or unintended by exploiting a vulnerability). Lines depict the interaction between the various layers.

The external access to the component is provided through a known *Interface Access*, which receives the input from the outside (e.g., by network packets or disk files), and eventually returns some output. Whether the component is a simple function that performs a specific task or a complex system, its intended functionality is, or should be, protected by *Input Data Validation* layers. These additional layers of control logic are supposed to regulate the interaction with the component, allowing it to execute the service specification only when the appropriate circumstances are present (e.g., if the client messages are in compliance with the protocol specification or if the procedure parameters are within the acceptable input domain). In order to achieve this goal, these layers are responsible for the parsing and validation of the arriving data. The purpose of a component is de-

fined by its *Implemented Functionality*. This last layer corresponds to the implementation of the service specification of the component, i.e., it is the sequence of instructions that controls its behavior to accomplish some well-defined objective, such as responding to client requests accordingly to a standard communication protocol.

By accessing the interface, an adversary may persistently look for vulnerabilities by stressing the component with unusual forms of interaction, such as sending wrong message types or opening malformed files. These *attacks* are malicious interaction faults against the component's interface (Verissimo et al., 2006). Nevertheless, a dependable system should continue to operate correctly, even in the presence of these faults, i.e., it should keep executing in accordance with the service specification. However, if one of these attacks causes an abnormal behavior of the component, it suggests the presence of a *vulnerability* somewhere in the execution path of its processing logic.

Vulnerabilities are faults caused by design, configuration, or implementation mistakes, susceptible of being exploited by an attack to perform some unintended and usually illegal activity. The component, failing to properly process the offending attack, enables the attacker to access the component in a way unpredicted by the designers or developers, causing an *intrusion*. This further step towards failure is normally succeeded by the production of an *erroneous* state in the system (e.g., a root shell). Consequently, if nothing is done to handle the error (e.g., prevent the execution of commands in the root shell), the system will *fail*.

After finding an attack that caused the component to fail, the adversary can further refine it in order to gain more control over the component. In fact, there is an undetermined number of different instances of the original attack that could be

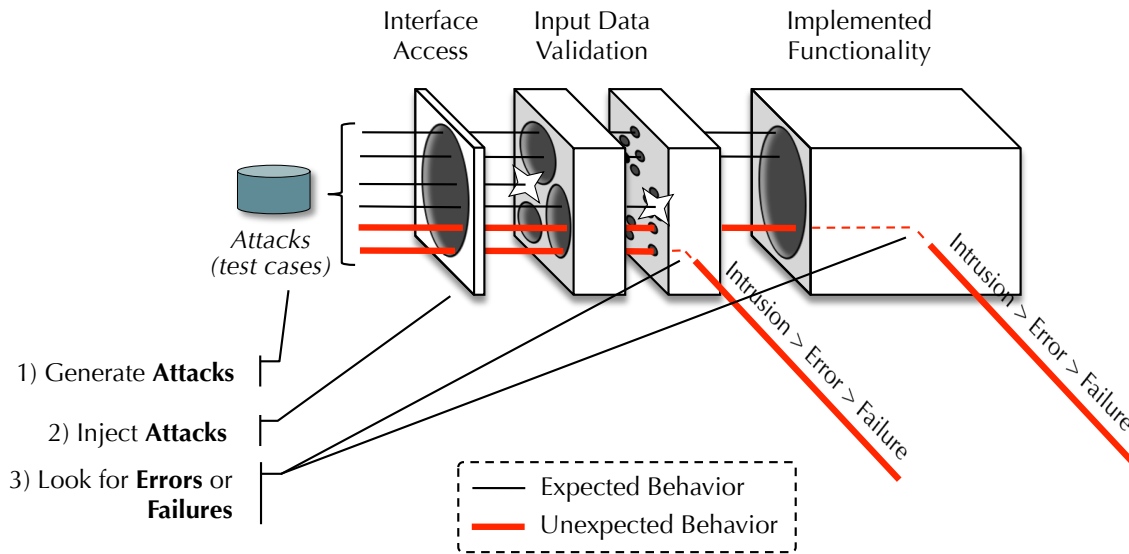


Figure 3.2: The attack injection methodology.

created to exploit the same vulnerability. Depending on the type of vulnerability and on the skill of the adversary, the range of the security compromise can vary greatly, from a simple [DoS](#) to the full control of the system. Usually, an adversary will try to gain full access by executing arbitrary commands in a root shell, to escalate its privileges as a stepping stone to further exploits.

The attack injection methodology adapts and extends classical fault injection techniques to look for security vulnerabilities. The methodology can be a useful asset in increasing the dependability of computer systems because it addresses the discovery of this elusive class of faults. An attack injection tool implementing the methodology, mimics the behavior of an external adversary that systematically attacks a component, hereafter referred as the *target system*, while monitoring its behavior. An illustration of the main the actions that need to be performed by an attack injection tool are represented in Figure [3.2](#).

First, numerous attacks are generated in order to fully evaluate the target sys-

tem's intended functionality (step 1). In order to get a higher level of confidence about the absence of faults, the attacks have to be exhaustive and should look for as many classes of vulnerabilities as possible. It is expected that the majority of the attacks are deflected by the input data validation mechanisms, but others will be allowed to proceed further along the execution path, testing deeper into the component. Each attack is a single test case that exercises some part of the target system, and the quality of these tests determines the coverage of the detectable vulnerabilities.

The attacks should then be injected (step 2) while the state of the component is monitored, looking for any unexpected behavior (step 3). Depending on its monitoring capabilities, the completeness and precision of the information can vary from the simple output of the target system, to the amount of allocated system resources or even the sequence of system calls it executed. Whenever an error or failure is observed, it indicates that a new vulnerability has potentially been discovered. For instance, a vulnerability is likely to exist in the target system if it crashes during (or after) the injection of an attack—this attack at least compromises the availability of the system. Likewise, if what is observed is the abnormal creation of a large file, this can eventually lead to disk exhaustion and subsequent [DoS](#), so it should be further investigated.

The collected evidence provides useful information about the location of the vulnerability, and supports its subsequent removal. System calls and the component responses, along with the offending attack, can identify the protocol state and the execution path, to find the flaw more accurately. If locating and removing the vulnerability is unfeasible or a more immediate action is required, for instance if the target system is a Commercial Off-The-Shelf ([COTS](#)) component or a funda-

mental business-related application, the attack description could be used to take preventive actions, such as adding new firewall rules or [IDS](#) filters. By blocking similar attacks, the vulnerability can no longer be exploited, thus improving the dependability of the system.

3.2 Overview of the Framework

The network attack injection framework was developed as a continuous iterative process to support the attack injection methodology and to accommodate its various approaches for the discovery of vulnerabilities. However, many challenges emerge when designing an overall solution with this objective in mind. For instance, how to *detect* whether an attack triggered a vulnerability or how to *produce* the attacks in the first place? There are thus several sub-problems that revolve around, and must be considered by, attack injection. In order to gain insight on the different challenges and solutions for the discovery of vulnerabilities, each of these sub-problems is addressed individually.

Figure [3.3](#) outlines the proposed framework for attack injection. The general methodology is decomposed in various components, each addressing a particular problem, such as the monitoring of the target system or the generation of attacks. The components of the framework were arranged to reflect their natural precedence order, as depicted by the background circular arrows.

The first component is the *Protocol Specification*, which provides a *specification* of the external interface implemented by the target system. For instance, in the case of an e-mail server, the interface could be defined by the Post Office Protocol ([POP](#)) and/or Internet Message Access Protocol ([IMAP](#)) protocols to provide

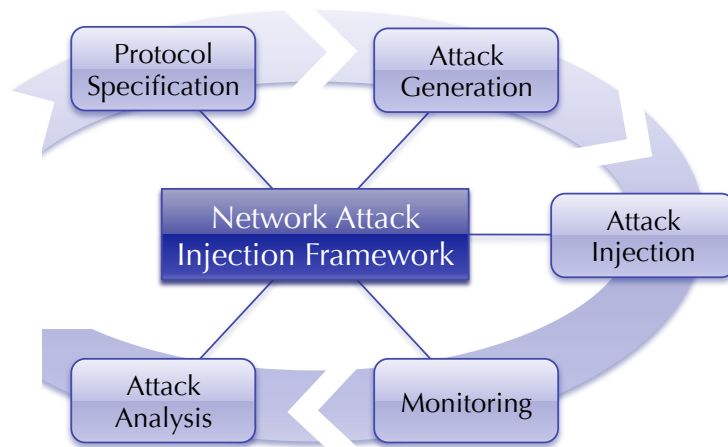


Figure 3.3: Framework of network attack injection.

clients access to their respective electronic messages. This component deals with the problem of modeling the protocol, i.e., the syntax of the protocol messages and the rules that dictate how the interaction between the server and its clients should be processed. This specification is fundamental to the methodology because it supports the generation of more effective attacks and it can also provide a means to evaluate the correctness of the target system's behavior.

The next component is the *Attack Generation*. The *attacks* aim at triggering existing vulnerabilities, causing the server to behave in some unexpected way (e.g., a crash or the execution of an unusual sequence of system calls). For instance, an adversary can exploit a buffer overflow vulnerability by providing a special memory address in the attack payload that points to a local buffer containing the root shell code. The generated attacks can find this kind of vulnerabilities by employing illegal memory addresses in the *testing payloads*—there is no need for specially crafted memory addresses because the server will crash or behave in some unusual way that can be detected by the monitoring capabilities of the framework.

The attack generation component should employ algorithms for the creation of test cases that may use the specification of the protocol. Understanding the protocol is crucial because it allows the systematic creation of attacks that are able to exercise as much of the functionality as possible. At the same time, the attack generation should strive to be independent of the protocol itself to support a wide range of target systems (implementing other protocols).

The actual injection of the attacks is addressed by the *Attack Injection* component. The injection is closely linked with the *Monitoring* of the target system, depicted by the underlying arrow that connects both components. The attack injection component deals with the execution of the test cases, i.e., it sets the testing environment for a new attack injection and sends the messages that compose the attack to the network server. While the attacks are being performed, the target system should be monitored. *Monitoring* is essential to determine which attacks have led to some form of intrusion, by observing and distinguishing any abnormal behavior.

The *Attack Analysis* component corresponds to the last step of the methodology, whose aim is to identify the presence of vulnerabilities. This component depends on the outcome of the previous steps and, more importantly, on the information collected during the monitoring of the attacks, but it may also benefit from knowledge about the specification of the target system.

3.3 Protocol Specification

As its name suggests, this component should produce a specification of the communication protocol used by the network server. Since the communication with

the server is performed through this protocol, incorporating the knowledge about the data and format of the messages can result in the creation of potentially better test cases. In addition, information about the states and operation of the protocol can also improve the effectiveness of testing. Test case generation algorithms can thus take advantage of the protocol specification to exercise most of the input space, such as covering all the protocol functionality and experimenting with more interesting testing values (with respect with security vulnerabilities).

The specification of the communication protocol explicitly and unambiguously defines the expected behavior of the target system, i.e., it determines the server's evolution while processing the different types of client requests. This information can also be useful in identifying abnormal conditions, such as when the server response indicates that it is not in the correct protocol state.

Usually, standard protocols are very well-documented and their specifications are publicly available from standards organizations such as [IETF](#), [ISO](#), Institute of Electrical and Electronics Engineers ([IEEE](#)), and World Wide Web Consortium ([W3C](#)). However, translating the respective documentation, which is usually a mix of informal and formal definitions, can be a tedious and slow process. To ease the task of writing protocol specifications we developed a Graphical User Interface ([GUI](#)) that allows an operator to define protocol states, messages, fields, and data types.

Unfortunately, not all protocols are documented or have an open source implementation available, which prevents their specification from being obtained without resorting to reverse engineering. To address this problem, we have developed a technique based on grammar induction that is able to automatically extract an approximate specification of a protocol using network traffic traces.

Chapter 4 details the solutions that were developed for obtaining and introducing the specification of communication protocols.

3.4 Attack Generation

The test cases to be injected in the target system should be able to trigger existing vulnerabilities, which will potentially lead the server to behave in some incorrect way (e.g., reply with private information or take too long to respond). In the context of this work, test cases are special network interactions, with the intention of forcing the server into a faulty state. The attacks should comply with the specification of the communication protocol that the network server implements in order to be accepted and processed by the target system. The attacks are thus specific to the interface of the target system and are designed to explore it with different testing patterns. There are therefore two essential elements that need to be considered: the protocol specification and the actual generation of test cases. The latter addresses the creation of the attacks, which should take advantage of the information provided by the protocol specification, i.e., the syntax rules that dictate how the interaction between the server and its clients is processed. The protocol specification allows the test case generation to restrict the space of possible test cases into a subset of relevant ones, i.e., attacks that are not immediately rejected by the parsing mechanisms of the server, but allowed to be further processed and eventually reach critical sections of the code. Test cases with protocol messages that clearly violate the communication protocol will most likely be immediately rejected by the server. However, if the attacks comply with the protocol, even if just partially (e.g., only one field of the message has invalid data), intuitively, they

will provide a higher coverage of the server's functionality and, consequently, a higher chance of encountering faults.

We have developed different test case generation algorithms to accommodate the most important classes of errors (Beizer, 1990): delimiter test definition, syntax test definition, and value test definition. While all algorithms should exhaust every protocol state and message type, they target different aspects of testing, such as the way the message fields are organized, or more importantly, their content. To further improve the chances of triggering potential vulnerabilities, some of these algorithms can resort to malicious payloads that are manually defined. These payloads can be obtained by looking at the exploits of previously reported vulnerabilities (e.g., buffer overflow shell code or format strings) and at the actual configuration of the target system (e.g., known usernames and file system paths).

In addition, we have also studied another type of solution that uses the payloads of existing test cases (for instance, from testing and vulnerability assessment tools). This approach automatically extracts the testing payloads included in the messages that compose the test cases for a specific protocol, by inferring the respective specification. This way, the already available test cases can be recycled to create new test cases for a particular target system.

Chapter 5 details the attack generation component and describes our solutions to this problem.

3.5 *Injection & Monitoring*

The attack injection is the component of the framework that deals with the execution of the test cases. Test cases are composed of one or more messages from a

particular communication protocol. They are run by transmitting the corresponding messages to the port where the server is listening (e.g., Transmission Control Protocol (TCP) packets sent to IMAP server's port 143).

One of the goals of the attack injection is to reliably detect any potential faulty behavior during the execution of the test cases. However, protocol specifications are usually incompletely defined, focusing on the most important aspects of interoperability standardization and thus some details are left to the implementation. For instance, a specification of a protocol may not define how to respond to malformed messages, and consequently, some server implementations may not reply, while others may choose to send an error message. These potentially small differences may affect the performance or even the final results of the injection campaign—a test case may be misidentified as failed. For this reason, it is important to contemplate such differences when implementing the attack injection methodology, and in particular in the execution of the test cases.

Another objective of the injection process is to ensure the reproducibility of the test cases that find the vulnerabilities. Hence, an important concern is the setup of the overall test environment, which includes the target system. The most obvious approach is to define each injection test as independent as possible. With this in mind, the experimental environment is reset after each test case execution, i.e., the network server is restarted before the injection of a new attack. If the effects of previous test cases are eliminated, it is easier to reproduce and identify the offending attacks.

However, other interesting questions arise. For instance, instead of regarding each attack as a single and independent test case, it could also be useful to consider an entire injection campaign as a single testing unit, therefore allowing the

accumulation of the effects of the test cases. One may claim that such approach is unsound or that it may distort the attack injection results and possibly render the whole process useless. However, injecting several attacks during the same execution of the network server is comparable to accelerate the effects of software aging, which may actually trigger some existing vulnerabilities that would otherwise remain undetectable.

Another way of accumulating the effects of the test cases in a more reproducible manner is to repeatedly inject the same attack without restarting the target system. The experimental environment is only reset when a different attack is used, which can happen after some predefined number of injections or some interval of time. This allows the server to accumulate the effects of only one attack, allowing the cause of any deviant behavior to be effectively attributed.

Discerning the abnormal from the normal behavior is crucial to determine if some fault has occurred. The attacks should be injected while monitoring how the state of the network server is evolving. Whenever an error or failure is observed, it most likely indicates that a vulnerability has been triggered. For instance, a vulnerability is likely to exist if it crashes the network server during (or after) the injection of an attack—this attack at least compromises the availability of the system. On the other hand, there are other observed behaviors that may not be immediately perceived as an indication of a potential vulnerability. An increase in a regular log file, for example, may actually exhaust all disk space (unless there is some automatic truncation), therefore making the target system susceptible to a resource-exhaustion attack.

From the monitor point of view, the target system is not a complete black-box, but some sort of grey-box. It has access to the hardware, OS, libraries, the network

server application, remaining running processes, etc., and as such, it can get detailed information about the execution. The collected evidence provides useful information to discover vulnerabilities and to assist in their removal. OS calls and network responses, along with the offending attack, can identify the protocol state and the execution path, to more accurately locate the vulnerabilities.

There are several approaches that can be applied to monitoring, from the creation of custom internal monitors that provide more extensive and detailed information, to other less revealing solutions. As a rule of thumb, it is usually better to have more monitoring data available. However, this normally requires the implementation of a monitor that relies on specific external libraries or on the underlying OS, which might dramatically narrow down the range of supported target systems. As a consequence, one should carefully balance the costs of the monitoring requirements versus the value of its information.

Another aspect inherent to the monitor component is that it can be used as a remote controller for the network server. Depending on the injection approach, it may be necessary to restart the network server at different points in its execution. Although not related to monitoring, it is only natural that controlling the network server should be one of the capabilities of the monitor. Fortunately, the degree of control required for restarting the network server is very simple to achieve—the monitor, running in the target system with sufficient privileges, only has to kill and re-spawn the application process of the network server at specific events.

Chapter suggests several approaches for attack injection and monitoring.

3.6 *Attack Analysis*

Once the attack injection is over and the monitoring results are collected, the accumulated evidence must be analyzed for any suspicious behavior. The most straightforward solution is to carry out a form of manually-assisted log inspection, where the operator goes through the monitoring data looking for any conspicuous server activity, such as a message response providing access to some illegal request, an anomalous resource consumption, or just a fatal crash.

Fortunately, this tedious and time-consuming process can be automated to some degree if one knows exactly what to look for, e.g., a SIGSEGV signal for segmentation fault errors or a network response providing illegal access to some resource. The operator can thus build custom script programs that effectively look for known abnormal behavior.

Naturally, the attack analysis can make use of the available monitoring information and in particular take full advantage of the singular capabilities of some custom monitors. In one of the investigated solutions, an automated analysis component performs regression analysis on very accurate resource usage data to predict the depletion of some resource used by the server.

However, unknown or subtly incorrect behavior could be overlooked, which could lead to false negatives. A complementary approach frees the operator from defining the anomalous symptoms, and instead looks for anything outside the normal behavioral pattern, similar to an anomaly-based IDS ([Hoagland and Staniford, 2009](#); [Barbará et al., 2001](#)). This solution can extend the specification of the communication protocol implemented by the network server, with internal monitoring data about its execution. The extended specification provides a be-

havioral profile of the target system, which can then be used to automatically identify any strange behavior.

Chapter investigates some approaches to carry out the analysis of the attack injection results.

3.7 Conclusions

This chapter introduced the attack injection framework and provided an overview of the capabilities that should be offered by each one of its components. In the next chapters, we develop solutions for each of the components, some of which were integrated in three prototype attack injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#). These tools are presented and evaluated in Chapter 7.

CHAPTER 4

PROTOCOL SPECIFICATION

Network attack injection is a black box testing approach that uses the network interface of the target system to inject attacks. For that reason, it can resort to a specification of the communication protocol that the server implements to assist in the generation of more effective attacks. The ability to obtain a protocol specification can, however, play an important role in some contexts. Testing can use protocol specifications to produce test cases covering the protocol space for conformance testing ([Dahbura et al., 1990](#)). Defense mechanisms, such as firewalls and [IDS](#), can also resort to a specification to assist them in detecting attacks. For example, they can use the specifications of the protocols to accurately identify malicious traffic by performing deep packet inspection ([Paxson, 1999](#)).

Network protocols determine how different entities communicate by defining

the syntax and semantics of the messages, and the order in which they need to be exchanged. Examples of commonly used protocols are standardized by the [IETF](#), and they cover areas such as remote file management, distributed name resolution or e-mail access. However, public specifications, such as those published by the [IETF](#), are not always available. In fact, one of the premises of this work is that the attack injection methodology can be applied to closed target systems, which may be implementing proprietary protocols. Obtaining a specification for this type of protocols requires reverse engineering the seemingly arbitrary set of bytes that compose each message to determine their meaning and structure. Open protocols, on the other hand, are well-documented and their human-readable specification is readily available. Even so, translating the documentation into a machine-readable specification is also hard and time consuming because developers have to carefully analyze the textual description of the protocol and translate it into a well-defined and unambiguous formal model.

This chapter begins by describing the concepts of communication protocol and protocol specification and by presenting a manual approach to define the specification of protocols. We then present a protocol reverse engineering approach that is able to derive an approximate specification by using only network traces and an extension to complement existing specifications.

4.1 *Communication Protocol*

A protocol is a set of rules that dictates the communication between two or more parties and is commonly described in a formal specification language and complemented with informal documentation (textual). The protocol specification can

serve as a blueprint for future implementations, making it possible for different developers to produce interoperable implementations of the protocol. In addition, the specifications can also be used for analyzing and testing the protocol design as well as the respective implementations. For this reason, entities such as the [IETF](#), [W3C](#) and [ISO/IEC](#) publicly release their specifications in order to facilitate the widespread dissemination of the protocols and to foster their rapid development and assessment.

However, protocols can be designed and implemented with different policies regarding their openness. *Closed protocols* are protocols for which there is incomplete or no documentation to describe their behavior (e.g., message formats, states, transitions between states). *Open protocols* correspond to the opposite case, where this documentation is available.

We employ the concepts of input and output, following the stimulus-response behavior of communication protocols, to describe messages produced by clients and servers, respectively. The communication originating from the clients is depicted as the input for the protocol execution. Conversely, the servers process these requests and respond with some output.

In this section, we use the [FTP](#) ([Postel and Reynolds, 1985](#)) as an example of a protocol specification. [FTP](#) defines a standard way where clients can access files stored remotely in a server. Clients have first to authenticate by providing a username and the respective password (USER command followed by the PASS command). An authenticated user can then navigate through the remote file system similarly to a local file system. The complete [RFC 959](#) specification defines 33 commands, allowing clients to authenticate, download or upload files, create directories, delete or rename files or directories, or to obtain status information

about files and directories.

The specifications of communication protocols usually define the *language* and the *state machine* of the protocol.

4.1.1 Protocol language

A protocol can be depicted as a formal language whose syntax rules are specified through a grammar, describing how symbols of an alphabet can be combined to form valid words. In this sense, the language of the protocol should determine the *message types* (or *formats*) that are composed of a sequence of fields organized with certain rules and that can take values from a given domain. Specifications can be written in formal specification languages (Plat and Larsen, 1992; Spivey, 1992) and by special notations (Backus, 1959; ITU-T, 1997; ITU, 1999a; Crocker and Overell, 2008). Grammars can be represented by deterministic FSM automata, which are commonly used to describe language recognizers, i.e., computational functions that determine whether a sequence of symbols belongs to the language.

Figure 4.1 shows the FSM of the language of the FTP protocol (Postel and Reynolds, 1985) that is recognized by the servers, i.e., it defines the FTP client requests that can be sent to the servers. Messages are usually composed of one or more fields, which are represented in the automaton by the transitions. Each transition corresponds to a message field and is labeled with a regular expression that defines its payload. For instance, the top transition in the figure corresponds to a message field that can hold the command names HELP, LIST NLST or STAT. The automaton defines two message types that start with one of these commands, depicted by the transitions that follow the command name. One message type

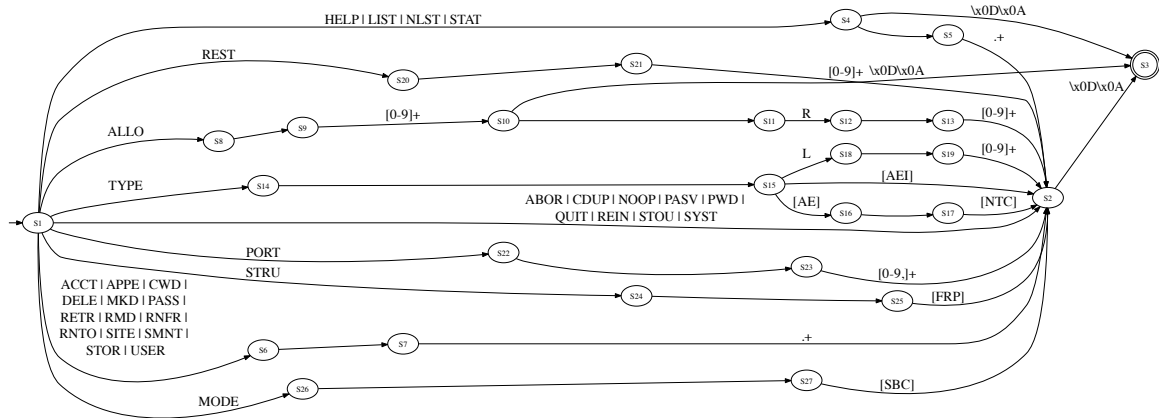


Figure 4.1: Protocol input language (as recognized by the server) of the FTP protocol (RFC 959).

has no additional parameters and is immediately terminated with `\x0D\x0A` (i.e., Carriage Return (CR) and Line Feed (LF) characters). The other path in the automaton defines a message format with the command name, followed by a space and a parameter (regular expression `.+`), and terminated with the message delimiter.

A [FSM](#) is an extremely useful mathematical representation to describe the language of the protocol because there are several formal methods can then be applied for various purposes, such as language or automata minimization, correctness evaluation, and test data generation.

4.1.2 Protocol state machine

Likewise, the protocol specification should also determine the order in which the messages can be transmitted, and consequently, a [FSM](#) can also be utilized to represent the relations among the different types of messages. This part of the specification depicts the progress of the protocol execution while visiting the different states of the protocol.

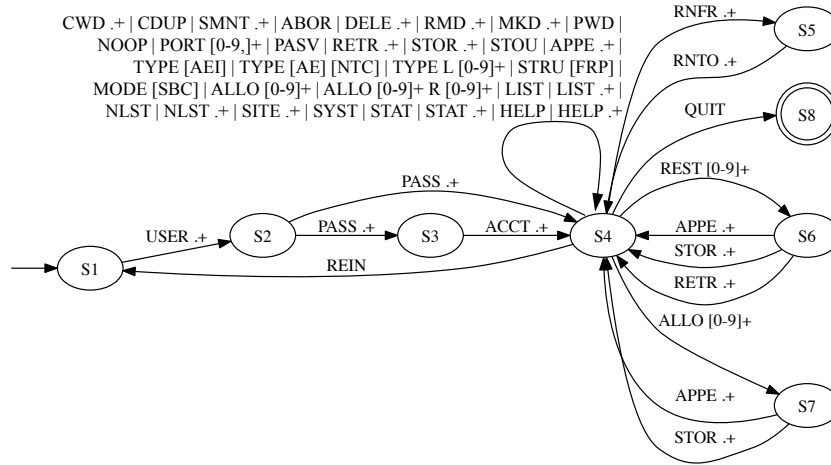


Figure 4.2: Protocol input state machine (as recognized by the server) of the FTP protocol (RFC 959).

Figure 4.2 shows the automaton for the state machine of the [FTP](#) protocol and it defines the evolution of the server when processing the client requests. For instance, at the initial state the server expects a `USER .+` type of message. After that, the client should also provide a `PASS .+` message and possibly a `ACCT .+`. Upon a successful authentication, the protocol goes to state `S4` where the server expect most of the protocol commands, e.g., `CWD .+` and `CDUP`. Some functions provided by the protocol require changing to a temporary state, such when renaming a file the client must send a `RNFR .+` and then a `RNT0 .+`. The protocol execution ends when the client issues a `QUIT` command.

4.2 Manual Protocol Specification

Open protocols provide human-readable documentation that can be used to manually define a protocol specification. [AJECT](#), an open source attack injection tool, provides a graphical user interface component that supports the spec-

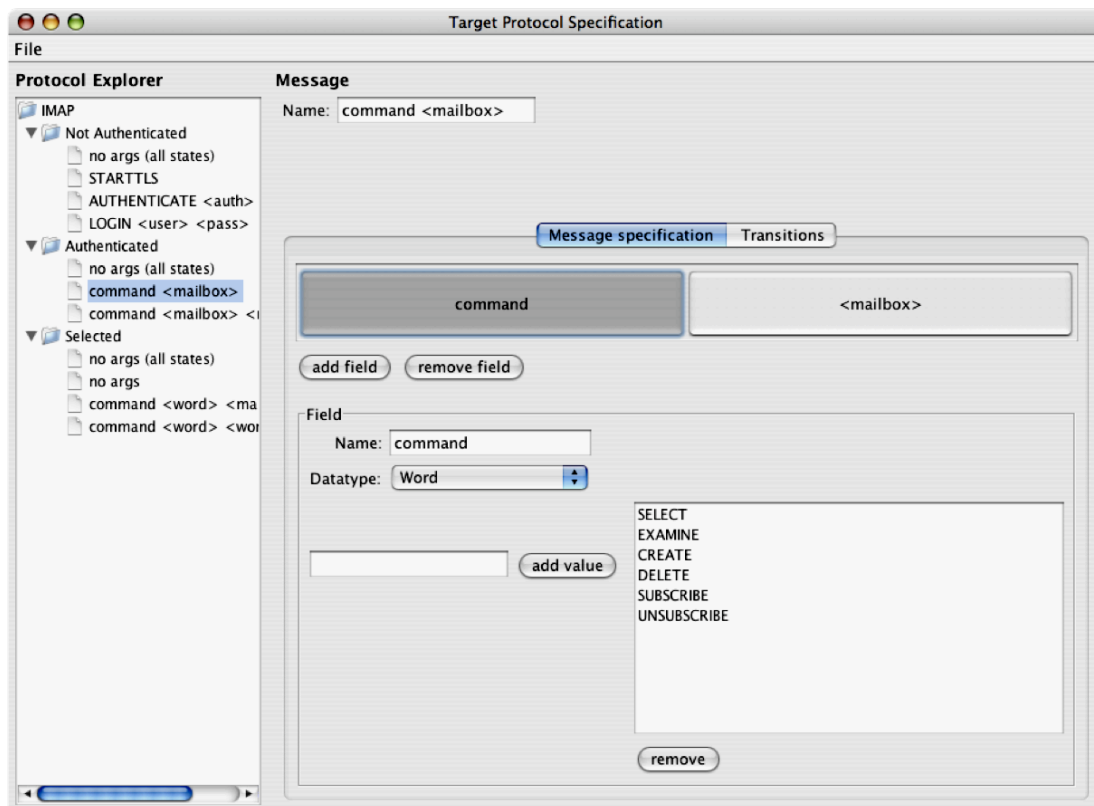


Figure 4.3: Screenshot of the AJECT protocol specification.

ification of the communication protocol used by the server. Figure 4.3 shows a screenshot of the graphical user interface component while introducing the specification of an [IETF](#) protocol ([IMAP](#)). The operator of the tool can describe the protocol states and messages, and identify the data types and acceptable ranges of values of each field of a message.

The tool can define two kinds of messages: messages that request the execution of some specific operation (but do not change the state of the protocol), and transition messages that make the protocol jump from one state to another (e.g., a login message). [AJECT](#) uses this information to explore the entire protocol state space, by creating test cases with innocuous transition messages preceding the

attack message. This procedure is exhaustive because all states are eventually tested with every operation that is defined for each state.

4.3 *Protocol Reverse Engineering*

Manually, defining a protocol specification is only possible when human-readable documentation is available. Proprietary protocols have very limited information accessible to discourage or prevent third-party developers to implement them. Protocol reverse engineering can help to deduce an approximate specification of a protocol from information about its operation and with minor assumptions about its structure.

This section presents a new method for automatically inferring the language and state machine of a protocol. This approach uses *only* the network messages to and from the target system and is based on the problem of inducing grammars (Biermann and Feldman, 1972; Fu and Booth, 1986). However, to derive such a formal model and to extrapolate it from a finite subset of examples is known to be NP-complete (Gold, 1978). Thus, our solution makes a few assumptions from the problem domain (e.g., we focus on text based protocols) and resorts to some heuristics (e.g., to identify identical states of the protocol) to derive the language syntax of the protocol and its state machine. Our solution is focused on text-based protocols, such as those developed in the context of IETF for popular client-server applications (e.g., FTP, IMAP, or SIP), exploring a few of their characteristics like the way text fields are usually organized and delimited in a message. The approach constructs automata from the sequences of messages in network traces, and then, generalizes and reduces them in order to create a

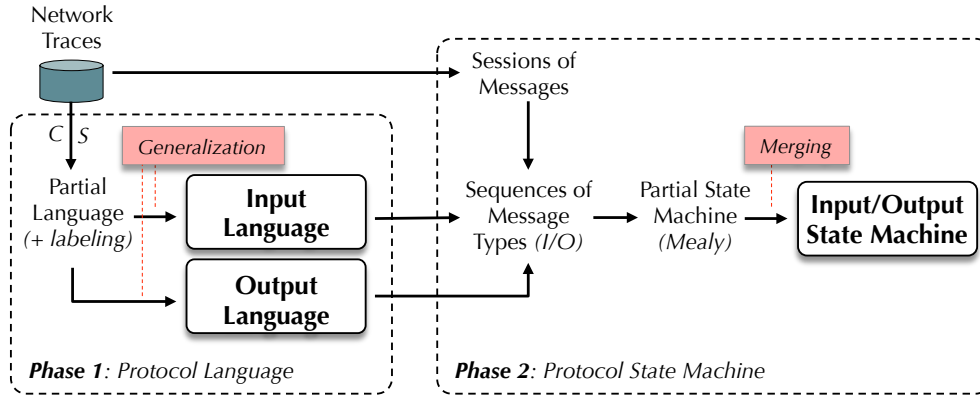


Figure 4.4: ReverX overview for inferring an input/output specification.

concise representation.

The network traces that are used by the inference process can be easily obtained by listening to the communication between clients and servers, thus making this approach suitable for extracting specifications of open and closed protocols alike. As in other learning-based approaches, the quality of the derived specifications depends on the correctness and coverage of the sample sequences. Therefore, the network traces should provide a good protocol coverage and must not have any illegal or malformed messages (as they would introduce incorrect message formats or corrupt existing ones). In addition, to simplify the presentation, we assume that protocol messages are not fragmented in several packets and that no encryption is performed.

This approach is able to derive two types of automata: one that defines the format of the messages (or the language), and another that recognizes the order and causal relations among the messages (or state machine). Figure 4.4 depicts an overview of the developed reverse engineering technique.

The reverse engineering process starts by deriving the *protocol language(s)*, which is then used to infer the *protocol state machine*. As mentioned earlier,

we employ the concepts of input and output to describe messages produced by clients and servers, respectively. Depending on the application scenario, it might be desirable to construct a state machine that captures only the protocol requests that a server can receive, together with its state transitions, and this is called the *input-only protocol specification*. In this case, we only need to obtain the input language specification. In other cases, one might need to obtain a state machine that covers both the client requests and the responses returned by the server, named as *input/output protocol specification*, where both input and output languages have to be inferred.

This reverse engineering approach was implemented as an open source tool called ReverX¹. An experimental evaluation of the tool was carried out with publicly available network traces, to determine if an inferred specification can capture the main characteristics of a protocol. For this experiment, we chose the [FTP](#) protocol for two main reasons. First, it is a non-trivial protocol with a reasonable level of complexity that is well-known to most readers, and therefore, it becomes simpler to provide examples in the text. Second, since [FTP](#) is documented in an [IETF RFC \(Postel and Reynolds, 1985\)](#), it facilitates the assessment of the results and allows an intuitive comparison between a manually written specification and the inferred automata. The experiments show that the generated automata can recognize the [FTP](#) protocol with a high f -score level, even for training sets with a relatively small number of messages.

¹<http://code.google.com/p/reverx/>

4.3.1 Inferring the language

The problem of *grammar induction*, or *automata inference*, refers to the process of learning a language L (or obtaining the FSM that recognizes L) from a set of sample sequences. If the sequences given to infer the language are exhaustive and complete, constructing a FSM that accepts all sequences is a matter of adding new paths to the automaton whenever a sequence is not accepted. However, several problems arise when the language is complex and the alphabet is rich, such as in the specification of some network protocols. First, if the language supports arbitrary values from a large domain, it can denote an extremely large alphabet. One example is the language defined by a communication protocol whose message fields can hold variable data. Another problem is concerned with the difficulty to obtain a representative set of sample sequences for languages with very large alphabets. Without some sort of generalization, this would imply that one must obtain a network trace containing *all the messages* of the protocol with *all possible variations and combinations* of parameter data². Consequently, the problem of grammar induction consists in generalizing from an incomplete set of sample sequences, to obtain a FSM that also accepts the missing sequences but not invalid messages (e.g., due to over-generalization).

Our approach derives an input and an output language for the protocol requests and responses respectively. The method to generate either language is equivalent, the only difference is the set of network messages used as the training set. Throughout this section, we exemplify the language inference through the

²This is unreasonable to obtain, for example, in a text field with a variable size of 1 to 256 characters, since there can be over 10^{89} different values.

generation of the input language automaton. The automaton $L = (Q, \Sigma, \delta, \omega, q_0, F)$ is defined as:

Q is a finite, non-empty set of states,

Σ is the input alphabet, i.e., a finite set of fields extracted from all messages,

δ is the state-transition function: $\delta : Q \times \Sigma \rightarrow Q$,

ω is the frequency-labeling function: $\omega : Q \times \Sigma \rightarrow \mathbb{N}$,

q_0 is the initial state, and

F is the set of final states.

Each state in the automaton defines a point in the message that potentially corresponds to a message field and each transition is a possible payload data of that field. In this context, the alphabet of the automaton, i.e., the set of symbols, is that whole set of field payloads that were observed in the training data.

The method for deriving the protocol language consists of two parts: First, a [PTA](#) is constructed to accept the protocol messages of the training set. A [PTA](#) is an ordered tree structure with states and transitions similar to a [FSM](#). Each common prefix of the messages is accepted by the same states and transitions and suffixes are always accepted by branchless paths. The [PTA](#) is derived by the following method: Every network message is composed of an arbitrary sequence of bytes, however, text-based protocols usually resort to well-known delimiter characters to separate the message fields (e.g., a space or a tab). Therefore, by providing a regular expression that defines the delimiters, each message can be decomposed as a sequence of fields and separators. Whenever a symbol (field or delimiter) is rejected by the [PTA](#), a new path of states and transitions is added to accept the remainder of the message. As the [PTA](#) is built, the transitions are labeled with the number of times they are visited, to keep track of the frequency that each payload





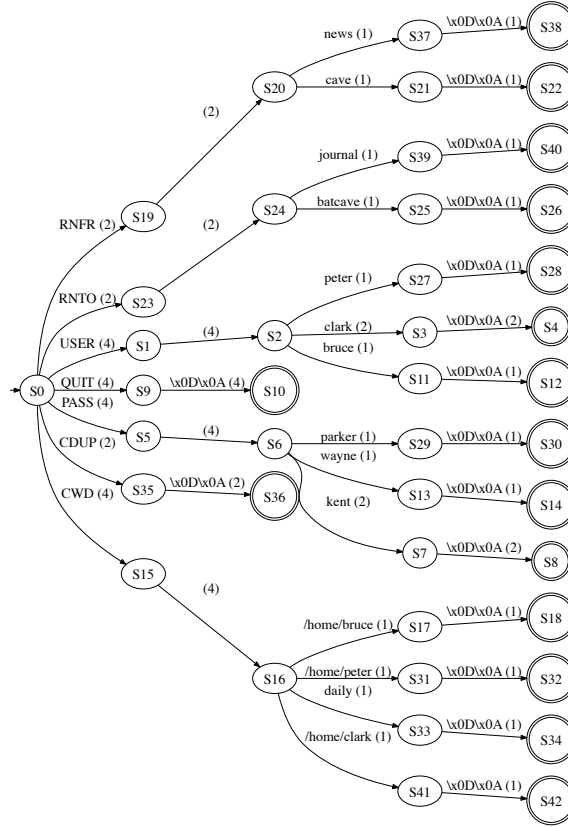
	Requests	Responses
Session 1	 IP1 connects to server IP1: USER clark IP1: PASS kent IP1: QUIT	220 FTP Server ready. 331 User name clark ok. 230 User logged in. 221 Goodbye.
Session 2	 IP2 connects to server IP2: USER bruce IP2: PASS wayne	220 FTP Server ready. 331 User name bruce ok. 230 User logged in.
Session 3	 IP3 connects to server IP3: USER peter IP3: PASS parker IP3: CWD /home/peter IP2: CWD /home/bruce IP2: RNFR cave IP2: RNT0 batcave IP2: QUIT IP3: CWD daily IP3: CDUP IP3: RNFR news IP3: RNT0 journal IP3: QUIT	220 FTP Server ready. 331 User name peter ok. 230 User logged in. 250 CWD command successful. 250 CWD command successful. 350 Specify new name. 250 RNT0 command successful. 221 Goodbye. 250 CWD command successful. 250 CDUP command successful. 350 Specify new name. 250 RNT0 command successful. 221 Goodbye.
Session 4	 IP1 connects to server IP1: USER clark IP1: PASS kent IP1: CWD /home/clark IP1: CDUP IP1: QUIT	220 FTP Server ready. 331 User name clark ok. 230 User logged in. 250 CWD command successful. 250 CDUP command successful. 221 Goodbye.

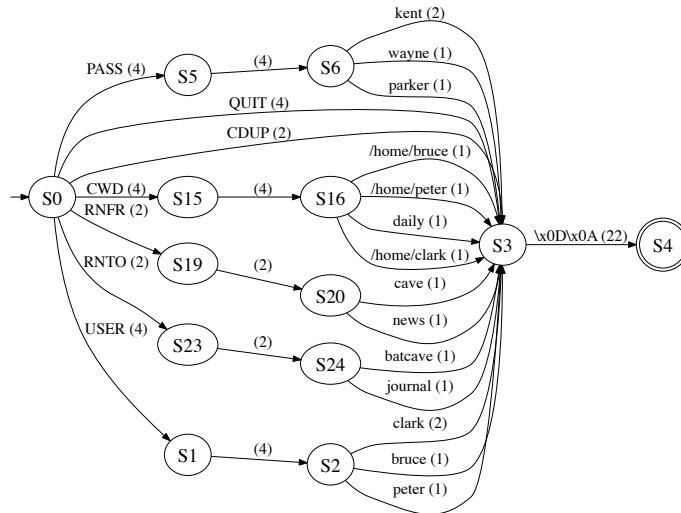
Figure 4.5: Example of an FTP network trace.

has been observed in that message field. The resulting frequency-labeled FSM is the *partial language* automaton, which is similar to a probabilistic automaton, where instead of a probability value, each transition has associated an absolute frequency.

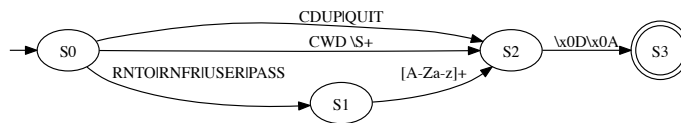
Figure 4.6 displays a complete example of inferring the input protocol language from a simple network trace of Figure 4.5. The trace was obtained from four FTP sessions. Every message is delimited by a CR and a LF characters, and each field is separated by a space character, as specified by RFC 959 (Postel and Reynolds, 1985). Figure 4.6a shows the PTA derived at the end of this part, where \x0D\x0A depicts the hexadecimal representation of the message delimiter. Ad-



(a) Partial language automaton (PTA).



(b) Minimization.



(c) Generalization.

Figure 4.6: Inference of the protocol language.

ditionally, each transition is labeled with the frequency that it was visited (in parenthesis).

The second part in inferring the protocol language consists in generalizing the partial language, so that it accepts other messages of the protocol besides the ones present in the training set. In order to produce a more generic [FSM](#), one needs to identify and abstract the segments of the message formats that correspond to the same group of payloads (e.g., parameters of a command). At the same time, we also want to produce a concise automaton with a minimal number of states and transitions. Otherwise, the same message fields could be scattered among equivalent states and transitions, needlessly augmenting the complexity of the derived specification.

We employ a Moore reduction procedure for deterministic finite automata minimization ([Hopcroft et al., 2006](#)) that produces an equivalent [FSM](#) with a minimum number of states and transitions. During the minimization process, equivalent states are merged, i.e., states with similar transitions that lead to equivalent states. Merging two states also causes equivalent transitions to be merged and the resulting label is combined by adding the respective frequencies. Figure 4.6b shows the partial language automaton after minimization.

Once the automaton is minimized, the labeled frequencies are analyzed to identify parts of the automaton that should be generalized. Our approach is based on the idea that most protocols make use of the concepts of messages with fixed command names and variable parameters. This occurs to facilitate the parsing of the protocol messages—some predefined fields define how each message should be processed, establishing the meaning of the remaining bytes. Most textual protocols, for instance, resort to command fields (usually the first) with the command

name, optionally followed by the respective parameters with variable data (or by some other sub-commands). The different keywords that each command field can have are specified by the protocol and should be derived. However, the individual instances of parameter data should be abstracted away and identified as parameter fields.

Intuitively, fields associated with predefined values, such as command keywords, should appear often in the network traces, as opposed to the variable and less recurrent nature of each instance of a parameter data. Parameters can therefore be recognized in states of the automaton that accept a wide range of different symbols (each one is a particular instance of that parameter). Additionally, these symbols should appear with relatively low frequency, since each individual instance of a parameter should be much less common than a command keyword. Alongside, one cannot rely only on the individual frequency of each symbol, or else commands that appear rarely in the traces could be misidentified as parameters.

Based on these ideas, the generalization algorithm is depicted in Algorithm 4.1. Two generalization parameters are employed, the *Variability Threshold* (*VT*) and the *Command names Threshold* (*CT*). Any state of the automaton is selected for generalization if one of two conditions is satisfied (Line 16):

- C1) the ratio of the number of symbols its accepts over the total frequency of its transitions is above *VT*;
- C2) the total number of symbols is larger than *CT*.

Condition C1 determines that a field is a parameter by looking for states that accept a wide range of symbols relative to the total number of times they were observed in the trace (i.e., the sum of frequency labels on that state). Therefore,

```

1 Function generalizeLanguage
2   Input: Automaton  $L: (Q, \Sigma, \delta, \omega, q_0, F)$ 
3      $VT$ : Variability threshold ( $0 \leq VT \leq 1$ )
4      $CT$ : Command names threshold ( $CT > 0$ )
5   Output: Automaton  $L$ 
6
7    $generalize \leftarrow \mathbf{TRUE}$ 
8   while  $generalize$  is TRUE do
9     .  $generalize \leftarrow \mathbf{FALSE}$ 
10    .
11    . foreach  $q \in Q$  do
12    .    $\#Trans_q \leftarrow |\{\delta(q, s) \neq \mathbf{UNDEFINED}, \text{with } s \in \Sigma\}|$ 
13    .    $Freq_q \leftarrow \sum_{s \in \Sigma} \omega(q, s)$ 
14    .    $Var_q \leftarrow \#Trans_q / Freq_q$  // variability of state  $q$ 
15    .
16    . if  $Var_q > VT$  or  $\#Trans_q > CT$  then
17    .    $Generic_q \leftarrow$  create a generic symbol that represents all transitions in  $q$ 
18    .   that are not delimiters
19    .   foreach  $a \neq \text{delimiter} \in \Sigma : \delta(q, a) \neq \mathbf{UNDEFINED}$  do
20    .   .  $\delta(q, Generic_q) \leftarrow \delta(q, Generic_q) \cup \{\delta(q, a)\}$ 
21    .   .  $\omega(q, Generic_q) \leftarrow \omega(q, Generic_q) + \omega(q, a)$ 
22    .   .  $\delta(q, a) \leftarrow \mathbf{UNDEFINED}$ 
23    .   .  $\omega(q, a) \leftarrow 0$ 
24    .   .  $generalize \leftarrow \mathbf{TRUE}$  // keep generalizing
25    .    $DeterminizeFSM(L)$  // converts to deterministic FSM
26    .    $MinimizeFSM(L)$ 
27 return  $L$ 

```

Algorithm 4.1: Generalization of the protocol language.

VT should be set to a value that captures the variability and sporadic nature of parameters. Consider for example a message field that can hold four distinct command names. In 200 messages, the value of the field will be distributed among the four commands (not necessarily evenly), and therefore the ratio of symbols over the total frequency will be $4/200 = 0.02$. On the other hand, a state that represents a parameter field (e.g., a pathname) is not bound to a limited number of fixed symbols. On 200 messages, such a field could have 150 different values and the ratio would be $150/200 = 0.75$. The evaluation section studies the sensitivity of the reverse engineering procedure to VT , and it is possible to observe

that the generalization works effectively for a wide range of values.

Condition C2 says that every state accepting more symbols (payload instances in that message field) than what a typical command field would, should also be considered a parameter. The purpose of this condition is to prevent traces that may be skewed towards a few commands to affect the inference of the specification—unusually common parameters could otherwise be incorrectly regarded as commands. However, *CT* is quite generic and only needs to be greater than the maximum number of different commands that a field can have (e.g., *CT* = 30 is an acceptable value because it is unlikely for a command field to accept more than 30 different command names).

A state identified as a field parameter is generalized by replacing the symbols of its transitions with a special *generic symbol* (Lines 17-22). However, transitions that define delimiters (such as the symbol space or the *CR* and *LF* characters) are ignored and their symbols are kept unchanged. This causes the automaton to keep its most important structural features (such as delimiters) and prevents it from being over-generalized.

The new generic symbol is defined as a regular expression that tries to capture the nature of the data that is defined by the respective transitions (Line 17). ReverX considers the following generalization classes:

- $[A-Za-z]^+$ when the transitions define only letter characters;
- $[0-9]^+$ when the transitions define only numbers;
- $\backslash S^+$ when the transitions define letters, digits, and at least one other type of character (e.g., comma, dot, tab).

In addition, some generalization classes can be combined, such as to produce $[A-Za-z0-9]^+$. The generalization of a previously generalized symbol with an-

other symbol, produces their union. For instance, creating a generic symbol from USER and $[0-9]^+$ results in $[A-Z0-9]^+$.

Setting the symbols of the transitions of a given state, to the same generalized symbol, renders the FSM nondeterministic. Therefore, we employ a standard determinization algorithm (Hopcroft et al., 2006) that will merge all nondeterministic transitions into a single transition, effectively producing a new generalized version of the language L (Line 25). The minimization algorithm is again applied to produce a simpler, yet equivalent, automaton (Line 26). This procedure is repeated until no more states can be generalized.

Returning to the FTP example, Figure 4.6c represents the inferred Language L after generalization. In this example, we configured ReverX with $VT = 0.5$ and $CT = 30$. State S0 was not generalized because its variability is $7/22 = 0.32 < VT$. However, states S2, S6, S16, S20, and S24 were identified as parameters (e.g., in S6 $3/4 = 0.75 > VT$) and were therefore generalized. State S16 corresponded to a parameter of the CWD and was generalized with symbol $\backslash S^+$, while the other generalizations were set to the symbol $[A-Za-z]^+$. The minimization and determinization operations were then responsible for merging similar transitions and states, thus producing the final automaton that corresponds to the inferred input language. ReverX also concatenates sequences of trivial transitions and states, which accounts for the transition CWD $\backslash S^+$ in S0 (symbol CWD was joined with symbol $\backslash S^+$).

The output language is derived following a similar process, although using the server responses from the network trace (see Figure 4.7).

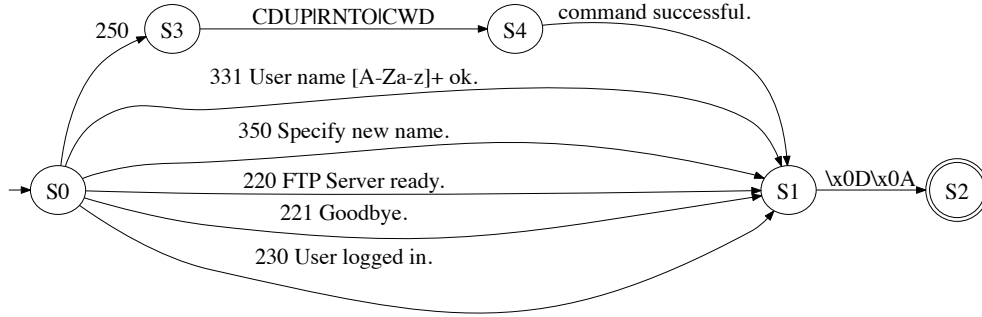


Figure 4.7: Inferred output language.

4.3.2 Inferring the protocol state machine

Similarly, the process of inferring the protocol state machine is usually difficult to perform with network protocols. In this case, one needs to consider the possible ways to combine the different message types, and derive an automaton that only accepts the valid interactions among the communicating parties. For example, in some situations, the messages have to be transmitted in some pre-defined order (e.g., a message with a username has to be followed by a message with a password), while in others they can be interchanged (e.g., a server accepts messages with commands to list/add/remove files). Both behaviors need to be learnt and correctly abstracted in the [FSM](#), using a process that is based in incomplete data.

The second phase of the specification inference uses the same traces and the previously derived languages [FSM\(s\)](#) to generate the protocol state machine. The automaton of the input state machine models only the interaction from the client to the server, i.e., what the servers should expect from the clients, and should capture the rules that dictate the client's behavior, such as the login requests must always precede the other types of requests. The input/output state machine provides a more complete view of the protocol execution by also modeling the

interaction between servers and clients.

In the following discussion, we will concentrate on deriving the input/output state machine, and in the next section we will compare the input and input/output state machines. The automaton for the input/output state machine is a Mealy machine $M = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$, where³:

Q is a finite, non-empty set of states,

Σ is the input alphabet, i.e., a finite set of input message types extracted with the support of the input language automaton,

Ω is the output alphabet, i.e., a finite set of output message types extracted with the support of the output language automaton,

δ is the state-transition function: $\delta : Q \times \Sigma \rightarrow Q$,

λ is the output function: $\lambda : Q \times \Sigma \rightarrow \Omega$,

q_0 is the initial state, and

F is the set of final states.

Automaton M is constructed from the application sessions. Each session corresponds to a sequence of messages that were exchanged during the same interaction between a client and a server (see Figure 4.5 for an example with four sessions). To identify individual sessions in the trace, we cluster messages that share similar network characteristics, such as network addresses and time proximity. In the current version, ReverX groups each application session based on the following criteria:

- same source and destination Internet Protocol (IP) and port addresses;

³The input state machine uses a simplified version of M without Ω and λ because the outputs are not considered (Moore machine).

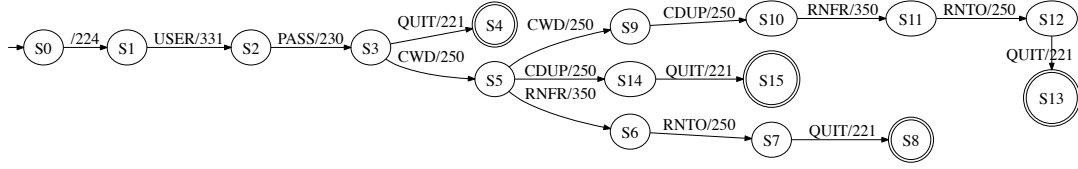
- **TCP** sequence numbers follow a monotonic increasing function, with the next sequence number being at most the sum of the last sequence number with the length of that last packet;
- temporal gaps between messages smaller than one hour.

Then, we use the previously inferred languages to convert each session into a sequence of message types. Each path in the input or output language automata corresponds to a distinct message format to which can be assigned a unique identifier. Therefore, the inferred type of a request or response is identified by processing it with the respective input or output language. By following this approach iteratively, ReverX transforms each session into a sequence of message type identifiers.

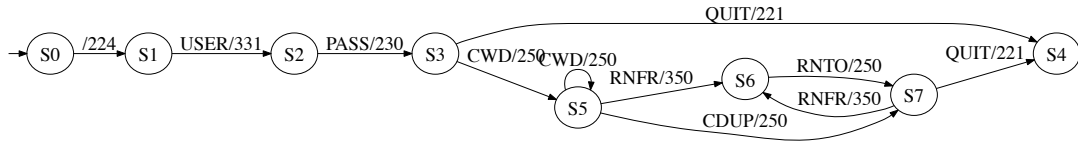
Analogous to the inference of a protocol language, a **PTA** is built to accept the sequences of message types present in the application sessions (although without requiring the frequency-labeling scheme). To derive an input state machine, we resort only to the input messages and ignore the responses, while with the input/output state machine we need both types of messages.

Figure 4.8a exemplifies how a **PTA** is inferred from the network trace. After clustering the network messages in individual protocol sessions, as depicted in Figure 4.5, the tool uses the language **FSM** to convert the sessions into sequences of message types and builds a **PTA** from those sequences. For the sake of readability, each message type is denoted in the figure with the command name of the request and the reply code of the response.

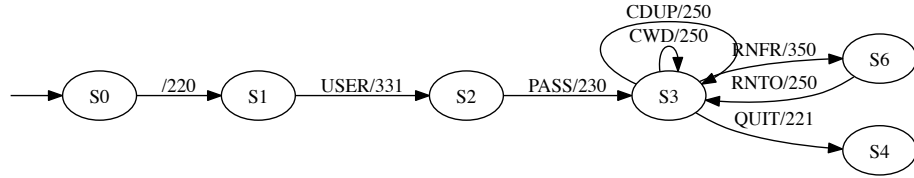
The **PTA** only captures the sequence of transitions between the protocol messages exactly as they appear in the traces. To derive the protocol state machine, it is necessary to identify and merge the automaton states that correspond to the



(a) Partial protocol state machine (PTA).



(b) Merging I: Merging of states reached from the same message types.



(c) Merging II: Merging of equivalent states.

Figure 4.8: Input/output state machine inference.

same protocol state. Algorithm 4.2 depicts the generalization procedure employed in the input/output state machine inference⁴. There are two conditions that identify and merge any pair of states:

- C1) both states are reached under the same input and output message types (Line 8), or
- C2) there is no causal relation between both states (Lines 19–20 and 22).

In the first place, we find out which pairs of states are reached under similar conditions (i.e., from the same input and output message types) because they probably represent the same protocol state. These states are merged by creating a new state with the transitions from each pair of states to merge—the two states are removed from the automaton, and any transition that pointed to either of these

⁴For the input state machine, the procedure is adapted by removing the conditions related to Ω and λ .

```

1 Function mergeStateMachine
2   Input: Automaton  $S: (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$ 
3   Output: Automaton  $S$ 
4
5   // merge states reached from similar message types
6   foreach  $q \in Q$  do
7     . foreach  $p \in Q$  do
8       . if  $\exists a \in \Sigma, x \in \Omega, r, s \in Q : \delta(r, a) = q \wedge \delta(s, a) = p \wedge \lambda(r, a) = x \wedge \lambda(s, a) = x$ 
9         . then MergeStates( $p, q$ )
10
11   // merge states without a causal relation that share
12   // at least one input/output message type
13   merging  $\leftarrow$  TRUE
14   while merging is TRUE do
15     . merging  $\leftarrow$  FALSE
16     . foreach  $q \in Q$  do
17       . foreach  $p \in Q$  do
18         . // if there is not a casual relation
19         . if  $(\exists a, b \in \Sigma : \delta(q, a) = p \wedge \delta(p, b) = q)$  or
20         .  $(\forall a, b \in \Sigma : \delta(q, a) \neq p \wedge \delta(p, b) \neq q)$  then
21         . // and if they share at least one message type
22         . if  $\exists a \in \Sigma, r, s \in Q : \delta(q, a) = r \wedge \delta(p, a) = s$  then
23         . MergeStates( $p, q$ )
24         . merging  $\leftarrow$  TRUE
25     . MinimizeFSM( $S$ )
26   return  $S$ 

```

Algorithm 4.2: Merging process to produce the protocol input/output state machine.

states is updated to point to the new state. As with most heuristic approaches, extrapolating from a smaller subset may incur in over-optimistic lossy generalizations. For instance, this procedure may fail if the protocol specification defines the same message type (e.g., the same command name) for different purposes, something that is normally avoided in practice.

The second merging condition tries to identify equivalent states that may still be reached from different message types (Lines 13–25). For instance, after logging in, a user may create, edit, or delete files, all seemingly interchangeable protocol commands. With respect to the protocol state machine, the order of these

messages is irrelevant as they are executed in the same protocol state. However, the network traces are most likely incomplete, in the sense that many causal relations between protocol messages are absent. To deduce a more compact protocol state machine, the algorithm needs to make a few assumptions about the equivalence of some states. First, if there is a transition from one state to another, but not vice versa, it establishes a causal relation between them and therefore they cannot be considered equivalent. Second, protocol states without similar transitions (i.e., not accepting at least one common message type) are also never considered equivalent. These two conditions are enforced by the algorithm to determine the cases where states can be merged (Lines 19–20 and 22). At each merging iteration, the algorithm minimizes the produced FSM to obtain a simpler but equivalent automaton (Line 25). This procedure is repeated until no more states can be merged, and the resulting automaton is the protocol state machine.

Figure 4.8b displays the automaton M after the first merging condition (Line 9). Then, the merging process further searches the automaton for potentially equivalent states. Figure 4.8c results in merging several states that do not have causal relations and that share at least one message type (Line 26).

4.3.3 *Input versus input/output state machine*

This reverse engineering approach can generate two kinds of automata for the protocol state machine, which can be used by different applications depending on their needs. However, there are other considerations that may lead us to choose one alternative over the other besides application requirements.

The input state machine models the client's behavior leaving out the server responses, which typically causes a simpler automaton to be derived. Since this

machine only resorts to one language, it eases the whole reverse engineering process making it less prone to errors, for example due to over-generalization or difficulties in recognizing the output messages. In fact, in several communication protocols the output language is not fully defined, allowing different implementations to specify the contents and format of the messages. This usually translates into a more intricate and extensive language, which is more difficult to reverse engineer, thus producing complex state machine automata.

On the other hand, using both input and output messages provides more information to derive the protocol specification. Since the input/output state machine also resorts to the output messages, it can better distinguish transitions and protocol states, which would otherwise be indistinguishable. In addition, the input/output state machine automaton captures the natural pattern of the protocol execution that consists in sequences of action/reaction pairs corresponding to input/output messages.

Figure 4.9a and 4.9b shows the result of inferring the input state machine and the input/output state machine from the network trace of Figure 4.5. We can see that the automata are almost isomorphic (the only distinction is an additional initial state in the input/output state machine that corresponds to the welcome banner sent by the server).

There are cases, however, where some different protocol states may be indistinguishable when resorting only input messages, and therefore, inferring an input/output protocol state machine produces a more accurate specification. Consider the same trace with an extra fifth session (see Figure 4.10) that contains the input message type PASS, but a different output message type (530). The automaton of the new input state machine is changed considerably (Figure 4.9c), some-

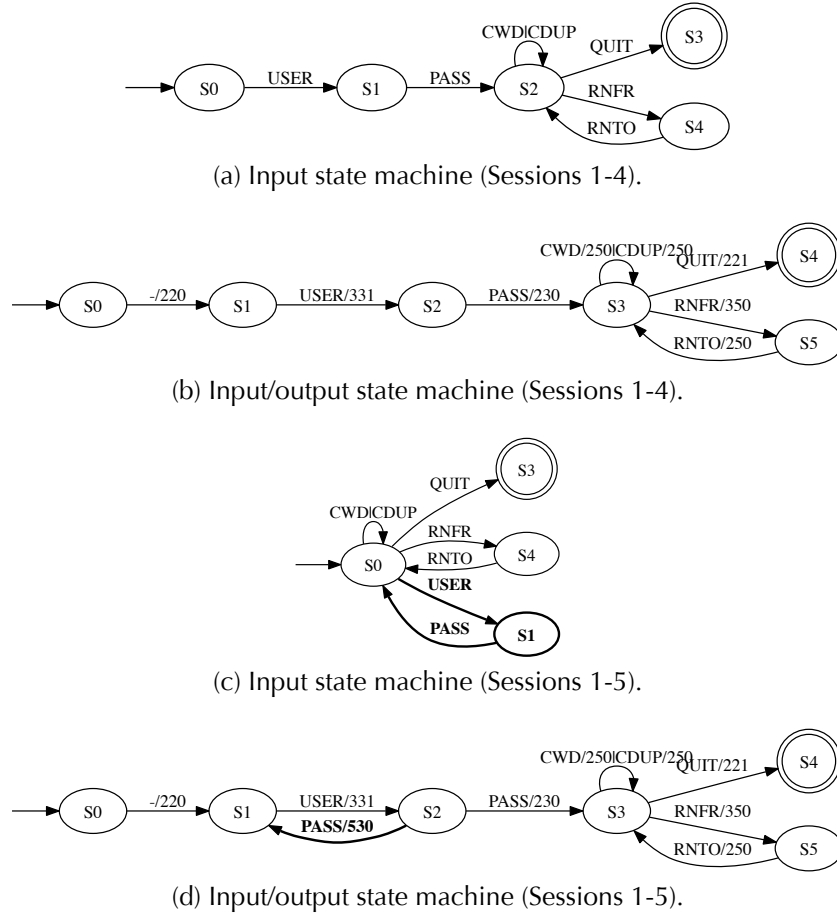


Figure 4.9: Input and input/output state machine inference.

thing that does not occur for the input/output state machine (4.9d). The sequence of message types in the fifth session: $\text{USER} \rightarrow \text{PASS} \rightarrow \text{USER} \rightarrow \text{PASS}$ causes states S_0 , S_1 , and S_2 to be merged in the input state machine (darker lines) because they are undistinguishable using only the input messages. However, since the input/output state machine also uses output messages, it can correctly distinguish the input/output symbols $\text{PASS}/230$ and $\text{PASS}/530$ (Figure 4.9d). The additional transition in the input/output state machine (in a darker line) accurately models the case when the password provided in the command PASS is incorrect and the protocol reverts to the initial state.

	<i>Requests</i>	<i>Responses</i>
Session 5	<i>IP2 connects to server</i>	220 FTP Server ready.
	USER bruce	331 User name bruce ok.
	PASS wrong	530 Login incorrect.
	USER bruce	331 User name bruce ok.
	PASS wayne	230 User logged in.
	QUIT	221 Goodbye.

Figure 4.10: Additional session in the network trace.

4.3.4 Experimental evaluation

This section evaluates the quality of the inferred language and state machine automata derived by ReverX. To achieve this objective, we chose to generate a specification of the **FTP** protocol because it is a well-known protocol with a reasonable level of complexity. Since **FTP** is documented by a **IETF RFC** (Postel and Reynolds, 1985), it allows the comparison between the inferred automaton and a reference automaton, manually produced from the documentation. The network traces were obtained from a public repository to facilitate the reproducibility of the results and to demonstrate that our solution can use unbiased network traces collected from the regular utilization of the protocol (without being specifically produced for reverse engineering purposes).

Network traces

The evaluation uses a large publicly available **FTP** network trace obtained from 320 public **FTP** servers located at the Lawrence Berkeley National Laboratory (LBL)⁵. The trace spans a period of ten days and contains over 3.2 million packets from 5832 clients. Even though this trace had been previously processed to anonymize the clients (Pang and Paxson, 2003), a few packets still contained mal-

⁵<http://ee.lbl.gov/anonymized-traces.html>

formed messages that had to be removed (such as illegal command names). This resulted in a clean packet capture file containing more than 2 million [FTP](#) packets, which corresponded to approximately 40.7% client requests and 50.3% servers responses. We further found that the traces were heavily skewed towards an excessive amount of PORT commands (roughly 57.9% of the traffic was related to PORT requests and responses). Although this affects the richness of the traces, our approach is still able to overcome such limitation as long as the sample used to infer the automata has sufficient variability and protocol coverage.

Experimental evaluation

The evaluation focuses on deriving the language and state machine of the [FTP](#) protocol. Overall, ten independent experiments were conducted, each one employing different subsets of the trace as training data and as test sets. In more detail, the procedure for each experiment was: We randomly selected eleven points in the trace to select 8000 consecutive [FTP](#) messages. This inevitably splits the protocol sessions that were taking place in the trace at the arbitrary cut points, rendering them unsuitable for the state machine inference (and for the evaluation). Thus, we ignored the packets that belonged to these incomplete sessions and selected further messages from the trace to complete each block with 8000 consecutive messages from complete sessions. From the eleven resulting blocks, the first was used as the training set, and the remaining ten as test sets.

ReverX inferred the language and state machine of the protocol for various configurations. Table [4.1](#) shows the average characterization of the ten training sets randomly chosen for the experiments. Each column of the table corresponds to a different sample size of the training set (e.g., the “250” column means that

	250	500	1000	2000	4000	8000
Sessions	1.1	1.1	1.1	3.1	44.9	63.1
Input language	110.9	235.9	485.9	970.0	1858.2	3655.0
Output language	139.1	264.1	514.1	1030.0	2141.8	4345.0
Input state machine	110.9	235.9	485.9	970.0	1858.2	3655.0
Input/output state machine	250.0	500.0	1000.0	2000.0	4000.0	8000.0

Table 4.1: Characterization of the training sets.

only 250 packets of the 8000 are used for the training set). Since to infer the input (or output) language, only the client (or server) messages are needed from the training set, this value is presented in row “Input language” (or “Output language”). For instance, given the different ratio of requests and responses, to obtain the input language from a sample of 250 messages, an average of 110.9 messages are used (whereas the output language derivation employs about 139.1 messages). In some cases, there is a very small number of protocol sessions present in the training sets, namely for samples with less than 4000 messages (row “Sessions”). Our evaluation will show that these sets have insufficient information to support the creation of representative *FSMs*. The input state machine inference only needs the input messages (row “Input state machine”), while the input/output state machine requires both requests and responses, thus using all the messages in the sample (row “Input/output state machine”).

For the language inference, we experimented six sample sizes from the training set, doubling from 250 to 8000 messages, and varied *VT* (the variability threshold) from 0.0 to 1.0, in increments of 0.1. Parameter *CT* (the command name threshold) was fixed to 30 because this value is adequate for many protocols, including *FTP*. Overall, the six sample sizes and eleven values of *VT* produced 66 input language automata and 66 output language automata. Each language automaton was then evaluated using the ten test sets of 8000 messages (regardless

of the sample size).

For the state machine inference, we derived both the input state machine and the input/output state machine of the protocol. We relied on the same configurations of sample size and *VT* because the state machine inference is based on the previously inferred language(s). However, since the input/output state machine uses two languages (input and output), its evaluation requires the configuration of two *VT* variables, instead of only one. Therefore, each experiment produced a total of 66 input state machines and 726 input/output state machines. As before, to evaluate each one of these automata, we resorted to the ten test sets.

The quality of the inferred automata (language and state machine) is assessed with the following metrics:

Recall: measures the coverage of the inferred automaton, i.e., how much of the protocol specification has been captured by the *FSM*. Recall is calculated as the ratio that a randomly selected set of *valid* protocol messages (or protocol sessions) is accepted by the inferred automaton.

$$Recall = \frac{\# \text{ accepted messages (or sessions)}}{\# \text{ messages (or sessions)}}$$

Precision: determines the soundness of the automaton, i.e., if the inferred automaton is not overly-generalized. We calculate precision as the ratio that a randomly selected (*valid* or *invalid*) set of protocol messages (or sessions) accepted by the inferred *FSM* is in fact *valid*.

$$Precision = \frac{\# \text{ accepted valid messages (or sessions)}}{\# \text{ accepted messages (or sessions)}}$$

F-score: measures the accuracy of a test by computing a score that considers the precision and the recall.

$$f\text{-score} = 2 \times \frac{\text{Recall} \times \text{Precision}}{\text{Recall} + \text{Precision}}$$

These metrics are inspired on widely used metrics from other fields, such as information retrieval (Van Rijsbergen, 1979; Manning et al., 2008) and intrusion detection (Alessandri, 2004). It is very simple to have a recall of 1 with over-generalized automata that accept all messages (or sessions) or a high precision with over-specialized automata. However, our goal is to achieve both a high recall *and* precision (Manning et al., 2008), which is reflected by *f*-score values close to 1.

Testbed

The experiments were carried out with ReverX in an Intel Pentium Dual Core 2.8GHz with 2GB of memory running Ubuntu 11.10. ReverX is an open source tool developed in Java and is freely available at <http://code.google.com/p/reverx/>. It resorts to some Java and native libraries to access the packet capture files (in *Tcpdump* format), such as LibPcap⁶ and *jnetpcap*⁷, and uses the *dot*⁸ program to generate the high-quality diagrams of the automata.

⁶<http://www.tcpdump.org/>

⁷<http://jnetpcap.com/>

⁸<http://www.graphviz.org/>

Experimental results – protocol language

All experiments described in this section were executed with $CT = 30$. Each entry of recall, precision, and f -score is an average of 100 values (ten experiments, each with one training set and ten different test sets).

To calculate the recall of the language **FSM**, we used the 8000 messages of each test set in the inferred automata to find out which packets were accepted or rejected. A recall with a value near 1 means that most of the messages are recognized and, therefore, that the **FSM** was able to capture most of the protocol language used between the **FTP** clients and server.

To calculate the precision, we require messages that are accepted by the derived automaton and then we need to determine if they are accepted or rejected by the reference language of **FTP**. This gives us a measure of how accurate the inferred **FSMs** are, since a higher precision value indicates that fewer extraneous messages are recognized. To get data for the experiment, we decided to follow an approach based on the mutation of valid messages in order to (potentially) produce invalid messages that are still accepted by the inferred automaton. We configured the *editcap* tool⁹ to mutate each byte of every packet with a probability of 0.2, and checked if the inferred **FSMs** accepted them. This process was applied repeatedly to the original test set until we had 8000 mutated messages that were accepted by the inferred automaton under evaluation. To calculate the precision, we then resorted to a reference **FSM** to find how many of those messages were in fact in conformance with **FTP**.

⁹<http://www.wireshark.org/docs/man-pages/editcap.html>

VT		Input language						Output language					
		250	500	1000	2000	4000	8000	250	500	1000	2000	4000	8000
0.0	Rec.	0.98	0.98	0.98	1.00	1.00	1.00	0.96	0.96	0.96	0.97	0.99	1.00
	Prec.	0.59	0.59	0.60	0.79	0.76	0.76	1.00	1.00	1.00	1.00	1.00	1.00
	F-sc.	0.73	0.74	0.74	0.88	0.86	0.86	0.98	0.98	0.98	0.98	0.99	1.00
0.1	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.96	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.98	0.88	0.88	0.88	0.98	0.99
0.2	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.3	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.4	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.5	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.6	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.7	Rec.	0.11	0.12	0.94	0.95	1.00	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.8	Rec.	0.11	0.12	0.94	0.95	0.99	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
0.9	Rec.	0.11	0.12	0.94	0.95	0.99	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99
1.0	Rec.	0.11	0.12	0.94	0.95	0.99	1.00	0.78	0.78	0.78	0.78	0.97	1.00
	Prec.	1.00	1.00	1.00	1.00	0.99	0.98	1.00	1.00	1.00	1.00	0.99	0.98
	F-sc.	0.20	0.21	0.97	0.97	0.99	0.99	0.88	0.88	0.88	0.88	0.98	0.99

Table 4.2: Evaluation of the inferred input and output languages.

Table 4.2 shows the recall, precision, and f -score of the inferred language automata (darker cells have higher values). For each value of the generalization parameter VT , we produced FSMs from different sizes of the training set (ranging

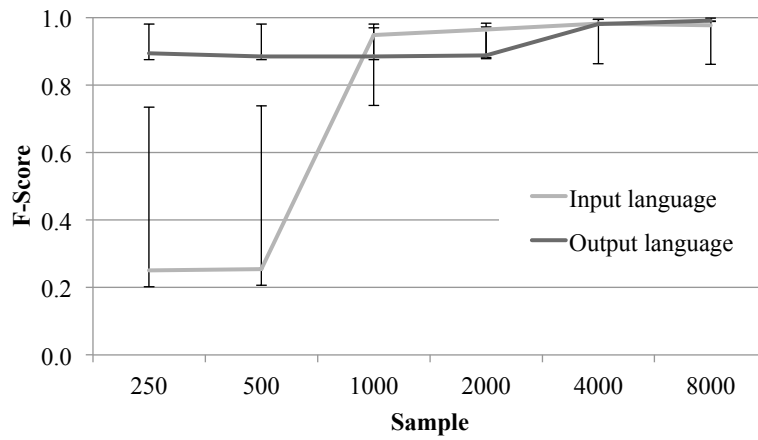


Figure 4.11: Impact of the sample size on the language inference.

between 250 to 8000 messages). It is possible to conclude from the table that smaller training set sizes lead to worse quality automata. This is expected since smaller traces lack enough message diversity to make them representative enough of the protocol language.

The impact of the sample size on the inference process is also graphically represented in Figure 4.11 in terms of the f -score metric (the error bars indicate the minimum and maximum f -scores among the different configurations of VT). The derived input language FSMs show a significant improvement for samples of 1000 messages (an average of just 485.9 protocol requests) and above, reaching f -scores in the order of 0.97 for almost all VT settings. The output language requires larger samples with at least 4000 messages (an average of 2141.8 protocol responses) to produce automata with an f -score with the same level of performance (there is only one exception that ended up also providing good results, with $VT \leq 0.1$ and very small samples). This suggests that the input language is simpler than the output language, requiring fewer messages. In fact, the output messages are usually much more complex due to the lack of a complete formal

definition—the server responds with a reply code and some custom description, which can be divided in multiple consecutive packets. In general, of course, there is no exact number for the minimum training set size because it depends on both the protocol complexity and coverage of the trace.

The generalization parameter VT can also affect the quality of the generated automata. For instance, a value of $VT = 0.0$ results in the generalization of most of the states, consequently producing over-generalized FSMs, which accounts for a recall of 1.0 but also for the lowest precision values. These over-generalized automata can recognize any type of FTP message, but they also accept illegal messages. On the other extreme of the spectrum, a threshold $VT = 1.0$ creates FSMs that rarely generalize. These automata reject some legal FTP packets but they are unlikely to accept any illegal messages. In any case, ReverX is relatively insensitive to most values of VT . We believe that ReverX produces good results for such a large range of generalization values due to the restrictions in the merging of states presented in Subsection 4.3.1. Even if some states have a variability larger than VT , transitions that lead to final states or that define delimiters are always preserved, and thus the automata is rarely over-generalized.

There is one special case in the generated output language FSMs that appears to have unexpectedly good results. Some automata obtained from smaller sample sizes (250 messages) have better f -scores than those obtained from larger ones for $VT \leq 0.1$. This occurs because a smaller sample produces a simpler PTA, but given that VT is very small, the result is a wide generalization across the automata. This does not happen for medium sized samples (between 500 and 2000 messages) because they lack the necessary variability for generalization if $VT \geq 0.1$.

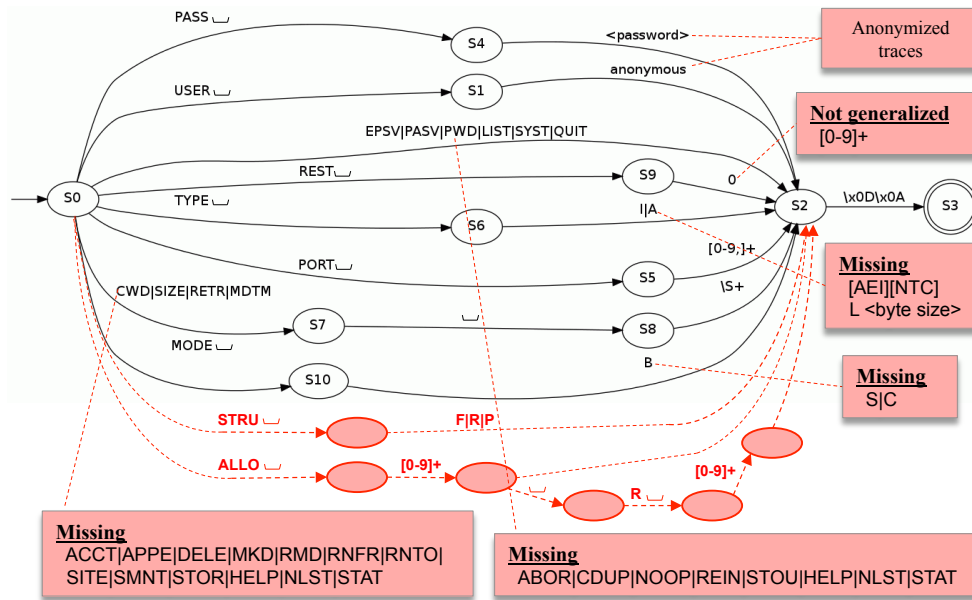


Figure 4.12: Inferred protocol language versus RFC 959.

Figure 4.12 compares one of the derived input language automaton (produced from a sample of 4000 messages and with generalization parameter $VT = 0.2$) against the reference FSM for FTP. The image shows how natural and legible is the generated automaton, and how close it resembles to the original specification. The annotated boxes and dashed lines indicate parts of the RFC 959 specification that were not inferred by ReverX. This incomplete inference is to be expected due to a generic limitation of trace-based solutions—they cannot infer what is absent from the training set. In fact, all missing commands and parameters were not available in the network trace (e.g., ALLO, ACCT, CDUP). Additionally, some transitions were not generalized due to the low diversity of the trace (e.g., REST 0) or because of the anonymization procedure (e.g., USER anonymous).

We further note that the approach is particularly resilient to over-generalization. Besides preventing some kinds of transitions from generalizing (such as those that

lead to a final state or that define delimiter characters), it tries to deduce the data domain of the parameters. For instance, command PORT, which receives as parameter a string with digits separated by commas (e.g., PORT 65,240,185,205,-66,11), is generalized as PORT [0-9,]+ instead of PORT \S+ that would accept a wider range of characters. Other transitions, such as CWD \S+, had to be abstracted in such a way because they actually contained many classes of characters (e.g., letters, digits, and at least another non-whitespace character).

Experimental results – protocol state machine

The generated protocol state machines are also evaluated using the same metrics, in particular we derived input state machines and input/output state machines for the same configurations of sample sizes and *VT* values. Recall is calculated with the protocol sessions extracted from the test sets, which are then tried in the inferred *FSMs*. To obtain the precision, we use a similar approach as in the language evaluation using instead mutated protocol sessions. Each message in a session has a 0.2 probability of mutation, which in this case corresponds either to its deletion or to its order exchange with the following message. This simple method is a very convenient way to create potentially invalid protocol sessions that could be accepted by an over-generalized *FSM*. For instance, if a USER command must always precede a PASS command, a mutation on the first message would effectively render the session invalid. To verify if the sessions accepted by the inferred *FSM* are in fact valid, we built a reference *FSM* for RFC 959 that only recognizes legal *FTP* protocol sessions.

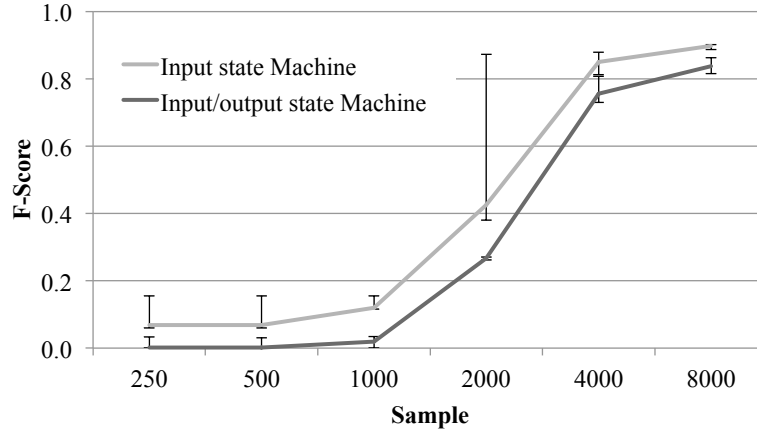
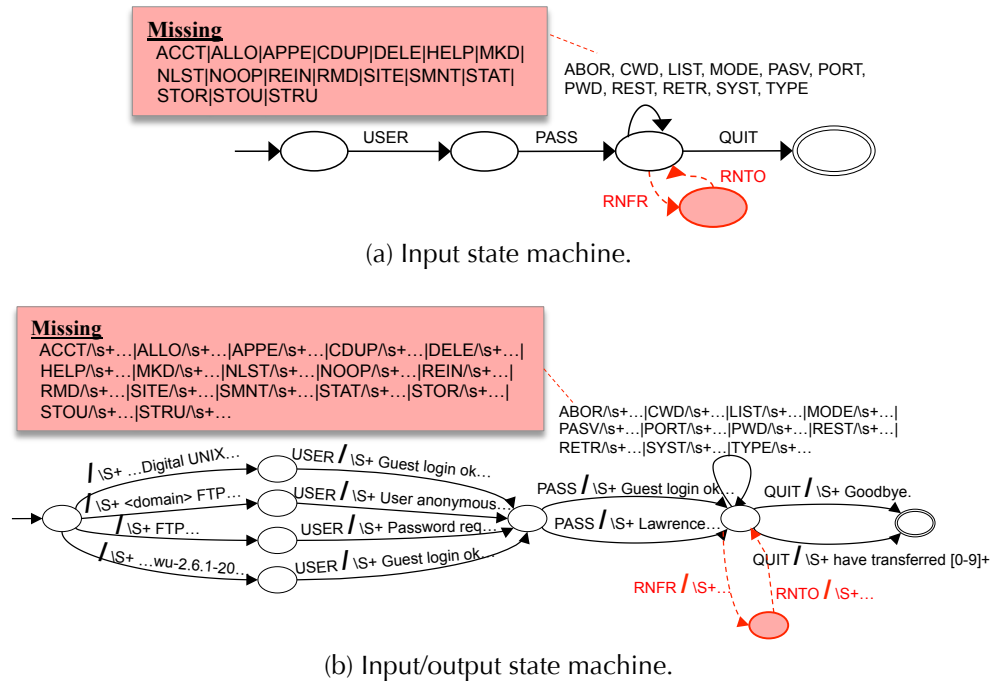


Figure 4.13: Impact of the sample size on the state machine inference.

Figure 4.13 shows the f -score values for the generated state machines (the error bars correspond to the minimum and maximum f -scores among the different configurations of VT). As with the protocol language inference, the best FSMs are built from larger samples of the training set (i.e., around 4000 messages or more). Here, the impact of the sample size is more pronounced because smaller training sets have an insufficient number of protocol sessions. In fact, recalling Table 4.1, we see that the average number of sessions even in samples with 2000 messages is only 3.1, which is clearly inadequate to generate good FSMs. The small error bars in most configurations show that the approach is relatively insensitive to the generalization parameter VT used to infer the input (and output) language(s).

The best derived input and input/output state machines (for training set size of 4000 and $VT = 0.2$) and the respective reference state machines are depicted in Figure 4.14. The annotated boxes and dotted transitions correspond to the part of the FTP specification not captured by ReverX. The approach was able to deduce all states (and commands) observed in the trace, such as the USER and PASS commands issued before any others, and the QUIT command as the final state.



Even though the state due RNFR and RNT0 is missing because these commands do not appear in the trace, our approach would be able to correctly identify this state if they were added to the trace—this is shown in Figure 4.8.

Experimental results – execution time

The average times that the tool takes to infer the protocol language and state machine are displayed in Figure 4.15. The overall execution time has been decomposed in the times spent on the construction of the PTAs, on the generalization and merging procedures (Gen), and on the minimization and determinization algorithms (Min).

The input-only specification (input language plus input state machine) with a sample size between 2000 and 4000 messages takes only an average of 1.6

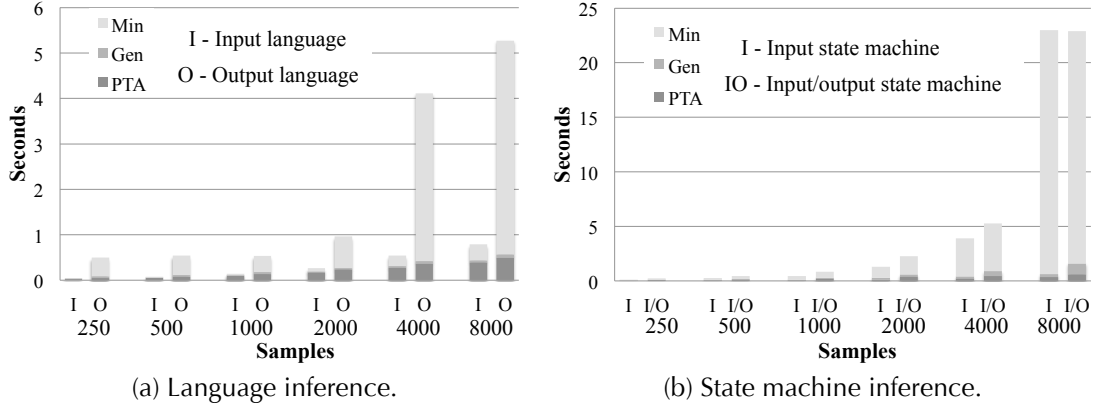


Figure 4.15: ReverX average execution time.

to 4.5 seconds, with most of the time being spent on minimization. To infer an input/output specification, the tool requires more time because it also has to derive the output language. Therefore, to produce a good input/output specification with a sample size between 4000 and 8000 messages, it requires an average of 9.9 and 29.0 seconds.

We also note that the time grows exponentially as the number of messages increases, which is due to the minimization steps of the reverse engineering process. However, since the approach does not need large traces to produce good specifications, this limitation is not relevant for the applications scenarios being considered. Furthermore, advances in minimization algorithms can be applied here to reduce the execution time for larger sets.

4.4 Automatically Complementing a Specification

Protocols are constantly evolving, as new functionality and different message formats are added, rendering the previously defined specifications incomplete or

deprecated. Old specifications must therefore be updated with the new extensions, which typically requires a careful analysis to identify *where* the old specification was changed and *how* it should be updated. Sometimes developers must even incorporate multiple changes from more than one extension, making it an even more challenging task.

Currently, the existing solutions aimed at obtaining protocol specifications in an automated way do not make use of the older versions of the specification, creating the specifications completely from scratch. This means that to complement an existing specification, one must not only get data traces that cover the new features, but also re-create the old data traces in order to conserve the previous coverage of the protocol. Additionally, since these approaches ignore the old specification, one cannot easily identify the new part of the specification that pertains to the extensions, which might be useful, for instance, to prioritize the testing of the new features.

This section presents an approach to automatically complement an existing specification with extensions to the protocol by analyzing the contents of the messages in network traces. The solution is based on our protocol reverse engineering technique and it takes advantage of the older version of the specification. Hence, the data traces it uses are only required to include information concerning the *new extensions* (although they can also have data pertaining to the old specification). This approach can be applied to both open and closed protocols. In fact, closed protocols are a very interesting target for security purposes because, as opposed to open protocols, they are not subject to the public scrutiny and testing. Nevertheless, this approach can shed some light on the specification of closed protocols and on their latest changes. Even without a publicly available

description, existing reverse engineering techniques can infer an approximate specification from network or execution traces, which can then be incrementally complemented using our method as newer traces are captured.

We implemented a prototype tool and evaluated our method with the current specification of the [FTP](#) ([Postel and Reynolds, 1985](#)) and with traffic data collected from 320 public [FTP](#) servers containing several extensions to the protocol. We found that the tool correctly complemented the specification with commands described in five different [RFC](#) extensions. The complemented specification also captured two non-standard protocol commands that were being used by a few [FTP](#) clients. This more complete specification is much closer to the real utilization of the protocol than the original document-based specification. It can provide valuable information as an unifying specification, which we use for testing and security purposes. Features not present in former versions of the specification should be given higher priority in testing. In particular, non-standard or undocumented extensions must be given special attention, since more obscure features are usually less tested.

4.4.1 *Overview of the approach*

We focus on clear-text protocols, which are often used by network servers, such as many of the standard protocols published by the [IETF](#). It is assumed that an older version of the specification already exists and that there are network traces with messages covering the new features (or that some implementation is available from which the network traces can be produced). Although in this section we are using open protocols as an example, our solution can also be applied to closed protocols. The lack of a public protocol description would require an

approximate specification to be inferred instead of being manually translated from the documentation, for instance, by reverse engineering the execution of a server such as described in the previous section.

In this solution, the original protocol specification is modeled as a [FSM](#) that describes the rules of communication between the clients and the servers (Section 4.3). The automaton must capture both the language (i.e., the formats of the messages) and the state machine (i.e., the relation between the different types of messages) of the protocol. Separate specifications are devised for the client and server dialects, i.e., one [FSM](#) defines the messages recognized by the clients and their respective states, whereas the specification pertaining the server is defined by another.

The approach consists in two distinct phases, one dedicated to the language of the protocol and another phase addressing its state machine.

Phase 1: protocol language

One of the things that might change with a more recent version of a protocol is the set of messages that are accepted, i.e., the language it recognizes. Novel messages or formats might be introduced, and therefore, the first step consists in complementing the protocol language with the messages in the network traces.

Algorithm 4.3 depicts the logical steps of the approach to extend the language of a given specification from network traces. Notice that the client requests and server responses are treated separately, so the procedure has to be applied to both specifications. For this reason we use indiscriminately the terms specification, [FSM](#), or automaton while referring to either the client or server specifications.

First, we extract a list of the message formats that are already defined in the

```

1 Function complementLanguage
2   Input: Automaton  $A$ : Original specification of the protocol
3      $NetworkTraces$ : Messages of the protocol
4      $VT$ : Variability threshold ( $0 \leq VT \leq 1$ )
5      $CT$ : Command names threshold ( $CT > 0$ )
6   Output: Automaton  $L$ :  $(Q, \Sigma, \delta, \omega, q_0, F)$ 
7
8    $L \leftarrow$  Empty automaton for the complemented language
9    $Formats \leftarrow$  list of message formats (regular expressions) taken from transitions of  $A$ 
10  foreach Format  $f \in Formats$  do
11    .  $Seq_f \leftarrow$  Sequence of text tokens from  $f$ 
12    . Add a new path to  $L$  to accept  $Seq_f$  // iteratively creates a PTA
13  foreach Message  $m \in NetworkTraces$  do
14    .  $Seq_m \leftarrow$  Sequence of text tokens from  $m$ 
15    . If needed, Add new path to  $L$  to accept  $Seq_m$  // iteratively creates a PTA
16    . Label newly created transitions with New
17    . Update frequency label of visited states
18  MinimizeFSM( $S$ ) // converts PTA to FSM
19
20  // Apply ReverX's generalization procedure for new states and transitions .
21  generalize  $\leftarrow$  TRUE
22  while generalize is TRUE do
23    . generalize  $\leftarrow$  FALSE
24    .
25    . foreach  $q \in Q$ 
26    . . if all transitions in  $q$  are labeled as New
27    . .    $\#Trans_q \leftarrow |\{\delta(q, s) \neq \text{UNDEFINED}, \text{with } s \in \Sigma\}|$ 
28    . .    $Freq_q \leftarrow \sum_{s \in \Sigma} \omega(q, s)$ 
29    . .    $Var_q \leftarrow \#Trans_q / Freq_q$  // variability of state  $q$ 
30    . .
31    . . if  $Var_q > VT$  or  $\#Trans_q > CT$  then
32    . .    $Generic_q \leftarrow$  create a generic symbol that represents all transitions in
33    . .    $q$  that are not delimiters
34    . .   foreach  $a \neq \text{delimiter} \in \Sigma : \delta(q, a) \neq \text{UNDEFINED}$  do
35    . . .    $\delta(q, Generic_q) \leftarrow \delta(q, Generic_q) \cup \{\delta(q, a)\}$ 
36    . . .    $\omega(q, Generic_q) \leftarrow \omega(q, Generic_q) + \omega(q, a)$ 
37    . . .    $\delta(q, a) \leftarrow \text{UNDEFINED}$ 
38    . . .    $\omega(q, a) \leftarrow 0$ 
39    . .   generalize  $\leftarrow$  TRUE // keep generalizing
40    .
41    . DeterminizeFSM( $L$ ) // converts to deterministic FSM
42    . MinimizeFSM( $L$ )
43  return  $L$ 

```

Algorithm 4.3: Complementing the language of an existing protocol specification.

original specification (Line 9, and also see Figure 4.1 for an example specification). Since we are addressing text-based protocols, message formats are modeled as regular expressions. For example, messages `USER jantunes` and `USER nneves` can be modeled as the regular expression `USER .+`. The list of extracted message formats is a comprehensive account of the language recognized by the protocol, i.e., any protocol message must be accepted by at least one of the regular expressions, unless the message follows some extension yet to be specified.

We use the list of extracted message formats to build a *FSM* L for the original protocol language (Lines 8–12). Each message format (regular expression) of the extracted list is tokenized in words and word separators (e.g., spaces, punctuation and any other special characters) (Line 11). Hence, every message format corresponds to a sequence of tokens, and when added to L it will cause the creation of a new path of states and transitions (Line 12). For example, a message type `REST [0-9]+` would be divided in tokens `REST`, the space character, and `[0-9]+`, and the path would therefore be: state S_1 is connected to S_2 by transition `REST`, S_2 is connected to state S_3 by a transition accepting the space character, and finally S_3 is connected to S_4 by transition `[0-9]+`. At the end of this process, a *FSM* that can recognize all messages from the original specification is produced.

The next step consists in identifying and adding new message formats not present in the original language of the protocol (Lines 14–17). The network traces are parsed, and each message is tokenized into a sequence of words and word separators (Line 14) and given to the automaton L . Whenever the automaton fails to recognize a new symbol (i.e., a word or a word separator) in a particular state, a new transition and destination state is created to accept it (Line 15). The frequency that each state is visited during the construction of the new paths is

recorded, and every new transition is labeled for later analysis (Lines 16 and 17). The process used in the construction of the automaton creates a prefix-tree acceptor that can be minimized in order to produce a reduced FSM (Line 18). The final FSM L accepts both the previously defined message formats and the new messages present in the network traces.

However, the newly created paths are not generic enough to accept different instances of the same types of messages (e.g., if a path was created in L to accept the new message `SIZE xfig`, it would not accept similar requests with different parameters like `SIZE newfile`). Therefore, the new paths of states and transitions do not yet represent a message format, which must describe the composition and arrangement of fields of a given type of message. In our approach, a few additional steps must be followed in order to identify messages related to similar requests and to produce a regular expression that captures their common format. In another words, we must identify transitions in L that are associated with predefined values (e.g., command names), which should be explicitly defined in the new specification, and transitions concerning undefined data (e.g., parameters of commands).

To achieve this objective, we apply the same generalization procedure used in ReverX (Subsection 4.3.1) where transitions with data that should be abstracted, such as specific parameters and other variable data, are identified and generalized (Lines 21–41). Naturally, only the transitions created for the new part of the protocol and labeled as “New” should be subject for generalization (Line 26). The other transitions correspond to the definition of message formats that were extracted from the original specification, and are consequently already generalized.

Phase 2: protocol state machine

In the second phase, we process individual application sessions from the network traces to complement the state machine of the protocol with the new message formats and corresponding protocol states.

Algorithm 4.4 shows the part of our approach dedicated to complement the state machine of the protocol, and therefore producing the final extended specification.

Individual sessions are extracted from the traces in order to ascertain the logical sequence of types of messages that were exchanged between the clients and the servers (Line 8). Different sessions can be distinguished by the client IP addresses and ports used in the connection, TCP sequence numbers, temporal gaps between messages, or simply by knowing which messages are used in the initial protocol setup as defined in the original specification.

Since the traces were already used to complement the protocol language, we use the respective message formats that were derived (i.e., the path in the automaton L that accepts the message). Thus, every application session, which is a sequence of messages, is converted into a sequence of message formats (Line 9). Each sequence is fed to the FSM of the original specification and new states and transitions are added whenever the automaton fails to accept the complete session (Line 10). For example, a session composed of messages USER jantunes, PASS xyz, and REST 10 is first converted into the corresponding message formats USER .+, PASS .+, and REST [0-9]+; then, it is fed to the original specification, and all messages are accepted (see Figure 4.1). If the session included a novel message type such as LPTR, then a new transition would be created in the au-

```

1 Function complementStateMachine
2   Input: Automaton  $A$ : Original specification of the protocol ( $A = (Q, \Sigma, \delta, q_0, F)$ )
3           Automaton  $L$ : Complemented language
4           NetworkTraces: Messages of the protocol
5   Output: Automaton  $A'$ : ( $Q, \Sigma, \delta, q_0, F$ )
6
7    $A' \leftarrow$  Automaton  $A$  to be extended
8   foreach Session  $s \in \text{NetworkTraces}$  do
9     .  $\text{Seq}_s \leftarrow$  Sequence of message formats from  $L$  that accepts the sequence of
10      messages of session  $s$ 
11     . Add new path to  $A'$  to accept  $\text{Seq}_s$  // iteratively creates a PTA
12     MinimizeFSM( $A'$ ) // converts PTA to FSM
13
14     // Apply ReverX's merge procedure for the entire automaton.
15     // merge states reached from similar message types
16     foreach  $q \in Q$  do
17       . foreach  $p \in Q$  do
18         . . if  $\exists a \in \Sigma, r, s \in Q : \delta(r, a) = p \wedge \delta(s, a) = q$  then
19           . . . MergeStates( $p, q$ )
20
21     // merge states without a causal relation that share at least one message type
22      $\text{merging} \leftarrow \text{TRUE}$ 
23     while  $\text{merging}$  is TRUE do
24       .  $\text{merging} \leftarrow \text{FALSE}$ 
25       . foreach  $q \in Q$  do
26         . . foreach  $p \in Q$  do
27           . . . // if there is not a casual relation
28           . . . if  $(\exists a, b \in \Sigma : \delta(q, a) = p \wedge \delta(p, b) = q)$  or
29             . . .  $(\forall a, b \in \Sigma : \delta(q, a) \neq p \wedge \delta(p, b) \neq q)$  then
30               . . . . // and if they share at least one message type
31               . . . . if  $\exists a \in \Sigma, r, s \in Q : \delta(q, a) = r \wedge \delta(p, a) = s$  then
32                 . . . . . MergeStates( $p, q$ )
33                 . . . . .  $\text{merging} \leftarrow \text{TRUE}$ 
34       . MinimizeFSM( $A'$ )
35   return  $A'$ 

```

Algorithm 4.4: Complementing the state machine of an existing protocol specification.

tomation so that it could be accepted. The automaton is also minimized because the insertion of new paths potentially creates a prefix-tree acceptor.

Since we are dealing with potentially incomplete data sets (the network traces are a sample of the protocol utilization), the automaton only captures the sequence of messages exactly as they appear in the traces. Cycles and equivalent

states must therefore be inferred. In this work, we use a similar technique to ReverX to identify and merge potentially equivalent states and cycles (Subsection 4.3.2). The resulting automata are the new complemented specification of the protocol state machine. The newly labeled transitions also reveal more clearly the changes brought by the network traces, which can help developers and testers to focus on that part of the specification.

4.4.2 *Experimental evaluation*

For the purpose of evaluation, we applied the method to complement a specification of a well-known protocol [FTP](#), with publicly available network traces that contained message types introduced in subsequent extensions. Since the responses produced by the server are relatively similar to the original specification—it mostly defines reply codes and implementation-specific response messages—we opted to use and complement only the input state machine specification.

A specification was manually produced for the original [FTP](#) protocol standard presented in [RFC 959](#) ([Postel and Reynolds, 1985](#)), which is represented by the [FSM](#) of Figure 4.2. It defines eight states, and the transitions are related to the various commands that can be executed in each state. For example, the first two states (S1 and S2) correspond to the initial authentication process where the client starts by indicating the username with command `USER` and then provides the associated password with command `PASS`. We used the same source of network traces from [LBL](#) utilized in the evaluation of ReverX (Subsection 4.3.4).

A prototype tool was written in Java, extending the original ReverX code base, to implement the method for complementing specifications described in the pre-

Message Types	Introduced in
XCWD, XPWD	RFC 775
LPRT	RFC 1639
FEAT, OPTS	RFC 1839
EPSV, EPRT	RFC 2428
SIZE, MDTM, MLSD	RFC 3659
MACB, CLNT	non-standard
169 illegal requests	N/A

Table 4.3: Discovered message formats and respective RFC extensions.

vious subsection. The tool uses as input the [FSM](#) of the original protocol specification, or a previously inferred specification from ReverX, and the [FTP](#) client requests (i.e., [TCP](#) messages from the traces transmitted to port 21). First, the tool produces a [FSM](#) recognizing the known language of the protocol, which is then extended with the new messages that were not recognized (*Phase 1*). Then, the tool complements the protocol specification using the language inferred previously, placing the new message formats in the corresponding protocol states, as determined by the causal relations observed in the application sessions in the traces (*Phase 2*).

Table 4.3 shows the new types of messages that the tool found in the [FTP](#) traces and the respective [RFC](#) document where they were published. A total of twelve new message types were extracted and their format inferred. Additionally, the tool detected 169 malformed protocol requests that consisted mainly of misspelled command names. To separate these erroneous messages from the rest, we just ignored command names that appeared only once in the traces, effectively preventing these messages from being further used in the experiments¹⁰.

Among the twelve commands, the tool discovered two commands (MACB and

¹⁰Notice that any approach that uses data traces to infer or to learn some model must assume the correctness of its training data, so it is acceptable to ignore these erroneous messages from the evaluation.

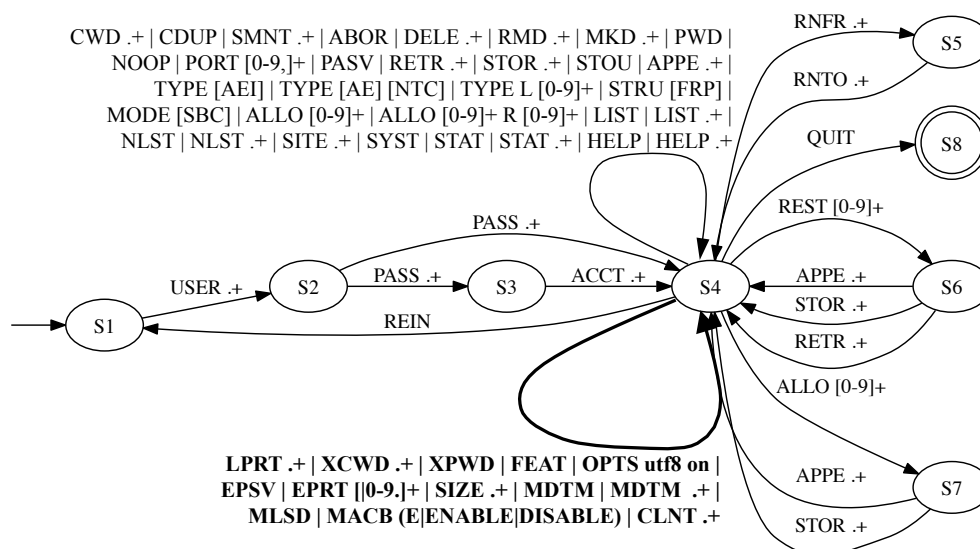


Figure 4.16: FTP input state machine complemented with message types and protocol states from subsequent extensions to the protocol (*in boldface*).

CLNT) that were never published or documented by any RFC extension. MACB command is sometimes used by FTP clients running in the Macintosh OSs (e.g., CuteFTP or WebTen) to transfer files in MacBinary mode, while CLNT refers to an obscure feature of a particular FTP client (NcFTP) apparently used to identify it and to access shell utilities. Little more information is available for these two non-standard commands, as they are not specified by any RFC or other official document. After identifying the new messages, the tool complemented the original specification with the observed extensions (Figure 4.16 shows the complemented specification with changes in boldface). By analyzing the traces, the tool was able to discover the correct state of the protocol where the message formats were specified as extensions, i.e., the protocol state after the user logged in (state S4).

Naturally, the quality of the derived specification for the protocol language and state machine depends on the values of the generalization parameters (*VT* and

CT) and on the comprehensiveness of the network traces, which should cover the protocol extensions one wishes to infer. Accordingly, any message type missing from the traces cannot be extracted, and therefore cannot be used to complement the original specification. This problem can be addressed if one has access to a client and server implementation that supports the new features. In this case, the new functionality of the client can be exercised, thus producing a network trace that covers the entire protocol extensions, allowing the creation of a full protocol specification.

4.5 Conclusions

This chapter presented a new method and a tool to derive the protocol language and state machine from network traces. This approach resorts only to a sample of the regular protocol usage between clients and servers and does not require any access to a protocol implementation or its source code, making it suitable to be used on open or closed protocols.

This approach can produce an input-only protocol specification, which describes the behavior of the server as it processes the requests from the clients, and also a more complete input/output specification, which defines the interaction between clients and servers. Furthermore, by taking advantage of particular characteristics of network protocols, we devised more aggressive and optimistic approaches for both the language and the protocol state machine inference, while still preventing the automata from being over-generalized. Our experimental results have shown that ReverX is able to derive the language and state machine of the FTP protocol, using a sample of 4000 messages in as little as 4.5 seconds for

the input-only protocol specification and 9.9 seconds for the input/output specification.

In addition, a method to complement existing protocol specifications from network traces was also presented. This solution has the advantage of not creating a complete specification from scratch, but by taking advantage of the previously defined (open protocols) or inferred (closed protocols) specifications and from network traces to capture new protocol interactions between the clients and the servers. The approach was implemented in a prototype tool and was evaluated by complementing the standard [FTP](#) specification ([RFC 959](#)) with a trace collected from 320 public [FTP](#) servers. Several protocol extensions and two non-standard [FTP](#) types of requests were discovered and integrated in the [FTP](#) specification.

The proposed method also has the advantage of obtaining a more complete and realistic specification because it integrates the rules and message formats from multiple and different extensions into a single specification. This unified specification captures the realistic utilization of the protocol, including unpublished or undocumented features present in the traces.

CHAPTER 5

ATTACK GENERATION

For a higher level of confidence in the attack injection results, the test cases should exercise the functionality provided by the target system and accessible to potential attackers. To this end, the attack generation method should use the specification of the protocol implemented by the network server to create a set of attacks that is both exhaustive (covering the whole specification) and comprehensive (searching for many classes of vulnerability).

Ideally, the attacks should explore the entire input domain, i.e., check all commands with every possible parameter value, as well as invalid commands with incorrect parameters (negative testing). However, this is impractical since it would require a potentially infinite set of attacks. Even if the set of commands and testing values is finite and/or the number of combinations of values is restricted, the

total number of attacks would still be too high for a viable solution. On the other hand, it has been proven that there exists a finite set of test cases that can reliably determine whether a given program is correct, although obtaining such a set is an undecidable problem (Howden, 1976). We propose two solutions to this problem. Both solutions can resort to malicious payloads, which have been observed in most of the reported network attacks, in order to circumvent the difficulty of generating interesting test cases.

One solution is based on a combinatorial generation of test cases, where different values consisting in legal and malicious payloads are combined. This solution offers several test case generation algorithms with different purposes, such as searching for a particular type of vulnerability (e.g., information disclosure vulnerabilities) or aimed at testing the syntax of the messages (e.g., field order and delimiters).

The second solution recycles test cases from several sources, even if designed for other protocols. It resorts to protocol reverse engineering techniques to build parsers that are capable of extracting the testing payloads from the test cases, which are then applied to the creation of new test cases tailored for the target system.

5.1 Combinatorial Test Case Generation

Attack injection uses a set of attacks to test the correctness of the target system. These attacks are special test cases that look for security vulnerabilities. One of the purposes of attack generation is to create a series of test cases, in a systematic way, that can be injected in the target system. The generation can resort to a

specification of the communication protocol of the server, which in practice characterizes the server's external interface to the clients, to explore the input space of the protocol more efficiently. Therefore, the attacks can exercise much of the intended functionality of the target, i.e., the parts of the code that are executed when processing the clients' requests. Contrarily to source code, which is often inaccessible, communication protocols tend to be reasonably well-documented, at least for standard servers (e.g., the Internet protocols produced by [IETF](#) or [W3C](#)). Consequently, even if the information about a server implementation is scarce, it is still possible to create good test cases as long as there is some knowledge about the communication protocol. Moreover, if the target system implements a closed protocol, an approximate protocol specification could be inferred by using the reverse engineering techniques presented in Chapter 4.

This section describes a series of generation algorithms that create test cases with different characteristics. The algorithms aim to maximize the coverage of the protocol space in order to exercise most of the server's functionality, leading to a better code coverage. However, one should keep in mind that full protocol (or code) coverage does not guarantee the complete and comprehensive discovery of defects. Vulnerabilities may remain dormant if the faulty portion of code is not executed with the required conditions (e.g., a message field data was not large enough to overflow a local buffer) or if the erroneous code is just never reached (e.g., the protocol documentation did not contemplate some non-standard or legacy feature). This problem is in general unsolvable for non-trivial servers, and therefore, our objective should be to locate as many flaws as a remote attacker would, within the time and resources available for testing.

In the following subsections, we provide some details about the set of currently

implemented algorithms, which were developed to accommodate several classes of common errors, such as delimiter, syntax, and field value errors (Beizer, 1990), plus a variation of the latter to look for vulnerabilities related to privileged access violations. This approach was implemented in three attack injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#), which are evaluated in Chapter 7, where they were used to discover vulnerabilities in [POP](#), [IMAP](#), and [DNS](#) servers.

5.1.1 Delimiter test definition

The specification of the protocol implemented by the server determines, among other things, the format of the messages (*protocol language*) and the rules for exchanging them with the clients (*protocol state machine*). It also indicates the expected behavior of the server, such as the responses that should be returned. We define a *test case* as a sequence of network messages that check the implementation of a specific protocol feature (e.g., the correctness of directory traversal or the ability to handle very long file names). The message that actually contains the test is called the *testing message* and is usually a protocol request with some special testing feature that depends on the generation algorithm, such as an invalid field value or a wrong delimiter. If the testing feature is a particular payload as in the value test (e.g., a frontier value or random data), we call it the *testing payload*.

The whole set of test cases should, however, cover as much of the protocol implementation as possible and each test should be executed in the correct protocol state. For this purpose, a sequence of messages, called the *prelude* of the test case, should precede the testing message to take the target server from the initial protocol state to the desired state. The messages used in the prelude must

be carefully constructed. Malformed messages can be rejected by the input validation mechanisms at the server, and therefore it will not go to the intended protocol state. Messages with wrong parameter data, such as invalid credentials or non-existing path names, will also fail to take the server into the desired state. For these reasons, the prelude of messages must comply with the protocol specification and with the configuration of the target server.

Usually, applications are thoroughly tested for the expected functionality, occasionally disregarding their robustness in dealing with malformed messages. The delimiter test generation creates messages with illegal or missing delimiters of a message field (see Algorithm 5.1). For example, in text-based protocols, each field is usually delimited by a *space* character and messages terminate with a **CR** and a **LF** characters. The attack generation algorithm cycles through all types of messages of the protocol and generates messages with variations of their delimiters, such as messages without one of the delimiters or messages with some delimiters replaced by a predefined set of illegal delimiter characters. Note that, with the exception of the delimiters, the generated messages contain only legal data taken from the specification. Moreover, one can use any custom data to experiment as illegal delimiters.

For the sake of simplicity, imagine a simple plain text protocol with two messages: a first login command with the username and the second message with the password. One could test a NULL character (ASCII code 0) or some non-printable set of characters as illegal delimiter in both messages, although a larger set of illegal delimiters should be used in a real life example. This test definition would generate various messages with valid usernames and passwords, but either without the delimiters or with one of the predefined illegal delimiters to test

```

1 Function TestDelimiter
2   Input: Protocol  $\leftarrow$  specification of the network protocol used by the server
3           IllegalDelimiters  $\leftarrow$  predefined list of illegal delimiters
4   Output: Attacks
5
6   S  $\leftarrow$  set of all states of the Protocol specification
7   foreach State s  $\in S$  do
8     . M  $\leftarrow$  set of message specifications of s
9     . Preludes  $\leftarrow$  set of ordered network packets necessary to reach s
10    . P  $\leftarrow$  empty_set
11    .
12    . foreach MessageSpecification m  $\in M$  do
13    . . foreach FieldSpecification f  $\in m$  do
14    . . .
15    . . . // remove field delimiters
16    . . . m'1  $\leftarrow$  copy of m removing f's initial field delimiter
17    . . . m'2  $\leftarrow$  copy of m removing f's final field delimiter
18    . . . P1  $\leftarrow$  set of all network packets based on m'1 specification
19    . . . P2  $\leftarrow$  set of all network packets based on m'2 specification
20    . . . P  $\leftarrow P \cup P_1 \cup P_2$ 
21    . . .
22    . . . // replace field delimiters with illegal delimiters
23    . . . foreach d  $\in$  IllegalDelimiters do
24    . . . . m'3  $\leftarrow$  copy of m replacing f's initial field delimiter with d
25    . . . . m'4  $\leftarrow$  copy of m replacing f's final field delimiter with d
26    . . . . P3  $\leftarrow$  set of all network packets based on m'3 specification
27    . . . . P4  $\leftarrow$  set of all network packets based on m'4 specification
28    . . . . P  $\leftarrow P \cup P_1 \cup P_2$ 
29    .
30    . foreach attack_packet  $\in P$  do
31    . . attack  $\leftarrow$  Preludes  $\cup$  {attack_packet}
32    . . Attacks  $\leftarrow$  Attacks  $\cup$  {attack}
33    .
34  return Attacks

```

Algorithm 5.1: Algorithm for the Delimiter Test generation.

(i.e., the NULL character or the non-printable characters). In addition, the test cases with the password message as the testing message, contain a prelude that is composed of a valid username message (with the correct delimiter).

```

1 Function TestSyntax
2   Input: Protocol  $\leftarrow$  specification of the network protocol used by the server
3   Output: Attacks
4
5    $S \leftarrow$  set of all states of the Protocol specification
6   foreach State  $s \in S$  do
7     .  $M \leftarrow$  set of message specifications of  $s$ 
8     .  $Prelude_s \leftarrow$  set of ordered network packets necessary to reach  $s$ 
9     .  $P \leftarrow$  empty_set
10    .
11    . foreach MessageSpecification  $m \in M$  do
12    . . foreach FieldSpecification  $f \in m$  do
13    . . . // remove and repeat fields
14    . . .  $m'_1 \leftarrow$  copy of  $m$  without  $f$ 
15    . . .  $m'_2 \leftarrow$  copy of  $m$  with  $f$  repeated
16    . . .  $P_1 \leftarrow$  set of all network packets based on  $m'_1$  specification
17    . . .  $P_2 \leftarrow$  set of all network packets based on  $m'_2$  specification
18    . . .  $P \leftarrow P \cup P_1 \cup P_2$ 
19    . . .
20    . . . // swapped fields
21    . . . foreach FieldSpecification  $g \neq f : g \in m$  do
22    . . . .  $m'_3 \leftarrow$  copy of  $m$  with  $f$  and  $g$  swapped
23    . . . .  $P_3 \leftarrow$  set of all network packets based on  $m'_3$  specification
24    . . . .  $P \leftarrow P \cup P_3$ 
25    .
26    . foreach attack_packet  $\in P$  do
27    . . attack  $\leftarrow Prelude_s \cup \{attack\_packet\}$ 
28    . . Attacks  $\leftarrow Attacks \cup \{attack\}$ 
29    .
30  return Attacks

```

Algorithm 5.2: Algorithm for the Syntax Test generation.

5.1.2 Syntax test definition

This kind of test generates attacks that infringe the syntax of the protocol. The currently implemented syntax violations consist on the addition, elimination, or re-ordering of each field of a correct message (see Algorithm 5.2). Note that as the previous algorithm, the message fields are tried with valid values. Like all other test definitions, after generating new message specifications (i.e., variations from the original messages), each will result in many test cases with different

combinations of testing features.

As an example, consider a message containing two different fields (e.g., a command with one parameter) represented as [A] [B]. Below are depicted some of the variations of the original message specification from which test cases are going to be created:

- [A] (removed field [B]);
- [B] (removed field [A]);
- [A] [A] (duplicated field [A]);
- [B] [B] (duplicated field [B]);
- [B] [A] (swapped fields);

5.1.3 Value test definition

This test determines if the server can cope with messages with bad field data, which we call the testing payloads. For this purpose, a mechanism is used to derive illegal data from the message specification, in particular from each field's data domain. Ideally, one would like to experiment all possible illegal values, however, this proves to be unfeasible when dealing with a large number of messages and fields with arbitrary content. To overcome such impossibility, a heuristic method was conceived to reduce the number values that have to be tried (see Algorithm 5.3). The algorithm has the following structure: all states and message types of the protocol are traversed, maximizing the protocol space; then test cases are generated based on one message type at a time. This algorithm differs from the others because it systematically populates each field with wrong values, instead of only resorting to the legal values.

```

1 Function TestValue
2   Input: Protocol  $\leftarrow$  specification of the network protocol used by the server
3   Output: Attacks
4
5    $S \leftarrow$  set of all states of the Protocol specification
6   foreach State  $s \in S$  do
7     .  $M \leftarrow$  set of message specifications of  $s$ 
8     .  $Transition_s \leftarrow$  set of ordered network packets necessary to reach  $s$ 
9     .  $P \leftarrow$  empty_set
10    .
11    . foreach MessageSpecification  $m \in M$  do
12    . . foreach FieldSpecification  $f \in m$  do
13    . . . if  $f$  is type Numbers then
14    . . . .  $f' \leftarrow$  all boundary values, plus some intermediary illegal values, from
14    . . . .  $f$  specification
15    . . . elseif  $f$  is type Words then
16    . . . .  $f' \leftarrow$  combin. of predefined malicious tokens
17    . . . .
18    . . .  $m' \leftarrow$  copy of  $m$  replacing  $f$  with  $f'$ 
19    . . .  $P \leftarrow P \cup \{\text{set of all network packets based on } m' \text{ specification}\}$ 
20    .
21    . foreach attack_packet  $\in P$  do
22    . .  $attack \leftarrow Transition_s \cup \{attack\_packet\}$ 
23    . .  $Attacks \leftarrow Attacks \cup \{attack\}$ 
24    .
25  return Attacks

```

Algorithm 5.3: Algorithm for the Value Test generation.

In the case of a field that holds a number, deriving illegal values is rather simple because they correspond to the complementary domain, i.e., the numbers that do not belong to the legal data set. Additionally, to decrease the number of values that are tried, and therefore to optimize the injection process, this attack generation algorithm employs boundary values and a limited subset of the complementary values. The current implementation of the illegal data generation uses two parameters to select the complementary values, an *illegal coverage ratio* and a *random coverage ratio*. The illegal coverage ratio chooses equally distant values based on the total range of illegal values. On the other hand, the random coverage ratio selects the illegal values arbitrarily, from the domain of

```

1 Function generateIllegalWords
2   Input: Words  $\leftarrow$  specification of the field
3           Tokens  $\leftarrow$  predefined list of malicious tokens, e.g., taken from hacker exploits
4           Payload  $\leftarrow$  predefined list of special tokens to fill in the malicious tokens
5           max_combinations  $\leftarrow$  maximum number of token combinations
6   Output: IllegalWords
7
8   // step 1: expand list of tokens
9   foreach  $t \in Tokens$  do
10    if  $t$  includes keyword $(PAYLOAD) then
11      . foreach  $p \in Payload$  do
12      .   .  $t' \leftarrow$  copy of  $t$  replacing $(PAYLOAD) with  $p$ 
13      .   .  $Tokens \leftarrow Tokens \cup \{t'\}$ 
14      .  $Tokens \leftarrow Tokens \setminus \{t\}$ 
15      .
16   // step 2: generate and append all  $k$ -combinations of tokens
17    $k \leftarrow 1$ 
18   while  $k \leq max\_combinations$ 
19     .  $k\text{-combinations} \leftarrow \binom{Tokens}{k}$  // combinations of  $k$  elements from Tokens
20     .  $IllegalWords \leftarrow IllegalWords \cup k\text{-combinations}$ 
21     .  $k \leftarrow k + 1$ 
22     .
23   return IllegalWords

```

Algorithm 5.4: Algorithm for the generation of malicious strings.

illegal numbers. For instance, if there are only 100 illegal numbers, from 0 to 99, a 10% illegal coverage ratio would add numbers 5, 15, 25, etc., whereas the same ratio of random coverage would select 10 random numbers from 0 to 99. This simple procedure allows most illegal data to be evenly distributed, while still keeping a reduced set of test cases.

Creating testing payloads for textual fields, however, is a much more complex problem because there is an infinite number of character combinations, making such an exhaustive approach impossible. Our objective is to design a method to derive potentially desirable testing payloads, i.e., values that are usually seen in exploits, such as large strings or strange characters (see Algorithm 5.4, which is called in the underlined line of Algorithm 5.3). Basically, this method produces

payloads by combining several tokens taken from two special input files. One file holds malicious tokens or known expressions, collected from the exploit community, previously defined by the operator of the tool (see examples in Listings 7.1 and 7.2 from Section 7.1). The algorithm expands the special keyword \$(PAYLOAD) with each line taken from another file with payload data. This payload file could be populated with already generated random data, long strings, strange characters, known usernames, and so on. The resulting data combinations from both files are used to define the illegal word fields.

As an example, here are a few attacks generated by this method, and that were used to detect known IMAP vulnerabilities (<A × 10>, means character 'A' repeated ten times):

- A01 AUTHENTICATE <A × 1296>
- <%s × 10>
- A01 LIST INBOX <%s × 10>

5.1.4 Privileged access violation test definition

This algorithm produces tests to determine if the remote server allows unauthorized accesses to its resources. The existing implementation is a specialization of the Value Test Definition that employs very specific testing payloads related to existing resources, such as known usernames or path names to existing files. These testing payloads, when applied to messages related to privileged operations, result in attacks that evaluate the access control mechanisms of the network server. If the server is authorizing any of these operations, such as disclosing private information, granting access to a known file, or writing to a specific disk location that should not have been allowed, it reveals that the server holds some access

violation vulnerability. Further investigation on the collected data, including the server replies, and its configuration, throws light into the cause of the problem, whether it is due to misconfiguration, bad design, or some software bug.

To keep the number of test cases manageable, a special focus must be taken on the selection of the tokens. Examples of good malicious tokens are directory path names (relative and absolute), well-known filenames, or existing usernames, which can also be combined with payload tokens such as `/`, `../`, or `..`. The created test cases can stress the server with illegal requests, such as reading the contents of the `../.../etc/passwd` file, accessing other user's data, or just writing to an arbitrary disk location. Here are a few examples of generated IMAP test cases that were able to trigger some vulnerabilities in previous experiments:

- `A01 CREATE /<A × 1000>`
- `A01 SELECT ../../../other-user/inbox`
- `A01 SELECT "{localhost/user=\""`

5.1.5 Experimental evaluation

The experimental evaluation of this approach is provided in Chapter 7 where two attack injection tools, [AJECT](#) and [PREDATOR](#), were used to discover vulnerabilities in various network servers. Overall, the results revealed that the delimiter and syntax test algorithms create test cases that are too simple to discover vulnerabilities in mature network servers, and are only useful during the initial software development phases. On the other hand, the value test algorithm (the privileged access violation test algorithm is actually a specialization of the former) was very successful in detecting different classes of vulnerabilities in e-mail and domain name servers.

5.2 Recycling-based Test Case Generation

Over the years, several black-box approaches have been devised to discover security vulnerabilities in software, namely in network facing components such as servers. For example, scanners resort to a database of checking modules to automate the task of identifying the presence of known vulnerabilities ([Tenable Network Security, 2002–12a](#); [Rapid7, 2012](#)). They typically start by determining the type and version of a target server running in a remote machine, and then they perform a pre-defined message exchange to find out if a certain vulnerability exists. Other tools such as fuzzers automatically generate many test cases that include unexpected data, which may cause incorrect behavior in the component (e.g., a crash), allowing the detection of a flaw.

When they started, fuzzers mainly used random data to evaluate the component's robustness ([Miller et al., 1990](#)), but eventually evolved to include more knowledge. Servers are usually protected with a set of validation routines that decide on the correctness of an interaction, letting the computation progress only when the appropriate conditions are present (e.g., non-compliant protocol messages are immediately discarded). Therefore, specialized fuzzers were developed, which understand the component interfaces, and are able to generate valid interactions carrying malicious payloads ([Codenomicon, 2012](#); [Infigo Information Security, 2012](#); [Rapid7, 2012](#); [AutoSec Tools, 2012](#); [Navarrete and Hernandez, 2012](#)). The downside of these approaches is that as they become increasingly knowledgeable about the target component, and therefore more effective at finding vulnerabilities, they lose generality and become useless to assess other types of servers.

To address this issue, we describe an approach that recycles existing test cases, making it particularly useful in the following scenarios. First, servers implementing newly designed protocols are typically hard to test because of lack of support from the available tools. For instance, the [IETF](#) publishes a few hundred new [RFCs](#) every year, some of them introducing novel protocols ([Internet Engineering Task Force, 2012](#)). This fast-paced standard deployment would require a constant upgrade of the testing tools, but what happens in practice is that tools typically lag behind. Second, many application areas are very dynamic, which is usually reflected in protocol specifications that are constantly being updated. For instance, from the publication of the current [FTP](#) protocol standard ([Postel and Reynolds, 1985](#)), the [IETF](#) has published 42 related [RFCs](#). When this happens, old test cases are unable to check the new features, and extensions to the protocol may even render them ineffective. Last, servers implementing proprietary (or closed) protocols are difficult to assess because little information is available, and therefore, they end up being evaluated only by their own developers.

Our approach takes test cases available for a certain protocol and re-uses them to evaluate other protocols or features. It resorts to protocol reverse engineering techniques to build parsers that are capable of extracting the relevant payloads from the test cases, i.e., the malicious parameter data of the messages. These payloads are then applied in new test cases for the target server, by generating malicious interactions that cover the particular features under test. In case the whole server needs to be checked, tests are created to explore the entire state space of the implemented protocol.

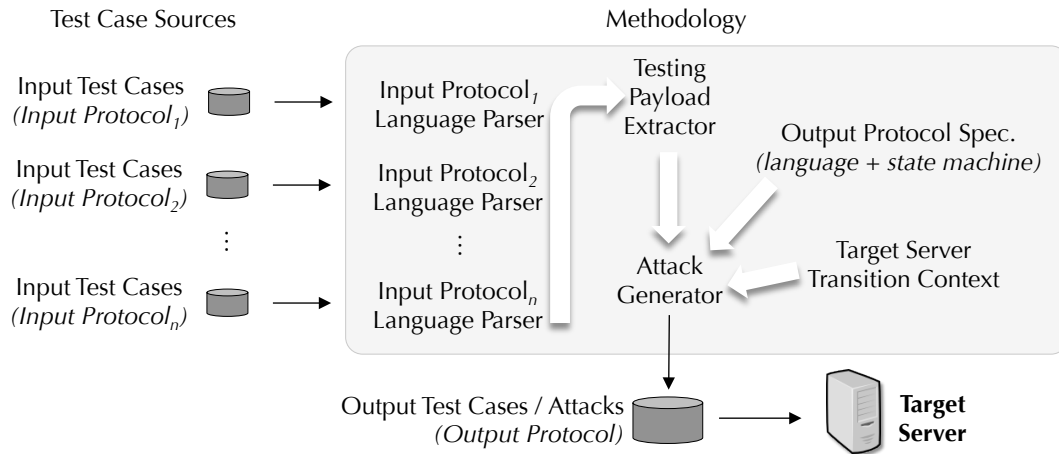


Figure 5.1: Recycling-based test case generation approach.

5.2.1 Overview of the approach

The goal of our approach is to produce attacks in an automated way for a particular protocol implementation, i.e., the target server. The solution resorts to test cases from different sources to maximize the richness of the testing payloads, together with a specification of the protocol implemented by the target server to increase the overall coverage of the generated attacks. This allows testing payloads for a particular protocol to be re-used in other kinds of servers, as well as the assessment of the added features to newer releases of a server.

The main steps of this approach are summarized in Figure 5.1. The test case generation approach resorts to existing test cases obtained from several sources, potentially for diverse protocols, in order to get a rich set of testing payloads. To parse and extract the payloads of the messages included in the input test cases, we employ input protocol language parsers that recognize the formats of the messages. The obtained payloads are forwarded to the testing payload extractor to be organized and selected for the next phase. An output protocol specification

provides the definition of the protocol that the target server implements. This specification gives the ability to construct messages according to the recognized formats and state machine of the protocol. To make the target server go from the initial protocol state to another state requires that a sequence of messages of the correct type and with the correct payload is transmitted—the prelude of the test case. Hence, a target server transition context provides the necessary information about the server’s configuration and execution to create protocol messages with the correct parameter values (e.g., valid credentials and/or file system structure) so that it can reach any protocol state.

The attack generator produces the output test cases to be sent to the target server. It generates attacks that cover the whole protocol space, containing the extracted testing payloads. To exhaustively test each protocol feature (or type of request), several approaches can be employed, such as using all combinations of the testing payloads in each message field or pairwise testing. Therefore, various strategies can be implemented in the generator, which tradeoff the time and resources with the completeness and coverage of the testing.

Table 5.1 shows some examples of test cases created for three different protocol implementations. The test cases were obtained from a fuzzer tool (Bruteforce Exploit Detector) and executed in a [POP](#) and [SMTP](#) server (Microsoft Exchange Server) and in a [FTP](#) server (Microsoft IIS). The messages exchanged between the fuzzer and the servers were collected. A subset of this messages is presented in the first column of the table. The second column displays the message fields as parsed by the input protocol language parsers. For instance, the language parser for the [POP](#) protocol recognizes the first request as a message with two fields, a

	Input test cases	Parsed requests
POP	#1 Req: USER AAAAAA1 Resp: +OK Req: QUIT Resp: +OK	C: USER; P: AAAAAA1 C: QUIT
	#2 Req: USER AAAAAA2 Resp: +OK Req: QUIT Resp: +OK	C: USER; P: AAAAAA2 C: QUIT
	...	
SMTP	#2560 Req: HELO Resp: 220 server.testing Microsoft ES (...) Req: MAIL FROM: BBBBBB1 Resp: 250 2.1.0 BBBBBB1@testing....Se (...) Req: QUIT Resp: 221 2.0.0 server.testing Servic (...)	C: HELLO C: MAIL FROM;; P: BBBBBB1 C: QUIT
	...	
	#5431 Req: USER CCCCCC1 Resp: 331 Password required for CCCCCC1. Req: QUIT Resp: 221 Goodbye.	C: USER; P: CCCCCC1 C: QUIT
FTP	...	
	#5828 Req: USER peter Resp: 331 Password required for peter. Req: PASS spiderman Resp: 230 User logged in. Req: CWD CCCCCC2 Resp: 550 CCCCCC2: Access is denied. Req: QUIT Resp: 221 Goodbye.	C: USER; P: peter C: PASS; P: spiderman C: USER; P: CCCCCC2 C: QUIT

Extracted Payloads = {AAAAAA1, AAAAAA2, BBBBBB1, CCCCCC1, peter, spiderman, CCCCCC2}

Table 5.1: Example of a set of test cases and the respective extracted payloads by three input protocol language parsers (*C: Command; P: Parameter*).

USER command and the respective parameter payload of AAAAAA1¹.

¹For this example in particular, we configured the fuzzer to use the strings AAAAAA1, AAAAAA2, BBBBBB1, CCCCCC1, and CCCCCC2 in its tests.

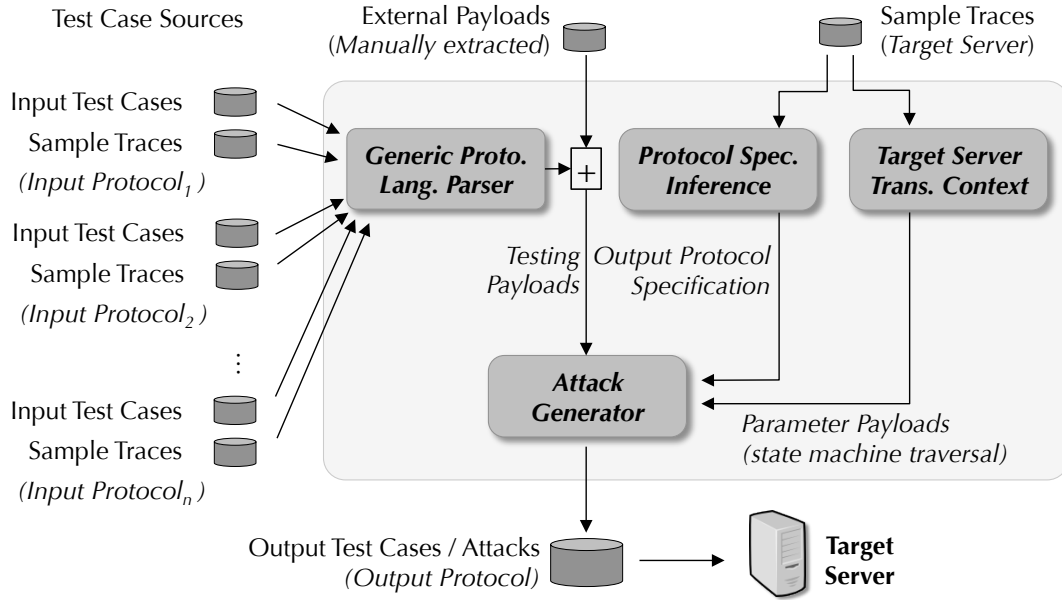


Figure 5.2: Architecture of the implemented tool.

5.2.2 Tool implementation

This section provides details about an implementation of the recycling-based test case generation approach. The general architecture of the tool is depicted in Figure 5.2. The implementation does not require individual protocol language parsers and a protocol specification to be provided, since it resorts to additional samples of network traces and protocol reverse engineering techniques to infer the language of the protocols.

Generic protocol language parser

The tool implements a *generic protocol language parser* component to infer input protocol language parsers from additional samples of network traces. The additional network traces can be collected from custom network servers that implement the protocol used in the input test cases, or downloaded from the Internet

as packet capture files. They only need to include the messages of regular client-server interactions. Then, a reverse engineering method is employed to derive the message formats (or the language) of the protocols.

The current implementation is based on the reverse engineering techniques developed in ReverX for text-based protocols (Section 4.3). Messages in text-based protocols are usually composed of a sequence of text fields. Some fields hold a limited range of predefined command names, which discriminate the type of the protocol request. Following the field with the command name, the message usually has one or more parameter fields that further refine the function of the protocol request. Based on this idea, reverse engineering divides the messages of the trace in their respective fields, and then builds a [PTA](#) that recognizes all messages. The [PTA](#) is minimized to obtain a more condensed representation, which corresponds a [FSM](#) automaton. Next, a statistical analysis is performed on the automaton to identify the command and parameter fields. The parameters are generalized as a regular expression, to produce a new automaton that is capable of accepting other messages of the protocol that are not present in the traces. This automaton is, however, non-deterministic, and therefore a determinization and a minimization procedure are performed to create the final automaton, which understands the formats of the messages.

Naturally, the sample traces that are used to infer the language automata must be sufficient and representative enough. Otherwise, the automata may be incomplete and fail to recognize some of the protocol messages of the test cases. To guarantee that the test cases are fully supported, our tool simply adds the messages of the test cases to the sample traces during construction of the input parser.

To extract the testing payloads, the tool feeds the test cases to the language

parser and extracts the parameter data, i.e., the content of the fields that are identified as parameters. Normally, the most interesting payloads are those found in the last message of the test cases (the preceding messages act as prelude to take the server to the designated state). However, it happens sometimes that the fault is triggered by a message in the middle or even the beginning of the test case. Therefore, it is preferable to collect the payloads from all messages, than risking neglecting payloads that could detect vulnerabilities.

Protocol specification inference

The tool resorts to the same techniques on protocol reverse engineering used in the generic parsers to infer the output protocol specification from a sample of network traces from the target server. As before, the traces only need to have normal protocol sessions.

This reverse engineering process derives the language of the protocol, and in addition infers the state machine of the protocol. The state machine is obtained by identifying the implicit causal relations among the various types of messages as observed in the traces. Naturally, parts of the protocol that are missing from the traces cannot be inferred, and therefore will not be contemplated in the attack generation. The client sessions must therefore exercise all the functionality that one wants to test. This approach supports the creation of attacks for new protocol extensions that may be missing from the standard specifications, simply by including in the traces the client-server messages related to those features. The server implementing the custom modifications (or the features still under development) only needs to be experimented with a client and the messages corresponding to the new extensions have to be captured.

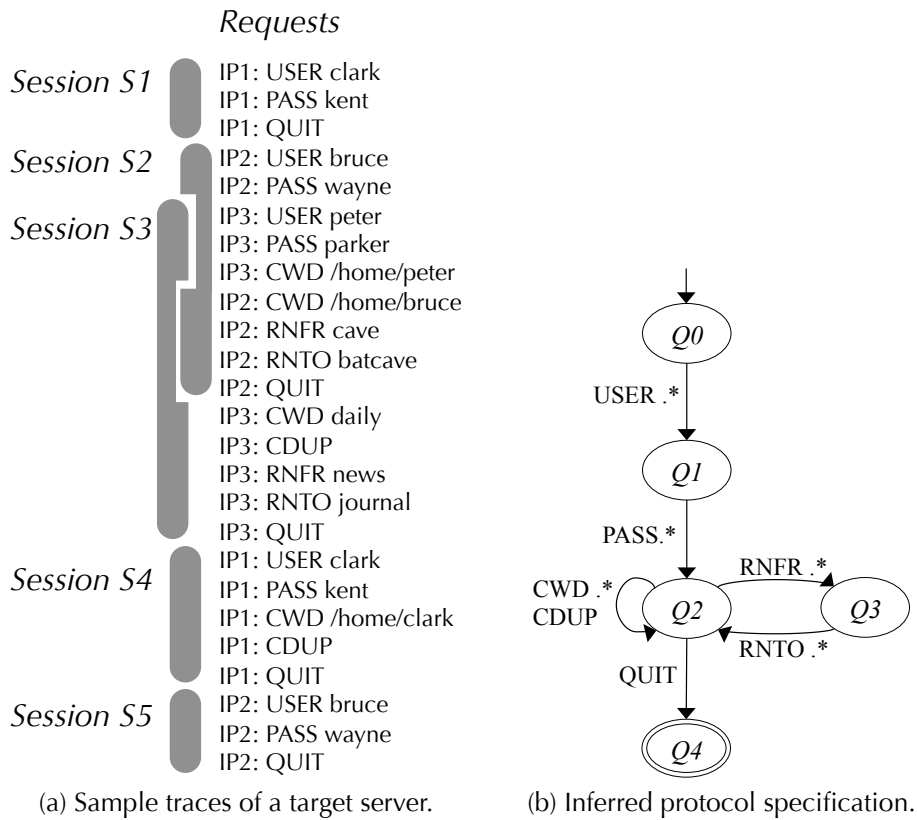


Figure 5.3: Reverse engineering the protocol specification.

Figure 5.3 shows the inferred protocol specification from a small traces containing five [FTP](#) sessions. The reverse engineering process starts by deriving the language automaton, where each path corresponds to a unique message format. Seven distinct [FTP](#) types of requests were identified—CDUP and QUIT with no parameters, and USER, PASS, CWD, RNFR and RNTD each with one parameter. Then, the protocol state machine is obtained by converting each client session into a sequence of message formats. Next, an automaton is built to accept the various sequences of messages of each session and to represent any implicit causal relation between message formats. For example, CWD and CDUP messages are used interchangeably after the PASS messages, so they are merged into

the same protocol state ($Q2$). On the other hand, since PASS messages always follow USER messages (and not the inverse), and the causal relation is captured with three states ($Q0 \rightarrow Q1 \rightarrow Q2$).

Target server transition context

The sample traces from the target server contain messages from the clients that make the server go from the initial state to other protocol states. The information included in these messages consists of valid usernames and passwords, and other parameter data that are specific to the target server. Examples of these parameter values are `clark` and `bruce` in USER messages and `kent` and `wayne` in PASS messages (see Figure 5.3a). The target server transition context component can take advantage of this data to obtain the necessary information about the server's configuration.

The component implements a table that correlates the parameter values in the sample traces with the previously inferred output protocol specification. The table is built by parsing each message that causes the protocol to make a state transition and extracting the values of each field. Table 5.2a shows an example for the traces of Figure 5.3a and using derived protocol specification of Figure 5.3b. The first two columns have the source and destination states; the third column depicts the message format from the language automaton (here abbreviated as the command name); and the forth column contains the list of observed parameter values and their respective sessions.

The table is then used to build the preludes of the test cases, i.e., the sequences of protocol messages that take the target server from the initial state to any other protocol state. The first step to create a message sequence is to select the shortest

From	To	Message type	Parameter values (from session)
Q0	Q1	USER	clark (S1,S4); bruce (S2,S5); peter (S3)
Q1	Q2	PASS	kent (S1,S4); wayne (S2,S5); parker (S3)
Q2	Q3	RNFR	cave (S2); news (S3)
Q3	Q2	RNTO	batcave (S2); journal (S3)
Q2	Q4	QUIT	none

(a) Target server transition table.

Dest. state	Preludes (from session)
Q0	none (initial state)
Q1	USER clark (S1)
Q2	USER clark; PASS kent (S1)
Q3	USER bruce; PASS wayne; RNFR cave (S2)
Q4	USER clark; PASS kent; QUIT (S1)

(b) Generated prelude to test cases.

Table 5.2: Target server transition table and prelude generation.

path in the protocol state machine from the initial state to the desired state. For instance, the smallest sequence of messages to reach state Q3 would be: USER . *, PASS . *, and RNFR . *. The second step is to select the correct parameter values, which cannot come from different sessions because this could take the server to an unexpected protocol state. Therefore, the table is searched to find which sessions reach the desired protocol state. When more than one session is found, the one that reaches the desired state with a smaller number of messages is chosen. For instance, session S2 is selected to go to state Q3 (S3 also reaches Q3, but only after the six messages). Then, the parameter values of the respective messages are used. In the example, the prelude would be: USER bruce, PASS wayne and RNFR cave. Table 5.2b shows the generated preludes, allowing each one of the protocol states to be reached.

Attack generator

The attack generator component aims at creating a set of test cases that covers the entire protocol space. So, it exhaustively creates tests for all states and transitions of the output protocol specification. Each test case is composed of a prelude of messages plus a testing message.

The generation iteratively selects a protocol state and one of its transitions. To reach the selected protocol state, the component uses the previously obtained prelude of messages. The transition is associated to a message type, so the generator uses the inferred message format for the creation of a testing message. The testing message consists in a variation of the message format with a special payload—one of the extracted testing payloads. Therefore, a large number of attacks can be produced for a single protocol state and transition. The existing implementation resorts to single parameter testing, i.e., only one of the parameters has a testing payload, which is the normal scheme used by most test case generation tools (e.g., fuzzers). Naturally, other techniques, such as pairwise testing, exhaustive testing or random testing, could also be programmed.

Table 5.3 presents a subset of the generated test cases for the example of Figure 5.3. The test cases were created for the entire inferred protocol using extracted testing payloads (e.g., AAAAAA1 or CCCCCC2). The tool starts by generating test cases for the initial state, which only accepts one transition (or message type) `USER . *`. Since this is an initial state, the prelude for these attacks is empty, and therefore they consist of only the testing message (e.g., `USER AAAAAA1`). As this message only has one parameter, the number of test cases that is produced is equal to the number of different testing payloads.

Dest. state	Prelude	Transition	Generated test case
Q_0	none (<i>initial state</i>)	$USER . *$	#1 USER AAAAAA1 #2 USER AAAAAA2 ...
Q_1	USER clark	$PASS . *$	#8 USER clark PASS AAAAAA1 #9 USER clark PASS AAAAAA2 ...
Q_2	USER clark, PASS kent	$CDUP$	none (<i>no parameters</i>)
		$CWD . *$	#15 USER clark PASS kent CWD AAAAAA1 ...
		$RNFR . *$	#22 USER clark PASS kent RNFR AAAAAA1 ...
		$QUIT$	none (<i>no parameters</i>)
Q_3	USER bruce, PASS wayne, RNFR cave	$RNTO . *$	#29 USER bruce PASS wayne RNFR cave RNTO AAAAAA1 ...
			#35 USER bruce PASS wayne RNFR cave RNTO CCCCCC2
Q_4	USER clark, PASS kent, QUIT	none (<i>final state</i>)	none (<i>no transition</i>)

Table 5.3: Test case generation example.

Test cases for transitions of non-initial states require a prelude of messages. For instance, to test the $RNTO . *$ transition, the generator would first get the corresponding prelude (i.e., USER bruce, PASS wayne and RNFR cave), and then generate several variations of the $RNTO . *$ message, each one with a different testing payload (e.g., test cases #29 and #35).

Optionally, the testing payloads could be used as testing messages to be sent at every protocol state, such as sending AAAAAA1 in states *Q0-4*. Another extension could be to generate attacks where the command field could hold the testing payloads as well, such as creating the testing message AAAAAA1 bruce to test the *USER . ** transition.

5.2.3 *Experimental evaluation*

This section presents the results of the evaluation of our approach by analyzing and comparing it with several commercial and open source fuzzers and vulnerability scanners. The experiments focus on the *FTP* protocol ([Postel and Reynolds, 1985](#)) because it is a well-known protocol with a reasonable level of complexity. Furthermore, we were able to create a large database with 131 *FTP* vulnerabilities, which supports the study on test case coverage.

We conducted three experiments. In the first experiment we wanted to evaluate the coverage and quality of the generated attacks, i.e., how much of the protocol space is tested and how many vulnerabilities can be (potentially) caught. For this experiment, we used ten *FTP* test case generation tools with varying levels of *FTP* coverage and types of payloads. In the second experiment we evaluate how our approach, using test cases designed for other protocols, compares against specialized *FTP* testing tools. We used ten non-*FTP* test case generation tools and compared the test cases generated by our approach with the ones produced by the *FTP* testing tools from the previous experiment. Finally, we conducted a detailed case study to compare two particular testing tools with our approach.

Type of tool	Testing tool
Specialized FTP fuzzers	Codenomicon Defensics FTP test suite 10.0
	Infigo FTPStress Fuzzer 1.0
Framework fuzzers (FTP setup)	Metasploit Framework (FTP Simple Fuzzer) 4.0
	FuzzTalk Fuzzing Framework 1.0
	Bruteforce Exploit Detector (BED) 5.0
	DotDotPwn 2.1 fuzzer.py 1.1
Vulnerability scanners (FTP test cases)	Nessus 4.4.1
	OpenVAS 4
	NeXpose (Community Edition) 2011

Table 5.4: Test case generation tools for FTP protocol.

Protocol Space Coverage

Table 5.4 shows the test case generation tools used in the experiments. We chose different types of tools that have varying levels of protocol coverage and payloads: two commercial fuzzers specialized in FTP, five fuzzer frameworks that support FTP out-of-the-box, and three vulnerability scanners with the most up-to-date versions of their vulnerability databases (which included some test cases for known FTP vulnerabilities). All tools were setup to be as complete and thorough as possible, without modifying the default configuration too much (only some minor options that were specific to the server configuration were changed, such as the network address and login credentials).

We analyzed the protocol space coverage achieved by the tools by observing which FTP commands were tested. Columns “Basic FTP” and “Full FTP” of Table 5.5 indicate the percentage of the commands that were tried by the tools when compared to the basic standard RFC 959 and the complete specification (with all known non-obsolete extensions defined in nine extra RFCs), respectively. With regard to “Basic FTP”, the specialized FTP fuzzers have a high command

	Basic FTP (RFC959)	Full FTP (10 RFCs)	State machine	Total test cases
Codonomicon	97%	61%	67%	71658
Infigo	86%	66%	17%	292644
Metasploit	94%	72%	17%	3125
FuzzTalk	91%	80%	17%	257892
BED	69%	44%	17%	24918
DotDotPwn	6%	3%	17%	28244
fuzzer.py	63%	44%	17%	197616
Nessus	20%	13%	17%	148
OpenVAS	14%	8%	17%	53
NeXpose	23%	14%	17%	54
Our approach (w/ traces from LBL)	97%	92%	67%	1463151

Table 5.5: Coverage of the FTP protocol space.

coverage because they are focused on generating test cases for a single protocol. The vulnerability scanners, on the other hand, have a very low coverage because they create test cases designed to confirm the existence of vulnerabilities that are known to exist in specific versions of [FTP](#) servers. These test cases thus contain very precise testing payloads, addressing only a small part of protocol space.

Although a tool may have a good basic coverage of the standard, it is important to test the full command space, including protocol extensions and even non-standard commands. This less used part of the protocol is also often less tested, and therefore its implementation might be more prone to vulnerabilities. In effect, Codonomicon’s fuzzer, which has the highest coverage of the [FTP](#) basic standard (97%), lacks support for many of the extensions (it has 61% of overall protocol coverage). The coverage of the fuzzer frameworks depends on how comprehensive is the setup or template for testing a specific protocol. In our study, FuzzTalk and Metasploit have the highest coverage with 80% and 72%, which implies that 20% or more of the commands are left without being checked. DotDotPwn has

a lower protocol coverage because it is designed to search for directory traversal vulnerabilities, and therefore, only path traversal commands are tried (e.g., CWD and RETR). Nevertheless, as we will see in the case study, DotDotPwn produces payloads that are useful to detect other types of vulnerabilities. We also found that none of the tools tested other parameters of the commands, besides the first one (e.g., TYPE A, TYPE E, HELP USER), thus ignoring these parts of the implementation.

We analyzed the coverage of the protocol state machine achieved by the tools, i.e., if they test the commands in their correct state (e.g., trying a RNT0 command after sending a RNFR command). Column “State machine” of the table depicts, for the basic [RFC 959](#) specification, the percentage of transitions that change the state of the protocol that are checked by the tools. A value of 17%, for instance, reveals that most tools only support the USER → PASS sequence in their test cases. Codenomicon’s fuzzer is the only tool that experiments some additional protocol transitions, such as RNFR → RNT0, REST → STOR, and REST → RETR. Even though testing the commands in the wrong state is useful to discover certain kinds of flaws, they should also be tried in their correct state. Otherwise, simple validation mechanisms implemented at the server (e.g., that discard arriving commands unexpected for the current state) could prevent the processing of the malicious payloads, rendering the test cases ineffective in practice.

The last row of Table 5.5 outlines the results of our approach. In these experiments we used a subset of the same network trace from [LBL](#) used in the experiments in Subsection 4.3.4². Our tool does not require such a large amount

²The full trace was obtained at [LBL](#) and contains more than 3.2 million packets exchanged between 320 public [FTP](#) servers and thousands of clients.

of messages to infer a protocol specification, so we randomly selected a 5 hour period from the trace. Next, we added a trace with the messages collected while the ten tools of Table 5.4 tested the FTP server. This second trace provides the messages with the execution context of the server and also complements the LBL trace with unusual message types, increasing the diversity of the exchanges. The complete trace was then used to infer a specification of the FTP protocol and to obtain the transition context of the target server.

The set of test cases generated by our tool covers 97% and 92% of the overall command space for the basic and full FTP specification, which is equal or better than the results achieved by any of the other tools. The second best coverage for the full standard, for instance, was obtained by FuzzTalk with 80% of the overall protocol space. With respect to the state machine coverage, our solution also matches the best coverage of 67%. These results are particularly interesting because our specification is automatically generated from network traces, while the others are manually programmed by experts in the protocol being tested. The downside of our better protocol coverage is a larger number of test cases (column “Total test cases”). For this experiment, the tool generated over one million attacks as a consequence of the combination of several testing payloads with a more complete protocol coverage.

The overall trace used by our tool contained the most used types of FTP requests as well as messages defined in protocol extensions. An analysis on the command coverage of the trace showed that the LBL trace only covered 45% of the full command space, which was complemented with the messages from the tools’ test cases in the second trace, thus resulting in the final coverage of 92%. In any case, the LBL trace contained some unconventional message types that

were not published, such as MACB and CLNT requests. MACB commands are sometimes used by [FTP](#) clients running in the Macintosh [OSs](#) (e.g., CuteFTP or WebTen) to transfer files in MacBinary mode, while CLNT refers to an obscure feature of a particular [FTP](#) client (NcFTP) apparently used to identify it and to access shell utilities. Little more information is available for these two non-standard commands, as they are not specified by any [RFC](#) or other official document. This further reveals the importance of using real traces, since it allows our approach to generate test cases that cover otherwise unknown parts of a server implementation.

FTP test cases

Another aspect analyzed in our evaluation is the quality of the payloads generated by the tools. In order to measure such attribute, we studied the potential coverage of these payloads in detecting vulnerabilities. Note that this metric is optimistic in the sense that it does not reflect the ability of the tool to detect those vulnerabilities, but rather how many vulnerabilities could be detected by these payloads *if applied to the correct commands in the right state*. For instance, a tool may be using a rich set of payloads, but if it does not generate test cases for the vulnerable commands, it will not be able to detect them.

To perform this experiment, we exhaustively searched the Web for all known [FTP](#) vulnerabilities and obtained the respective exploits (i.e., the [FTP](#) command and the payload). We retrieved a total of 131 known vulnerabilities and classified them according to the type of payloads that were used in the exploits. In total, seven classes of payloads were defined, covering six types of vulnerabilities, from buffer overflows to [SQL](#) injection (see Table 5.6 for examples).

Type of payload	Type of vulnerability	Test case example	Extrated payload
Arbitrary strings	BO, ID, DoS	MKD Ax255	Ax255
Strings w/ special prefix	BO, FS, DoS	LIST ~{	~{
Large numbers	BO, DoS	REST 1073931080	1073931080
Format strings	FS, DoS	USER %x%x%x	%x%x%x
Directory structure	BO, FS, ID, DoS	LIST ../.../..	../.../..
Known keywords	ID, DoS, DC	USER test	test
Special constructs	BO, EE	USER ')UNION SELECT (...))UNION SELECT (...)

Table 5.6: Taxonomy of payloads from exploits of 131 FTP vulnerabilities
(BO: Buffer Overflow; FS: Format String; EE: Ext. App Exec. and SQL injection; ID:Dir. Traversal and Inf. Disclosure; DoS: Denial of Service and Resource Exhaustion; DC: Default Config.).

# Vulns and type of payload	Codenomicon	Infigo	Nessus	OpenVAS	NeXpose	Our approach
50 Arbitrary strings	94%	94%	94%	10%	10%	94%
10 Strings w/ special prefix	25%	50%	0%	0%	25%	50%
3 Large numbers	100%	100%	0%	0%	0%	100%
11 Format strings	100%	91%	0%	0%	0%	100%
41 Directory structure	76%	71%	2%	0%	5%	90%
14 Known keywords	21%	0%	50%	29%	36%	64%
2 Special constructs	0%	0%	0%	0%	0%	0%
131 Potential coverage	78%	74%	42%	7%	10%	88% (actual)
Total test cases	71658	292644	148	53	54	1463151
Distinct payloads	5554	14116	45	12	23	23569

Table 5.7: Potential vulnerability coverage of FTP test cases (specialized FTP fuzzers, vulnerability scanners, and our approach).

We devised the following classes of payloads: arbitrary strings with different sizes (from 35 characters to over 135000), large numbers, strings starting with ~{, -, *?, or a:/, or strings containing wildcards, format strings, or directory structures, specific keywords (e.g., root, ~root, LPT1, etc.), and some special constructs (which included [SQL](#) queries).

Tables 5.7 and 5.8 depict the coverage of the payloads of the test cases generated by the tools with respect to the 131 vulnerabilities. Only the specialized fuzzers have a potential vulnerability coverage of over 70%, whereas the fuzzer frameworks achieved values between 36% and 66%. The vulnerability scanners

# Vulns and type of payload	Metasploit	FuzzTalk	BED	DotDotPwn	fuzzer.py	Our approach
50 Arbitrary strings	94%	94%	94%	94%	94%	94%
10 Strings w/ special prefix	0%	25%	25%	0%	50%	50%
3 Large numbers	0%	100%	33%	0%	100%	100%
11 Format strings	0%	73%	82%	64%	82%	100%
41 Directory structure	0%	61%	27%	63%	54%	90%
14 Known keywords	0%	14%	7%	7%	0%	64%
2 Special constructs	0%	0%	0%	0%	0%	0%
131 Potential coverage	36%	66%	58%	62%	63%	88% (actual)
Total test cases	3125	257892	24918	28244	197616	1463151
Distinct payloads	12	3450	139	10286	1023	23569

Table 5.8: Potential vulnerability coverage of FTP test cases (framework fuzzers and our approach).

posses the lowest coverage of 7% and 10%, with Nessus as exception with 42%, due to its high coverage in buffer overflow payloads and known keywords. In many cases, vulnerability scanners use passive test cases that only obtain the welcome banner of the server to check if the reported version is known to be vulnerable. This type of test case does not aim at triggering vulnerabilities and therefore its payload has a null potential vulnerability coverage.

Naturally, the highest potential vulnerability coverage was obtained by our tool since it extracts the testing payloads of all the input test cases. This result shows that our approach effectively combines the test cases from different sources to produce a new set of test cases with a higher vulnerability coverage than even the best specialized testing tools. Furthermore, given that our test cases had a very high protocol space coverage (92%), the *actual* vulnerability coverage should be very close to the observed potential vulnerability coverage (88%). See the case study for an experiment with two testing tools that gives evidence to confirm this observation.

An increased vulnerability coverage comes at the expense of executing a larger number of test cases, which may be too costly in some scenarios. We

Target protocol	Testing tool
SMTP	Bruteforce Exploit Detector (BED) 5.0
	Metasploit Framework (SMTP Simple Fuzzer) 4.0
	FuzzTalk Fuzzing Framework 1.0
	Simple Fuzzer 0.6.2 (sfuzz)
	Nessus 4.4.1
POP3	Bruteforce Exploit Detector (BED) 5.0
	Simple Fuzzer 0.6.2 (sfuzz)
	Nessus 4.4.1
Plain payloads	DotDotPwn 2.1
	Fuzz Generator (fuzz)

Table 5.9: Test case generation tools for non-FTP protocols.

believe, however, that in security testing maximizing the vulnerability coverage is crucial. In any case, a good coverage can still be achieved with our approach by resorting only to the test cases of two tools, as shown by the case study with Metasploit and DotDotPwn. Moreover, some kind of prioritization could be used to selected the actually tried test cases, therefore reducing the number of tests to some desired level, but this is left as future work.

Non-FTP test cases

This experiment studies the effectiveness of our approach when it resorts to test cases designed for implementations of other protocols. Table 5.9 presents the ten sources of test cases for non-FTP protocols that were used. It includes fuzzer frameworks and vulnerabilities scanners that create test cases for the SMTP (Klensin, 2008) and POP (Myers and Rose, 1996) protocols, as well as two plain payload generators, including the popular fuzz program from 1990 (Miller et al., 1990).

Tables 5.10 and 5.11 depict the potential vulnerability coverage of the extracted payloads. As with the previous experiment, our generated test cases got the best vulnerability coverage (83%). In fact, these test cases even obtained a

# Vulns and type of payload	SMTP					FTP
	BED	Metasploit	FuzzTalk	sfuzz	Nessus	Our approach
50 Arbitrary strings	94%	76%	94%	96%	94%	96%
10 Strings w/ special prefix	25%	0%	25%	0%	0%	75%
3 Large numbers	0%	0%	100%	0%	0%	100%
11 Format strings	82%	0%	73%	100%	0%	100%
41 Directory structure	27%	0%	0%	0%	0%	88%
14 Known keywords	7%	0%	0%	0%	0%	14%
2 Special constructs	0%	0%	0%	0%	0%	0%
131 Potential coverage	57%	29%	45%	45%	36%	83% (actual)
Total test cases	34288	1300	106136	3660	13	9939567
Distinct payloads	1087	304	303	156	29	160109

Table 5.10: Potential vulnerability coverage of non-FTP test cases (SMTP testing tools and our approach).

better potential vulnerability coverage than the best specialized [FTP](#) testing tools, such as Codenomicon's or Infigo's fuzzers, with 78% and 74% respectively.

This result indicates that recycling the test cases is helpful because the payloads from one protocol can also be successfully used in other protocols, since the classes of vulnerabilities they might experience are relatively similar (and therefore, they can be exploited in equivalent ways). The tables also suggest that using other sources of test cases can contribute to a richer set of payloads. For instance, fuzz (from Table 5.11), which is a simple random payload generator, has a high potential coverage for [FTP](#) vulnerabilities.

Case study with two FTP fuzzers

This experiment provides a closer look at how our approach can create test cases with greater coverage than the original input test cases. To better illustrate this, we restricted the sources of input test cases for our tool to two testing tools with very different values of protocol space and potential vulnerability coverage. We chose the [FTP](#) fuzzer from the Metasploit Project, which has a protocol coverage of

# Vulns and type of payload	BED	POP sfuzz	Nessus	Plain payloads		FTP Our approach
				DotDotPwn	fuzz	
50 Arbitrary strings	94%	94%	94%	94%	76%	96%
10 Strings w/ special prefix	25%	0%	0%	0%	75%	75%
3 Large numbers	0%	0%	0%	0%	0%	100%
11 Format strings	82%	100%	82%	64%	100%	100%
41 Directory structure	27%	0%	0%	63%	80%	88%
14 Known keywords	7%	0%	0%	0%	14%	14%
2 Special constructs	0%	0%	0%	0%	0%	0%
131 Potential coverage	57%	44%	43%	61%	71%	83% (actual)
Total test cases	16195	676	367	6984	150370	9939567
Distinct payloads	193	676	7	6984	150370	160109

Table 5.11: Potential vulnerability coverage of non-FTP test cases (POP and plain payloads generator tools and our approach).

72% and a potential vulnerability coverage of 36%, and the DotDotPwn, a fuzzer specialized at directory traversal vulnerabilities, which has the lowest protocol coverage of 3%, but a high potential vulnerability coverage of 62%.

We analyzed the test cases produced by the tools and evaluated their *true vulnerability coverage*. One way to calculate this coverage is by trying the tools with the various [FTP](#) servers to determine if they discovered the corresponding vulnerabilities. Unfortunately this is not feasible to do in practice because several of the vulnerable versions of the servers are no longer available, either because they are deleted from the Web to avoid the dissemination of the flaws or because they are no longer distributed by the commercial software vendors. Additionally, in other cases, some of the support software ([OS](#) and/or specific libraries versions) is also missing, since some of the vulnerabilities have a few years.

In alternative, we calculated the true vulnerability coverage by inspecting manually the generated test cases to find out if one of them was equivalent (from a testing perspective) to the exploit of the vulnerability. Figure [5.4a](#) shows the actual coverage of the two fuzzers and our tool. We can see that neither fuzzer

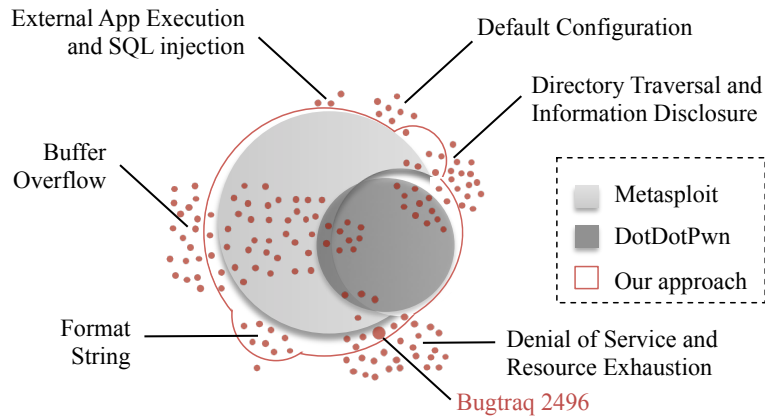
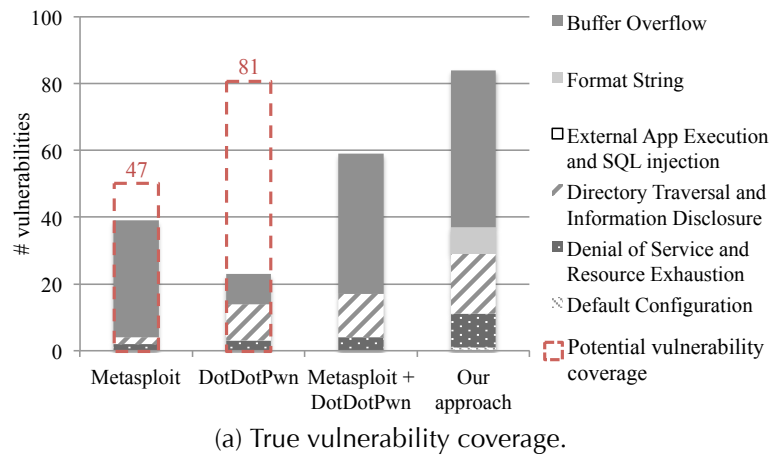


Figure 5.4: Analysis of the true vulnerability coverage of experiment 3.

achieved their full potential vulnerability coverage. Metasploit’s fuzzer, for instance, is able to detect 39 vulnerabilities, although it has a potential vulnerability coverage of 47 vulnerabilities (36%). DotDotPwn is a good example of a tool that even though it produces good payloads (with a potential vulnerability coverage of 62%), it is only able to cover 23 vulnerabilities (only 18% of the total number of vulnerabilities).

The graph further reveals the gain of combining both tools to detect vulnerabilities, where together they can discover 59 [FTP](#) vulnerabilities (about 45% of

the total number). Our tool, however, generated a set of test cases with a coverage of 64% of the known [FTP](#) vulnerabilities. This shows that our approach can cover an additional 25 vulnerabilities that none of the tools created test cases for. Figure 5.4b displays a representation of the coverage of the [FTP](#) vulnerabilities by Metasploit's [FTP](#) fuzzer, DotDotPwn and our own approach. Each dot corresponds to one of the 131 vulnerabilities, and vulnerabilities of the same type are graphically depicted near to each other. The area of the circles depicts the number and type of vulnerabilities found by each tool. For example, there are 65 dots for the buffer overflow type of vulnerability, where 35 were found by Metasploit, 9 by DotDotPwn, and 47 were disclosed by our tool (an additional 5 when compared with using both tools), while 18 still remained undetected from all tools.

Figure 5.4b shows that the Metasploit's [FTP](#) fuzzer has a much larger coverage than DotDotPwn, circumscribing many vulnerabilities. Nevertheless, DotDotPwn can cover many vulnerabilities that Metasploit cannot, such as directory traversal and information disclosure and buffer overflow. Our approach is depicted by the line surrounding both circles. The area delineated by this line is larger than the union of both circles because it combines a much better protocol space coverage with the payloads of both tools. As a last experiment to help to confirm this hypothesis, we actually installed the open source server wu-ftp 2.6.0 that is known to contain a [DoS](#) vulnerability (bugtraq 2496). Then, we used the three tools to test it, and found out that both Metasploit and DotDotPwn fuzzers were incapable of finding the flaw, but our tool was successful.

5.3 Conclusions

This chapter presented two alternative solutions to the attack generation problem. Both approaches resort to the specification of the communication protocol implemented by the target system in order to maximize the protocol coverage, and consequently, the amount of tested code. In addition, both solutions use special testing payloads, which have been observed in most of the reported network attacks, in order to generate test cases that are able to trigger potential vulnerabilities.

The first solution consists in a series of test case generation algorithms that generate attacks based on the original protocol messages. Each algorithm focuses on testing particular features, such as delimiters, syntax, and values. The latter test case generation algorithm was used by two attack injection tools to discover vulnerabilities in e-mail and domain name servers.

The second solution proposes to recycle existing test cases from several sources, including those designed to test implementations of other protocols. The approach automatically extracts the malicious payloads of the original test cases, and then re-uses them in a new set of attacks. These new attacks are generated based on a protocol specification inferred from network traces of the target server, allowing the inclusion of not only the normal protocol exchanges but also extensions and non-standard features. The experiments with ten [FTP](#) testing tools and ten other sources of test cases for non-[FTP](#) protocols, show that in both cases our approach is able to get better or equal protocol and vulnerability coverage than with the original test cases. In a experiment with two [FTP](#) fuzzer frameworks, our solution showed an improvement of 19% on the vulnerability coverage when

compared with the two combined fuzzers, being able to discover 25 additional vulnerabilities.

CHAPTER 6

INJECTION, MONITORING, AND ANALYSIS

This chapter covers the injection of the previously generated attacks in the target system, including the monitoring of the effects of the execution of the test cases, and the interpretation of the results. In the first part of the chapter, we introduce different strategies to inject the attacks, which may affect the kinds of vulnerabilities that can be discovered. Then, we present three types of monitor solutions that provide varying levels of information about the target system's execution. Finally, we show three approaches to the analysis of the injection results, with the objective of determining whether vulnerabilities were discovered.

6.1 *Injection*

The execution of the attacks is carried out by an injector component. This component decomposes each attack in its corresponding network packets, such as the prelude of messages to bring the protocol to the desired state and the actual testing message, and sends them to the network interface of the server.

In its simplest form, each attack is injected only once, the monitoring data is collected, and the environment is reset to the original experimental conditions (namely, the network server process is restarted) before the next test takes place. However, this requires a continuous synchronization between the injector and the monitor, and also that the monitor is capable of restarting the target system, which may not always be possible (e.g., the server runs in a machine that cannot be modified due to legal reasons). Furthermore, there may be some advantages in accumulating the effects of various attacks by not resetting the experimental conditions or by repeatedly injecting the same attack.

6.1.1 *Single injection campaign with restart*

In this approach, a test case corresponds to single attack, which requires the monitor to reset the experimental conditions after each test case execution. To accomplish this task, the injector and the monitor have to synchronize. The injector starts by notifying the monitor that a new test case is going to begin. The monitor then resets the target system to the original experimental setup, so that each test case is run under the same conditions. This includes a fresh execution of the network server with the original configuration and initial state, which can be obtained by restoring a disk image or a backup of the system. In addition, the

monitor should begin its monitoring operation of the target system and notify the injector once it is ready.

The injector then exchanges the network messages that constitute the attack with the network server and collects its responses. However, before sending each message and after receiving the response, the injector requests the monitor for the current monitoring data¹. This allows the injector to build up a continuous snapshot of the target system's evolution while it executes the test case.

At the end of the test case execution, the injector prepares the monitor for the next attack, and the process is repeated.

6.1.2 *Single injection campaign without restart*

Another injection strategy consists in accumulating the effects of multiple attacks without restarting the target system or the experimental conditions. This approach has the result of accelerating software aging (Parnas, 1994; Garg et al., 1998) and can thus be quite useful at detecting flaws related to resource-exhaustion and DoS. All the attacks are successively injected until a vulnerability is found or there are no more test cases available.

The injector performs in the same way as in the previous approach, however, the monitor should never reset the environment conditions after each test case execution. This particular approach is also useful to perform attack injection in the absence of a monitor component, although only fatal flaws are detectable (Subsection 6.2.1). The downside is that, since the target system is presented with many different attacks during the same execution, if a fault is detected it is difficult

¹Naturally, successive monitoring requests are merged into a single request because no additional processing has been performed in the meanwhile by the network server.

to identify which attack (or attacks) was responsible for activating it, in particular if the vulnerability is related to software aging. Moreover, the order of the attacks may be important in discovering certain vulnerabilities—some flaws may only be triggered if the attacks are made in a particular order. In any case, it is always possible to carry out injection campaigns with different sequences of attacks, but there is the risk of incurring in an excessive number of tests. Consequently, in our experiments, the sequence of attacks is injected once in the same order as they were generated.

6.1.3 Repeated injection campaign with restart

Another strategy to perform multiple injections to accelerate software aging is to repeated the same attack a number of times. Similar to the single injection campaign with restart, this approach also requires a monitor that synchronizes with the injector to reset the experimental conditions after each test execution.

This solution is comparable to performing a single injection campaign without restart because both approaches perform multiple injections during the same execution of the network server, although in this strategy, it is the same attack that is injected multiple times. Carrying out repeated injections of the same attack has the advantage of immediately identifying the attack that caused a vulnerability to be revealed. On the other hand, some vulnerabilities may be only activated under such unusual conditions that can be achieved by carrying out different successive attacks.

6.2 Monitoring

During the attack injection campaign, an injector component obtains the responses and execution data about the network server. The injector resorts to a monitor component, typically located in the actual target system and running alongside the server, to inspect the execution (e.g., Unix signals, Windows exceptions, and allocated resources).

In order for the monitor to support a more accurate identification of unexpected behavior, its implementation may require (1) access to the low-level process and resource management functions of the target system and (2) synchronization between the injector and the monitor for each test case execution. Although such monitoring component is desirable and very useful, it is not absolutely necessary and, in some circumstances, it might even be impractical. Some target systems, for instance, cannot be easily tampered or modified to execute an extra application, such as in embedded or proprietary systems. In other cases, it might just be too difficult to develop a monitor component for a particular architecture or OS, or for a specific a server due to some special feature (such as acting as a proxy to other servers).

To circumvent potential monitoring restrictions and to support as many target system as possible, three alternative monitoring components were developed (see Figure 6.1). The most general monitoring solutions, like the external and generic internal monitors, are quite easy to deploy because they are independent of the OS and libraries. However, they provide little information when compared with more sophisticated monitors, such as specialized internal monitors. These monitors can be created to make use of particular features of the OS (hardware drivers,

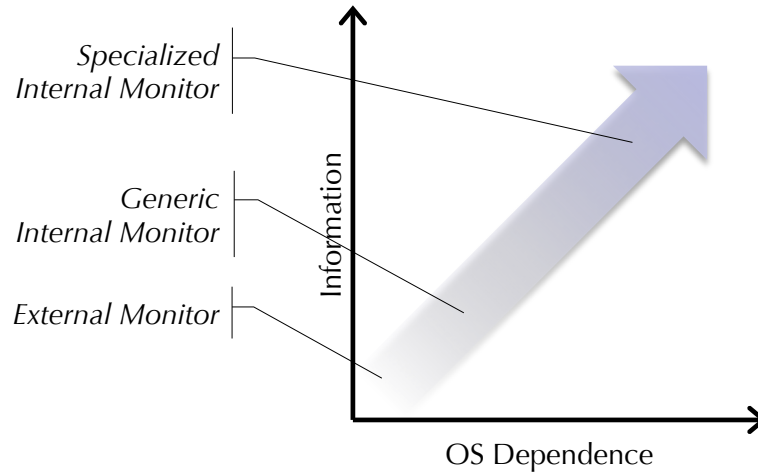


Figure 6.1: Range of monitoring solutions.

libraries, etc.) and therefore they can better characterize the target system and its internal state.

A monitor can also help to control the execution of the server, such as to restart the program when there is a hang. For instance, internal monitors can re-initiate the target system after transmitting to the injector the data collected about attack. This is, however, harder to accomplish with external monitors, unless there is specific hardware support.

6.2.1 External monitor

The external monitor infers the server's behavior remotely, mostly through passive and external observation, and ultimately through an additional network connection. It executes in the injector machine, and therefore, it has no in-depth execution monitoring capabilities. An external monitor should also synchronize with the injector, although it has little control over the network server, such as to force the restart of the server process after each attack injection.

This kind of monitors have to *infer* the state of the server based on the network connection (e.g., server replies and connections errors). However, the server implementation may opt not to respond to some attacks, e.g., if the attacks contain malformed messages, which may be interpreted as a fatal crash. In this case, the external monitor opens a new network connection with the server to determine if the server failed—if a communication error arises, it indicates that the last attack crashed the server.

This monitoring solution is the simplest and fastest approach when using an unknown or inaccessible target system (e.g., embedded or proprietary target systems), although it has reasonably limited capabilities.

6.2.2 *Generic internal monitor*

Internal monitors can obtain more information than an external monitor because they run in the target system. A range of internal monitors can be developed by resorting to more or less features of the underlying OS, which can allow them to be either more thorough or more system-independent.

At one side of the spectrum, generic internal monitors are platform-independent solutions that can be used in virtually any target system because they do not access the native features of the underlying OS. Although a generic monitor lacks in-depth tracing capabilities, it can still control the network server (i.e., restart the process) and collect the return status of the program's termination code. This code is useful because it allows the detection of fatal crashes and other evident abnormal behaviors.

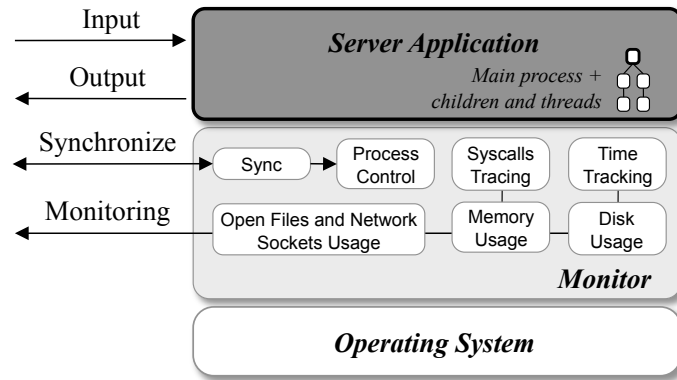


Figure 6.2: Specialized internal monitor.

6.2.3 Specialized internal monitor

Specialized internal monitors also run in the target system. However, contrary to the generic monitors, they are developed to take advantage of the native monitoring features of the hardware and OS to obtain the desired level of information and controllability over the network server and the system where it operates. Figure 6.2 illustrates a specialized internal monitor running in the target system.

In our current implementation of a specialized monitor for Unix-based systems, the tracing capabilities are achieved with the PTRACE family functions to intercept system calls and signals received by the network server processes (i.e., the main process, forked children, and threads). A signal interrupts the process execution to indicate that a special condition, such as an error, was triggered. A SIGSEGV signal, for instance, indicates a memory access violation, which is usually related to buffer overflow vulnerabilities. System calls are also interesting to monitor because any service provided by the OS that is not available to user level processes must be made through these special calls, e.g., allocate memory, write to disk, or send a message to the network. Therefore, the sequence (or more

simply the set) of system calls can provide a pattern of the execution of the server for a given task ([Warrender et al., 1999](#)). If the server executes a set of instructions distinct from the expected, this could reveal, for instance, that the flow of control is being hijacked. The PTRACE facility is also employed to obtain resource usage data at specific system calls related to its utilization (e.g., the amount of used memory is probed after a memory allocation or de-allocation call). This mode of operation supports the collection of monitoring data, while minimizing the interference with the server's normal behavior.

In more detail, the following local resources are currently being monitored:

- *total number of processes*, including forked children and threads of the target server. PTRACE signal interceptions are used to track new process IDs (PIDs);
- *memory pages*, given by the number of resident set pages minus the shared pages, are obtained through the LibGTop library ([Baulig and Kacar, 2012](#));
- *file descriptors*, such as those identifying opened disk files or network sockets, are kept in a updated list of file descriptors. LSOF ([Abell, 2012](#)) calls are used to keep track of the open files;
- *disk usage*, specified by the number of bytes written to disk. This value is obtained by parsing the LSOF's output for the files in use, and recording their size throughout the execution;
- *CPU cycles*, which corresponds to the work performed by the processor for all server's processes, is obtained by performance hardware counters. The Linux kernel had to be patched to associate to each process a private set of *virtual* hardware counters ([Pettersson, 2002–2011](#)). The monitor controls

and accesses these counters through the PAPI library ([London et al., 2001](#); [Innovative Computing Laboratory, 2012](#));

- *wall time*, measured as the elapsed time from the beginning of the main process execution, is computed by simple `gettimeofday()` calls. This resource is monitored mainly to compare its value with the number of CPU cycles. Large CPU and wall time discrepancies normally indicate a non-active wait, which suggests the presence of some timeout or deadlock.

Specialized internal monitors can impose harder implementation and operational requirements than other monitor solutions (e.g., they may need low-level OS access or a patched kernel) and can cause some perturbation on the server, potentially affecting its execution (e.g., intrusive probing or increased system overhead). Nevertheless, this is the most comprehensive and thorough monitor solution, and as such, it can provide a more accurate picture of the target system's internal state. We extensively use specialized internal monitors in many of the solutions presented in subsequent chapters, namely in the attack injection tools.

6.3 *Analysis of the Attacks*

Attack injection aims at triggering existing vulnerabilities, which when activated should result in some unexpected behavior. Depending on the type of flaw, it may be quite simple to identify the problem or it may require a more careful observation and analysis on the collected data. Vulnerabilities that result in fatal crashes are easily perceived through the raised software or hardware exceptions, returned error codes, or even network connection timeouts. However, not all vulnerabilities result in such evident effects. Privileged access violations, for in-

stance, are only discovered if some illegal access is granted by the server, which can be found by studying both the attack (i.e., the type of protocol request) and the server's response (i.e., the authorization).

This section presents three approaches for the analysis of the attack injection results. The simplest solution involves searching for known faults or patterns of faulty behavior. Then, we describe two methods that perform automatic detection of vulnerabilities: one approach builds resource usage profiles to discover vulnerabilities related to resource-exhaustion, and the other solution consists in inferring a complete behavioral profile of the server's correct execution, which is then used to identify deviant behaviors.

6.3.1 Analysis through fault pattern detection

Searching for known patterns of faults is a straightforward procedure to identify flaws. Typically, the monitoring data is inspected and searched for known faulty behaviors after the injection campaign. However, it is limited to known signs of faults. For instance, one may discover buffer overflow vulnerabilities if the monitoring data contains a SIGSEGV signal, which indicates that an illegal memory address has been referenced. However, if one does not know the particular signal (or other monitoring feature) that is indicative of a vulnerability, the flaw will remain hidden in the midst of other innocuous signals.

Depending on the type of monitor and on the level of information it provides, different kinds of vulnerabilities may be detected. An external monitor, for instance, is limited to detect vulnerabilities that result in fatal hangs or that can be perceived by looking at the content of the server's responses. Specialized internal monitors, on the other hand, can provide a rich set of monitoring

information that allows for more kinds of vulnerabilities to be found. An [SQL](#) injection, for instance, may cause an additional sequence of system calls to be executed. Therefore, if a particular attack triggers this kind of vulnerability, the monitoring data will contain the new sequence of system calls (as well as a surge on the resource usage).

Detecting known fault patterns requires the knowledge and the unique characterization of these patterns in the first place. Faults that result in subtle effects or whose effects are unknown, are very difficult to discover because they may be hard to characterize and to search for. Nevertheless, we have used fault pattern detection in one of the attack injection tools, [AJECT](#), and it was able to find new vulnerabilities in several network servers (Section [7.1](#)).

6.3.2 *Analysis with a resource usage profile*

An effective way to carry out a [DoS](#) attack consists in exploiting some vulnerability in the network server. In other words, the adversary resorts to a malicious interaction to activate the vulnerability and as a result, the server suffers an immediate crash or some sort of service degradation. In the presence of a fatal crash, the fault usually brings the server into an erroneous state that cannot be handled, abruptly ending the execution. Example vulnerabilities that usually produce such behavior are the well-known buffer overflows and divide by zero. On the other hand, in the case of a service degradation, faults are subtler but they can also lead to a complete halt if they occur at a higher rate. These faults appear due to resource-exhaustion vulnerabilities.

A resource-exhaustion vulnerability is a specific type of fault that causes the consumption or allocation of some resource in an undefined or unnecessary way,

or the failure to release it when no longer needed, eventually causing its depletion. As this definition indicates there are two classes of problems that can lead to resource-exhaustion. The first one is related to a bad design or an inefficient implementation of the server, forcing it to spend more resources than required. As a consequence, the overloading of the system can be accomplished with much less effort, when compared with an efficient design or implementation. For example, a component with poor resource management or slow algorithms can reserve large chunks of memory that are only partially utilized or waste valuable CPU cycles.

The second problem is associated with resource leakages. Here, the resource is indeed necessary but the server fails to make it available after use. Examples are a component that neglects to close a file descriptor or to free some memory, or a log file that grows indefinitely due to some error condition. These flaws are particularly important in long running servers, since the cumulative resource consumption can build up through time (also known as software aging ([Vaidyanathan and Trivedi, 2005](#))).

On a malicious setting this is even more serious because the particular situation that causes the resource leakage can be continuously forced by the adversary. [TCP SYN](#) attacks, for instance, cause the server to create half-open connections until it fills up the maximum number of available connections ([Eddy, 2007](#)). Other attacks, such as memory leak attacks, cause [DoS](#) by continuously forcing the server to execute a particular task that uses some chunk of memory that is not freed upon completion ([BugTraq, 2003](#)).

Keep in mind, however, that these vulnerabilities can have other impact besides [DoS](#), such as data corruption or some other exceptional illegal state. Con-

sider a scenario where a server with very little disk space does not properly verify the error status of a write call. If the write call terminates abruptly due to space shortage, the data stored in the disk is left in an inconsistent state.

In the rest of this section, we describe a method specifically designed for the detection of resource-exhaustion vulnerabilities. This approach was implemented in the [PREDATOR](#) attack injection tool that is presented in Section 7.2.

Discovering resource-exhaustion vulnerabilities

Typical solutions for vulnerability discovery, such as scanners or fuzzers, inject faults (i.e., malicious attacks) in the target system and look for an abnormal outcome that indicates some sort of problem ([Miller et al., 1990](#); [Roning, J. et al. 1999–2003](#); [Tenable Network Security, 2002–12a](#)). This approach is usually confined to locate vulnerabilities that produce quite visible effects, normally a crash. More subtle results, like those associated with resource-exhaustion vulnerabilities, are much more difficult to observe. The resource loss cannot be detected just by looking at a single snapshot of its utilization. Only a continuous and careful monitoring can perceive the overall tendency in which the resource depletion is developing into.

Our method searches for vulnerabilities through a comprehensive analysis on the system resource utilization of many (potentially malicious) client/server interactions. This is accomplished by injecting attacks into the network interface of the target system while monitoring the server's evolution. The whole procedure is performed in three basic phases:

Test case generation phase: Given a specification of the communication protocol utilized by the server, the test case generation creates valid and invalid network interactions (as explained in Chapter 4). When transmitted to the target system, these messages will test its ability to cope with some erroneous data, such as an out-of-bounds value or a missing field. This phase can employ different attack generation algorithms specialized in detecting specific classes of vulnerabilities. As more knowledge is gained on how to activate resource-exhaustion vulnerabilities, additional generation algorithms can be implemented to produce test cases that could trigger those flaws.

Exploratory phase This phase runs the entire universe of the generated test cases using the *repeated injection with restart* strategy presented in Section 6.1.3. Each attack has to be performed many times, so that the monitoring data can be gradually collected to build a profile of each resource usage. After the profile usage has been created, the target system is restarted.

The resource usage profile is a statistical model that describes the consumption of each resource. As the number of repeated injections increases, so does the accuracy of the resource usage profile. However, it may not be feasible to perform a large number of injections for the whole set of attacks. Therefore, in the exploratory phase, each attack is injected just the number of times that allows any resource utilization variation to be detected. The minimal number of repeated injections depends on the granularity of the resource monitoring to guarantee that changes on the resource use can be observed. For example, if the monitoring mechanism can measure exactly how much memory is allocated and released by each attack, then two repeated injections might be sufficient to detect any

memory usage variation. On the other hand, if the memory leak is small and the monitoring mechanism works with a page size granularity (e.g., counts the number of pages assigned to the process), then it is necessary to re-inject the attacks as many times as needed to force the allocation of an additional memory page.

In any case, enough information must be obtained to construct a usage profile for each considered attack and resource. We use regression analysis on the collected data to produce a statistical model of the actual resource consumption. Linear regression, however, also imposes constraints on the minimum number of repeated injections. If p coefficients have to be estimated, then $n \geq p + 1$ data samples are necessary to determine the regression, and therefore at least n injections have to be performed. Consequently, the minimum number of repeated injections is defined by both the monitoring capabilities and the type of regression analysis.

At the end of this phase there is, for all test cases, a profile for each monitored resource. With this information it is possible to recognize which attacks are more dangerous by looking for the profiles with higher growth rates.

Exploitive phase The execution of this last phase is optional, though it should provide more accurate profiles, which supports for better forecasts of the resource utilization and to remove false positives. The second injection campaign is initiated exclusively for the small subset of attacks that showed highest DoS potential. Essentially, it performs a larger number of repeated injections for these attacks and calculates new and more precise projections.

Using this method has several useful applications. First, it allows the discovery of resource-exhaustion vulnerabilities, whether caused by simple bugs, e.g.,

buffer overflows, or by more subtle faults that also result in the excessive consumption of a resource. Resource usage profiles can also contribute to the identification of the root of the problem, by discovering which resources are being depleted and by providing the test cases that activate the vulnerabilities. Developers can then use this information to fix the problem on the server. Additionally, the resource usage models can support further analysis since they let us forecast the consumption of resources in various scenarios with distinct attack magnitudes. For example, it is possible to find out: what are the main resource bottlenecks of a system; how many attacks can be sustained before the execution halts, and therefore estimate the critical level of the attack; compare the robustness of two implementations of the same server given some hardware configuration.

Modeling resource usage

A resource usage profile corresponds to a model of the use of a given resource in a particular test case. Since complex models typically require more computation time, there is usually a tradeoff between the accuracy of the model and the number of tests that can be performed within a reasonable time frame. In order to maximize tests, and increase confidence on the correctness of the server, the model has to be relatively simple to allow a rapid calculation. Among the different mathematical models that were considered, we opted for linear regression because it gave excellent results for the kind of analysis and vulnerabilities we were focusing.

Least-square analysis is employed to compute the parameters of the linear regression fitting. In its simplest form, linear regression estimates one parameter plus the constant intercept, which results in a straight line. This idea can be

generalized to higher degree polynomials by estimating p parameters. For example, with $p = 2$ the projection is a quadratic function (i.e., a parabola), and with $p = 3$ a cubic function. The number of parameters is also related to the number of inflections, or “curves”, that the polynomial has. A linear regression with p parameters will result in a polynomial of degree p with $p - 1$ inflections. A cubic function, for instance, will be useful to represent data that follows a cubic polynomial pattern, i.e., that has two inflections.

Therefore, it is important to study the nature and pattern of the data itself in order to determine the best polynomial that fits it. First, consider that it is the same attack that is injected multiple times, which results in the execution of the same task over and over again. Also note that the monitoring mechanism measures the total amount of resources spent by the server throughout n injections. Consequently, the resource usage data is the *accumulated value* since the beginning of the injections of that attack (and not per injection). This translates into a *nondecreasing monotonic* resource utilization (e.g., CPU time never decreases, memory consumption is constant or increasing).

The following two propositions result from an understanding of the data, and help us determine the degree of the linear regression polynomial:

- P1) Intuitively, since the resource usage data is nondecreasing, it should be suitably represented by straight line or a degree-two polynomial, i.e., by a curve with zero or one inflection. Therefore, the number of parameters to be estimated can be $p \leq 2$, plus the constant intercept.
- P2) Additionally, if resource waste is increasing (e.g., a memory leak), it does not necessarily grow at a constant rate. For example, some additional overhead may increase the resource consumption even further. This means that

a straight line may not be enough to accurately represent the loss of the resource. Therefore, in some cases a polynomial with at least one inflection ($p \geq 2$) may be required.

From these propositions we conclude that $p = 2$ estimated parameters are necessary and sufficient to correctly model the consumption of a resource due to the repeated execution of the same attack (i.e., $y = ax^2 + bx + c$).

6.3.3 Analysis with a behavioral profile

This section presents a method for deriving a behavioral profile of the correct execution of the server and for finding deviations from the expected behavior. A behavioral profile of the server is obtained by combining data from its internal execution with information about the implemented protocol. For this purpose, it uses the protocol specification (inferred or manually specified) and a monitor component that gives execution data. This approach thus provides a complete and detailed picture of the expected behavior of the target system. By inferring the behavior of the correct execution of the protocol, it allows the immediate identification of any vulnerability whose effects were unknown, and secondly, the automation of the analysis of results—a human operator is no longer needed to study the result of each test.

The behavioral profile is created in a *learning phase*. This phase is responsible for deriving a specification of the communication protocol used to exchange messages between the server and clients. The specification is complemented with detailed operational data about the server, obtained by monitoring the execution while it processes regular requests from the clients. This extended specification

is then used in the *testing phase* to discover abnormal behavior while the server executes a battery of test cases. A deviation from the expected behavior provides a strong indication about the test cases that were effective at triggering faults, which can then be further investigated by the developers to correct them. This approach was implemented in the [REVEAL](#) attack injection tool that is presented in Section 7.3.

Defining the behavioral profile

We want to automatically obtain a rich set of information about the server's execution, so that it can later be employed to provide evidence about incorrect behavior. Since we want to treat the server as a black box², thus avoiding dependencies on programming languages and run-time environments, the data that can be externally collected is: 1) the way the server interacts through the network (for example, in response to malicious requests); and 2) how the server relates to the local environment (namely, how it uses the resources offered by the [OS](#)). Therefore, in our approach, we combine both sources of information by capturing the server message exchanges in a protocol specification, which is complemented with monitoring data.

We use [FSMs](#) to model protocol specifications because they are a useful mathematical representation to describe the *language* of the protocol and its *state machine* (Chapter 4). Mealy machines are regular [FSM](#) that can model the interactions between the two protocol entities—they define both a transition function (i.e., give the next state for a given state and input) and an output function (i.e.,

²The reader, however, should notice that our approach can be easily extended if white-box monitoring data is available, by incorporating it in our model as an extra data source.

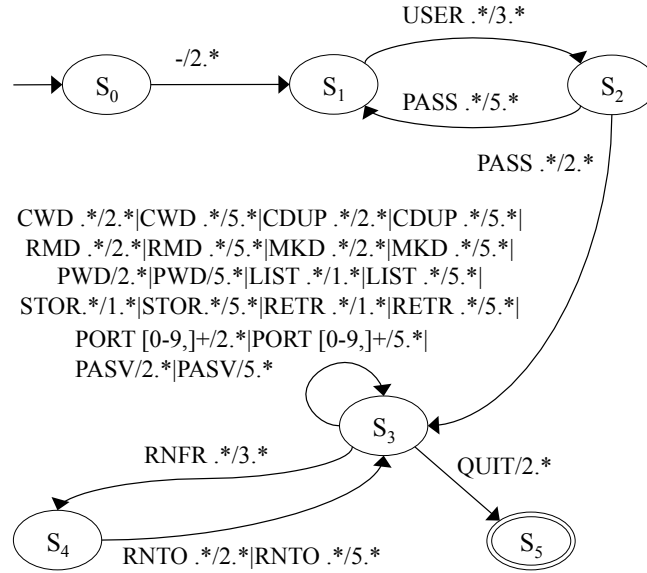


Figure 6.3: Subset of the FTP specification.

the output for a given state and input). Hence, we model the specification of the protocol as a Mealy machine, where each transition is defined by the client message (input), the server response message (output), and the following state. More formally, the Mealy machine $(S, I, O, f_t, f_o, s_0, F)$ is defined as:

S is a finite, non-empty set of states,

I is the input alphabet (finite set of requests),

O is the output alphabet (finite set of responses),

f_t is the transition function: $f_t : S \times I \rightarrow S$,

f_o is the output function: $f_o : S \times I \rightarrow O$,

s_0 is the initial state, and

F is the set of final states.

Figure 6.3 shows an example of a Mealy machine that models a subset of the FTP protocol. This automaton identifies the states and the input and output symbols (messages) for each transition (labeled as regular expressions in the form of

input/output). The protocol has an initial login exchange (encompassing states S_0 , S_1 , and S_2), states where the client can issue many different types of commands to access the remote file system (states S_3 and S_4), and a final state (S_5). Server replies start with a numeric code: 2 means a positive completion reply, 3 a positive intermediate reply, and 5 a permanent negative reply. For instance, after the server's welcome banner in state S_0 (labeled as 2 . *, a regular expression that recognizes messages similar to 220 Welcome to FTP server .), the client must transmit its credentials. First, the client sends the username (e.g., USER john that is labeled as USER .*) and then the respective password (labeled as PASS .*). If the password is correct, the server responds with a positive completion reply (labeled as 2 .*) and the protocol goes to state S_3 . If the password is incorrect, the server responds with a permanent negative reply (labeled as 5 .*) and the client must restart the login process.

Automata, such as Mealy machines, are well-suited representations for the expected external behavior of protocol implementations, such as the messages exchanged between the clients and the server, and for that reason they have been used in testing (Lai, 2002; Bosik and Umit Uyar, 1991; Hierons et al., 2009). While an analysis based on these models may be sufficient to detect some types of anomalies (e.g., wrong message or lack of response), there are many other types of vulnerabilities that are difficult to address with this information alone. In fact, more elusive classes of flaws can trigger faults that either remain dormant or imperceptible, not immediately affecting the server's compliance with the protocol specification (e.g., an attacker exploits a command injection flaw to force the server to execute some local program).

To support the detection of a larger range of vulnerabilities, we extend the

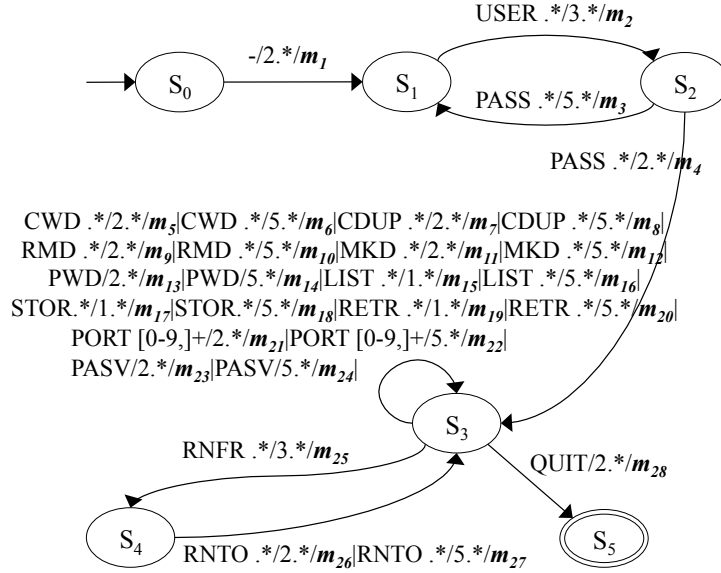


Figure 6.4: Subset of the FTP specification extended with monitoring data.

Mealy machine with information about the server's local execution. We call this extended Mealy machine the *Behavioral Profile* of the server and it is defined as the tuple $(S, I, O, M, f_t, f_o, f_m, s_0, F)$, where:

M is the monitoring alphabet (finite set of monitoring data), and

f_m is the monitoring function: $f_m : S \times I \rightarrow M$.

This automaton is depicted in Figure 6.4 and is obtained by monitoring the server's execution while it progresses through the various states (as defined in the original Mealy machine of Figure 6.3). Hence, besides the input and output symbols, we also associate to each transition the respective monitoring information (m_i). m_i corresponds to a tuple $(source_1, \dots, source_n)$, where each $source_i$ captures one of the dimensions of the server's internal execution as it interacts with the underlying OS and hardware (e.g., the range of memory consumption or the execution of specific OS calls).

The scope, accuracy, and thoroughness of the monitor determine the ability

to detect deviations from the correct behavior. A specialized internal monitor (Subsection 6.2.3) can provide a more complete view of the server's execution, which can be crucial to detect the pattern of execution caused by some classes of vulnerabilities. Vulnerabilities that do not affect the server's external messages, will nevertheless affect the server's local execution (Leon et al., 2005). Thus, the extended Mealy machine provides a more complete and accurate model of the server's behavior, which can then be explored during a testing phase to detect flaws.

Learning phase

The *learning phase* is responsible for obtaining the *Behavioral Profile* of the server. As input to this phase, we need a set of *Benign Test Cases* that correspond to sequences of interactions between the clients and the target server. Currently, they can be provided in two ways. Functional test cases, created by the developers to test the correct implementation of the server, can be used as benign test cases since they do not fail (i.e., they do not trigger any abnormal server's behavior). Alternatively, the messages from the clients can be extracted from the network traces and replayed to the server. In both cases, however, these tests should exercise the whole protocol specification (or the part of the specification that is going to be analyzed in the testing phase³) and thus they should cause positive and negative responses from the server (e.g., login with correct and wrong credentials).

Figure 6.5 shows the main components involved in the learning phase. The *Test Case Executor* component is responsible for processing each test case and

³Naturally, any test case used later in the testing phase that exercises some missing part of the specification, cannot be automatically evaluated.

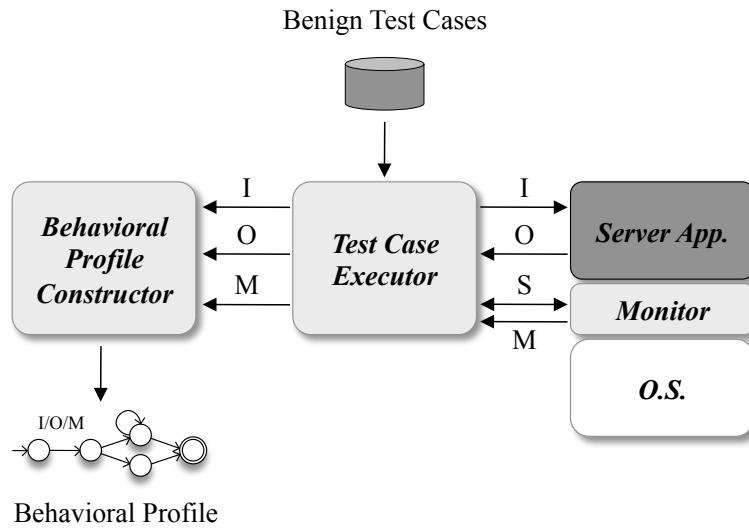


Figure 6.5: Obtaining a behavioral profile (learning phase).

for synchronizing with the *Monitor* component. Before each test case is started, the Test Case Executor instructs the Monitor to setup the testing environment, such as putting the server into a predefined initial state (launch a new instance of the server, discarding any changes that were previously made) and prepare the monitoring tasks. Then, once the server is running and waiting for new requests, the Test Case Executor starts emulating a client by sending request messages to the server and storing the corresponding responses. In addition, at each response received from the server, the Test Case Executor asks for new monitoring data. The Monitor then sends all information it has gathered since the last monitoring request. Therefore, the monitoring data depicts the server's progress and provides a snapshot of its current internal state.

Next, the Test Case Executor provides the input, output, and monitoring data of each test case to the *Behavioral Profile Constructor* component. This component uses the input and output to reverse engineer an approximate protocol specification (Section 4.3) and adds the monitoring data to the respective tran-

sitions in the form: $f_m(s_j, i_k) = m_l$ with $s_j \in S$, $i_k \in I$, and $m_l \in M$. If the server replies with more than one message to a request (because the response was split into multiple packets), the response messages are all concatenated as a single output symbol. Additionally, multiple instances of monitoring data for the same transition (typically, one after each response) are also combined into a single monitoring instance. In the existing implementation, the Behavior Profile Constructor uses two strategies for combining different instances of monitoring data. For resource usage, it just keeps track of the maximum observed values (e.g., maximum time, last count of CPU cycles, largest memory consumption, total disk utilization, total number of open files). If the monitor tracks sequences of operations (such as instructions, system calls, or signals) then they are concatenated, since they belong to the same protocol transition.

At the end of this phase, the Behavioral Profile Constructor produces the Behavioral Profile of the server, i.e., a Mealy machine modeling the server's protocol execution with the additional monitoring data.

Testing phase

The test cases used in the *testing phase* are also designed to command the server into performing various tasks, while covering as much of the protocol space as possible (or they can be focused on a subset of the server's functionality). However, besides exercising the normal server operations, they should mainly evaluate the server's robustness when confronted with unexpected and/or malicious input in the form of malformed sequences of messages or messages containing exploit data for known vulnerabilities. Figure 6.6 depicts the main tasks and components involved in this phase.

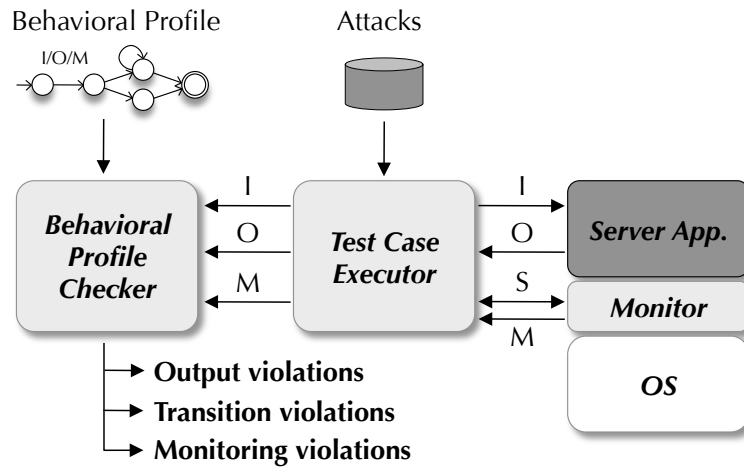


Figure 6.6: Using the behavioral profile (testing phase).

The *Test Case Executor* carries out each test case like in the learning phase, collecting the input, output, and the respective monitoring data. A synchronization with the Monitor guarantees that each test case starts executing in the same initial state. A special evaluator component, the *Behavioral Profile Checker*, analyzes the server's behavior by looking at the current state of the protocol, the input and output messages, and at the respective execution data. If the server's response is not in accordance with the Behavioral Profile, the evaluator found an output or transition violation.

An *output violation* is characterized by having the server in the correct/expected state producing an erroneous response (e.g., provides an incorrect code). A *transition violation* occurs when the server moves to a state distinct than the anticipated, due to a flaw that was activated while processing a request. A transition violation can also cause the server to send erroneous responses, possibly because the problem was in a previous transition that made the server jump to the wrong state. In order to distinguish from the output violation, we employ the following approach: if the server response is not in accordance with any of the

responses/output recognized in the same state, we consider that the server is in the wrong state, and thus it has a transition violation for that test case; otherwise it is an output violation.

Conversely, if the server's response is in accordance with the expected output, the evaluator checks the server's local execution. Depending on the type of information provided by the monitor, the evaluator can verify if the resources used by the server have exceeded the expected values for that transition, or even compare the sequence of operations (e.g., system calls). If the server's local execution deviates from the expected, the evaluator has found a *monitoring violation*.

Besides the type of violation that was detected and the respective test case, this approach also provides valuable information to identify the source of the flaw. By looking at the protocol request (input) that triggered the faulty behavior, the server's response (output), utilized resources, and signals or system calls (monitoring), one can better ascertain the type of the vulnerability and its location (e.g., a misconfiguration or a programming bug).

6.4 Conclusions

This chapter addressed the problem of the injection and monitoring of attacks, and the subsequent analysis of the results. Three strategies for the injection of attacks were described: single injection with restart, single injection without restart, and repeated injection with restart. These approaches differ mainly in two aspects, whether it is a single or more attacks that are injected throughout the same execution of the network server and the moment when the experimental conditions are reset.

This chapter also studied different monitoring solutions that are required to trace the execution of the target system during the injection of attacks. External monitors are able to remotely (and less intrusively) infer the state of the network server by resorting to an additional network connection. Internal monitors execute in the target system and can be made more generic or specialized. At one end of the spectrum, generic internal monitors do not make use of any native feature of the OS and can thus support more target systems, but are also incomplete. At the other side of the spectrum, specialized internal monitors can take advantage of advanced features of the hardware and OS to provide very precise and comprehensive monitoring information.

Finally, the chapter covers the analysis and interpretation of the attack injection results to identify the presence of vulnerabilities. A straightforward approach is to inspect the monitoring information to look for known fault patterns, such as typical error conditions and fatal crashes. Another approach consists in building resource usage profiles to find vulnerabilities related to the misuse of local resources and to automatically predict the utilization of every monitored resource. A last solution infers a behavioral profile modeling the server's correct execution of the protocol, combined with local monitoring data. This profile is then used as a reference to detect any deviation from the expected behavior while the server executes the test cases. Violations to the behavioral profile indicate that a test case has triggered some flaw and provide additional information about the faulty behavior.

CHAPTER 7

ATTACK INJECTION TOOLS

This chapter presents the attack injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#), that implement many of the solutions described previously. [AJECT](#) is a typical attack injection tool and it implements most of the injection and monitoring approaches discussed in the previous chapter, as well as the combinatorial test case generation algorithms. The tool was designed to look for vulnerabilities in network server applications, although it can also be utilized with local daemons. To assess the usefulness of this approach, several attack injection campaigns were performed with sixteen publicly available [POP](#) and [IMAP](#) servers. The results show that [AJECT](#) could effectively be used to locate vulnerabilities, even in well-known servers tested throughout the years.

[PREDATOR](#) is an attack injection tool based on [AJECT](#) that is able to perform

post-processing analysis to build accurate resource usage profiles of the target server. It implements a custom monitor and a two-phase injection campaign in order to detect small deviations in the utilization of the local resources, and is thus suitable to discover vulnerabilities related to resource-exhaustion. The validity of the approach was demonstrated in several experiments with synthetic programs and seven publicly available [DNS](#) servers.

Finally, [REVEAL](#) is an attack injection tool that identifies the presence of vulnerabilities by resorting to the inference of a behavioral profile. This approach is based on the creation of a behavioral profile that models the network server's correct execution, combining the information about its internal execution with the progress of the protocol. Flaws are automatically detected if the server's behavior deviates from the inferred profile while processing the test cases. The evaluation of this tool focused on determining its ability to detect abnormal behavior that result from different types of known vulnerabilities. Therefore, the tool was used to analyze [FTP](#) vulnerabilities, showing that it can effectively find the distinct patterns of faulty behavior automatically.

7.1 *AJECT*

[AJECT](#) (*Attack InJEction Tool*) is designed to discover new vulnerabilities in network servers. It does not require access to the source code of the server, however, in order to generate more effective attacks, [AJECT](#) resorts to a specification of the protocol that the network server implements (Section 4.2) and to a predefined set of test case generation algorithms (Section 5.1). The tool also implements two injection strategies, the single injection campaign with and without restart

(Section 6.1), and the three kinds of monitors: an external monitor, a specialized internal monitor, and a generic internal monitor (Section 6.2).

To demonstrate the usefulness of our approach, we have conducted 58 attack injection experiments with sixteen e-mail servers running POP and IMAP services. The main objective was to investigate whether AJECT could automatically discover previously unknown vulnerabilities in fully developed and up-to-date server applications. Although the number and type of target applications was not exhaustive, they are nevertheless a representative sample of the universe of the network servers.

Our evaluation confirmed that AJECT could find different classes of vulnerabilities in five of the servers, and assist the developers in their removal by providing the respective test cases. These experiments also lead to other interesting conclusions. For instance, we confirmed the expectation that complex protocols are more prone to vulnerabilities than simpler ones, since all detected vulnerabilities were related to the IMAP protocol. Additionally, based on the sixteen e-mail servers, we found that closed source applications appear to have a higher predisposition to contain vulnerabilities (none of the open source servers were found vulnerable whereas 42% of the closed source servers had problems).

7.1.1 Architecture and implementation

The architecture and main components of AJECT are depicted in Figure 7.1. The tool defines an attack generation phase, for producing the test cases or attacks, followed by an injection campaign phase for executing the test cases. The architecture was developed to achieve automatic injection of attacks independently of the network server implementation.

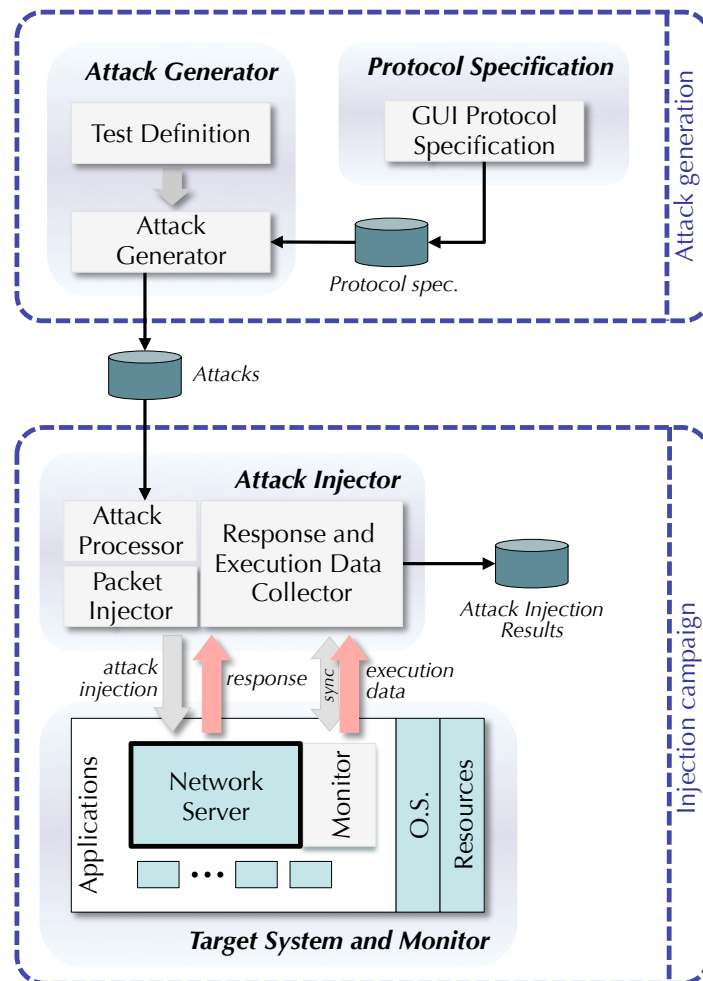


Figure 7.1: Architecture of the AJECT tool.

The *Target System* is the entire software and hardware components that comprise the network server and its execution environment, including the OS, the software libraries, and the system resources. The *Network Server* is typically a software program that provides a service that can be queried remotely from client applications (e.g., an e-mail server).

The network server uses a well-known protocol to communicate with the clients. The specification of the communication protocol is manually defined in AJECT through a GUI component (Section 4.2), which it uses to generate nu-

merous attacks by using any of the four test definitions that it implements (Section 5.1). Furthermore, AJECT was also built to be flexible regarding the method used to monitor the target system. In particular, it implements the three main types of *Monitors*: an external monitor, a generic internal monitor, and a specialized internal monitor. The current implementations of the generic and specialized internal monitors cannot trace the execution of background processes (also known as Unix *daemons* or Windows services) as they immediately detach themselves from the main process. However, this is not a major concern if applications are tested during development, since they are usually built as regular processes for debugging purposes.

Overall, AJECT's implementation provides a framework to create and evaluate the impact of different test case generation algorithms, injection strategies, and monitoring approaches. Moreover, other attack injection tools can be built upon this framework and further extend it.

7.1.2 Experimental evaluation

This section provides the details about the laboratory conditions in which the following experimental evaluation took place. It includes a description of the network server protocols that were specified and tried with AJECT, and the testbed configuration.

Communication protocols: POP and IMAP

The experimental evaluation was designed to assess the advantages of using attack injection to discover vulnerabilities in real applications. To that purpose we chose fully developed and commonly used standard protocols, POP3 and

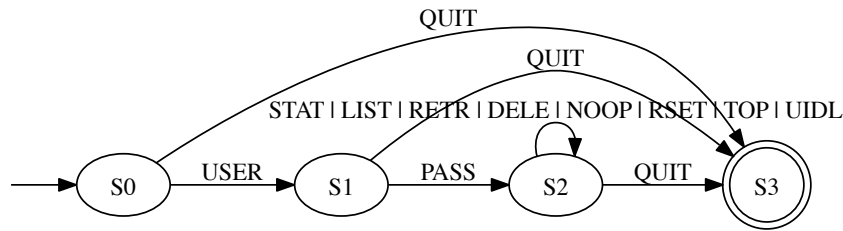


Figure 7.2: Input state machine of the POP3 protocol.

[IMAP4Rev1](#). Therefore, finding bugs in these targets is usually harder because applications have gone through several revisions, where all vulnerabilities had an opportunity to be removed. Additionally, the selected protocols are well-documented and not overly complex, which leads to simple and less error-prone implementations.

[POP](#) is a widely used protocol for e-mail retrieval ([Myers and Rose, 1996](#)). It was designed to allow users without a permanent connection to remotely view and manipulate messages. [POP3](#) servers listen on port number 110 for incoming connections, and use a reliable data stream ([TCP](#)) to ensure the transfer of commands, responses, and message data.

Figure 7.2 shows the [FSM](#) used in the specification of the [POP](#) protocol in [AJECT](#) based on the standard documentation. The client initiates the connection with the server in state S0. The actual authorization process is performed by the message types `USER` and `PASS`, which if contain valid credentials bring the protocol to state S2, where the client can perform all e-mail message access and retrieval transactions. When the client is finished, it issues the `QUIT` command allowing the server to perform various housekeeping functions, such as removing all messages marked for deletion, before closing the connection.

All interactions between the client and the server are in the form of text strings

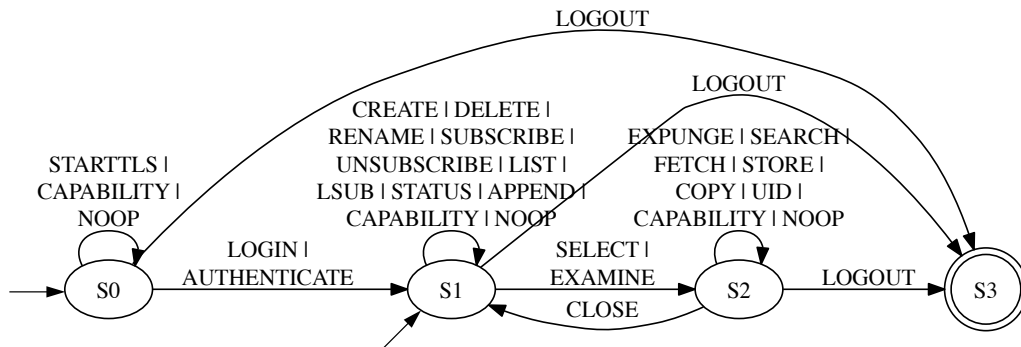


Figure 7.3: Input state machine of the IMAP4Rev1 protocol.

that end with **CR** and **LF** characters. Client messages are case-insensitive commands, with three or four letters long, followed by the respective parameters. Server replies are prefixed with **+OK** or **-ERR**, indicating a successful or unsuccessful command completion.

IMAP is a popular method for accessing electronic mail and Usenet newsgroup messages maintained on a remote server (Crispin, 2003). This protocol is specially designed for users that need to view e-mail messages from different computers, since all management tasks are executed remotely without the need to transfer the messages back and forth between these computers and the server. A client program can manipulate remote message folders (also known as mailboxes) in a way that is functionally equivalent to a local file system.

The client and server programs communicate through a reliable data stream (**TCP**) and the server listens for incoming connections on port 143. Once a connection is established, it goes into state S0 or S1 as depicted in the **FSM** of Figure 7.3. Normally, the protocol starts in an unauthenticated state, where most operations are forbidden. If the client is able to provide acceptable authentication credentials (i.e., an **AUTHENTICATE** or a **LOGIN** message with a valid

username and password), the protocol goes to state S1 where the she can choose a mailbox (with the SELECT or EXAMINE messages) to execute commands that manipulate respective e-mail messages. The connection is terminated when the client logs out from the server or when some exceptional action occurs (e.g., server shutdown).

All interactions between the client and the server are in the form of strings that end with the CR and LF characters. Depending on the type of the protocol request, the server's response may contain zero or more lines with data, ending with one of following completion results: OK (indicating success), NO (indicating failure), or BAD (indicating a protocol or syntax error). To simplify the matching between the requests and the responses, both the client commands and the respective server completion result are prefixed with the same distinct alphanumeric tag (e.g., A01, A02).

IMAP is more complex than POP, offering a wider functionality to its clients. It provides an extensive number of operations, which include: creation, deletion, and renaming of mailboxes; checking for new messages; permanently removing messages; server-based RFC 2822 and MIME parsing and searching; and selective fetching of message attributes and texts. Although some of the protocol commands are very simple (e.g., composed of a single field), most of them are much more intricate than in POP, and require various parameters.

Experimental environment

One of AJECT's features that we wished to test was its interoperability, i.e., its ability to support different target systems, independently of the hardware, OS, or software implementation. Therefore, it was necessary to utilize a flexible testbed

to ensure that the distinct requirements of the diverse network servers could be accommodated.

The testbed consisted of three physical machines with Intel Pentium 4 at 2.80 GHz and 512 MB of main memory, and running Windows XP SP2. Two of the machines acted as target systems and the remaining one as the injector. The target systems were executed in VMware Virtual Machine (VM) software on top of the native OS, each configured with 256 MB of RAM. A total of sixteen VMs were set up, split between the two physical machines, each installed with its own OS (either Windows XP SP2 or Ubuntu 6.10) and the respective network server (one of the sixteen e-mail servers). A generic internal monitor was installed in every Windows VM, whereas the Ubuntu VMs also featured the specialized and generic internal monitors (Subsections 6.2.3 and 6.2.2, respectively). The external monitor (Subsection 6.2.1) was executed remotely from the injector machine, so no additional installation was required on the VMs.

AJECT carried out the injection campaigns from the dedicated physical machine (injector) and collected the monitoring data from the monitor component in the target system. The protocol specification and the attacks were also previously generated in this injector machine.

The network servers were carefully selected from an extended list of e-mail programs supporting POP3 and IMAP4Rev1. All servers were up-to-date applications and fully patched to the latest stable version (at the time of the experimental evaluation), with no known vulnerabilities. All server applications had gone through many revisions and were supported by an active development team. Since we wanted to experiment with targets with distinct characteristics, we chose servers running in different OSs (Windows vs. Linux) and programmed under op-

E-mail servers (POP3/IMAP4Rev1)	OS	Version	Monitor
602LAN Suite Groupware (<i>602 Software</i>)	Windows	5.0.08.0403	External
Citadel* (<i>Uncensored Communications Group</i>)	Linux	7.32	External
dovecot*	Linux	1.1.rc3	Any
Hexamail Server Corporate	Windows/Linux	3.1.0.002	External
hMailServer	Windows	4.4.1	External
IMail Server (<i>Ipswitch</i>)	Windows	2006.23	External
Kerio MailServer (<i>Kerio Technologies</i>)	Windows/Linux	6.5.1	External
Mailtraq Email Server (<i>Fastraq</i>)	Windows	2.12.1.2364	External
Mdaemon Email Server (<i>Alt-N Technologies</i>)	Windows	9.6.5	External
Merak Mail Server (<i>IceWarp</i>)	Windows/Linux	9.1.0	External
NoticeWare Email Server NG	Windows	4.6.2	External
Softalk Mail Server Corporate	Windows	8.5.1.431	External
SurgeMail Mail Server (<i>NetWin</i>)	Windows/Linux	3.9e	External
uw-imap* (<i>University of Washington</i>)	Linux	2007b	Any
WinGate Email Server (<i>Qbik</i>)	Windows	6.2.2	External
xmail*	Windows/Linux	1.25	Any

Table 7.1: Target POP and IMAP e-mail servers (* open source).

posing philosophies regarding the availability of the source code (closed source vs. open source).

Table 7.1 lists the e-mail servers used in the experiments. Each server was subject to more than one attack injection campaign. For instance, since all programs supported both [POP](#) and [IMAP](#), they were tested at least twice, with each set of specific protocol attacks. Moreover, if the servers supported Windows and Linux, then both versions were also tested. Finally, to evaluate the advantages and drawbacks of the different monitoring solutions, each server was experimented individually with every supported monitor. This information is summarized in the last column of Table 7.1, which shows the monitor components employed with each e-mail server. The external monitor, for instance, was used with all target systems because it probes network servers remotely. Since the internal monitors (specialized and generic) were implemented to test open source Linux servers, they were utilized with servers developed for this [OS](#) (except for Citadel, which

could only be run as a daemon process, and therefore it automatically detaches from the monitor). Overall, each protocol was used in the 29 possible combinations of server/OS/monitor tuples, resulting in 58 independent attack injection campaigns.

Test case generation

AJECT implements the four test case generation algorithms from Section 5.1. A limited number of experiments was carried out initially to get a first understanding of how effective were these attacks. These preliminary results showed that delimiter and syntax test algorithms create test cases too simplistic for the discovery of security vulnerabilities in mature network servers. The attacks produced from these methods are not representative of the universe of attacks created by hackers or penetration testers—the messages are immediately rejected by the parsing and validation mechanisms of the servers and thus, they are only useful during the initial software development phases.

Furthermore, as explained in Subsection 5.1.4, the privileged access violation test algorithm is actually a specialization of the value test algorithm, requiring very specific malicious tokens and a more careful and time-consuming analysis of the results.

For this reason, and to maximize the automation of the various injection campaigns, we decided to center our efforts on testing the networks servers with the attacks produced by the value test algorithm (Subsection 5.1.3). However, should the focus be on a specific server (e.g., to debug a particular network server), more time and work should be invested on experimenting larger sets of attacks, generated with different algorithms.

```

1 $(PAYLOAD)
2 aject@$(PAYLOAD)
3 $(PAYLOAD)@aject
4 <aject<$(PAYLOAD)>
5 <$(PAYLOAD)<aject>
6 "$(PAYLOAD)"aject"
7 ./ private/passwd
8 D:\home\jantunes\AJECT\private\passwd

```

Listing 7.1: File with malicious tokens for POP protocol.

```

1 $(PAYLOAD)
2 aject@$(PAYLOAD)
3 $(PAYLOAD)@aject
4 <aject<$(PAYLOAD)>
5 <$(PAYLOAD)<aject>
6 "$(PAYLOAD)"aject"
7 ./ private/passwd
8 D:\home\jantunes\AJECT\private\passwd
9 ($(PAYLOAD))
10 "{ localhost/user=\"$(PAYLOAD)}"
11 (FLAGS BODY[$(PAYLOAD) (DATE FROM)])
12 (FLAGS $(PAYLOAD))

```

Listing 7.2: File with malicious tokens for IMAP protocol.

Listings 7.1 and 7.2 shows the contents of the malicious tokens files used in the value test algorithm. The ability to generate good illegal data is of the utmost importance, i.e., values that are *almost correct* in order to be accepted by the parsing and input validation mechanisms, but *sufficiently erroneous* in order to create such unusual conditions that were never achieved in testing, and that may trigger existing vulnerabilities. Therefore, picking good malicious tokens and payload data is essential. We defined some known usernames and hostnames (e.g., `aject`), as well as path names to sensitive files (e.g., `./private/passwd`). The special keyword `$(PAYLOAD)` is recognized by `AJECT` and expanded into various words: 256 random characters, 1000 and 5000 characters `A`, a sequence of format string characters (i.e., `%n%p%s%x%n%p%s%x`), a string with many non-

printable ASCII characters, and two strings with many relative pathnames (i.e., `../../../../` and `././././`). Depending on the size of the messages and fields, a different number of attacks was produced.

The attacks to the [POP](#) and [IMAP](#) protocols were generated by [AJECT](#) only once, taking a negligible amount of time—from a few seconds to a couple of minutes. The attack generation component created one attack file for each protocol, containing the prelude messages (to make the server go to the correct protocol state) and the testing messages. For the sake of fairness in the experiments, only the mandatory protocol functionality (i.e., no protocol extensions) was tested, which allowed all servers to be tested on the same stance. Based on the thirteen message specifications of the [POP](#) protocol, the algorithm created a 90 MB attack file with 35,700 test cases, whereas for [IMAP](#), which has a larger and more complex set of messages, there were 313,076 attacks in a 400 MB file. Given the size of the attack files, [AJECT](#)'s injector only reads and processes one attack at a time, thus keeping a small memory footprint.

POP and IMAP experimental results

Each experiment involved the injector in one machine and the target system in another (i.e., a [VM](#) image configured with either Windows or Linux, one of the sixteen target network servers with one of the supported monitor components). The monitor component was used to trace and record the behavior of the e-mail server in a log file for later analysis. If a server crashed or gracefully terminated the execution, for instance, the fatal signal and/or the return error code was automatically logged (depending on the type of monitor). The server was then restarted (automatically, or manually in case of the external monitor) and the injection

campaign resumed for the next attack. At the end of the experiment, we analyzed the output results, looking for any unexpected behavior. The log file presents in each line the result of a test case, making it easier to visually compare different attacks and to perceive divergent behavior. Any suspicion of an abnormal execution, i.e., any line in the log file with dissimilar monitoring data, such as an unusual set of system calls, a large resource usage, or a bad return error code, was further investigated. [AJECT](#) allows us to replay the last or latter attacks in order to reproduce the anomaly and thus confirm the existence of a vulnerability. The offending attacks were then provided as test cases to the developers to debug the server application.

The monitoring data also allows other less evident abnormal behaviors to be detected. Nevertheless, to identify these vulnerabilities, a more careful (and less automated) analysis is required. For example, the inspection of the server responses lets the operator detect when the server is disclosing some private information. However, in these experiments, we focused on the most automated log analysis, and only the most evident abnormal behaviors, related to [OS](#) signals, exceptions and resource usages, were looked for.

The actual time required for each injection experiment depended on the protocol (i.e., the number of attacks), the e-mail server (e.g., the time to reply or to timeout), and the type of monitor (e.g., the overhead of an additional software component constantly intercepting system calls and restarting the server application, as opposed to the unobtrusive external monitor). Overall, the [POP](#) injection campaigns took between 9 to 30 hours to complete, whereas the [IMAP](#) protocol experiments could last between 20 to 200 hours.

[AJECT](#) found vulnerabilities in five e-mail servers, which could eventually be

Vulnerable servers	Version	Corrected	Attacks / Observable behavior
hMailServer (IMAP)	4.4.1	4.4.2 beta	over 20k CREATE and RENAME messages <i>Server becomes unresponsive until it crashes</i>
NoticeWare (IMAP)	4.6.2	5.1	over 40 A01 LOGIN Ax5000 password <i>Server crashes</i>
Softalk (IMAP)	8.5.1.431	8.6 beta 1	over 3k A01 APPEND messages <i>Server crashes after going low on memory</i>
SurgeMail (IMAP)	3.9e	3.9g2	A01 APPEND Ax5000 (UIDNEXT MESSAGES) <i>Server crashes</i>
WinGate (IMAP)	6.2.2	-	A01 LIST Ax1000 * <i>Server denies all subsequent connections: NO access denied</i>

Table 7.2: E-mail servers with newly discovered vulnerabilities.

exploited by malicious hackers. Table 7.2 presents a summary of the problems, including the attacks that triggered the vulnerabilities, along with a brief explanation of the unexpected behavior as seen by the monitor (last column). All vulnerabilities were detected by a fatal condition in the server, such as a crash or a DoS. Two servers, hMailServer and Softalk, showed signs of service degradation before finally crashing, suggesting that their vulnerabilities are related to bad resource management (e.g., a memory leak or an inefficient algorithm). In every case, the developers were contacted with details about the newly discovered vulnerabilities, which included the attacks that triggered them, so that they could reproduce and correct the problem in the following software release (third column of Table 7.2). Since the servers were all commercial applications, we could not perform source code inspection to further investigate the cause of the anomalies, but had to rely on the details disclosed by the developers instead.

The first vulnerabilities were discovered in the IMAP service provided by the hMailServer. This Windows server gave signs of early service degradation when running with the external monitor. The telltale sign was a three-minute delay, with 100% peaks of CPU usage, when processing the attacks with the LIST com-

mand. Since the external monitor does not rejuvenate the system, the server was responding to the request with an extensive list of mailboxes created from the previous attacks. Further investigation on this issue showed that the server eventually crashed when processing over 20 thousand different CREATE and RENAME messages continuously. After helping the developers to pinpoint and correct the exact cause of the problem, they informed us that there was in fact a stack buffer overflow vulnerability, triggered when creating several deep-depth mailboxes (e.g., `../../../../../../../../mailbox`). Additionally, there was a second problem, which could also be remotely exploited to create a DoS—an inefficient parsing algorithm that was very slow on large buffers caused the high CPU usage.

NoticeWare also ceased to provide its service when the attacks were injected. In the experiment with the external monitor, the server was successively crashing after a series of different attacks. Then, after manually restarting the server and resuming the experiment, the server eventually failed again. The attacks were all related to the LOGIN command, for example, crashing after processing over 40 LOGIN messages with long arguments. This indicates that some problem was present in the parsing routine of that command, and the number and nature of the attacks hinted that some resource-exhaustion vulnerability existed. However, no details were disclosed about the vulnerability by the developers (even though they showed great interest in using AJECT themselves), and therefore we could not confirm this conclusion.

Softalk was another Windows IMAP server with clear service degradation problems, also probably caused by some resource-exhaustion vulnerability. The IMAP server was consuming too much memory while processing several APPEND requests, until it eventually depleted all memory resources and crashed. This

particular issue was aggravated by the fact that the attacker did not need to be authenticated in order to exploit the vulnerability. The overall number of attacks until resource depletion obviously depended on the amount of available memory and on the actual hardware configuration. Nevertheless, it did not require the attacker much effort to cause the DoS. The developers were contacted with details to reproduce the anomaly and eventually corrected their software, but they revealed no details about the vulnerability.

Both versions of the SurgeMail IMAP server, running on Windows and on Linux, were also found vulnerable. However, contrarily to the other three cases, only a single message was needed to trigger the vulnerability. An authenticated attacker could crash the server by sending one APPEND command with a very large parameter. Even though the presence of the large parameter suggests a buffer overflow vulnerability, the developers indicated that the problem was related to the execution of some older debugging function that was left in the production code—this function intentionally crashed the server when facing some unexpected conditions.

WinGate could also be exploited remotely by an adversary. In spite of the simplicity of the attack, oddly enough it was quite difficult to reproduce the anomaly with the developers. The DoS vulnerability that was found, stopped the IMAP service by denying further client connections. The attack was quite trivial, and consisted in a single LIST command with two parameters: several A's and a * (an asterisk). After processing the command, the server never crashed but replied to all new connections with the following message: NO access denied, refusing further access to new clients. This problem affected all new connections, independently of the IP source address, so this was not a blacklist defense mechanism.

However, in the few communications with the developers they were unable to reproduce the issue. Given this apparent difficulty, we repeated the same experiment in a separate Windows XP machine with a fresh server install (instead of the cloned VM image) still obtaining the same DoS result. Over the course of the experiments, we kept the developers up to date with our observations, but we received no further contact from them.

There was a sixth IMAP server, the 602LAN Suite, that revealed some intermittent problems, but we were unable to deterministically reproduce them (in order to contact the developers). Although most of the time the IMAP server operated uninterruptedly, on occasion, it would hang after 100 or 200 thousand attacks. Even though it is likely that there was some vulnerability in the network server, we did not have the resources (e.g., source code) to confirm it.

Additional observations

During the experiments, we came upon many differences in the way the network servers handled the client requests, such as: when facing a malformed message, some servers opted not to reply; server responses in the format of multi-line messages, which require the client to wait and receive each message separately to obtain the complete response; the utilization of IP blacklist defense mechanisms, preventing AJECT from successively injecting malformed packets in the server (naturally, such servers had to be excluded from the evaluation¹). These observations prompted several minor improvements in the implementation of AJECT to self-adapt to the various patterns of message handling, and therefore, to minimize

¹It would be possible to test these servers if the attack injector implemented IP spoofing, which would allow the execution of each test case with a different IP address, thus fooling the protection mechanism.

the number of false positives.

Additionally, the characteristics of the vulnerable servers hinted at other interesting conclusions, which would, naturally, benefit from a more extensive validation with a larger sample of network servers. First, all discovered vulnerabilities were related to the [IMAP](#) protocol, which seems to indicate that more complex protocols lead to more error-prone implementations, as one would expect. A rough analysis between the [POP](#) and [IMAP](#) protocols, in terms of number of states and messages types, indicates that [POP](#) is a much simpler protocol. Therefore, implementations of the [IMAP](#) protocol, usually require more lines of code and, consequently, more sophisticated tests, which makes the debugging operations a much harder process, increasing the chances of some defect being missed.

Second, all vulnerabilities were detected in closed source commercial servers. In fact, 42% of the tested closed source servers had confirmed vulnerabilities, which is a quite significant number. In part, this result could be explained because the number of open source servers was significantly lower than closed source servers, accounting for only 25% of all target systems. Naturally, a more complete experimental evaluation with a larger and more balanced sample of network servers could clarify this particular aspect. Nevertheless, we conjecture that the real reason why open source servers have fewer flaws is probably related to the larger, and usually more active, community behind these servers (testing, utilizing, and sometimes even developing them). Moreover, only the mandatory protocol functionality was specified in the attack generation. Therefore, any extra functionality or complexity potentially present in some commercial servers was not an issue, since all testing was restricted to the same common set of features.

Another interesting result is related to the monitoring approach required for

the detection of the vulnerabilities. Unfortunately, none of the servers that were found vulnerable with the external monitor were supported by other monitoring approaches. However, based on the attacks and on the observable behavior, one can infer that both internal monitors could also detect the vulnerabilities discovered in the SurgeMail and WinGate servers (using the single injection without restart approach from Subsection 6.1.2).

The results also shed some light on the process of conducting an injection campaign. As seen earlier, not restarting the target application between injections has proven to be advantageous in detecting some vulnerabilities that require software aging (e.g., in hMailServer). However, since each attack potentially changes the state of the target application, it might be possible that two attacks cancel each other out, thus invalidating the effects of an unpredictable number of test cases. Therefore, the order in which the attacks are carried out becomes important, if not determinant. Injecting the same attacks in a different order might yield different results, making this approach interesting and worthy of further study. Whenever possible, both methods should be used: first by exhausting all protocol messages individually, and then by continuously re-injecting all attacks without restarting the server, thus emulating the effects of software aging.

The results presented here show that [AJECT](#) can be very useful in discovering vulnerabilities even in fully developed and tested software. Actually, even though developers were not forthcoming into disclosing the details of the vulnerabilities, which is understandable because all servers were commercial applications, most of them showed great interest in using [AJECT](#) as an automated tool for vulnerability discovery. The attack injection methodology, sustained by the implementation of [AJECT](#), could be an important asset in constructing more dependable systems

and in enforcing the security of the existing ones.

7.2 PREDATOR

PREDATOR (*PREDicting ATtacks On Resources*) is a specialization of **AJECT**'s architecture to implement the resource usage profile analysis method (Subsection 6.3.2). It computes resource usage profiles for every attack in order to discover resource-exhaustion vulnerabilities (e.g., memory leaks) and to identify the protocol requests that are more susceptible to **DoS**.

The tool performs a *thorough resource and process monitoring*, making it capable of automatically detecting small resource usage variations, such as CPU cycles, number of child processes or threads, memory, disk, or open files. For this reason it can discover various kinds of bugs, like deadlocks or memory/disk leaks, that can be exploited to produce a service disruption. Furthermore, **PREDATOR** is capable of generating *resource usage profiles* through a post-processing analysis of the collected data. This allows the discovery of attacks that can compromise the availability of the system, as well as the most dangerous protocol interactions. Additionally, it does *test case prioritization*, which is achieved by a two-phase attack injection campaign using the repeated injection with restart strategy (Subsection 6.1.3). Only a minimum number of injections are performed in an initial exploratory phase to identify the most promising attacks, which are then evaluated in an exploitive phase with a larger number of injections.

A post-processing analysis on the collected data is performed to build accurate resource usage profiles and to find vulnerabilities. The attacks that triggered the vulnerabilities and the respective monitoring data can be provided to the de-

velopers to assist debugging. On the other hand, the resource profiles give a forecast on the amount of effort necessary for an attacker to deplete the server's resources. This information can also be used by administrators to assign critical levels to vulnerabilities, and to assist on decisions regarding hardware upgrades to sustain stronger attacks, at least until a patch is available.

The analysis approach using the resource usage profiles was experimentally evaluated with synthetic leak servers, i.e., programs purposely developed to contain different kinds of resource leaks, and with seven public-domain DNS servers. The results revealed that PREDATOR is quite suitable not only to discover remotely exploitable vulnerabilities that lead to the server crash, but also with the profiling of different resource usages. In particular, the experiments demonstrated the usefulness of the tool by disclosing two resource-exhaustion vulnerabilities in one of the DNS servers.

7.2.1 Architecture and implementation

The architecture of the PREDATOR tool is presented in Figure 7.4. The overall operation of the tool can be divided in the attack generation phase and the two injection campaigns (exploratory and exploitive).

The attack generation phase is similar to AJECT's, including the same test case generation algorithms and the manual definition of the protocol specification. It is performed off-line and only once for each target communication protocol (e.g., DNS, FTP), allowing the attacks to be reused to test network servers that implement the same protocol.

PREDATOR stands out from AJECT by defining two injection campaigns and building resource usage profiles. The injection campaigns follow the repeated

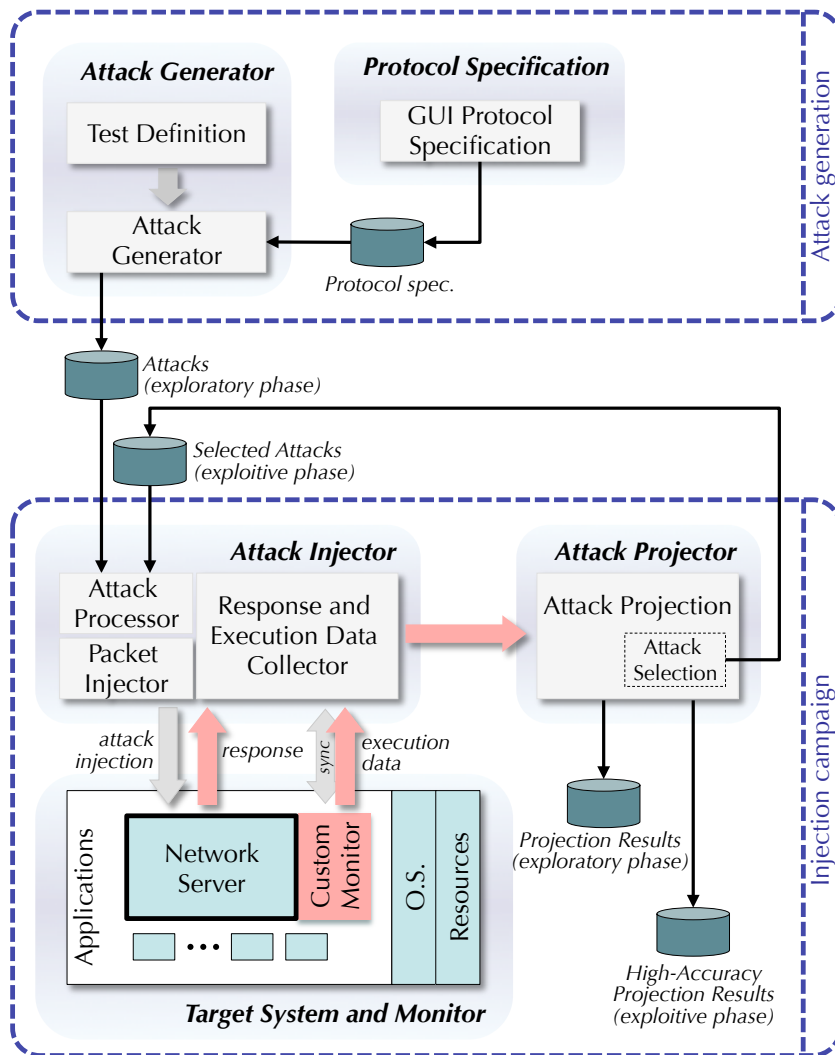


Figure 7.4: Architecture of the PREDATOR tool.

injection with restart approach from Subsection 6.1.3, differing only on the subset of attacks and on the number of injections. In the exploratory phase, the entire set of attacks is injected with a limited number of repeated injections. The small number of injections allows the tool to build a rough model of the resources utilization for each particular attack. This allows the most relevant subset of the attacks to be identified for the next exploitive phase, which due to the larger

number of repeated injections allows for more accurate resource usage profiles. For example, given the granularity of the memory usage monitoring (i.e., allocated memory pages), 256 injections is the minimal number of repeated injections that can capture the smallest memory leakage. The exploitive phase, on the other hand, can use any number of repeated injections, although we have empirically found that 1024 provides accurate projections for most situations.

PREDATOR resorts to third-party libraries to implement a highly specialized internal monitor (Section 6.2). This Custom Monitor was designed with resource monitoring in mind and was developed specifically to use CPU performance counters present in modern Intel-based systems and other resource tracking features from Unix OS derivatives. The monitor maintains a global table with resource usage data, which it regularly updates with the following local resources: total number of processes, memory pages, file descriptors, disk usage, CPU cycles, and wall time. Since inspecting the resources utilization incurs in a non-negligible overhead, the monitor only probes the usage data at the relevant system calls, such as `sys_brk` for the allocation of memory or `sys_write` when writing data to a file.

The resource usage data acquired by the monitor after every response, and the contents of the responses themselves, are gathered by the *Response and Execution Data Collector*. The *Attack Projector* then uses linear regression on this data to build the statistical profiles of the resource usage. In the exploratory phase, a list with the most dangerous attacks, i.e., those with higher estimated coefficients, is continuously updated. At the end, **PREDATOR** outputs the resource usage profiles for all attacks.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Length (TCP only)															
ID.															
QR	Opcode				AA	TC	RD	RA	Z			RCODE			
QDCOUNT															
ANCOUNT															
NSCOUNT															
ARCOUNT															
Question Section															
Answer Section															
Authority Section															
Additional Information Section															

Figure 7.5: DNS message format.

7.2.2 Experimental evaluation

The main purpose of the evaluation was to validate the analysis based on resource usage profiles and to demonstrate its usefulness by detecting resource-exhaustion vulnerabilities in [DNS](#) servers. This section details the experimental conditions in which the evaluation was conducted. It includes a description of the [DNS](#) protocol and the testbed configuration.

Communication protocol: DNS

[DNS](#) is a network component that performs a crucial role in the Internet ([Mock-apetris, 1987](#)). It is a hierarchical and distributed service that stores and associates information related to the Internet domain names. [DNS](#) employs a query/response stateless protocol. Messages have a large number of fields (see [Figure 7.5](#)), which can take a reasonable range of possible values (e.g., 16-bit binary fields or null-delimited strings), and the erroneous combination of these fields can be

utilized to perform attacks.

Experimental environment

The experiments were carried out in two Intel Pentium Dual-core 2.8 Ghz PCs, with 512 MB of main memory. One PC was running the injector, while the other corresponded to the target system (also running the monitor component). Each target network server was installed in a separated cloned partition of a basic Ubuntu Linux Distribution, with approximately 360 MB of free disk space.

The target servers were chosen to be representative of different types of resource leaks (synthetic leak servers) and actual production code (DNS servers). The synthetic leak servers provided a simple, yet controllable approach for the experimental validation. As for the DNS servers, their development and testing evolved and stabilized throughout the years. They are relatively small (few lines of code) and sustain continuous execution and testing, making them a challenging target.

Synthetic programs results

PREDATOR computes a statistical model that represents the utilization of resources, and therefore, is able to predict, within a small and acceptable error, its future consumption. To verify and validate this hypothesis several synthetic programs were developed containing distinct types of resource leaks. The programs were based on a simple TCP echo server—it has only one kind of interaction, where the client sends a hello message, and the server returns it back. Every time the server receives a message, it creates some sort of resource waste. Table 7.3 presents the fundamental characteristics of various synthetic leak servers.

Server ID	Leak type	No. of injections	Type of exhaustion
A	no leak	5,000	predefined no. of injections
B ₁	CPU leak (add)	5,000	predefined no. of injections
B ₂	CPU leak (mult)	5,000	predefined no. of injections
C	fork leak	4,888	memory exhaustion
D ₁	pthread leak (stack size 16 KB)	2,659	memory exhaustion
D ₂	pthread leak (stack size 8 MB)	383	memory exhaustion
E ₁	memory leak (malloc 4 B)	3,652,789	memory exhaustion
E ₂	memory leak (malloc 30 KB)	86,482	memory exhaustion
F	file open leak	1,019	open file limit
G	socket leak	1,019	open file limit
H	disk leak (write 30 KB)	13,042	disk exhaustion

Table 7.3: Synthetic leak servers with resource leaks.

It shows the identifier of the server, the type of leak, the number of attack injections until terminating the experiment (5000) or until the machine stopped responding, and the kind of resource that was exhausted. Since the size of the leak could be decisive to the conclusions, in some cases variations of the same server were used. Two CPU leak servers were configured to execute additional instructions, a cumulative sum and a cumulative multiplication (B₁ and B₂); a server which created another process per interaction (C); similarly, two servers which wasted a thread, with a stack of 16 KB and 8 MB, respectively (D₁ and D₂); two memory leak servers with 4 B and 30 KB (E₁ and E₂); a disk leak server with 30 KB (H) ; and two servers that did not close a file or socket descriptor per interaction (F and G).

Minimum number of injections: As stated previously, the minimum number of repeated injections for each attack must meet two requirements: allow the calculation of linear regression projections, and guarantee the detection of any resource usage variation. Since we need potentially to estimate $p = 2$ parameters, three independent data points are required, i.e., at least three injections have to

	Disk leak (4 bytes)	memory leak (4 bytes)
$n = 3$	$\hat{y}_d = 4\mathbf{x}$	$\hat{y}_m = 114$
$n = 256$	$\hat{y}_d = 4\mathbf{x}$	$\hat{y}_m = 0.0001\mathbf{x} + 113.99$

Table 7.4: Projections for a disk and memory leak created from n injections.

be done.

On the other hand, monitoring mechanisms can have distinct levels of precision and granularity. For instance, in the current version of [PREDATOR](#), all resources have a fine-grain type of monitoring with the exception of memory—the tool can only assess the number of memory pages assigned to the process. This affects the minimum number of injections because changes to the memory usage are only perceived when the process requests an additional page. The glibc *malloc()* implementation allocates at least 16 bytes on a 32-bit system (4 bytes for the preceding size field + 4 bytes for trailing size field + at least 8 bytes for the user block²). Therefore, no matter how many bytes a program requests, *malloc()* will reserve a block of at least 16 bytes. If a memory page is 4096 bytes long, in the case of the smallest memory leak of 1 byte, there must be 256 attack injections to force a new page request ($4096/16 = 256$). One should notice that other memory management implementations, which do not waste so much memory with control data, may require a larger number of injections.

Table 7.4 shows the linear regression projections for resources disk space (denoted by \hat{y}_d) and memory pages (denoted by \hat{y}_m) of two synthetic leak servers. The first of them had a 4 B disk leak and the other wasted 4 B of memory per interaction. The calculated models with 256 injections estimated parameters that reflected the increase in resource consumption (non-zero coefficients in the x

²The minimum of 8 bytes for the user block is imposed because when the chunk is freed, the memory manager must store two free list pointers (double-linked list) in this space.

variable), therefore, they identify the vulnerabilities. The table also demonstrates the impact of the granularity of monitoring. Since the mechanism measuring disk usage is capable of detecting variations of a single byte, three injections ($n = 3$) were enough to completely determine the model for the first server. However, for the memory case it was necessary to inject 256 times to reflect in the model the growing memory consumption, i.e., the 0.0001 coefficient.

Resource usage projection: To validate the linear regression model, **PREDATOR** generated resource usage profiles for the various synthetic leak servers, which were calculated with the measurements made on first 1024 attack injections. Then, the injector continued to attack the server until it stopped, so that real data could be obtained about the behavior of the resources as the server became exhausted. Finally, the real data was compared with the predictions.

The resource usage projections with $p = 2$ estimated parameters are presented in Table 7.5. We tried to use polynomials with degree higher than 2, but in all cases the additional coefficients were always zero. As it is possible to observe, in most cases resource consumption was either constant (sometimes zero) or had a constant growth (x^2 coefficient was zero). Highlighted in boldface are the coefficients that reflect the curved-shaped lines. These are the most dangerous vulnerabilities because the consumption increase is “accelerating”.

Another point worth of notice is the implicit correlation between the resources, in particular the CPU and the memory. For instance, servers D_1 and D_2 have a thread leak, i.e., a new thread is created at each request (which is not terminated). The creation of a thread has an impact on three resources: the number of processes/threads, CPU, and memory. Therefore, there is a correlation among these

Server	CPU M cycles	Processes	Memory pages
A	$\hat{y} = 0.93x - 1.04$	$\hat{y} = 1.00$	$\hat{y} = 106.00$
B ₁	$\hat{y} = \mathbf{0.01x^2} + 0.35x - 22.89$	$\hat{y} = 1.00$	$\hat{y} = 91.00$
B ₂	$\hat{y} = \mathbf{0.41x^2} + 13.74x - 522.30$	$\hat{y} = 1.00$	$\hat{y} = 91.00$
C	$\hat{y} = 0.22x - 14.05$	$\hat{y} = 1.00x + 1.00$	$\hat{y} = 69.99x + 114.62$
D ₁	$\hat{y} = 0.08x - 0.33$	$\hat{y} = 1.00x + 1.00$	$\hat{y} = \mathbf{2.04x^2} + 132.58x + 124.69$
D ₂	$\hat{y} = 0.12x + 0.03$	$\hat{y} = 1.06x - 2.01$	$\hat{y} = \mathbf{3.03x^2} + 134.90x + 137.42$
E ₁	$\hat{y} = 0.04x + 0.29$	$\hat{y} = 1.00$	$\hat{y} = 0.0001x + 103.63$
E ₂	$\hat{y} = 0.04x + 0.32$	$\hat{y} = 1.00$	$\hat{y} = 1.00x + 104.00$
F	$\hat{y} = 0.14x + 1.56$	$\hat{y} = 1.00$	$\hat{y} = 1.09x + 110.51$
G	$\hat{y} = 0.12x + 0.31$	$\hat{y} = 1.00$	$\hat{y} = 94.00$
H	$\hat{y} = 0.12x + 0.27$	$\hat{y} = 1.00$	$\hat{y} = 112.00$

(a) CPU, processes, and memory.

Server	File descriptors	Disk bytes
A	$\hat{y} = 0.00$	$\hat{y} = 0.00$
B ₁	$\hat{y} = 0.00$	$\hat{y} = 0.00$
B ₂	$\hat{y} = 0.00$	$\hat{y} = 0.00$
C	$\hat{y} = 0.00$	$\hat{y} = 0.00$
D ₁	$\hat{y} = 0.00$	$\hat{y} = 0.00$
D ₂	$\hat{y} = 0.00$	$\hat{y} = 0.00$
E ₁	$\hat{y} = 0.00$	$\hat{y} = 0.00$
E ₂	$\hat{y} = 0.00$	$\hat{y} = 0.00$
F	$\hat{y} = 1.00x + 5.00$	$\hat{y} = 0.00$
G	$\hat{y} = 1.00x + 5.00$	$\hat{y} = 0.00$
H	$\hat{y} = 6.00$	$\hat{y} = 28672.00x$

(b) File descriptors and disk usage.

Table 7.5: Resource usage projections for the synthetic leak servers (with $p = 2$).

resources for this sort of vulnerability (e.g., the correlation coefficient between the number of processes and memory pages is $R = 0.97$). This reveals the potential of the resource usage profile analysis to better understand the dependencies and relationships between the different resources, which can contribute to the identification of specific vulnerabilities.

The results of the goodness-of-fit of the linear regression projections, for $p = 1$ and $p = 2$ parameters, are depicted in Table 7.6. The table shows two well-known statistical measures for each of the major resources: the *adjusted coefficient of determination* (R_a^2) and the *Mean Square Error* (MSE). Since we want to evaluate

Server	CPU		Processes		Memory		Files		Disk	
	R_a^2	MSE	R_a^2	MSE	R_a^2	MSE	R_a^2	MSE	R_a^2	MSE
A	$p = 1$	1.00	0.0012	—	0.00	—	0.00	—	0.00	0.00
	$p = 2$	1.00	0.0011	—	0.00	—	0.00	—	0.00	0.00
B ₁	$p = 1$	-0.23	2.6×10^9	—	0.00	—	0.00	—	0.00	0.00
	$p = 2$	0.99	1.5×10^7	—	0.00	—	0.00	—	0.00	0.00
B ₂	$p = 1$	-0.27	1.2×10^{13}	—	0.00	—	0.00	—	0.00	0.00
	$p = 2$	1.00	2.0×10^8	—	0.00	—	0.00	—	0.00	0.00
C	$p = 1$	0.99	45.9905	1.00	1.002	1.00	1.8×10^7	—	0.00	0.00
	$p = 2$	0.96	0.026	1.00	1.003	1.00	1.8×10^7	—	0.00	0.00
D ₁	$p = 1$	0.98	94.582	1.00	1.001	0.31	1.3×10^{13}	—	0.00	0.00
	$p = 2$	0.91	0.047	1.00	1.001	1.00	4.2×10^7	—	0.00	0.00
D ₂	$p = 1$	1.00	0.86	1.00	0.038	0.95	1.1×10^9	—	0.00	0.00
	$p = 2$	0.99	2.1574	1.00	0.038	1.00	2.2×10^6	—	0.00	0.00
E ₁	$p = 1$	0.96	3.7×10^7	—	0.00	0.99	8.9×10^4	—	0.00	0.00
	$p = 2$	0.98	2.2×10^7	—	0.00	0.99	8.9×10^4	—	0.00	0.00
E ₂	$p = 1$	0.99	0.0088	—	0.00	1.00	0.0095	—	0.00	0.00
	$p = 2$	0.99	0.0098	—	0.00	1.00	0.0095	—	0.00	0.00
F	$p = 1$	1.00	0.69	—	0.00	1.00	1.2611	1.00	1.002	—
	$p = 2$	0.99	12.6607	—	0.00	1.00	1.2524	1.00	1.003	—
G	$p = 1$	1.00	2.3726	—	0.00	—	0.00	1.00	1.002	—
	$p = 2$	0.97	32.7006	—	0.00	—	0.00	1.00	1.003	—
H	$p = 1$	1.00	1.1951	—	0.00	—	0.00	—	0.00	$1.00 \times 8.2 \times 10^8$
	$p = 2$	1.00	37.0782	—	0.00	—	0.00	—	0.00	$1.00 \times 8.2 \times 10^8$

Table 7.6: R_a^2 and MSE for the resource usage profiles for the synthetic leak servers.

the goodness-of-fit of the projection for the entire data, and not only for the subset used in the linear regression, the coefficient of determination³ had to be *adjusted* to the sample size. An R_a^2 of 1.0 indicates that the regression line perfectly fits the data (some values are missing because of the regular resource usage). However, since the entire data is much larger than the 1024 injections used in the regression, the residual standard deviation can sum up to produce negative R_a^2 values. The other measure, [MSE](#), is the expected value of the square of the error. It measures the amount by which the estimator differs from the quantity to be estimated. When comparing two estimators for the same data, the one that gives a smaller

³The coefficient of determination (R^2) is the proportion of variability (defined as the sum of squares) in a data set that is accounted for by a statistical model.

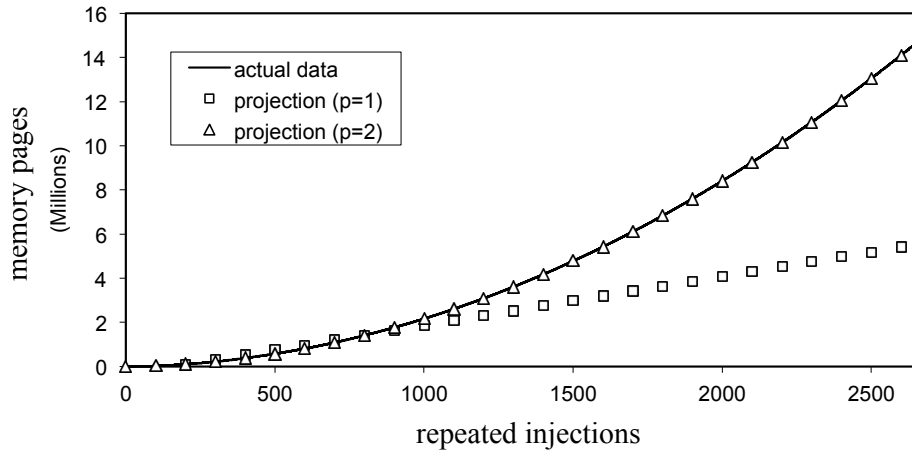


Figure 7.6: Memory usage profiles for the D_1 synthetic server (with thread leak).

[MSE](#) is better.

The table clearly shows that in all cases but three, the projections were extremely accurate (R_a^2 of approximately 1 and [MSE](#) for $p = 1$ and $p = 2$ in the same order of magnitude). The exceptions (highlighted in boldface) are the two CPU projections for the CPU leak servers, and the memory projection for the thread leak server (D_1). In all three cases a straight line projection ($p = 1$) did not correctly represent the actual resource usage data. Figure 7.6, for instance, shows the memory usage profiles and the actual consumption for the D_1 server. These results confirm our initial intuition that for certain types of resource consumption we may need curve-shaped projections. Overall, the experimental results show that linear regression with $p = 2$ estimated parameters produces valid and accurate projections for the entire lifetime of the server, i.e., until the depletion of the resources.

DNS experimental results

The experimental validation was conducted with seven known DNS servers: BIND 9.4.2 (Internet Systems Consortium, Inc. 1984-2012), MaraDNS 1.2.12.05 (Trenholme, 2001-2012), MyDNS 1.1.0 (Moore, 2002–2006), NSD 3.0.6 (NLnet Labs, 2002–2012), PowerDNS 2.9.21 (PowerDNS.COM BV, 2002–2012), Posadis 0.60.6 (Veeningen, 2002–2005), and rbindsd 0.996a (Tokarev, 2003–2008). All these servers are highly customizable, with several options that could affect the monitoring data gathered during the experiments. To make our tests as reproducible as possible, we chose to run the servers with no (or minimal) changes to the default configuration.

PREDATOR generated a total of 19,104 different attacks from the DNS protocol specification, using a test case generation algorithm that created message variations with illegal data. The exploratory phase repeated the injection of each attack 256 times and selected the best attack for each resource (i.e., the attack that caused higher consumption) to be used in a second injection campaign. In the exploitive phase, each of the selected attacks was injected 1024 times.

The final resource usage profiles (from the exploitive phase) are presented in Table 7.7. Four profile projections are highlighted in boldface, the higher CPU resource projection (BIND), the higher processes resource projection (PowerDNS), and a couple of increasing memory resource projections (MaraDNS and PowerDNS). The CPU increase is expected because as more tasks are executed, more CPU cycles are spent. However, it is interesting to note that the most CPU intensive server happens to be also the most used DNS server in the Internet, BIND. This means that BIND is more susceptible to a CPU exhaustion, i.e., the CPU has

Server	CPU M cycles	Processes	Memory pages
BIND-9.4.2	$\hat{y} = \mathbf{0.39x} + 25.31$	$\hat{y} = 1.00$	$\hat{y} = 1251.00$
MaraDNS-1.2.12.05	$\hat{y} = 0.18x + 4.85$	$\hat{y} = 1.00$	$\hat{y} = \mathbf{0.14x} + 170.85$
MyDNS-1.1.0	$\hat{y} = 0.14x + 0.21$	$\hat{y} = 1.00$	$\hat{y} = 494.00$
NSD-3.0.7	$\hat{y} = 0.02x + 0.78$	$\hat{y} = 3.00$	$\hat{y} = 534.00$
PowerDNS-2.9.21	$\hat{y} = 0.19x - 19.61$	$\hat{y} = \mathbf{0.01x} + 7.04$	$\hat{y} = \mathbf{2.40x} + 4983.49$
Posadis-0.60.6	$\hat{y} = 0.29x - 0.02$	$\hat{y} = 2.00$	$\hat{y} = 812.00$
rbldnsd-0.996a	$\hat{y} = 0.02x + 1.50$	$\hat{y} = 1.00$	$\hat{y} = 175.00$

(a) CPU, processes, and memory.

Server	File descriptors	Disk bytes
BIND-9.4.2	$\hat{y} = 0.00$	$\hat{y} = 0.00$
MaraDNS-1.2.12.05	$\hat{y} = 0.00$	$\hat{y} = 0.00$
MyDNS-1.1.0	$\hat{y} = 0.00$	$\hat{y} = 0.00$
NSD-3.0.7	$\hat{y} = 0.00$	$\hat{y} = 0.00$
PowerDNS-2.9.21	$\hat{y} = 0.00$	$\hat{y} = 0.00$
Posadis-0.60.6	$\hat{y} = 0.00$	$\hat{y} = 0.00$
rbldnsd-0.996a	$\hat{y} = 0.00$	$\hat{y} = 0.00$

(b) File and socket descriptors and disk usage.

Table 7.7: Resource usage profiles for the DNS servers.

no idle times, than the remaining target systems.

PowerDNS increases the total number of processes/threads from 7 to 8, which results in the highlighted processes projection. Further inspection, i.e., by running the exploitive phase with more injections, showed that the PowerDNS server was in fact limited to 8 processes/threads. The observed raise in the memory consumption profile was also due to the same cause, since the OS allocates memory when starting a new process/thread on behalf of the same application. Therefore, no vulnerability actually existed in both cases as the resource consumption eventually stabilized. This example demonstrates the usefulness of the last phase of the resource usage profile analysis—it allows the user to tradeoff some additional time executing extra injections on a small set of attacks, with a better accuracy of the resource profiles. In some exceptional cases, such as to confirm the attacks that potentially could exploit a vulnerability, we deliberately increased the

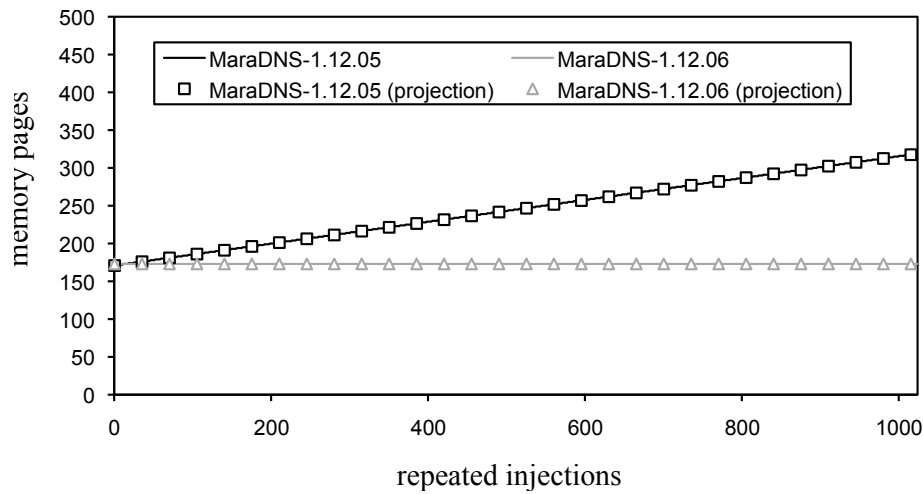


Figure 7.7: Memory consumption in MaraDNS.

number of injections (> 1024) to take the server closer to the exhaustion point.

Another [DNS](#) server, MaraDNS, also showed a raising memory usage profile. In fact, the memory consumption of MaraDNS is a clear example of a memory leak vulnerability. But the memory exhaustion was not restricted to the selected attack. Several of the generated attacks caused the same abnormal behavior, which allowed us to identify the relevant message fields that triggered the vulnerability. Any attack requesting a reverse lookup, or a non-Internet class records, made memory usage grow. A closer look at the server's code path of execution, showed that the server stops processing these queries once they are detected because they are not currently supported. However, the respective parsing function fails to free a couple of previously allocated variables. Successively injecting any of these two kinds of attacks caused the server to constantly allocate more memory, eventually requesting a new memory page. Both resource-exhaustion vulnerabilities could be exploited remotely to halt the server. They were deemed by the MaraDNS developers as fairly serious and credited to [PREDATOR](#). Figure 7.7

compares the projections for memory consumption of the vulnerable (1.12.05) and corrected (1.12.06) versions of the server.

7.3 REVEAL

REVEAL (*REvealing Vulnerabilities with bEhAvioral profile*) is a specialization of the **AJECT** architecture designed to infer a behavioral profile of the network server and to use it to identify potential vulnerabilities (Subsection 6.3.3).

To evaluate this approach, we used the tool to detect behavioral deviations in **FTP** network servers caused by vulnerabilities. We looked at all publicly available reports of **FTP** vulnerabilities and organized them into nine classes. Then, we selected some of these vulnerabilities whose fault pattern were representative of the different classes and devised five distinct experiments. Since the vulnerabilities occurred in different server applications, it was necessary to create several testbeds. The experiments show that an accurate behavioral profile can be an interesting solution for the automatic discovery of the flaws. Depending on the class of vulnerability, different sources of data were more insightful in the detection, suggesting that there are gains in employing a holistic approach to build the profiles.

7.3.1 Architecture and implementation

The architecture of the tool is depicted in Figure 7.8, which is based on **AJECT**'s architecture and extended with the construction and assessment of the behavioral profile. **REVEAL** defines two separate phases (see Figures 6.5 and 6.6 from Subsection 6.3.3). In a *learning phase*, a *Behavioral Profile Constructor* uses the

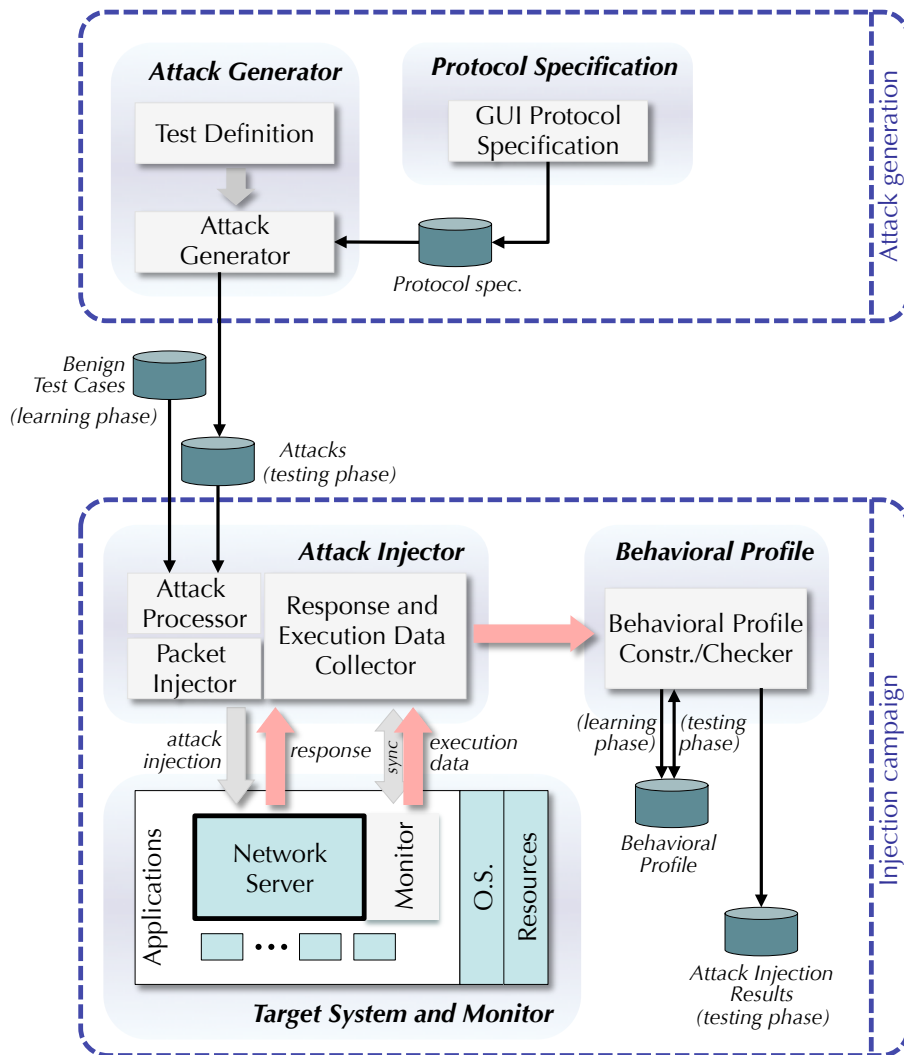


Figure 7.8: Architecture of the REVEAL tool.

network server's requests/responses and monitoring data gathered by the monitor component to build a Behavioral Profile. This phase starts by using a set of benign test cases to obtain an approximate protocol specification, resorting to the reverse engineering techniques presented in Section 4.3. Alongside the inference of the specification, the *Network Server* execution is continuously monitored, namely, its internal execution and resource usage data are traced and collected. The *Be-*

havioral Profile is obtained by extending the protocol specification (in particular, each transition of the input/output state machine) with the monitoring data. To later compare different instances of monitoring data, we resort to a condensed representation that maintains the maximum values of each resource utilization, the set of unique system calls and signals, and the sequence of system calls (specific examples are given in the evaluation).

In the attack injection campaign (or testing phase), the *Behavioral Profile Checker* is able to detect abnormal behavior by analyzing the requests, responses, and monitoring data that resulted from the test cases. It uses each input/output to follow the server's execution of the protocol and to identify the current state and transition. Then, it compares the server's current behavior with the one defined in the *Behavioral Profile* to look for inconsistencies.

7.3.2 Experimental evaluation

This section presents the results of the experimental evaluation of [REVEAL](#) to detect different classes of security vulnerabilities. For this purpose, we exhaustively searched the Internet for known [FTP](#) vulnerabilities in specialized websites like *SecurityFocus* (Bugtraq) and Common Vulnerabilities and Exposures ([CVE](#)), and we found a total of 122 reported flaws. The vulnerabilities were grouped in the following classes: buffer overflow, format string, [SQL](#) injection, external application execution, resource-exhaustion, [DoS](#), directory traversal, information disclosure, and default account vulnerabilities (see Table [7.8](#)).

Furthermore, we analyzed the typical fault patterns of each type of vulnerability (forth column in Table [7.8](#)) and devised a set of experiments covering the respective faulty behaviors (last column), such as the execution of a different set

Vulnerability	Bugtraq	CVE	Main faulty pattern	Experiment
Buffer overflow	113, 269, 599, 612, 679, 726, 747, 818, 961, 966, 1227, 1638, 1675, 1690, 1858, 2120, 2124, 2242, 2242, 2342, 2496, 2548, 2550, 2552, 2782, 3507, 3581, 3884, 4482, 5427, 7251, 7278, 7307, 8315, 8668, 8679, 8893, 9483, 9675, 9767, 9953, 10078, 10834, 11069, 11508, 11589, 11645, 11772, 12139, 12155, 12463, 12487, 12632, 12704, 12755, 13454, 14077, 14138, 14339	CVE-1999-0911, CVE-1999-1058, CVE-2001-0247, CVE-2001-0248, CVE-2001-0325, CVE-2001-0550, CVE-2003-0466, CVE-2003-0831, CVE-2004-0185, CVE-2004-0340, CVE-2004-2111, CVE-2004-2111, CVE-2004-2533	executes a different set of instructions	1, 3, 4
Format string	1387, 1425, 1560, 2296, 6781, 7776, 9800, 14380, 14381, 14381	CVE-2000-0573, CVE-2000-0574, CVE-2005-2390, CVE-2005-2390		
SQL injection	7974, 34288			
External app. execution	2240, 2241			
Resource exhaustion	271, 2698, 8875, 14382	CVE-2003-0853, CVE-2003-0854, CVE-2004-0341	resource consumption increases	3, 4
Denial of service	217, 859, 1456, 1506, 1677, 3409, 6297, 6341, 7425, 7473, 7474, 7900, 9573, 9585, 9627, 9651, 9657, 9668, 9980, 11065, 12384, 12790, 12865, 13054, 17398	CVE-1999-0838, CVE-2000-0644, CVE-2000-0645, CVE-2000-0647, CVE-2000-0648, CVE-2001-1156, CVE-2004-0252, CVE-2004-0342, CVE-2005-0779, CVE-2005-1034		
Directory traversal	301, 2444, 2489, 2618, 2655, 2786, 2789, 5168, 6648, 7472, 7718, 9832, 11159, 12586, 13292, 16321	CVE-2002-0558, CVE-2004-0148	output differs	2, 5
Information disclosure	1016, 1452, 2564, 3331, 3333, 7825, 11430, 13479, 14124, 14653	CVE-2000-0176, CVE-2000-0646		
Default account	5200, 7359, 20721			

Table 7.8: Reported known FTP vulnerabilities.

of instructions, an increase in some resource utilization, or an incorrect server response. For instance, vulnerabilities of buffer overflow, format string, [SQL](#) injection, or external application execution, force the server into inadvertently change its control-flow of execution, making it execute code that was not intended by the developers. Unusual resource consumption is a clear faulty characteristic

of resource-exhaustion vulnerabilities and also of [DoS](#) flaws (CPU/time is also a resource). Directory traversal and information disclosure vulnerabilities can be characterized by the server's behavior in granting illegal access to some private resources. This behavior is depicted in the server's output by responding positively when it should be denying access. The same faulty behavior is observed in misconfiguration issues, such as the existence of default accounts.

We then created a series of experiments based on specific security flaws that are representative of the faulty patterns of the different classes of vulnerabilities:

Experiment 1 – buffer overflow: a test case that triggers this vulnerability should either cause a memory access exception (e.g., SIGSEGV signal) or the execution of a different set of instructions.

Experiment 2 – directory traversal: this vulnerability can be detected by looking at the server's output (e.g., if it grants access when it should deny it).

Experiment 3 – [SQL](#) injection: this kind of vulnerabilities usually cause the server to execute more instructions, such as additional [SQL](#) queries, and consequently, to consume more resources, even though the server may still respond in accordance.

Experiment 4 – illegal program execution: executing an external program will cause the server to fork its execution: the main process will continue to handle the protocol requests, while a child process carries out additional instructions.

Experiment 5 – default account: configuration vulnerabilities, such as accessible default accounts, typically permit some illegitimate protocol transition that may

lead the server to reply and/or execute in some unexpected way.

Experimental environment

The experiments were devised to activate various security flaws that were reported in [FTP](#) servers and that can trigger distinct types of faulty behavior. Three of the five experiments are based on vulnerability reports for specific server implementations, while the other two are more general security problems, usually due to misconfiguration issues. To replicate the environment conditions described in the vulnerability reports, we created five distinct [VM](#) images (with Oracle's VirtualBox virtualization product). Whenever stated, we installed the appropriate version of the [OS](#), including the libraries and applications, and always compiled and installed the vulnerable version of the server from the source. In addition, we installed our monitor component and the required libraries in each [VM](#).

In the first and second experiment, we installed proftpd version 1.2pre1 (taken from the CVS repository) in two Ubuntu 10.10 [VM](#) images. The third experiment was also related to a vulnerability in proftpd server, however, it also required the use of a database—as described in the vulnerability report, we configured the [VM](#) with Debian 5.0 and MySQL 5.0. The forth experiment was related to a vulnerability present in wu-ftp, which we installed from the source files obtained from a mirror repository that contained old versions of the server. The server was configured to enable SITE EXEC commands, whose vulnerability would allow users to execute programs from `/bin` directory, instead of `~/bin`. Finally, for the fifth experiment, we used a recent build of proftpd (1.3.3) in the latest Ubuntu version (11.04) to detect the existence of default (and unwanted) accounts, such as the anonymous [FTP](#) account.

The same set of benign test cases was used to automatically infer the protocol specification and the internal server's behavior. To correctly infer the [FTP](#) specification with a good protocol coverage we used a total of 1000 messages taken from the same [FTP](#) trace used in the evaluation of ReverX in Subection [4.3.4](#). The subset of messages we selected contained the most used types of [FTP](#) requests, which exercised the server into executing a series of tasks while our tool collected the requests, responses, and monitoring data. The messages covered most of the protocol specification, including positive and negative responses (e.g., from requests with correct and incorrect parameters, such as wrong usernames or non-existent files).

For the injection campaign, and since we have already evaluated the attack generation approaches, we manually defined the test cases based on the proof of concept exploits that are provided in the vulnerability reports. This way, the evaluation is strictly focused on the assessment of the behavioral profile capabilities.

FTP experimental results

Figures [7.9–7.13](#) show the relevant results of the test cases that triggered vulnerabilities in each of the five experiments. For the purpose of clarity, we are ignoring some of the monitoring details, such as the actual sequence of systems calls or the utilization of some resources.

In the first experiment we tested a [FTP](#) server with a buffer overflow vulnerability (see Figure [7.9](#)). Normally, these vulnerabilities are easy to detect because they often crash the server while it tries to access an illegal memory address. However, modern [FTP](#) servers usually either create a child process to handle each

Experiment 1 - buffer overflow		
Server: proftpd 1.2pre1		
Vulnerability: buffer overflow in command MKD (bugtraq 612)		
	Behavioral Profile	Test Case
output	2.*	-
# processes	2 \pm 0	2
# unique syscalls	51 \pm 0	40
# seq of syscalls	1639 \pm 3	1015
seq of signals	5,19,13	5,19,11
memory usage	250	250
disk usage	0	0

Figure 7.9: Vulnerability detection of test cases (Experiment 1).

client and/or intercept this specific type of signals to gracefully terminate and automatically restart the execution. This effectively masquerades the fault from the clients and makes the detection unfeasible only by looking at the external behavior. However, our tool was able to detect these *monitoring violations* because it intercepted a different set of signals from those present in the behavioral profile for that particular protocol transition—the test case that triggered the abnormal behavior caused a signal 11 (SIGSEGV, segmentation fault). Additionally, the tool also provided evidence that the server executed significantly less code (i.e., less system calls), certainly due to the premature termination of the child process that received the signal 11. Moreover, since the server’s lack of response is not recognized by any of the acceptable responses for that protocol state, the tool identified a *transition violation* (instead of a simple output violation).

The second experiment is related to a directory traversal vulnerability and illustrates the necessity of looking at the execution of the protocol, i.e., at the messages exchanged (see Figure 7.10). The test case that triggered this vulnerability caused the server to respond affirmatively to an access request for a directory outside the user’s scope. The server responded with a 226 Transfer

Experiment 2 - directory traversal		
Server: proftpd 1.2pre1		
Vulnerability: accessing the root contents (bugtraq 2618, 2786, 5168, and 11159)		
	Behavioral Profile	Test Case
output	450 .*	226 Transfer complete.
# processes	2 ±0	2
# unique syscalls	49 ± 0	49
# seq of syscalls	1087 ±10	1244
seq of signals	5,19	5,19
memory usage	250	250
disk usage	0	0

Figure 7.10: Vulnerability detection of test cases (Experiment 2).

Experiment 3 - SQL injection		
Server: proftpd 1.3.1		
Vulnerability: SQL injection in USER command (bugtraq 33722)		
	Behavioral Profile	Test Case
output	331 .*	331 Password required for...
# processes	2 ±0	2
# unique syscalls	49 ± 0	49 ± 0
# seq of syscalls	1154 ±0	1704
seq of signals	5,19	5,19
memory usage	1180	1188
disk usage	10702	19903

Figure 7.11: Vulnerability detection of test cases (Experiment 3).

complete. when it should have denied access with a reply of the type 450 .*, thus it is an *output violation*. Because the server processed the request differently, it also executed a different sequence of system calls, thus the tool also identified a *monitoring violation*.

The following two experiments are related to vulnerabilities that cause the server to execute additional code. In the third experiment, a server vulnerable to SQL injection is lead to execute additional SQL queries (see Figure 7.11). The vulnerability, present in the USER command, cannot be detected by looking at the server's response alone because it always accepts the username parameter

Experiment 4 - illegal program execution		
Server: wu-ftpd 2.6.0		
Vulnerability: Illegal access in command SITE EXEC (bugtraq 2241)		
	Behavioral Profile	Test Case
output	200-.*	200-bash -c id
# processes	3 ± 0	5
# unique syscalls	30 ± 0	98
# seq of syscalls	1063 ± 0	1970
seq of signals	5, 5, 19, 19, 17	5,5,19,19,19,19,17,17,17
memory usage	794	1206
disk usage	0	0

Figure 7.12: Vulnerability detection of test cases (Experiment 4).

(whether it contains [SQL](#) statements or not). However, the expected behavior for processing a USER command is to execute 1154 system calls, as observed during the learning phase. Yet, the test case caused the server to execute 1704 system calls, i.e., an additional 550 system calls (*monitoring violation*). In addition, due the server's database logging mechanisms, our tool also detected a significantly larger number of bytes written to disk. This also illustrates the ability to detect discrepancies in the utilization of resources, such as memory or disk resource-exhaustion vulnerabilities.

The forth experiment also aimed at causing the server to execute additional code, although through the execution of an external program (see [Figure 7.12](#)). In this case, the server has a vulnerability that allows a remote user to run a program outside the user's *bin* directory. The previously inferred behavior of the server for processing a SITE EXEC command with an unauthorized program path, defined three processes and the execution of 1063 system calls. However, the test case caused the server to fork two additional child processes, to perform an additional 907 system calls, and to received four more signals, which is a result of the external program execution. These additional processes and code also

Experiment 5 - default accounts		
Server: proftpd 1.3.3		
Vulnerability: server accepts unexpected credentials (bugtraq 5200, 7359, and 10886)		
	Behavioral Profile	Test Case
output	530 .*	230 User anonymous logged in.
# processes	2 ±0	2
# unique syscalls	43 ± 0	47
# seq of syscalls	264 ±0	412
seq of signals	5, 19	5, 14 ,19
memory usage	610	1088
disk usage	1024	996

Figure 7.13: Vulnerability detection of test cases (Experiment 5).

caused the server to allocate more memory than the expected. Both discrepancies indicate a *monitoring violation* and therefore revealing that the test case triggered an existing fault in the server.

The fifth and last experiment is also related to a widely known class of vulnerabilities (see Figure 7.13). Some servers are configured by default with special accounts, such as for testing or debugging purposes, or simply with an anonymous account, which may also be considered a security risk. This particular experiment shows that besides the server's response being divergent from the expected on the behavioral profile (it accepts the credentials, when it should deny them), the server's internal execution is also in violation (it is carrying out a different protocol transition). In fact, the correct behavior for that test case would be to send a **530 .*** type of reply and to execute 264 system calls, which is being clearly violated by the server's execution (*output* and *monitoring violations*).

These five experiments demonstrate that although some vulnerabilities can be detected by looking solely at the server's output (i.e., experiment 2 and 5), other kinds of faulty behavior can only be detected if the server's internal execution is also observed and compared, such as by counting the number of system calls

and/or the amount of used resources. There are also a couple of points worth noting. First, as in any approach that learns the correct behavior and tries to detect any deviations as anomalies, it is susceptible to false positives. However, in the case of searching for security vulnerabilities it might be preferable to inspect a few false positives than to miss a false negative. Second, constantly probing the server for monitoring data is CPU intensive and may result in some overhead. We note, however, that our goal is to test network servers in a closed and controlled environment and not in live systems. Therefore, any overhead that may incur from the monitoring is negligible for detecting vulnerabilities and acceptable as it does not prevent the tests from being carried out at a good pace⁴.

7.4 Conclusions

This chapter presented three attack injection tools that implement many of the approaches described in the previous chapters. [AJECT](#) is an attack injection tool for the discovery of vulnerabilities in server applications. The tool is capable of generating numerous test cases and in carrying out an attack injection campaigns against a network server, while observing its execution. The monitoring information is later analyzed to determine if the server executed correctly or if it exhibited any suspicious behavior that suggests the presence of a vulnerability. The experimental evaluation with [AJECT](#) confirmed that it could detect different classes of vulnerabilities in e-mail servers and assist the developers in their removal by providing the required test cases. The sixteen servers chosen for the experiments were fully patched and up-to-date applications and most of them

⁴Typically, one test case takes one or two seconds to be executed, with a few more seconds to restart the environment conditions.

had gone through many revisions, making them challenging targets. In any case, [AJECT](#) successfully discovered vulnerabilities in five servers, which corresponded to 42% of all tested commercial applications.

[PREDATOR](#) is a specialization of [AJECT](#)'s architecture for the detection of resource-exhaustion vulnerabilities. The tool computes resource usage profiles to project the utilization of the monitored resources. Vulnerabilities related to the bad utilization of resources are automatically detected by the projected growth rate expressed in the profiles. The tool was experimentally validated with synthetic servers, which showed that it is quite suitable to profile different kinds of resource leaks. Moreover, the tool was used to test seven well-known public-domain [DNS](#) servers, which have been extensively tested throughout the years. Nevertheless, [PREDATOR](#) still found new vulnerabilities, which we believe is an important demonstration of the added value of the tool and of the solutions it implements.

[REVEAL](#) realizes an analysis method that infers and uses behavioral profiles to detect deviations in network servers, which are indicative of potential vulnerabilities. The tool defines two phases. During the learning phase, it infers a behavioral profile that models the server's correct execution of the protocol combined with local monitoring data. Then, it carries out the injection campaign, where it uses the profile as a reference to detect any deviation from the expected behavior. Violations to the behavioral profile indicate that a test case has triggered some fault and provide additional information about the faulty behavior. The experimental evaluation with [REVEAL](#) shows that since this approach uses external and internal sources of execution information, it is able to detect the different types of faulty behavior found in known [FTP](#) vulnerabilities, from wrong responses to

the execution of an unexpected set of instructions or to the unusual amounts of resource utilization.

CHAPTER 8

APPLICATIONS OF THE DEVELOPED TECHNIQUES

Some of the solutions and techniques that resulted from this work have also been applied to other areas of the security domain. This chapter presents two applications: one to assist the deployment of diverse [IT](#) systems and another to enhance [IDS](#).

[IT](#) services are usually implemented as replicated systems. If the server replicas execute identical software, then they share the same vulnerabilities and the whole system can be easily compromised if a single flaw is found. One solution to this problem is to introduce diversity by using different server implementations, but this increases the chances of incompatibility between replicas. We present a novel methodology to evaluate the compliance of diverse server replicas and to identify the various kinds of incompatibilities. This new approach collects

network traces to identify syntax and semantic violations, and to assist in their resolution. A tool called **DIVEINTO** was developed based on this methodology and applied to three **IT** scenarios. The experiments demonstrate that **DIVEINTO** is capable of discovering various sorts of violations, including problems related to nondeterministic execution.

Another explored area is related to the detection of intrusions in mission critical servers. An intrusion in a critical server can affect the security of an entire infrastructure that relies on it, including clients and other services. Hence, there is a constant concern in deploying and maintaining the correct execution of these servers. We present an approach for continuous monitoring of a server execution in an adaptive way, where fundamental tasks are thoroughly monitored, and compared against a previously inferred behavioral profile.

8.1 *DIVEINTO: Supporting Diversity in IT Systems*

Intrusion tolerance is a security and dependability paradigm that has been gaining momentum over the past decade because it lets system designers address both accidental faults and attacks in a seamless manner (Verissimo et al., 2003). A usual way to deploy an **IT** service is through a middleware that implements Byzantine state machine replication (Lamport, 1978; Schneider, 1990). Server replicas execute the same requests from the clients, and rely on the middleware protocols to carry out the coordination actions. The purpose of using replication is to keep the overall service correct even if some of the replicas are compromised. Typically, a system with n servers can tolerate up to f corrupted replicas, with $n \geq 3f + 1$ ¹.

¹Some authors have looked for particular solutions with lower bounds, e.g., $n \leq 2f + 1$, starting with the work of (Correia et al., 2004).

Malicious replica behavior is usually addressed by running the same request operation in all replicas, and then by selecting at the client the result that has more than a predetermined minimum number of votes ($f + 1$ equal votes).

Replication is normally carried out with the same software (e.g., OS and server application) at each (virtual) machine, so that correct servers produce identical results. However, in part due to their complexity, systems can remain to some extent faulty and/or vulnerable. Therefore, if an attacker is able to compromise one of the replicas, he or she can easily attack the remaining servers, defeating the original purpose of having replication. In the past, this issue has been mainly considered an orthogonal problem to the design of Byzantine replication protocols (Castro and Liskov, 2002; Correia et al., 2004; Abd-El-Malek et al., 2005; Cowling et al., 2006; Kotla et al., 2007). However, once the actual deployment of systems is considered, it becomes a fundamental problem for which there are very few practical approaches.

One potential solution is to take advantage of the inherent diversity provided by different software products that implement the same functionality, with the expectation that they are compromised in different ways. For example, at the OS level, one can employ distinct flavors of Unix and Windows (Garcia et al., 2011). Other good candidates are application level servers. For instance, there are several IMAP applications freely available on the Internet that could be utilized to build an IT e-mail service. However, distinct software implementations, even if complying with a formal specification (e.g., an IETF protocol specification), may behave differently and/or respond with small variations due to specification gaps, incorrect implementation or misconfiguration. Moreover, diversity also increases the difficulty of ensuring determinism on the execution of the replicas, an impor-

tant requirement to keep the various servers with consistent states. All these issues can have an impact on the replica responses, by causing them to look different, and thus preventing the required quorum on the results from being reached. In this case, the client is left with a number of responses from which it cannot distinguish the correct ones from the malicious, consequently stopping the progress of the computation.

This section studies various types of violations based on the content of the server replies, and presents a new methodology to evaluate the compliance of diverse *IT* replica configurations. The methodology employs network traces with the messages exchanged among the clients and the servers to identify syntax and semantic violations, and to assist developers in their resolution. We also developed a tool called *DIVEINTO*, based on our methodology, and used it to evaluate three different *IT* scenarios. In each of these scenarios, we resorted to a diverse set of servers (*FTP*, *SMTP*, and *POP*). The experiments demonstrate that *DIVEINTO* is capable of discovering various kinds of violations, some of them related to syntax and semantic errors, and others to nondeterminism in the executions.

8.1.1 Overview of an *IT* system

An *IT* replicated system offers a service to a group of clients under very weak failure assumptions (see Figure 8.1). Typically, it is assumed that the adversary can perform various attacks on the network (e.g., replay, delay, re-order, corrupt messages), and that he or she controls an undetermined number of clients and up to f servers (for a total of $n \geq 3f + 1$ replicas). Nevertheless, even under these challenging conditions, the system needs to provide correct responses to the good clients.

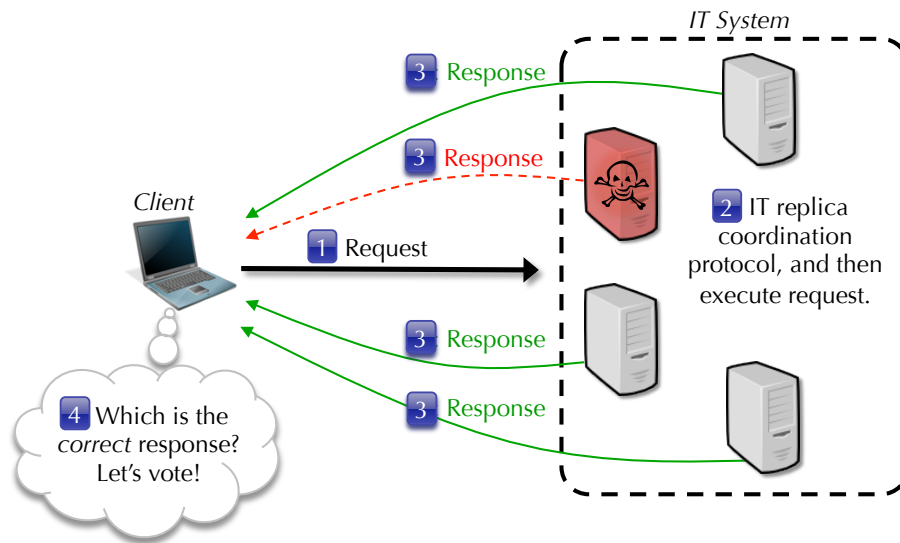


Figure 8.1: IT system architecture.

The design and implementation of an IT system has to address two main concerns. First, the protection of the communication between the client and the servers, where it is necessary that all correct replicas execute the request (step 1 of the figure) and then that the client is able to select a correct response (step 3). By employing re-transmissions and cryptographic methods, it is possible to secure the messages from the network attacks and ensure their delivery. The selection of the response is a bit more delicate because some of the replies might be produced by malicious replicas, and therefore contain erroneous data. Consequently, the usual procedure is to wait for $f + 1$ equal responses, and only then choose this answer (this ensures that at least one correct replica vouches for the reply). If no response has such a quorum, then the client is unable to use the service because there is no generic mechanism to distinguish correct from wrong answers².

²The client could for instance carry out validity checks on the data to exclude the wrong answers. However, in general this sort of solutions is ineffective because we assume that the malicious replicas are controlled by a human that can produce data to evade the checks.

Second, all correct servers should start to execute from an identical initial state, and then they should evolve through the equivalent states as they process the requests (Schneider, 1990). This is achieved by running a Byzantine replication protocol among the servers (step 2), which associates a processing order number to each request and makes sure the correct servers carry out the same requests. More formally, a server can be modeled by two functions: $f_t : S \times I \rightarrow S$ is the *transition function* that maps the current state and input to the corresponding next state; and $f_o : S \times I \rightarrow O$ the *output function* that maps the current state and input to the corresponding output, with S being the set of finite states, I being the input messages that a server can receive (i.e., the requests) and O being the respective output (i.e., the responses). The replica implementation should guarantee that correct servers always exhibit an equivalent behavior by returning similar replies, which implies that the state transition and output functions among the correct replicas should be indistinguishable.

8.1.2 Classification of violations

Syntax and semantic violations

Servers offer a service accordingly to some specification. Whenever a replica's output differs from what is expected, we call it a *syntax violation*. More specifically, we denote $f'_o : S \times I \rightarrow O'$ with output $O' \neq O$ as a non-compliant output function that returns a different response from the other replicas for the same current state and input. If a given replica has an output function $f'_o(s, i) \neq f_o(s, i)$ for some $s \in S$ and $i \in I$, we say it has a syntax violation in state s for input i .

Messages that are not in conformance with the syntax of the specified protocol

are clear syntax violations. However, there might be some gaps in a protocol specification that delegate a few decisions to the developers, such as the content of welcome banners or optional response descriptions, which naturally lead to differences among the various implementations. Such dissimilarities, however, should not occur in a replicated system, even if the internal state is the same across all replicas, because otherwise they may prevent voting mechanisms.

Another type of violation concerns with the meaning of the message, as opposed to its actual format. If the meaning of a message is different from the expected, we say that a *semantic violation* has occurred. Semantic violations are more serious because they typically reflect a deviation in the internal state of the replica. Hence, a semantic violation is usually, but not necessarily, also accompanied by a syntax violation. We denote semantic violations when the replica's transition function f'_t is different from f_t , i.e., when $f'_t(s, i) \neq f_t(s, i)$ for some $s \in S$ and $i \in I$.

Semantic violations might arise without violating the syntax of the messages, i.e., the content of messages might be the same, although one replica might be in a different state. This unusual circumstance might occur when the same kind of response is used in different states, such as when the responses are a simple yes or no. Nevertheless, if the replicas' state differs, their responses to the clients will eventually also be distinct as the execution progresses (otherwise, no problem exists for the clients).

Naturally, the cause for violations is linked to the actual implementation or configuration of the servers. Errors or differences in messages can be attributed to several reasons, from which some examples are:

- Incompatible configurations: an option to refuse a username without the

domain (*syntax violation*) or to support some particular feature (*semantic violation*);

- Undefined behavior by the specification: an additional explanation of the response (*syntax violation*) or an optional state in the implementation (*semantic violation*);
- Incorrect implementation: a misspelled word (*syntax violation*) or an error that leads the server into an incorrect state (*semantic violation*);
- Nondeterminism: a message field with the date or time (*syntax violation*) or a random number that leads the servers to divergent states (*semantic violation*).

Mitigation measures

Both syntax and semantic violations have an impact on the correctness of the [IT](#) system and ultimately they can prevent clients from utilizing the service because the required quorum on the answers might not be reached. Our solution is aimed at identifying violations between replicas and providing the necessary additional information to eliminate them. Below, we discuss the main approaches to remove the violations:

Eliminate at the source: Removing a syntax or semantic violation at its source requires specific knowledge about the server execution, configuration, or actual implementation. If $f_o : S \times I \rightarrow O$ is the output function that models the desired responses of the replicas and $f'_o : S \times I \rightarrow O'$ the output function of an invalid replica that produces a different output, removing the syntax violation at the source means changing the actual replica so that its output function f'_o con-

forms to f_o . For example, a welcome banner returned by a replica may be the source of a syntax violation if a server name is different from the expected. Many times, this problem can be corrected by changing the server configuration value to the right name. A semantic violation, such as a missing feature, could be removed by enabling the corresponding functionality in the configuration of the server or by implementing it from scratch, which is the equivalent of correcting the replica's non compliant transition function f'_t into f_t .

Normalization: Another approach to resolve a violation is to change the output of the invalid replica, in order to normalize the replies so that they conform to the other replicas. Being f'_o the incorrect output function of a replica that produces invalid output $O' \neq O$ for the same input I , this approach consists in creating a normalizer function $norm : O' \rightarrow O$ that transforms invalid responses into correct ones, and thus $norm(f'_o(s, i)) = f_o(s, i)$, $\forall s \in S, i \in I$. However, one must take into account that since the normalizer uses the output of the invalid replica, $f'_o(s, i)$, it must get the information required to construct $f_o(s, i)$. For example, if the invalid response, $f'_o(s, i)$, does not contain an expected identifier, then it may be difficult for the normalizer to produce $f_o(s, i)$.

Circumventing: A violation can also be prevented from occurring. If the request types that the replica is failing to process correctly are optional or not essential to the overall replicated service, the violation can be removed by blocking those requests of all replicas. A firewall component in front of the replicas could provide ingress filtering to ensure that only the requests that can be correctly processed by all servers are allowed. This firewall can be modeled as limiting the input set to $J \subset I$, so that $f'_o(s, j) = f_o(s, j)$ and $f'_t(s, j) = f_t(s, j)$, $\forall s \in S, j \in J$.

8.1.3 Methodology

There is a reasonable amount of research dedicated to the analysis of software components. One way to study the behavior of a replica is to look for instance at its internal execution and state, the contents of the files it reads or writes, and the resources it is using. However, performing *white-box* analysis on a replicated system with diversity can be extremely difficult. For one, as the number of replicas increases, so does the cost to analyze their behavior. Additionally, distinct server implementations may not be directly comparable. For instance, one server might use a single large file to maintain some of its state, while another server uses a hierarchy of several files, and still another server resorts to a database system.

Another way to study the behavior of a software component is to look at it as a black-box, and examine its input and output interfaces. Since this approach can be built around the specification of the component under analysis, it is well-suited for modeling the behavior of the replicas without having to scrutinize their internal states and configurations.

The approach we propose examines the behavior of the replicas by inspecting and comparing their state transition and output as they process the same input. If the servers deviate from the expected state, eventually this will be reflected in their behavior and thus in their output. Our methodology is depicted in Figure 8.2, and in general it consists in capturing the behavior of a reference replica and in verifying if other replicas conform to it.

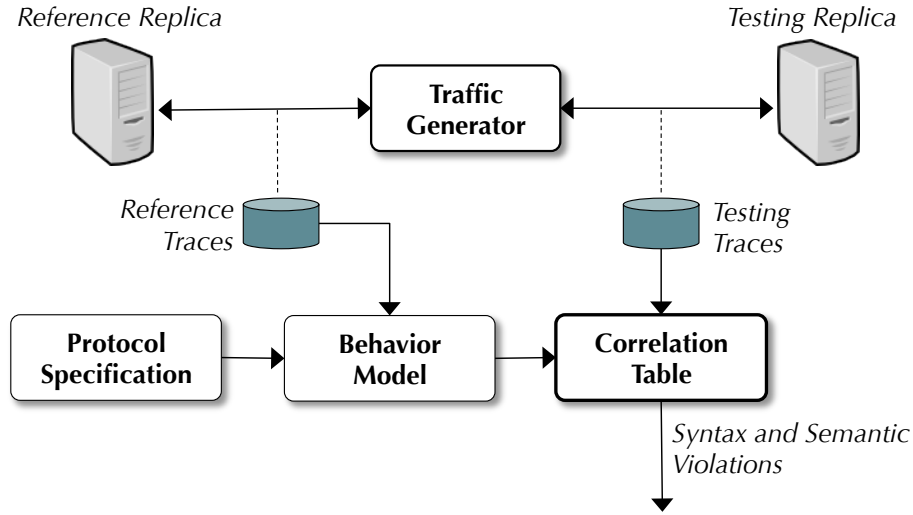


Figure 8.2: Methodology overview.

Detecting violations

Our methodology resorts to a specification of the protocol that the replicated system uses to communicate with the clients and to network traces that were collected from the replicas (see Figure 8.2). *Protocol Specification* formally defines the format of the messages (protocol language) and the rules for exchanging those messages between the clients and the servers (protocol state machine). A Mealy machine $(S, I, O, f_t, f_o, s_0, F)$ is a well-suited formal representation for the protocol specification:

S is a finite, non-empty set of states,

I is the input alphabet (finite set of requests),

O is the output alphabet (finite set of responses),

f_t is the transition function: $f_t : S \times I \rightarrow S$,

f_o is the output function: $f_o : S \times I \rightarrow O$,

s_0 is the initial state, and

F is the set of final states.

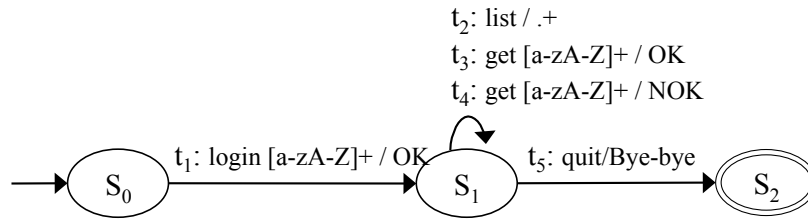


Figure 8.3: Input/output state machine of an example protocol.

This specification can be obtained in several ways. If the replicas use an open protocol and its standard is available (e.g., a standard [RFC](#)), then it can be translated into a [FSM](#). For closed protocols, however, an approximate specification must be inferred, using for instance reverse engineering techniques (Chapter 4).

The protocol specification is an *incompletely defined* [FSM](#) because it models the behavior of the client and the server for only a relevant set of transitions that are necessary to understand the protocol operation and to create compliant implementations—it does not necessarily define for every state and input, a next state and an output. Some parts of the protocol execution are usually loosely described or omitted, giving the developers some liberty for the implementation. For instance, some specifications may not be clear about how a server should reply to malformed protocol messages, and therefore, some implementations may choose to return an error message while others may opt not to respond. Another example is when the specification defines message types with optional descriptions that are server-dependent, such as welcome banners or the [FTP](#) protocol replies.

Let us consider a simple example. Figure 8.3 shows the specification of a protocol (input/output state machine) defined by a Mealy machine. Each transition defines an input and an output. For instance, transition t_1 in state s_0 accepts re-

quests of the type `login [a-zA-Z]3` and responses of the type `OK`. If successful, the protocol jumps to state s_1 where it expects three different transitions for the commands `list` and `get`. The command `get` in particular can take a parameter of the type `[a-zA-Z]+` and two types of responses: `OK` or `NOK`. This specification is an incompletely defined machine because it does not define transitions for the entire input and state space. For instance, the behavior is undefined if the client sends a `get` command without parameters or in state s_0 .

A server that implements a protocol specification, on the other hand, is a *completely defined* machine because its behavior is fully realized by its particular implementation and configuration—for every state and input, it defines a next state and an output (which may be null). Our methodology obtains the *Behavior Model* of a reference replica by creating a completely defined machine based on the *FSM* of the protocol specification, with the difference that it redefines the original transition and output functions (f_t and f_o) to accept only the messages from the reference replica.

The behavior model is constructed using network traces containing client-server interactions together with the protocol specification. Therefore, the coverage of this model depends on the network traces. A trace containing *all* possible client-server interactions would yield a completely defined machine with the entire protocol coverage. However, this is unpractical (if not impossible) and it may not even be desirable. In fact, in order to limit the exposure of the replicas, we may wish to restrict the service provided by the *IT* system to a subset of the servers' functionality. In this case, we would also like to limit the evaluation of

³We resort to regular expressions to represent message fields that can take different values depending on the specific message. In this case, this field corresponds to the *username* of the *login* command.

nth	Request _n	Response _n	Response' _n
1	login user1	OK	OK
2	list	file1	file1
3	quit	Bye-bye	Good bye
4	login user2	OK	OK
5	list	file2	file1, file2
6	get file2	OK	OK
7	quit	Bye-bye	Good bye
8	login user1	OK	OK
9	get file2	NOK no file	NOK no file
10	quit	Bye-bye	Good bye

Table 8.1: Reference and testing traces.

the diverse replicas to the respective subset of the protocol specification.

Returning to the example, the reference trace from Table 8.1 (columns “Request_n” and “Response_n”) and the protocol specification can be used to build a behavior model that completely defines the transition and output functions for the entire trace. The behavior model is shown in the second column of the correlation table in Table 8.2. One can see, for instance, that the behavior model defines transitions t_{11} and t_{12} at state s_0 based on the original transition t_1 of the protocol specification. These transitions accept respectively the first, the fourth, and the eighth requests/responses from the traces (column “nth”), and take the protocol to state s_1 (column “Protocol specification”). The behavior model defines these restrictive transitions to accept *only* the input/output present in the reference trace.

A *Traffic Generator* is used to exercise the replicas into performing equivalent kinds of tasks that are expected to be executed on the IT system. The traffic generator’s purpose is to perform each protocol task and transition individually, and therefore it is implemented as a single-threaded client, which is unrelated to the IT system support for multiple and concurrent connections. The generated re-

Protocol specification	Behavior model	nth (cont.)
$t_1(s_0, \text{"login [a-zA-Z]+/OK"}) = s_1$	$t_{11}(s_0, \text{"login user1/OK"}) = s_1$ $t_{12}(s_0, \text{"login user2/OK"}) = s_1$	1, 8 4
$t_2(s_1, \text{"list/.+"}) = s_1$	$t_{21}(s_1, \text{"list/file1"}) = s_1$ $t_{22}(s_1, \text{"list/file2"}) = s_1$	2 5
$t_3(s_1, \text{"get [a-zA-Z]+/OK"}) = s_1$	$t_{31}(s_1, \text{"get file2/OK"}) = s_1$	6
$t_4(s_1, \text{"get [a-zA-Z]+/NOK"}) = s_1$	$t_{41}(s_1, \text{"get file2/NOK ..."}) = s_1$	9
$t_5(s_1, \text{"quit/Bye-bye"}) = s_2$	$t_{51}(s_1, \text{"quit/Bye-bye"}) = s_2$	3, 7, 10

(a) First three columns.

nth	Response' _n	Conformance
1, 8	OK, OK	true
4	OK	true
2	file1	true
5	file1,file2	false
6	OK	true
9	NOK ...	true
3, 7, 10	Good bye, Good bye, Good bye	false

(b) Last three columns.

Table 8.2: Correlation table (violations in boldface).

quests and the observed responses are saved as the network traces. Our approach is flexible enough to control the degree of protocol coverage that will be evaluated. Therefore, this component must create the protocol requests that cover the desired part of the protocol specification, allowing the evaluation to focus on a subset of the replicas' functionality.

The requests produced by the traffic generator are used as input to both the *Reference Replica* and the *Testing Replica*. The messages exchanged with the reference replica are stored as the *Reference Trace*, and then they are utilized to construct the behavior model. Messages transmitted between the traffic generator and the testing replica are saved as the *Testing Trace*, and they are used to correlate the behavior of the testing replica with the behavior model. By giving the same input on both replicas, we ensure that their behavior can be directly compared—if

their transition and output functions are equivalent, then if they start in the same state and receive the same input, their output and next state (which is confirmed by the subsequent behavior) should also be identical.

Our approach to determine if the testing replica conforms to reference replica consists in comparing the responses of the testing replica with the behavior model. To compare the responses, we resort to a *Correlation Table* that correlates the order in which the responses appear in the traces (which is the same for both replicas since the requests were sent in the same order) with the expected transitions of the behavior model and the protocol specification. Thus, in every state, the n th request/response of the reference replica should be identical to the n th request/response of the testing replica. If the behavior model rejects the respective messages from the testing replica, we have a violation and we mark the transition of the protocol specification. Marked transitions mean that the testing replica is not equivalent to the reference replica for that part of the protocol.

In the previous example, we use the order of the request/response (columns “ n th” of Table 8.2) to obtain the transition of the behavior model (second column) that is expected to accept the corresponding response of the testing trace (column “Response’ $_n$ ”). If the transition rejects the response, which is basically a matter of comparing the response of the testing replica with the one from reference replica, it means that the testing replica is not complying with the behavior of the reference replica for the corresponding transition of the protocol specification (first column). For instance, the fifth response from the testing trace (file1, file2) is rejected by the transition t_{22} , which was expecting a message identical to the fifth response of the reference replica (file2). This violation indicates that the testing replica is not complying to the reference replica’s execution of that transition of the protocol

specification, i.e., in the command `list`, thus we mark transition t_2 as *false* (last column).

In addition, protocol states that are only reached by marked transitions are also considered to be in violation in the testing replica. Therefore, we also mark any transitions of the protocol specification leaving these states.

The methodology is able to detect all syntax violations automatically. To find out semantic violations, one has to look at the violations to understand if the messages differ because of a semantic difference. For instance, the violations in transition t_5 of the protocol specification are related to the behavior model expecting a response of the type `Bye-bye` and getting the response `Good bye` from the testing replica (last row of Table 8.2). This is just a syntax violation because messages are semantically expressing same thing—the state of both replicas is the same. On the other hand, the violation triggered in the 5th request/response shows that the behavior model was expecting an output of `file2` and it got instead a `file1, file2`. Here, not only is the text different, but it also indicates that both replicas have semantically different views at what should be the same state—one replica just sees `file2`, while the other sees the files `file1` and `file2`. Hence, this is also a semantic violation.

The methodology can also discover nondeterminism issues in responses, such as dates, times, or random numbers. To find out this type of violations, we get additional versions of the testing trace from the same replica at a different date and time. Each version consists of the same sequence of protocol requests produced by the traffic generator and should also have the same sequence of responses. In this case, we add extra columns to the correlation table (one for each additional version of the responses of the testing replica) and compare the different versions.

The comparison of responses from distinct trace versions allows the detection of discrepancies that are caused by nondeterminism. If the responses have some date or time field, or even a value derived from a random number or event, they will differ among the various versions.

The detected violations can then be resolved (i.e., eliminated at the source, normalized, or circumvented) by the [IT](#) architect or administrator, by looking at the form of violation, the message type, and the protocol state in which they appear.

8.1.4 Tool implementation

We developed a tool in Java, called [DIVEINTO](#), that implements the methodology to detect inconsistencies among diverse replicas. The tool automatically discovers syntax violations and provides additional information to help a human operator to identify semantic violations and to devise ways to mitigate them. This section provides an overview of the implementation details.

The traffic generator is implemented by a specific scripted client for a given protocol. A scripted client is an automated program that performs a sequence of predetermined tasks in order to interact with a server in a deterministic way. For instance, we developed a scripted client that executes a sequence of [FTP](#)-related tasks on a server, from navigating in the remote file system to downloading a particular file. While the scripted client interacts with each replica, a packet capture tool is used to collect and store the entire communication.

Currently, we are focusing our research in text-based protocols from the [IETF](#). Hence, we resort to a reverse engineering technique to infer an approximate specification of the protocol used by the replicas. [DIVEINTO](#) can derive automatically

a protocol specification based only on the reference trace (using the approach described in Section 4.3). For protocols that we are unable to infer at this moment, we have developed a graphical tool that supports the manual specification of the protocol, by creating a *FSM* that recognizes the protocol language and state machine.

After obtaining the protocol specification (inferred or manually), *DIVEINTO* derives the behavior model from the reference trace and uses the testing trace to detect violations as described in the methodology. In the end, the tool produces a report with the transitions of the protocol specification that it marked with violations and asks the operator of the tool to identify semantic violations. For every marked transition, the tool shows two regular expressions, one created from the messages of the reference replica and another from the messages of the testing replica. This helps the operator to determine in which state the violation occurred and if there was a syntax or/and semantic violation. Listing 8.1 and Figure 8.4 in the next section show an excerpt of the tool's output. This information is complemented with a trace of the requests that allowed the discovery of the violation, so that it can be reproduced if needed.

8.1.5 Experimental evaluation

To evaluate the methodology, we used *DIVEINTO* in three *IT* scenarios with the protocols: *FTP*, *SMTP*, and *POP*. These protocols are open standards from the *IETF* that have a large user base, but are quite difficult to replicate in an *IT* system. One of the challenges is that their specification is vague and undefined at times. For instance, most server replies contain a variable-length text field where the developers can further detail the reason of the response. Also, some protocol replies

OS	FTP	SMTP	POP
Fedora 14	wu-ftp	Sendmail	UW IMAP
Ubuntu server 10.04	Pure-FTPd	Postfix	dovecot
Windows XP SP2	IIS5	Surgemail	Surgemail
Windows Server 2003 SP2 Enterprise Edition	IIS6	Exchange Server 2003	Exchange Server 2003

Table 8.3: OS and server configuration of the three IT scenarios.

can be composed of an unspecified number of messages that must begin with the same reply code. This incompletely specified behavior creates an additional barrier when trying to implement a replication solution because the behavior of each replica must be directly comparable in order to detect intrusions.

The experimental procedure for each IT scenario follows our methodology: we generate and collect network traffic from a pair of replicas, which is used to derive the behavior model of the reference replica and to detect violations of the testing replica. In addition, the tool also infers an approximate protocol specification from the network traces of the reference replica. Each of the four replicas was evaluated both as a potential reference replica and as a testing replica.

Replication Scenarios

Each IT scenario simulates an IT system with four diverse replicas, with different OS and server implementations. Every replica is executed in a virtual image using VirtualBox⁴ and was configured with the minimal hardware and software requirements to run the respective server (FTP, SMTP, or POP). Table 8.3 shows the various configurations of the experimental scenarios. Despite the different software, each replica was started with the same initial state, by configuring every server as similar as possible (same authentication scheme, user accounts, user

⁴<http://www.virtualbox.org/>

files, e-mail messages, etc.).

The first scenario was set with four diverse [FTP](#) server replicas. As explained previously, [FTP](#) is a communication protocol for exchanging files between a client machine and a server ([Postel and Reynolds, 1985](#)), supporting many commands to remotely manipulate files and directories on the server. The communication is typically done in two separate [TCP](#) connections, a control connection where the clients and server exchange commands and status replies, and a data connection that the server uses to send data through. The data connection is established by either the client (active mode) or the server (passive mode). The second scenario involved four replicas with different [SMTP](#) implementations. [SMTP](#) is an [IETF](#) standard for outgoing electronic mail transport ([Klensin, 2008](#)). Email clients can send e-mails by connecting to the destination [SMTP](#) server or to a local or intermediary server that will relay the e-mail message to the destination [SMTP](#) server. And finally, the third scenario corresponds to a replicated [POP](#) service. [POP](#) is a standard protocol for retrieving e-mail messages from a remote server ([Myers and Rose, 1996](#)). Email clients can access their electronic mailbox by connecting to the [POP](#) server and by providing their credentials.

One scripted client was created for each [IT](#) scenario to generate the necessary traffic. The [FTP](#) scripted client executes 36 distinct [FTP](#) sessions using the most representative [FTP](#) commands, such as successful and unsuccessful login attempts, logouts, directory listings, traversing the remote file system, getting information from existing and non-existing files, and manipulating files and directories, which results in 354 [FTP](#) requests. To identify the [FTP](#) requests more likely of being used in a replicated system, we analyzed networks traces from 320 pub-

lic **FTP** servers⁵ and selected the 15 different **FTP** commands that accounted for more than 99% of the requests. To detect nondeterminism, we collected traces of each replica at two distinct dates and times, totaling 10,954 **FTP** packets.

The **SMTP** traffic generator creates 20 **SMTP** sessions with the most relevant commands, ranging from simple connections to the composition of malformed e-mail messages, and includes the successful delivery of 7 e-mails. Unfortunately, we did not find any public traces with the full **SMTP** payload, so we analyzed the most common behavior of e-mail clients of our laboratory while interacting with **SMTP** servers. We found the most representative **SMTP** commands to be: HELO, EHLO, MAIL, RCPT, DATA, RSET, VRFY, EXPN, HELP, NOOP, and QUIT. As with the **FTP** scenario, **SMTP** traffic was collected from each replica at two different days to support the identification nondeterminism violations. A total of 1640 packets were obtained from the **SMTP** replicas.

The **POP** scripted client produces 14 different sessions employing the mostly used **POP** commands (10 out of 12 commands), taken from an similar analysis of the most common commands of e-mail clients (e.g., retrieve, delete, list). Traffic data for each replica was collected at two different dates and times. The network traces accounted for a total of 900 **POP** packets.

Results on the identification of violations

Tables 8.4, 8.5, and 8.6 show a summary of the violations identified by **DIVEINTO** between each pair of replicas in the three **IT** scenarios. Each server acts a reference replica in one subset of the experiments. The tables provide values for the total number of distinct transitions of the inferred protocol specification (row

⁵<http://ee.lbl.gov/anonymized-traces.html>

Reference		IIS5	IIS6	Pure-FTPd	wu-ftp
IIS5	transitions	–	40	40	40
	syntax	–	4	38	20
	semantic	–	4	3	4
	nondeterm.	–	1	4	3
	rejected msgs	–	72/1580	858/1580	1133/2395
IIS6	transitions	40	–	40	40
	syntax	4	–	38	17
	semantic	4	–	1	1
	nondeterm.	1	–	4	3
	rejected msgs	72/1580	–	858/1580	1105/2395
Pure-FTPd	transitions	47	47	–	47
	syntax	45	45	–	45
	semantic	3	1	–	1
	nondeterm.	1	1	–	3
	rejected msgs	858/1580	858/1580	–	1717/2395
wu-ftp	transitions	38	38	38	–
	syntax	20	17	36	–
	semantic	4	1	1	–
	nondeterm.	1	1	5	–
	rejected msgs	318/1580	290/1580	902/1580	–

Table 8.4: Violations detected on the FTP replication scenario.

“transitions”), and on how many of those transitions were marked with violations (rows “syntax”, “semantic”, and “nondeterministic”). In addition, we provide the number of request/response messages from the testing trace that were rejected over the total count of messages exchanged between a pair of replicas (row “rejected msgs”).

The information about the number of rejected messages gives evidence of one of the benefits of our methodology. Without the methodology, an operator would normally have to look at each one of the messages to understand why and where they differ from the expected. For instance, the evaluation between the IIS5 server (reference replica) and the wu-ftp (testing replica) in Table 8.4, yielded 1133 different request/response messages between the replicas, which

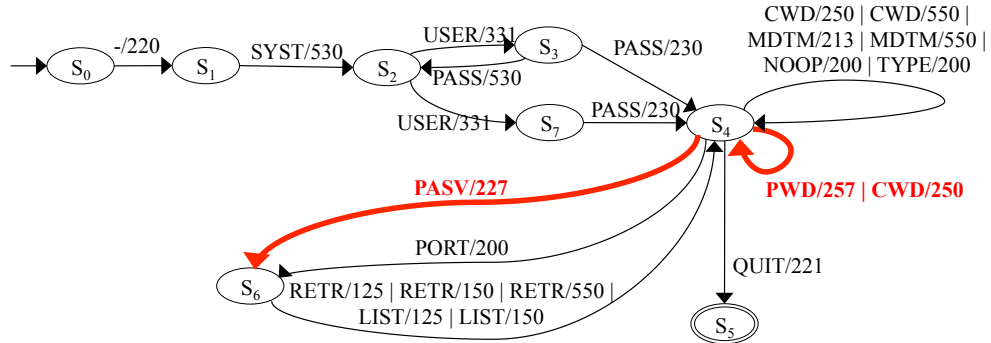


Figure 8.4: Behavior model for FTP server IIS6 with the identified conformance violations from IIS5 (in boldface).

corresponds to 20 syntax violations. **DIVEINTO** automatically clusters the 2395 messages in the 40 inferred transitions, where it detects inconsistencies in 20 of them. This approach allows an operator to look at a manageable number of clusters of messages, which pertain to a single state and type of message, instead of hundreds of messages without any additional information.

Table 8.4 demonstrates that **DIVEINTO** found several violations in every pair of **FTP** servers. The tool correlates the identified violations with the state and transition of the protocol specification, and provides regular expressions to help a human operator to determine the existence of semantic violations (and eventually to mitigate them). As an example, let us look more closely to a specific pair of **FTP** servers, the Internet Information Services version 6 (IIS6) and the Internet Information Services version 5 (IIS5). Figure 8.4 shows the IIS6 inferred protocol specification with the violation transitions (drawn in boldface) that were found when comparing with the IIS5 replica's testing trace. As expected, **DIVEINTO** found most of the protocol execution to be identical because replicas were running similar server implementations. However, it detected some violations related to the **FTP** commands **PASV**, **CWD**, and **PWD**.

```

1 State S4; Transition PASV/227
2   iis6 : 227 Entering Passive Mode (10,10,5,163,4,*).\r\n
3   iis5 : 227 Entering Passive Mode (10,10,5,31,19,1.*).\r\n
4 (Non-determinism detected!)
5
6 State S4; Transition CWD/250
7   iis6 : 250 CWD command successful.\r\n
8   iis5 : 550 /priv: The system cannot find the file specified.\r\n
9
10 State S4; Transition PWD/257
11   iis6 : 257 "/" is current directory.\r\n
12   iis5 : 257 "/" is current directory.\r\n
13
14 State S4; Transition PWD/257
15   iis6 : 257 "/p.*" is current directory.\r\n
16   iis5 : 257 "/" is current directory.\r\n

```

Listing 8.1: Detected violations between IIS6 and IIS5.

Listing 8.1 exhibits DIVEINTO's output when detecting the violations between IIS6 and IIS5. The majority of the discovered problems were syntax violations, mostly due to responses that conform to the transition function, but do not comply with the output function. This type of violations are typically easier to resolve, however, we will focus on semantic violations because are typically more difficult to analyze and to mitigate.

The tool found the first violation in State S4 (Lines 1–4), for transitions that accept PASV requests and replies with response code 227⁶. Although the regular expressions created from the responses of both replicas (Lines 2 and 3) show that they correspond to the same message type (i.e., 227), they still reveal different payloads and nondeterministic behavior. The PASV command is a special command to instruct the server to transfer data in *passive mode*. In this mode, the server replies with a free port from which it will send the data. Since the port

⁶For the sake of readability, the label of the transitions is trimmed to the first word of the request and response.

Reference		Exchange	Postfix	Sendmail	SurgeMail
Exchange	transitions	–	31	31	31
	syntax	–	31	31	31
	semantic	–	6	5	6
	nondeterm.	–	1	5	5
	rejected msgs	–	218/396	218/396	218/396
Postfix	transitions	25	–	25	25
	syntax	25	–	25	25
	semantic	6	–	3	2
	nondeterm.	2	–	2	5
	rejected msgs	218/396	–	218/396	218/396
Sendmail	transitions	33	33	–	33
	syntax	33	33	–	33
	semantic	5	3	–	3
	nondeterm.	7	1	–	5
	rejected msgs	218/396	218/396	–	218/396
SurgeMail	transitions	29	29	29	–
	syntax	29	29	29	–
	semantic	6	2	3	–
	nondeterm.	2	1	2	–
	rejected msgs	218/396	218/396	218/396	–

Table 8.5: Violations detected on the SMTP replication scenario.

number is not defined by the RFC specification, a server can choose any free port that with high probability will not be the same across the replicas. Therefore, this violation is also a semantic violation because it affects the state of the server.

In the second violation (Lines 6–8) the responses of each replica correspond to different types of messages (i.e., 250 and 550). Since both responses are syntactically and semantically different, the operator can safely determine that this is also a semantic violation. The remaining two problems are also semantic violations, although the type of the response messages is the same (response code 257). By inspecting the part of the messages in which they differ (i.e., / and / . *), one can conclude that they display different the path names, which implies conflicting states (e.g., some file or directory is missing or the current directory is different).

Reference		Dovecot	Exchange	SurgeMail	Uw-imap
Dovecot	transitions	–	22	22	22
	syntax	–	20	21	21
	semantic	–	1	0	1
	nondeterm.	–	10	10	10
	rejected msgs	–	80/190	104/190	104/190
Exchange	transitions	23	–	23	23
	syntax	21	–	22	22
	semantic	1	–	1	0
	nondeterm.	12	–	12	12
	rejected msgs	80/190	–	104/190	104/190
SurgeMail	transitions	22	22	–	22
	syntax	21	21	–	21
	semantic	0	1	–	1
	nondeterm.	6	6	–	6
	rejected msgs	104/190	104/190	–	104/190
Uw-imap	transitions	23	23	23	–
	syntax	22	22	22	–
	semantic	1	0	1	–
	nondeterm.	12	12	12	–
	rejected msgs	104/190	104/190	104/190	–

Table 8.6: Violations detected on the POP replication scenario.

As it turns out, the last three semantic violations (i.e., Lines 6–16) are the result of the lack of the *user isolation* feature⁷ present in IIS servers since version 6.

DIVEINTO also found problems on the other IT scenarios, some of them related to malformed protocol requests (see Tables 8.5 and 8.6). For instance, several SMTP servers refused RCPT or MAIL requests with non-existent e-mail addresses, while other servers accepted those requests. Sendmail was the only server to accept RCPT commands with an e-mail address from unknown servers, and all but the Exchange Server rejected RCPT commands with an e-mail address of a non-existent local user. Additionally, the Exchange Server was the only

⁷FTP user isolation prevents users from viewing or overwriting other users' Web content by restricting users to their own directories. Users cannot navigate higher up the directory tree because the top-level directory appears as the root of the FTP service.

one to allow [SMTP](#) HELO/EHLO requests without the server address (response with 250 message code), and Dovecot and Sargemail recognized a [POP](#) USER command without the username parameter, while the other two servers did not.

An interesting nondeterminism violation was discovered in Pure-FTPd. The tool detected a syntax violation in one of the [FTP](#) commands that was due to an *easter egg*—a hidden message—introduced by the developers. The message randomly appears in replies to the RETR command: 150-Connecting to port 5000\r\n150 The computer is your friend. Trust the computer\r\n226 file successfully transferred\r\n. This is an example of an unspecified behavior that could not have been detected by looking at the [FTP](#) protocol specification or by inspecting the configuration of the server.

8.1.6 Case-study

In this section, we focus on resolving the conformance violations of two [FTP](#) servers. We present this case-study as an example of how the tool can be used to identify and resolve conformance violations. We chose the servers IIS6 and Pure-FTPd because they illustrate well the point of having diversity in a replicated system. IIS6 is a Microsoft server that provides a [FTP](#) service for Windows Server 2003, as opposed to Pure-FTPd that only runs on Unix-based machines. Additionally, the evaluation of the Pure-FTPd identified a surprising syntax violation – a hidden message that appeared randomly in the traces.

To resolve the conformance violations between the two replicas we have to look at each one individually and decide which is the better approach (Subsection 8.1.2). Therefore, we analyzed all 38 syntax violations and the semantic violation between IIS6 and Pure-FTPd. We used only information provided by

the [DIVEINTO](#) to determine how to address each violation.

In the previous section we already identified the semantic violation, which is related to the [FTP](#) passive mode (PASV command). This violation could be resolved by any of the approaches pointed in Subsection [8.1.2](#), though some of them may require a greater effort. For instance, a patch to each server could potentially be created to allow the replicas to deterministically choose the port for the passive connection (removal at the source). Another approach would be to use a normalizer to change the output of the server to match the same port number as the other replica (mitigation at the output). Ultimately, we could just block any PASV requests, effectively preventing the replicas from behaving differently (circumventing the violation).

The remaining conformance violations detected in Pure-FTPd are related to the server's welcome banner and to 12 other [FTP](#) commands. These violations are syntax-only and can be simply resolved through a normalizer component that modifies the Pure-FTPd's responses to comply with the ones from IIS6. In some isolated cases, the normalizer must also be applied to the IIS6's responses to remove information that is missing on the responses provided by the Pure-FTPd server (e.g., the total number of bytes transferred after a RETR command).

We implemented a response normalizer in Python for the Pure-FTPd server. The normalizer could run in the same machine as the server, and it intercepts the requests from the clients, relays them to the Pure-FTPd server, and then gets the responses, transforms them so that they comply with the responses of IIS6, and sends them back to the clients. In Listing [8.2](#) we show some of the transformations that the normalizer performs on the responses from the Pure-FTPd replica, as well as the information provided by [DIVEINTO](#) (in the comments) that allowed

```

1 def fixResponse(request, response):
2
3     # Welcome banner
4     if re.search("^$", request):
5         # State S0; Transition : -/220
6         # IIS6: 220 Microsoft FTP Service\r\n
7         # Pure-FTPd: 220----- Welcome to Pure-FTPd.... [trimmed]
8         m = re.search('220-----\sWelcome\s\sto\sPure-FTPd.*', response)
9         if m: return "220_Microsoft_FTP_Service\r\n"
10
11     # SYST request
12     if re.search("^SYST", request):
13         # State S1; Transition : SYST/530
14         # IIS6: 530 Please login with USER and PASS.\r\n
15         # Pure-FTPd: 215 UNIX Type: L8\r\n
16         m = re.search('215_UNIX_Type:\sL8', response)
17         if m: return "530_Please_login_with_USER_and_PASS.\r\n"
18
19     # USER request
20     if re.search("^USER", request):
21         # State S2; Transition : USER/331
22         # IIS6: 331 Password required for .*\r\n
23         # Pure-FTPd: 331 User .* OK. Password required\r\n
24         m = re.search('331_User(.*)\sOK\sPassword_required', response)
25         if m: return "331_Password_required_for_" + m.group(1) + ".\r\n"
26
27         # State S2; Transition : USER/331
28         # IIS6: 331 Anonymous access allowed, send identity (e-mail name) as password.\r\n
29         # Pure-FTPd: 230 Anonymous user logged in\r\n
30         m = re.search('230_Anonymous_user_logged_in', response)
31         if m: return "331_Anonymous_access_allowed_send_identity(e-mail_name)_as_password.\r\n"
32     (...)
33
34     # RETR request
35     if re.search("^RETR", request):
36         # State S6; Transition : RETR/150
37         # IIS6: 150 Opening BINARY mode data connection for .*(\d+ bytes).\r\n226 Transfer (...)
38         # Pure-FTPd: 150.*Connecting to port 5000\r\n.*226 File successfully transferred\r\n
39         # State S6; Transition : RETR/150
40         # IIS6: 150 Opening ASCII mode data connection for *.html(\d+ bytes).\r\n226 Transfer (...)
41         # Pure-FTPd: 150.*Connecting to port 5000\r\n.*226 File successfully transferred\r\n
42         m = re.search('150.*Connecting\sto\sport_5000.*226_File_successfully_transferred', response)
43         if m: return "150_Opening_data_connection.\r\n226_Transfer_complete.\r\n"
44     (...)
45
46     return response

```

Listing 8.2: Excerpt of the normalizer (Python).

us to code them. Function *fixResponse* in the normalizer receives as input the FTP request that was sent to the Pure-FTPd and its respective response. For each violation the normalizer checks the type of the request and modifies the response accordingly. Since DIVEINTO provides regular expressions that capture the message content of the responses, the transformations are almost straightforward.

The first two transformations are trivial replacements for the welcome banner (Lines 4–9) and for the SYST request (Lines 12–17). In both cases the normalizer just returns the correct response if the request and the response match the regular expressions provided by DIVEINTO.

Other transformations involve using data from some fields of the original responses, such as the first transformation for the USER command (Lines 20–25). In this transformation, the normalizer must use the username provided in the response by using a regular expression with the capturing group `(.*)` (in Line 24) and retrieved with `m.group(1)` (in Line 25). The second transformation for the USER command is also a simple substitution.

The next transformation is related to the RETR command (Lines 35–43). We present this transformation for three reasons: (1) Pure-FTPd's response does not provide all information necessary to make it match the IIS6's response; (2) Pure-FTPd's responses match two different types of IIS6's responses; and (3) occasionally, Pure-FTPd server inserts a hidden message in its responses. For the resolution of this violation we had to analyze the type of responses from both servers. Because of (1) we cannot transform messages from one server into the other due to missing data (i.e., the `(\d+ bytes)` part in the response from IIS6). Therefore, we have to normalize both servers into the same common response, which in this case means to omit the total number of bytes. Also, because of (2) we would

have to convert messages that match one regular expression into a message that could have two different descriptions (i.e., either with `BINARY` or `ASCII`). However, and since we are already normalizing the response from IIS6, we could also merge the two descriptions into a more general one by leaving out the `BINARY` mode and `ASCII` mode. Finally, we just ignore the hidden message of (3), which is captured in one of the `. *` in Line 42.

The remaining transformations, omitted in Listing 8.2, were also resolved with simple regular expression substitutions, assisted by the information provided by the tool.

8.2 *Adaptive IDS for Critical Servers*

Nowadays, there is a significant reliance on several network servers for the execution of certain services, ranging from distributed name resolution to vital control operations in critical infrastructures. While providing these services, the servers are subject to a large number of threats and attacks, either carried out from the outside or internally, due to their necessary exposure.

However, despite the many efforts, vulnerabilities and exploits continue to surface everyday, compromising all kinds of servers. Often, when newer forms of attack are being used, intrusions are only detected when the server behavior is conspicuously incorrect or malicious, e.g., with a crash or due to client complains about abnormal actions. Still, a *post-mortem* analysis on the server can usually trace back the failure to the point where its behavior started to deviate from the normal execution. Therefore, through a detailed monitoring and analysis, it should be possible to detect the intrusion right from the first steps of the

attack. This has been tried by some IDS in the past, but these solutions are limited by a superficial model of the servers' behavior, such as logging the number, frequency and type of network connections and correlate that information with the users and other resources in the system (Paxson, 1999; Roesch, 1999).

We briefly present an approach that automatically provides an accurate and detailed model of the normal execution of a server, which is suitable to be used as a frame of reference to identify malicious behavior in production systems. The solution resorts to reverse engineering techniques to infer a model of the specific protocol being implemented by the server, which includes the expected formats of the messages (both requests and responses) and the relations among them (e.g., a USERNAME request should come before the PASSWORD). This model is augmented with comprehensive monitoring data, capturing for instance the resources and code that is executed while the server processes requests of a certain type. The result is a detailed *behavioral profile* of the server that allows the detection of an incorrect execution when an intrusion occurs.

8.2.1 Methodology

Our solution consists in learning a behavioral profile of an application server (*learning phase*) and in verifying if the server is operating in accordance with it (*operational phase*). The behavioral profile is an accurate model of the correct behavior of the server that captures how it follows the protocol and operates internally. The general architecture of our tool in both phases is presented in Figures 8.5 and 8.6.

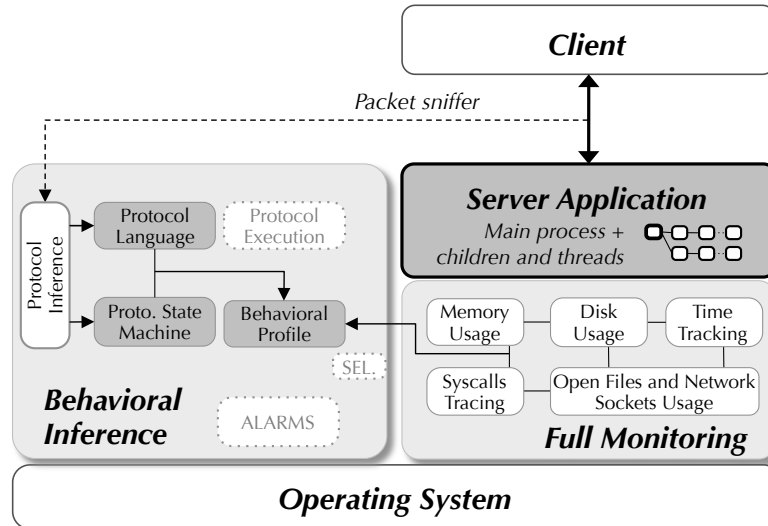


Figure 8.5: Inferring the behavioral profile (learning phase)

Learning phase – inferring the behavioral profile

During the learning phase, the Protocol Inference component listens to the traffic coming to and from the server (see Figure 8.5). Based on the collected messages, it uses a protocol reverse engineering technique to infer a specification of the protocol implemented in the server, which is capable of understanding the format of the messages (the Protocol Language) and their relations (the Protocol State Machine) (Section 4.3).

In parallel, a specialized internal monitor collects various kinds of data about the server execution. For example, the maximum amount of allocated memory *during* and *after* the handling of a request and the list of accessed files. The observed sequence of system calls and signals provides a pattern of the execution of the server for a given task. If a server executes a different set of instructions from the expected, such as when its flow of control is being hijacked, this will be revealed by a different set of system calls (Warrender et al., 1999; Leon et al.,

2005).

The Behavioral Profile component constructs a model that combines the inferred specification of the implemented protocol with the local monitoring data. We use a Mealy FSM for the model because it can represent the interactions between the two protocol entities (input/output). This type of automaton defines both a transition function (i.e., give the next server state for a given state and a client request) and an output function (i.e., the server response for a given state and request). The FSM is augmented with monitoring data that is requested from the monitor component at specific points of the server execution, in particular after replying to the clients. This monitoring data, therefore, depicts the server's progress with regard to the last monitoring request and provides a snapshot of the server's current internal state. Hence, besides the request and response, we also associate to each transition of the FSM the respective monitoring information (M_i). M_i corresponds to a tuple $(source_1; \dots; source_n)$, where each $source_i$ captures one of the dimensions of the server internal execution.

Operational phase – using the behavioral profile

In the operational phase, the tool uses the previously learnt Behavioral Profile to evaluate the server's real-time execution (see Figure 8.6). However, monitoring is an intensive task that may introduce a non-negligible overhead and affect the server's performance. Therefore, we resort to an *adaptive monitoring* approach, where only a subset of the protocol space is thoroughly monitored—the monitor gathers and collects internal execution information only at certain previously selected FSM transitions. The amount of monitoring data and which monitoring agents are used can also be adapted, thus providing a greater flexibility and

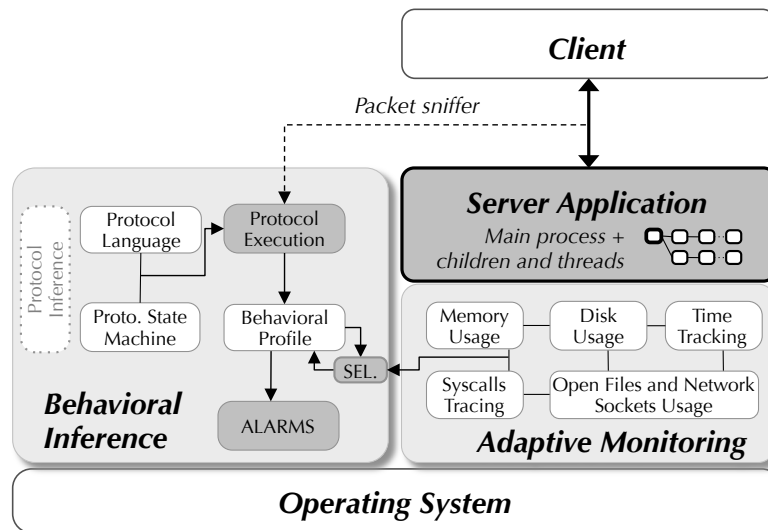


Figure 8.6: Using the behavioral profile (operational phase)

control over the monitoring process.

The Protocol Execution component analyses the incoming and outgoing traffic to identify the current state and transition of protocol and to verify if the client or the server are correctly following the protocol execution. If the server receives a protocol request that is not accepted by the protocol language of the client (e.g., malformed message) or by the protocol state machine (e.g., the current state does not accept that type of message), an alarm is raised. This helps to identify malicious clients that are not following the correct execution of the protocol. In addition, the server responses are also evaluated against the server protocol language and the protocol state machine. This allows the tool to track the server's protocol state and to identify any deviant behavior in the protocol execution of both parties, such as when an attacker tries to exploit some vulnerability in the server or as soon as the server is compromised and is responding to the attack in an unexpected (and therefore suspicious) way.

If a client-server interaction correctly follows the protocol execution, the tool

checks which monitoring data was selected for that particular transition (*SEL* component) and requests the respective monitoring agents to collect internal data. The data is then sent back and compared against the Behavioral Profile. Any deviant behavior is identified as a potential intrusion and an alarm is raised. For instance, the behavioral profile might define that for a particular protocol transition, the server executes 12 system calls, allocates 23 memory pages, and writes around 100k bytes to disk. An alarm is triggered if the server's memory usage is above that maximum threshold while executing that protocol transition (e.g., a potential memory leak). Or if an unusual number system calls and disk usage is observed, indicating that a different set of instructions is being executed (e.g., [SQL](#) injection attack). This approach automatically correlates different monitoring sources with the protocol execution, potentially improving the precision of the detection.

8.3 Conclusions

This chapter introduced two applications of some of the solutions and techniques that resulted from this work. The first application is related to the introduction of diversity in [IT](#) systems, which is a challenging and difficult problem because replicas must behave similarly and execute in a deterministic way. We presented a new methodology for the automatic discovery of inconsistencies among diverse replicas, and to assist on their correction. We also developed a tool, [DIVEINTO](#), that is able to infer the behavior of each replica by analyzing network traces collected from their execution, which is then utilized to identify syntax and semantic violations. We used [DIVEINTO](#) to evaluate different server implementations in three [IT](#) scenarios ([FTP](#), [SMTP](#), and [POP](#)), where it was able to automatically

detect a large number of violations.

The second application is related to the protection of critical servers against intrusions. These kind of servers play an important role in the organizations and thus deserve special intrusion detection measures. However, most *IDSs* are limited to passive monitoring solutions because thorough monitoring incurs in great overhead. This work introduces a solution for the detection of intrusions that is based on a behavioral profile, which captures the execution of the protocol together with the server's internal monitoring data. During the operational phase, the behavioral profile is compared against the observed execution to discover deviations that could indicate that an attack/intrusion is taking place. To minimize the impact on the overall performance, only the most sensitive part of functionality is selected to be thoroughly monitored.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

This chapter summarizes the main contributions of the work with regard to the attack injection framework, focusing on the various aspects that had to be addressed to produce a workable solution for the discovery of security vulnerabilities in network servers. It also provides some future research directions, namely to extend the monitoring capabilities with source code inspection and resource usage correlation, or to efficiently generate more focused attacks, for instance, by learning what kinds of vulnerabilities may or may not exist by exploring the inferred behavioral profile.

9.1 *Conclusions*

The thesis describes a general framework that encompasses the various phases associated with the injection of attacks, from the protocol specification and generation of test cases, to the respective injection and monitoring, and the analysis of the results.

The first contribution is related to obtaining a specification of the target system, which is used in several steps of the attack injection methodology. In the context of this work, the target system is a server that provides some service and communicates with its clients through the network. Therefore, the specification of the system can be modeled after the specification of the protocol that the server implements. Attack injection may be used in diverse kinds of servers without access to the source code, and possibly without a description of the protocol that is implemented. To address this issue, we propose a reverse engineering approach to automatically derive a protocol specification using samples of the communication. This solution has been implemented and released as an open source tool, ReverX¹, which has been successfully used to derive the specification some IETF protocols, namely FTP, SMTP, and POP.

Another contribution that resulted from the thesis is related to the generation of attacks, which are test cases that aim at covering as much of the input domain of the target system as possible. For that reason, the process of test case generation resorts to the specification of the protocol (manually defined or previously inferred) to create a finite set of attacks that may discover existing vulnerabilities. Two approaches to the test case generation problem are presented. One

¹<http://code.google.com/p/reverx/>

solution employs a combinatorial generation of test cases with four algorithms to test different aspects of the protocol messages. An alternative solution recycles existing test cases (and known exploits) to reuse them as testing payloads. This approach allows the testing of systems that are not supported by existing tools or that have been extended with new features. Experiments with ten [FTP](#) testing tools and ten other sources of test cases for non-[FTP](#) protocols, show that in both scenarios our approach is able to get better or equal protocol and vulnerability coverage than with the available tools. In an experiment with two known fuzzer frameworks, this approach showed an improvement of 19% on the vulnerability coverage when compared with their combined utilization, thus being able to detect 25 additional vulnerabilities.

Another problem tackled in this work is the monitoring and the automated analysis of the results. It is through the detection of anomalous behavior of the target system that the vulnerabilities can be discovered. The first major challenge was therefore to distinguish between correct and incorrect behavior. Three complementary approaches were studied. One starts by defining the anomalous behavior and explicitly looks for it in the attack result logs and monitoring data. Another solution computes resource utilization profiles and automatically reveals the most acute ones. A last solution looks for deviations in a behavioral profile. A behavioral profile is learnt from the target system's regular operation by automatically inferring a protocol specification combined with internal execution data. This behavioral profile is then checked in the injection phase in order to detect any unexpected behavior.

Some of the techniques mentioned above were integrated into three injection tools, [AJECT](#), [PREDATOR](#), and [REVEAL](#). [AJECT](#), which was released as an open

source project², provides a framework that implements different injection and monitoring approaches and that can be further extended. Our evaluation with sixteen heterogeneous e-mail servers, fully patched and up-to-date, confirmed that **AJECT** could detect different classes of vulnerabilities and assist the developers in their removal—five previously unknown vulnerabilities were discovered, corresponding to 42% of all tested commercial applications.

PREDATOR is an attack injection tool that takes a different injection approach and extends the monitor's capabilities to detect small resource usage variations. Thus, besides injecting the attacks continuously, the tool also computes resource usage profiles that predict the utilization of every monitored resource. This tool was experimentally validated with synthetic servers and with seven well-known public-domain **DNS** servers. Despite the fact that these were stable and extensively tested **DNS** servers, the tool still found a memory leak and identified the types of **DNS** requests that were most susceptible to resource-exhaustion problems.

REVEAL is an attack injection tool that implements the behavioral profile analysis to automatically detect faulty behaviors that occur during the activation of vulnerabilities. The experimental evaluation focused on the ability of the tool to successfully identify the different faulty patterns of several known **FTP** vulnerabilities.

Some of the contributions that resulted from this investigation were also applied to other domains of network security. The first of these applications is a methodology for the automatic identification of incompatibilities among diverse replicas, which was inspired by the protocol reverse engineering approach pre-

²<http://code.google.com/p/aject/>

sented here. A tool implementing this methodology, [DIVEINTO](#), successfully identified several violations among [FTP](#) replicas, which if not properly addressed, would prevent their utilization in the same [IT](#) system. The other application uses an inferred behavioral profile to augment the detection capabilities of [IDS](#). While this approach is still very preliminary, it shows good promise.

9.2 Future Work

The thesis described the main challenges of attack injection, and then went into solving them, opening new avenues for research. Future works can develop other tools and methodologies to support the development of more secure and dependable computer systems by building on these ideas and further explore them. We provide here some of the possible research directions:

White-box approach

In this work, attack injection was mainly used as a black-box testing approach, or at most as a grey-box solution because some monitor implementations are able to trace the internal execution of the binary. However, none of the developed monitoring solutions resorted to the source code of the target system.

Although not always available (specially in proprietary systems), having access to the source code can be useful, in particular to estimate the coverage of the tested code. This metric can then be used to evaluate the attack injection outcomes, quantifying the confidence of the results and other dependability metrics (e.g., how many vulnerabilities were discovered and what was the code coverage). Moreover, code coverage can also contribute to guide the generation of

attacks in order to maximize the testing coverage with the minimum number tests.

Correlation between different types of resources

PREDATOR is an attack injection tool focused on detecting vulnerabilities related to the bad management of resources. The results obtained with this tool hinted to the fact that the utilization of resources is not completely independent. Most resources are intrinsically co-related and their utilization may affect other types of resources. For instance, creating additional threads or child processes causes the **OS** to allocate memory and execute more CPU instructions. Due to this relation, identifying which resource is the source of the exhaustion problem can be a challenging task. One future research direction could be to extend the methodology implemented by **PREDATOR** to analyze the correlation of the different resources in order to more accurately identify the cause of the flaw.

Using the behavioral profile to generate more focused attacks

Other future works may build on top of the inferred behavioral profile (Section 6.3.3) to create more effective attacks. The behavioral profile can provide very detailed information about the protocol execution and the target system's internal execution, such as a sequence of intercepted system calls. Depending on the observed internal execution during the learning phase, the injected attacks can be built to target specific types of vulnerabilities. For instance, by identifying system calls in the execution path related to memory allocation, attacks can be specifically created with large payloads to verify if there are buffer overflows vulnerabilities. In addition, these special-purpose attacks can be restricted to the

parts of the specification (e.g., protocol requests) for which they are suitable.

Adaptive IDS

Another research line, which was briefly addressed, is the creation of more sophisticated protection mechanisms. This thesis presented some ideas about an adaptive [IDS](#) that is capable of providing different levels of inspection (packet-level and application-level) depending on which part of the protocol space is being executed. This approach is especially useful for critical servers whose correct execution is of paramount importance.

BIBLIOGRAPHY

- ABD-EL-MALEK, M., GANGER, G., GOODSON, G., REITER, M., and WYLIE, J. "Fault-Scalable Byzantine Fault-Tolerant Services". In *Proceedings of the ACM Symposium on Operating Systems Principles*. Vol. 39. 5. 2005, pp. 59–74.
- ABELL, V. A. *Isof – LiSt Open Files*. <http://people.freebsd.org/~abe/>. 2012.
- ADLER, S. "The Slashdot Effect: An Analysis of Three Internet Publications". In *Linux Gazette* (<http://ldp.dvo.ru/LDP/LG/>), issue 38. 1999.
- AIDEMARK, J., VINTER, J., FOLKESSON, P., and KARLSSON, J. "GOOFI: Generic Object-Oriented Fault Injection Tool". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2001, pp. 83–88.
- ALBINET, A., ARLAT, J., and FABRE, J.-C. "Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2004, pp. 867–876.
- ALESSANDRI, D. "Attack-Class-Based Analysis of Intrusion Detection Systems". PhD thesis. University of Newcastle upon Tyne, Newcastle, UK, 2004.
- AMMANN, P. and OFFUTT, J. *Introduction to Software Testing*. Cambridge University Press, 2008.
- ANDERSON, R. and NAGARAJA, S. *The Snooping Dragon: Social-Malware Surveillance of the Tibetan Movement*. Tech. rep. UCAM-CL-TR-746. University of Cambridge, 2009.

- ANDERSSON, C. and RUNESON, P. "Verification and Validation in Industry – A Qualitative Survey on the State of Practice". In *Proceedings of the International Symposium in Empirical Software Engineering*. 2002, pp. 37–47.
- ARLAT, J., COSTES, A., CROUZET, Y., LAPRIE, J.-C., and POWELL, D. "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems". *IEEE Transactions on Computers*, 42(8) (1993), pp. 913–923.
- ARLAT, J., CROUZET, Y., and LAPRIE, J.-C. "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1989, pp. 348–355.
- ARLAT, J., CROUZET, Y., KARLSSON, J., FOLKESSON, P., FUCHS, E., and LEBER, G. H. "Comparison of Physical and Software-Implemented Fault Injection Techniques". *IEEE Transactions on Computers*, 52(9) (2003), pp. 1115–1133.
- AUTOSEC TOOLS. *FuzzTalk Fuzzing Framework*. <http://www.autosectools.com/Page/FuzzTalk-Guide>. 2012.
- AVIZIENIS, A. and RENNELS, D. "Fault-Tolerance Experiments with the JPL STAR Computer". In *Proceedings of the IEEE Computer Society Conference*. 1972, pp. 321–4.
- AYEWAH, N., HOVEMEYER, D., MORGENTHALER, J., PENIX, J., and PUGH, W. "Using Static Analysis to Find Bugs". *IEEE Software*, 25(5) (2008), pp. 22–29.
- BACKUS, J. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference". In *Proceedings of the International Conference on Information Processing*. 1959.
- BARBARÁ, D., WU, N., and JAJODIA, S. "Detecting Novel Network Intrusions using Bayes Estimators". In *Proceedings of the SIAM International Conference on Data Mining*. 2001.
- BAULIG, M. and KACAR, D. *LibGTop*. <http://ftp.gnome.org/pub/GNOME/sources/libgtop/>. 2012.
- BEARDSLEY, T. *Manual Protocol Reverse Engineering*. <http://www.breakingpointsystems.com/community/blog/manual-protocol-reverse-engineering/>. BreakingPoint Systems. 2009.
- BEDDOE, M. A. "Network Protocol Analysis Using Bioinformatics Algorithms". <http://www.4tphi.net/~awalters/PI/PI.html>. 2005.
- BEIZER, B. *Software Testing Techniques*. 2nd. Van Nostrand Reinhold, 1990.

- BETOUIN, P. P. *BSS (Bluetooth Stack Smasher)*. <http://www.secuobs.com/news/05022006-bluetooth10.shtml>. 2006.
- BIEGE, T. *Radius Fuzzer*. <http://www.suse.de/~thomas/index.html>. 2002–2011.
- BIERMANN, A. and FELDMAN, J. "On the Synthesis of Finite-State Machines From Samples of Their Behavior". *IEEE Transactions on Computers*, 21(6) (1972), pp. 592–597.
- BIRD, D. and MUNOZ, C. "Automatic Generation of Random Self-Checking Test Cases". *IBM Systems Journal*, 22(3) (1983), pp. 229–245.
- BIRZNIKES, G. *Perl Taint Mode*. <http://gunther.web66.com/FAQS/taintmode.html>. 1998.
- BISHOP, M. and DILGER, M. "Checking for Race Conditions in File Accesses". *ACM Transactions on Computer Systems*, 9(2) (1996), pp. 131–152.
- BOSIK, B. and UMIT UYAR, M. "Finite state machine based formal methods in protocol conformance testing: From theory to implementation". *Computer Networks and ISDN Systems*, 22(1) (1991), pp. 7–33.
- BOUSQUET, L. du, OUABDESSELAM, F., RICHIER, J., and ZUANON, N. "Lutess: A Testing Environment for Synchronous Software". *Tool Support for System Specification, Development and Verification* (1998), pp. 48–61.
- BOYER, R., ELSPAS, B., and LEVITT, K. "SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution". In *Proceedings of the International Conference on Reliable Software*. 1975, pp. 234–245.
- BUGTRAQ. *eServ Memory Leak Enables Denial of Service Attacks*. <http://www.securityfocus.com/archive/1/321306>. 2003.
- BURR, K. and YOUNG, W. "Combinatorial Test Techniques: Table-based Automation, Test Generation and Code Coverage". In *Proceedings of the International Conference on Software Testing Analysis & Review*. 1998, pp. 503–513.
- BUSH, W. R., PINCUS, J. D., and SIELAFF, D. J. "A Static Analyzer for Finding Dynamic Programming Errors". *Software – Practice & Experience*, 30(7) (2000), pp. 775–802.
- CABALLERO, J., YIN, H., LIANG, Z., and SONG, D. "Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis". In *Proceedings of the ACM Conference on Computer and Communications Security*. 2007, pp. 317–329.

- CADAR, C., GANESH, V., PAWLOWSKI, P. M., DILL, D. L., and ENGLER, D. R. "EXE: Automatically Generating Inputs of Death". In *Proceedings of the ACM Conference on Computer and Communications Security*. 2006, pp. 322–335.
- CARREIRA, J., MADEIRA, H., and SILVA, J. G. "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers". *IEEE Transactions on Software Engineering*, 24(2) (1998), pp. 125–136.
- CASTLE, S. "Europe Suffers Worst Blackout for Three Decades". In *The Independent*. Newspaper article, November 6, 2006.
- CASTRO, M. and LISKOV, B. "Practical Byzantine Fault Tolerance and Proactive Recovery". *ACM Transactions on Computer Systems*, 20(4) (2002), pp. 398–461.
- CHEN, H. and WAGNER, D. "MOPS: An Infrastructure for Examining Security Properties of Software". In *Proceedings of the ACM Conference on Computer and Communications Security*. 2002, pp. 235–244.
- CHEN, H., DEAN, D., and WAGNER, D. "Model Checking One Million Lines of C Code". In *Proceedings of the Network and Distributed System Security Symposium*. 2004, pp. 171–185.
- CHESS, B. and MCGRAW, G. "Static Analysis for Security". *IEEE Security and Privacy*, 2(6) (2004), pp. 76–79.
- CHOI, G. S. and IYER, R. K. "FOCUS: An Experimental Environment for Fault Sensitivity Analysis". *IEEE Transactions on Computers*, 41(12) (1992), pp. 1515–1526.
- CHOW, T. "Testing Software Design Modeled by Finite-State Machines". *IEEE Transactions on Software Engineering*, SE-4(3) (1978), pp. 178–187.
- CHUNLEI, W., GANG, Z., and YIQI, D. "An Efficient Control Flow Security Analysis Approach for Binary Executables". In *Proceedings of the International Conference on Computer Science and Information Technology*. 2009, pp. 272–276.
- CLARKE, E. M., GRUMBERG, O., and PELED, D. A. *Model Checking*. The MIT Press, 2000.
- CLARKE, E. M., GRUMBERG, O., and LONG, D. E. "Model Checking and Abstraction". *ACM Transactions on Programming Languages and Systems*, 16(5) (1994), pp. 1512–1542.

- CLARKE, L. A. "A System to Generate Test Data and Symbolically Execute Programs". *IEEE Transactions on Software Engineering*, SE-2(3) (1976), pp. 215–222.
- CODENOMICON. *Defensics X*. <http://www.codenomicon.com>. 2012.
- COHEN, D. M., DALAL, S. R., KAJLA, A., and PATTON, G. "The Automatic Efficient Test Generator (AETG) System". In *Proceedings of the International Symposium on Software Reliability Engineering*. 1994, pp. 303–309.
- COHEN, D., DALAL, S., KAJLA, A., and PATTON, G. "The Automatic Efficient Test Generator (AETG) System". In *Proceedings of the International Symposium on Software Reliability Engineering*. 1994, pp. 303–309.
- COHEN, D., DALAL, S., PARELIUS, J., and PATTON, G. "The Combinatorial Design Approach to Automatic Test Generation". *IEEE Software*, 13(5) (1996), pp. 83–88.
- COMBS, G. ET AL. *Wireshark*. <http://www.wireshark.org/>. 2012.
- COMPARETTI, P. M., WONDRAČEK, G., KRUEGEL, C., and KIRDA, E. "Prospex: Protocol Specification Extraction". In *Proceedings of the IEEE Symposium on Security and Privacy*. 2009, pp. 110–125.
- CORREIA, M., NEVES, N., and VERISSIMO, P. "How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems". In *Proceedings of the International Symposium on Reliable Distributed Systems*. 2004, pp. 174–183.
- COWAN, C., MCNAMEE, D., BLACK, A., PU, C., WALPOLE, J., KRASIC, C., WAGLE, P., ZHANG, Q., and MARLET, R. *A Toolkit for Specializing Production Operating System Code*. Technical Report CSE-97-004. Oregon Graduate Institute of Science and Technology, 1997.
- COWAN, C., BEATTIE, S., JOHANSEN, J., and WAGLE, P. "PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilities". In *Proceedings of the USENIX Security Symposium*. 2003, pp. 91–104.
- COWAN, C., PU, C., MAIER, D., WALPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., ZHANG, Q., and HINTON, H. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In *Proceedings of the USENIX Security Symposium*. 1998, pp. 63–78.
- COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., and SHRIRA, L. "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance". In *Proceedings*

- of the Symposium on Operating Systems Design and Implementation*. 2006, pp. 177–190.
- COZ, C. L. “Social Media, Facebook Help People Stand Up in Tunisia, Egypt”. In *Public Broadcasting Service (PBS)*. Newspaper article, February 3, 2011.
- CRISPIN, M. *Internet Message Access Protocol (IMAP) – Version 4rev1*. RFC 3501. Internet Engineering Task Force, 2003.
- CROCKER, D. and OVERELL, P. *Augmented BNF for Syntax Specifications: ABNF*. RFC 5234. Internet Engineering Task Force, 2008.
- CUI, W., KANNAN, J., and WANG, H. J. “Discoverer: Automatic Protocol Reverse Engineering From Network Traces”. In *Proceedings of the USENIX Security Symposium*. 2007, pp. 199–212.
- CUI, W., PEINADO, M., CHEN, K., WANG, H. J., and IRUN-BRIZ, L. “Tupni: Automatic Reverse Engineering of Input Formats”. In *Proceedings of the ACM Conference on Computer and Communications Security*. 2008, pp. 391–402.
- DAHURA, A., SABNANI, K., and UYAR, M. “Formal Methods for Generating Protocol Conformance Test Sequences”. *Proceedings of the IEEE*, 78(8) (1990), pp. 1317–1326.
- DAVIS, M. “Hilbert’s Tenth Problem is Unsolvable”. *The American Mathematical Monthly*, 80(3) (1973), pp. 233–269.
- DERDERIAN, K., HIERONS, R., HARMAN, M., and GUO, Q. “Automated Unique Input Output Sequence Generation for Conformance Testing of FSMs”. *The Computer Journal*, 49(3) (2006), pp. 331–344.
- DESIGNER, S. *Getting Around Non-Executable Stack (and Fix)*. <http://seclists.org/bugtraq/1997/Aug/63>. 1997.
- DI LUCCA, G., FASOLINO, A., FARALLI, F., and DE CARLINI, U. “Testing Web Applications”. In *Proceedings of the International Conference on Software Maintenance*. 2002, pp. 310–319.
- DOBBINS, R. and MORALES, C. *Worldwide Infrastructure Security Report – Volume VII*. Tech. rep. Arbor Networks, 2011.
- DOROFEEVA, R., EL-FAKIH, K., MAAG, S., CAVALLI, A., and YEVTUSHENKO, N. “Experimental Evaluation of FSM-based Testing Methods”. In *Proceedings of the International Conference on Software Engineering and Formal Methods*. 2005, pp. 23–32.

- DURÃES, J. and MADEIRA, H. "A Methodology for the Automated Identification of Buffer Overflow Vulnerabilities in Executable Software without Source-Code". In *Proceedings of the Latin-American Symposium on Dependable Computing*. Vol. 3747. 2005, pp. 20–34.
- DURÃES, J. and MADEIRA, H. "Definition of Software Fault Emulation Operators: A Field Data Study". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2003, pp. 105–114.
- DURÃES, J. and MADEIRA, H. "Emulation of Software Faults: A Field Data Study and a Practical Approach". *IEEE Transactions on Software Engineering*, 32(11) (2006), pp. 849–867.
- DURÃES, J. and MADEIRA, H. "Emulation of Software Faults by Educated Mutations at Machine-Code Level". In *Proceedings of the International Symposium on Software Reliability Engineering*. 2002, pp. 329–340.
- EDDY, W. *TCP SYN Flooding Attacks and Common Mitigations*. RFC 4987. Internet Engineering Task Force, 2007.
- ETOH, H. and YODA, K. "ProPolice: Improved Stack-smashing Attack Detection". *Transactions of Information Processing Society of Japan*, 43(12) (2002), pp. 4034–4041.
- EVANS, D., GUTTAG, J., HORNING, J., and TAN, Y. M. "LCLint: A Tool for Using Specifications to Check Code". In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 1994, pp. 87–96.
- FERGUSON, R. and KOREL, B. "The Chaining Approach for Software Test Data Generation". *ACM Transactions on Software Engineering and Methodology*, 5(1) (1996), pp. 63–86.
- FEWSTER, M. and GRAHAM, D. *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.
- FONSECA, J. and VIEIRA, M. "Mapping Software Faults with Web Security Vulnerabilities". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2008, pp. 257–266.
- FONSECA, J., VIEIRA, M., and MADEIRA, H. "Vulnerability & Attack Injection for Web Applications". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2009, pp. 93 –102.
- FORTIFY SOFTWARE, INC. *RATS – Rough Auditing Tool for Security*. <https://www.fortify.com/ssa-elements/threat-intelligence/rats.html>. 2009.

- FOSSI, M., BLACKBIRD, J., EGAN, G., LOW, M. K., HALEY, K., MAZUREK, D., JOHNSON, E., MCKINNEY, D., MACK, T., WOOD, P., and ADAMS, T. *Symantec Internet Security Threat Report*. Tech. rep. Volume XVI. Symantec Corporation, 2011.
- FOSTER, J. S., FÄHNDRICH, M., and AIKEN, A. "A Theory of Type Qualifiers". In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1999, pp. 192–203.
- FU, K. and BOOTH, T. "Grammatical Inference: Introduction and Survey". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8 (1986), pp. 343–375.
- GARCIA, M., BESSANI, A., GASHI, I., NEVES, N., and OBELHEIRO, R. "OS Diversity for Intrusion Tolerance: Myth or Reality?" In *Proceedings of the International Conference on Dependable Systems and Networks*. 2011, pp. 383–394.
- GARG, A., CURTIS, J., and HALPER, H. "Quantifying the Financial Impact of IT Security Breaches". *Information Management & Computer Security*, 11(2) (2003), pp. 74–83.
- GARG, S., MOORSEL, A. V., VAIDYANATHAN, K., and TRIVEDI, K. S. "A Methodology for Detection and Estimation of Software Aging". In *Proceedings of the International Symposium on Software Reliability Engineering*. 1998, pp. 283–292.
- GNU BINUTILS. *ObjDump*. <http://www.gnu.org/software/binutils/>. 2012.
- CODEFROID, P., LEVIN, M. Y., and MOLNAR, D. "Automated Whitebox Fuzz Testing". In *Proceedings of the Network and Distributed System Security Symposium*. 2008, pp. 151–166.
- GOLD, E. "Complexity of Automaton Identification From Given Data". *Information and Control*, 37(3) (1978), pp. 302–320.
- GONENC, G. "A Method for the Design of Fault Detection experiments". *Computers, IEEE Transactions on*, 100(6) (1970), pp. 551–558.
- GOSWAMI, K. K., IYER, R. K., and YOUNG, L. T. "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis". *IEEE Transactions on Computers*, 46(1) (1997), pp. 60–74.
- GRAMMATECH. *CodeSurfer*. <http://www.grammatech.com/products/codesurfer>. 2012.
- GREENBONE NETWORKS GMBH. *OpenVAS*. <http://www.openvas.org/>. 2012.

- GREENE, A. *SPIKEfile*. <http://labs.idefense.com/labs-software.php?show=14>. 2005.
- GUNNEFLO, U., KARLSSON, J., and TORIN, J. "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1989, pp. 340–347.
- GUTJAHR, W. "Partition Testing vs. Random Testing: The Influence of Uncertainty". *IEEE Transactions on Software Engineering*, 25(5) (1999), pp. 661–674.
- HALL, P. "Relationship Between Specifications and Testing". *Information and Software Technology*, 33(1) (1991), pp. 47–52.
- HAMLET, D. and TAYLOR, R. "Partition Testing Does Not Inspire Confidence". *IEEE Transactions on Software Engineering*, 16(12) (1990), pp. 1402–1411.
- HAUGH, E. and BISHOP, M. "Testing C Programs for Buffer Overflow Vulnerabilities". In *Proceedings of the Network and Distributed System Security Symposium*. 2003, pp. 123–130.
- HEDAYAT, A., SLOANE, N., and STUFKEN, J. *Orthogonal Arrays: Theory and Applications*. Springer Verlag, 1999.
- HEX-RAYS. *IDA Pro*. <http://www.hex-rays.com/idapro/>. 2012.
- HIERONS, R. M., BOGDANOV, K., BOWEN, J. P., CLEAVELAND, R., DERRICK, J., DICK, J., GHEORGHE, M., HARMAN, M., KAPOOR, K., KRAUSE, P., LÜTTGEN, G., SIMONS, A. J. H., VILKOMIR, S. A., WOODWARD, M. R., and ZEDAN, H. "Using Formal Specifications to Support Testing". *ACM Computing Surveys*, 41(2) (2009), pp. 9–85.
- HIGUERA, C. de la. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.
- HINES, S., WYATT, B., and CHANG, J. M. "Increasing Timing Resolution for Processes and Threads in Linux". Unpublished. 2000.
- HOAGLAND, J. and STANIFORD, S. *SPADE (Statistical Packet Anomaly Detection Engine)*. <http://www.silicondefense.com/spice/>. 2009.
- HOPCROFT, J. E., MOTWANI, R., and ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley, 2006.
- HOWDEN, W. "Introduction to the Theory of Testing". *Tutorial: Software Testing and Validation Techniques* (1978), pp. 16–19.

- HOWDEN, W. "Reliability of the Path Analysis Testing Strategy". *IEEE Transactions on Software Engineering*, (3) (1976), pp. 208–215.
- HSUEH, M.-C., TSAI, T. K., and IYER, R. K. "Fault Injection Techniques and Tools". *IEEE Computer*, 30(4) (1997), pp. 75–82.
- HUANG, Y., KINTALA, C., KOLETTIS, N., and FULTON, N. D. "Software Rejuvenation: Analysis, Module and Applications". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1995, p. 381.
- IBM INTERNET SECURITY SYSTEMS. *Internet Scanner*. <http://www.iss.net>. 2006.
- INFIGO INFORMATION SECURITY. *Infigo FTPStress Fuzzer*. <http://www.infigo.hr/>. 2012.
- INGHAM, K. and FORREST, S. "Network Firewalls". *Enhancing Computer Security with Smart Technology* (2006), pp. 9–40.
- INNOVATIVE COMPUTING LABORATORY. *PAPI – Performance Application Programming Interface*. <http://icl.cs.utk.edu/papi/index.html>. 2012.
- INTERNET ENGINEERING TASK FORCE. *RFC Editor*. <http://www.rfc-editor.org/rfc-index.html>. 2012.
- INTERNET SYSTEMS CONSORTIUM, INC. *BIND – Berkeley Internet Name Domain*. <http://www.isc.org/sw/bind>. 1984–2012.
- ITU. *Key Global Telecom Indicators for the World 2005-2010*. Tech. rep. International Telecommunications Union, 2011.
- ITU. *Specification and Description Language (SDL): Overview of SDL-2010*. ITU-T Rec. Z.100. International Telecommunication Union, 1999.
- ITU. *Specification and Description Language (SDL): SDL-2010 Combined With ASN.1 Modules*. ITU-T Rec. Z.105. International Telecommunication Union, 1999.
- ITU-T. *Abstract Syntax Notation One (ASN.1): Specification of Basic Notation*. ISO/IEC 8824-1. Telecommunications Standardization Sector of ITU, 1997.
- JACOBSON Van, L. C. and MCCANNE, S. *Tcpdump/Libpcap*. <http://www.tcpdump.org/>. 1987–2012.
- JENN, E., ARLAT, J., RIMÉN, M., OHLSSON, J., and KARLSSON, J. "Fault Injection into VHDL Models: The MEFISTO Tool". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1994, pp. 66–75.

- KAÂNICHE, M., LE GUÉDART, Y., ARLAT, J., and BOYER, T. "An Investigation on Mutation Strategies for Fault Injection Into RDD-100 Models". *Computer Safety, Reliability and Security* (2001), pp. 130–144.
- KANAWATI, G. A., KANAWATI, N. A., and ABRAHAM, J. A. "FERRARI: A Tool for the Validation of System Dependability Properties". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1992, pp. 336–344.
- KANAWATI, G. A., KANAWATI, N. A., and ABRAHAM, J. A. "FERRARI: A Flexible Software-Based Fault and Error Injection System". *IEEE Transactions on Computers*, 44(2) (1995), pp. 248–260.
- KANICH, C., WEAVER, N., MCCOY, D., HALVORSON, T., KREIBICH, C., LEVCHENKO, K., PAXSON, V., VOELKER, G. M., and SAVAGE, S. "Show Me the Money: Characterizing Spam-advertised Revenue". In *Proceedings of the USENIX Security Symposium*. 2011, pp. 219–233.
- KING, J. C. "Symbolic Execution and Program Testing". *Communications of the ACM*, 19(7) (1976), pp. 385–394.
- KLENSIN, J. *Simple Mail Transfer Protocol (SMTP)*. RFC 5321. Internet Engineering Task Force, 2008.
- KOOPMAN, P. and DEVALE, J. "Comparing the Robustness of POSIX Operating Systems". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1999, pp. 30–37.
- KOREL, B. "Automated Software Test Data Generation". *IEEE Transactions on Software Engineering*, 16(8) (1990), pp. 870–879.
- KOREL, B. "Dynamic Method for Software Test Data Generation". *Software Testing, Verification and Reliability*, 2(4) (1992), pp. 203–213.
- KOREL, B. and AL-YAMI, A. "Assertion-Oriented Automated Test Data Generation". In *Proceedings of the International Conference on Software Engineering*. 1996, pp. 71–80.
- KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., and WONG, E. "Zyzyva: Speculative Byzantine Fault Tolerance Replication". In *Proceedings of the Symposium on Operating Systems Principles*. Vol. 41. 6. 2007, pp. 45–58.
- KOUFAREVA, I. and DOROFEEVA, M. "A Novel Modification of W-Method". *Joint Bulletin of the Novosibirsk Computing Center and A.P. Ershov Institute of Informatics Systems*, (18) (2002), pp. 69–81.

- KREKEL, B. *Capability of the People's Republic of China to Conduct Cyber Warfare and Computer Network Exploitation*. Tech. rep. Defense Technical Information Center, 2009.
- LAI, R. "A Survey of Communication Protocol Testing". *Journal of Systems and Software*, 62(1) (2002), pp. 21–46.
- LAMPORT, L. "Time, Clocks, and the Ordering of Events in a Distributed System". *Communications of the ACM*, 21(7) (1978), pp. 558–565.
- LAROCHELLE, D. and EVANS, D. "Statically Detecting Likely Buffer Overflow Vulnerabilities". In *Proceedings of the USENIX Security Symposium*. 2001, pp. 177–190.
- LAU, F., RUBIN, S., SMITH, M., and TRAJKOVIC, L. "Distributed Denial of Service Attacks". In *International Conference on Systems, Man, and Cybernetics*. Vol. 3. 2000, pp. 2275–2280.
- LEE, D. and YANNAKAKIS, M. "Principles and Methods of Testing Finite State Machines – A Survey". *Proceedings of the IEEE*, 84(8) (1996), pp. 1090–1123.
- LEON, D., PODGURSKI, A., and DICKINSON, W. "Visualizing Similarity between Program Executions". In *Proceedings of the International Symposium on Software Reliability Engineering*. 2005, pp. 311–321.
- LIN, Z., JIANG, X., XU, D., and ZHANG, X. "Automatic Protocol Format Reverse Engineering Through Context-Aware Monitored Execution". In *Proceedings of the Network and Distributed System Security Symposium*. 2008.
- LO, D. and KHOO, S. "QUARK: Empirical Assessment of automaton-based specification miners". In *Proceedings of the Working Conference On Reverse Engineering*. 2006, pp. 51–60.
- LO, D., MARIANI, L., and PEZZÈ, M. "Automatic Steering of Behavioral Model Inference". In *Proc. of the joint meeting of the European Software Engineering Conf. and the ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*. 2009, pp. 345–354.
- LONDON, K., MOORE, S., MUCCI, P., SEYMOUR, K., and LUCZAK, R. "The PAPI Cross-Platform Interface to Hardware Performance Counters". In *Proceedings of the Department of Defense Users' Group Conference*. 2001.
- MANNING, C., RAGHAVAN, P., and SCHÜTZE, H. "An Introduction to Information Retrieval". *Cambridge University Press*, 11 (2008), p. 2009.

- MANOLACHE, L. and KOURIE, D. "Software Testing Using Model Programs". *Software: Practice and Experience*, 31(13) (2001), pp. 1211–1236.
- MARIANI, L. and PASTORE, F. "Automated Identification of Failure Causes in System Logs". In *Proceedings of the International Symposium on Software Reliability Engineering*. 2008, pp. 117–126.
- MCAFEE. *In the Dark: Crucial Industries Confront Cyberattacks*. Tech. rep. Center for Strategic and International Studies, 2011.
- MCAFEE, INC. *FoundStone Enterprise (now named McAfee Vulnerability Manager)*. <http://www.foundstone.com>. 2003–2012.
- MEMON, A., POLLACK, M., and SOFFA, M. "Automated Test Oracles for GUIs". *ACM SIGSOFT Software Engineering Notes*, 25(6) (2000), pp. 30–39.
- MENDONÇA, M. and NEVES, N. "Robustness Testing of the Windows DDK". In *Proceedings of the International Conference on Dependable Systems and Networks*. 2007, pp. 554–564.
- MICROSOFT, CORP. *A Detailed Description of the Data Execution Prevention (DEP) Feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003*. <http://support.microsoft.com/kb/875352>. 2006.
- MICROSOFT, CORP. *Introducing AsmL: A Tutorial for the Abstract State Machine Language*. <http://research.microsoft.com/en-us/groups/foundations/asml/>. 2001.
- MILLER, B. P., FREDRIKSEN, L., and SO, B. "An Empirical Study of the Reliability of UNIX Utilities". *Communications of the ACM*, 33(12) (1990), pp. 32–44.
- MOCKAPETRIS, P. *Domain Names – Implementation and Specification (DNS)*. RFC 1035. Internet Engineering Task Force, 1987.
- MOORE, D. *MyDNS*. <http://mydns.bboy.net>. 2002–2006.
- MOORE, H. D. *Month of the Browser Bugs*. <http://browserfun.blogspot.com/>. 2006.
- MUSUVATHI, M., PARK, D. Y. W., CHOU, A., ENGLER, D. R., and DILL, D. L. "CMC: A Pragmatic Approach to Model Checking Real Code". In *Proceedings of the Operating System Design and Implementation*. 2002, pp. 75–88.
- MYERS, G., SANDLER, C., and BADGETT, T. *The Art of Software Testing*. Wiley, 2011.
- MYERS, G. J. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

- MYERS, J. and ROSE, M. *Post Office Protocol (POP) – Version 3*. RFC 1939. Internet Engineering Task Force, 1996.
- NAITO, S. and TSUNOYAMA, M. “Fault Detection for Sequential Machines by Transition Tours”. In *Proceedings of the IEEE Fault Tolerant Computer Symposium*. Vol. 81. 1981, pp. 238–243.
- NAUMOVICH, G. and MEMON, N. “Preventing Piracy, Reverse engineering, and Tampering”. *Computer*, 36(7) (2003), pp. 64–71.
- NAVARRETE, C. and HERNANDEZ, A. *DotDotPwn*. <http://dotdotpwn.blogspot.com/>. 2012.
- NETHERCOTE, N. and SEWARD, J. “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation”. In *Proceedings of the Programming Language Design and Implementation*. 2007.
- NEVES, N., ANTUNES, J., CORREIA, M., VERISSIMO, P., and NEVES, R. “Using Attack Injection to Discover New Vulnerabilities”. In *Proceedings of the International Conference on Dependable Systems and Networks*. 2006.
- NLNET LABS. *NSD – Name Server Daemon*. <http://www.nlnetlabs.nl/projects/nsd/>. 2002–2012.
- OEHLERT, P. “Violating Assumptions with Fuzzing”. *IEEE Security and Privacy*, 3(2) (2005), pp. 58–62.
- OFFUTT, A. and HAYES, J. “A Semantic Model of Program Faults”. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1996, pp. 195–200.
- OFFUTT, J., LIU, S., ABDURAZIK, A., and AMMANN, P. “Generating Test Data from State-based Specifications”. *Software Testing, Verification and Reliability*, 13(1) (2003), pp. 25–53.
- PANG, R. and PAXSON, V. “A High-level Programming Environment for Packet Trace Anonymization and Transformation”. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 2003, pp. 339–351.
- PARNAS, D. “Software Aging”. In *Proceedings of the International Conference on Software Engineering*. 1994, pp. 279–287.
- PAX TEAM. *PaX*. <http://pax.grsecurity.net/>. 2009.

- PAXSON, V. "Bro: A System for Detecting Network Intruders in Real-Time". *Computer Networks*, 31(23) (1999), pp. 2435–2463.
- PENG, T., LECKIE, C., and RAMAMOCHANARAO, K. "Survey of Network-based Defense Mechanisms Countering the DoS and DDoS Problems". *ACM Computing Surveys*, 39(1) (2007), p. 3.
- PETTERSSON, M. *Linux Performance-Monitoring Counters Driver*. <http://www.csd.uu.se/~mikpe/linux/perfctr>. 2002–2011.
- PHADKE, M. S. *Quality Engineering Using Robust Design*. Prentice Hall, 1989.
- PLAT, N. and LARSEN, P. "An Overview of the ISO/VDM-SL Standard". *ACM Sigplan Notices*, 27(8) (1992), pp. 76–82.
- POSTEL, J. and REYNOLDS, J. *File Transfer Protocol (FTP)*. RFC 959. Internet Engineering Task Force, 1985.
- POWERDNS.COM BV. *PowerDNS*. <http://www.powerdns.com>. 2002–2012.
- QUALYS, INC. *QualysGuard Enterprise (now called QualysGuard Vulnerability Management)*. <http://www.qualys.com>. 2008–2012.
- RAMAMOORTHY, C., HO, S., and CHEN, W. "On the Automated Generation of Program Test Data". *IEEE Transactions on Software Engineering*, 2(4) (1976), pp. 293–300.
- RAPID7. *Metasploit Project*. <http://www.metasploit.com>. 2012.
- RAPID7. *NeXpose*. <http://www.rapid7.com>. 2012.
- RASHID, F. Y. "Epsilon Data Breach to Cost Billions in Worst-Case Scenario". In *eWeek*. Newspaper article, May 3, 2011.
- RAUCH, J. "PDB: The Protocol DeBugger". In *BlackHat USA*. 2006.
- RAUSAS, M. P. du, MANYIKA, J., HAZAN, E., BUGHIN, J., CHUI, M., and SAID, R. *Internet Matters: The Net's Sweeping Impact on Growth, Jobs, and Prosperity*. Tech. rep. McKinsey Global Institute, 2011.
- ROESCH, M. "Snort – Lightweight Intrusion Detection for Networks". In *Proceedings of the USENIX Conference on System Administration*. 1999, pp. 229–238.
- RONING, J. ET AL. *PROTOS – Security Testing of Protocol Implementations (now called Codenomicon Defensics)*. Computer Engineering Laboratory, University of Oulu. <http://www.ee.oulu.fi/research/ouspg/protos/>. 1999–2003.

- SABNANI, K. and DAHBURA, A. "A Protocol Test Generation Procedure". *Computer Networks and ISDN Systems*, 15(4) (1988), pp. 285–297.
- SAINT CORP. *SAINT Network Vulnerability Scanner*. <http://www.saintcorporation.com>. 2012.
- SAKAKIBARA, Y. "Grammatical Inference in Bioinformatics". *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(7) (2005), pp. 1051–1062.
- SCHNEIDER, F. "Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial". *ACM Computing Surveys*, 22(4) (1990), pp. 299–319.
- SEGALL, Z., VRSALOVIC, D., SIEWIOREK, D., YASKIN, D., KOWNACKI, J., BARTON, J., RANCEY, D., ROBINSON, A., and LIN, T. "FIAT – Fault Injection Based Automated Testing Environment". In *Proceedings of the International Symposium on Fault-Tolerant Computing*. 1988, pp. 102–107.
- SHANKAR, U., TALWAR, K., FOSTER, J. S., and WAGNER, D. "Detecting Format String Vulnerabilities with Type Qualifiers". In *Proceedings of the USENIX Security Symposium*. 2001, pp. 201–220.
- SHARMA, A. "Cyber Wars: A Paradigm Shift from Means to Ends". *Strategic Analysis*, 34(1) (2010), pp. 62–73.
- SHEVERTALOV, M. and MANCORIDIS, S. "A Reverse Engineering Tool for Extracting Protocols of Networked Applications". In *Proceedings of the Working Conference On Reverse Engineering*. 2007, pp. 229–238.
- SIMÃO, A. and PETRENKO, A. "Checking Sequence Generation Using State Distinguishing Subsequences". In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops*. 2009, pp. 48–56.
- SINGPURWALLA, N. and WILSON, S. *Statistical Methods in Software Engineering: Reliability and Risk*. Springer Verlag, 1999.
- SPIVEY, J. *The Z Notation: A Reference Manual*. Prentice Hall International (UK) Ltd., 1992.
- STOCKS, P. and CARRINGTON, D. "A Framework for Specification-based Testing". *IEEE Transactions on Software Engineering*, 22(11) (1996), pp. 777–793.
- SUTTON, M., GREENE, A., and AMINI, P. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.
- SUTTON, M. *FileFuzz*. http://labs.idefense.com/software/fuzzing.php#more_filefuzz. 2005.

- SYMANTEC. *Endpoint Security Best Practices Survey*. Tech. rep. Symantec Corporation, 2012.
- TAN, L., SOKOLSKY, O., and LEE, I. "Specification-based Testing With Linear Temporal Logic". In *Conference on Information Reuse and Integration*. 2004, pp. 483–498.
- TAYLOR, O. *MemProf: Profiling and Leak Detection*. <http://www.gnome.org/projects/memprof>. 1999-2007.
- TENABLE NETWORK SECURITY. *Nessus Vulnerability Scanner*. <http://www.tenable.com/products/nessus>. 2002–12.
- TENABLE NETWORK SECURITY. *Passive Vulnerability Scanner*. <http://www.tenable.com/products/tenable-passive-vulnerability-scanner>. 2002–12.
- TOKAREV, M. *rbldnsd*. <http://www.corpit.ru/mjt/rbldnsd.html>. 2003–2008.
- TRENHOLME, S. *MaraDNS – A Security-Aware DNS Server*. <http://www.maradns.org>. 2001-2012.
- TRIFILÒ, A., BURSCHKA, S., and BIRSACK, E. "Traffic to Protocol Reverse Engineering". In *Proceedings of the IEEE International Conference on Computational Intelligence for Security and Defense Applications*. 2009, pp. 257–264.
- TSAI, T. and SINGH, N. "Libsafe 2.0: Detection of Format String Vulnerability Exploits". *White paper, Avaya Labs* (2001).
- TSAI, T. K. and IYER, R. K. "Measuring Fault Tolerance with the FTAPE Fault Injection Tool". In *Proceedings of the International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*. Vol. 977. 1995, pp. 26–40.
- US-CANADA POWER SYSTEM OUTAGE TASK FORCE. *Blackout 2003: Blackout Final Implementation Report*. Tech. rep. Natural Resources Canada & U.S. Department of Energy, 2006.
- VAIDYANATHAN, K. and TRIVEDI, K. S. "A Comprehensive Model for Software Rejuvenation". *IEEE Transactions on Dependable and Secure Computing*, 2(2) (2005), pp. 124–137.
- VAN RIJSBERGEN, C. J. *Information Retrieval*. Butterworth-Heinemann, 1979.
- VANMALI, M., LAST, M., and KANDEL, A. "Using a Neural Network in the Software Testing Process". *International Journal of Intelligent Systems*, 17(1) (2002), pp. 45–62.

- VEENINGEN, M. *Posadis*. <http://posadis.sourceforge.net>. 2002–2005.
- VERISSIMO, P., NEVES, N., CACHIN, C., PORITZ, J., POWELL, D., DESWARTE, Y., STROUD, R., and WELCH, I. "Intrusion-Tolerant Middleware: The Road to Automatic Security". *IEEE Security and Privacy*, 4(4) (2006), pp. 54–62.
- VERISSIMO, P., NEVES, N. F., and CORREIA, M. "Intrusion-Tolerant Architectures: Concepts and Design". In *Architecting Dependable Systems*. Ed. by R. Lemos, C. Gacek, and A. Romanovsky. Vol. 2677. Springer-Verlag, 2003, pp. 3–36.
- VIEGA, J., BLOCH, J. T., KOHNO, Y., and MCGRAW, G. "ITS4: A Static Vulnerability Scanner for C and C++ Code". In *Proceedings of the Computer Security Applications Conference*. 2000, pp. 257–267.
- WAGLE, P. and COWAN, C. "StackGuard: Simple Stack Smash Protection for GCC". In *Proceedings of the GCC Developers Summit*. 2003, pp. 243–256.
- WAGNER, D., FOSTER, J. S., BREWER, E. A., and AIKEN, A. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". In *Proceedings of the Network and Distributed System Security Symposium*. 2000, pp. 3–17.
- WANG, W. "BEST: An Assembler Structural Representation Tool Based on Flow Analysis". In *Proceedings of the International Conference on Management and Service Science*. 2010, pp. 55–59.
- WARRENDER, C., FORREST, S., and PEARLMUTTER, B. "Detecting Intrusions Using System Calls: Alternative Data Models". In *IEEE Security and Privacy*. 1999, pp. 133–145.
- WEYUKER, E. "On Testing Non-Testable Programs". *The Computer Journal*, 25(4) (1982), pp. 465–470.
- WHALEN, M. W., RAJAN, A., HEIMDAHL, M. P. E., and MILLER, S. P. "Coverage Metrics for Requirements-Based Testing". In *Proceedings of the International Symposium on Software Testing and Analysis*. 2006, pp. 25–36.
- WHEELER, D. *Flawfinder*. <http://www.dwheeler.com/flawfinder/>. 2007.
- WIEBALCK, A., STEINBECK, T. M., and LINDENSTRUTH, V. "Fluctuating Processors". In *Linux Magazine* (<http://www.linux-magazine.com>), issue 31. 2003, pp. 46–49.
- WILANDER, J. and KAMKAR, M. "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention". In *Proceedings of the Network and Distributed System Security Symposium*. 2003, pp. 149–162.

- WONDRACEK, G., COMPARETTI, P., KRUEGEL, C., KIRDA, E., and ANNA, S. "Automatic Network Protocol Analysis". In *Proceedings of the Network and Distributed System Security Symposium*. 2008, pp. 336–354.
- YANG, J., TWOHEY, P., ENGLER, D., and MUSUVATHI, M. "Using Model Checking to Find Serious File System Errors". *ACM Transactions on Computer Systems*, 24(4) (2006), pp. 393–423.
- ZHIVICH, M., LEEK, T., and LIPPMANN, R. "Dynamic Buffer Overflow Detection". In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. 2005.
- ZÚQUETE, A. "StackFences: A Run-time Approach for Detecting Stack Overflows". In *Proceedings of the International Conference on E-Business and Telecommunication Networks*. 2004, pp. 76–84.

INDEX

- 2-way testing, *see* Pairwise testing
- ABNF, *see* Augmented Backus-Naur Form
- Abstract Syntax Notation One, 37
- Abstract syntax tree, 25
- AJECT, 76, 126, 166, 167, 170–172, 175, 196
- All-pairs testing, *see* Pairwise testing
- ASN.1, *see* Abstract Syntax Notation One
- Attacks, 56, 60
- Augmented Backus-Naur Form, 37
- Autoformat, 38
- Automata inference, *see* Grammatical induction
- Backus-Naur Form, 28, 37, 38
- Behavioral profile, 183, 184, 186, 188, 190, 230, 277–279
- BEST, 26
- BNF, *see* Backus-Naur Form
- BOON, 23
- Buffer overflow, 22–28, 30–32, 60, 162, 209, 211, 232, 234, 236
- Canary, 31
- CFSM, 26
- Characterization set, *see* W-method
- CMC, 20
- Complete oracle, 18
- Conformance testing, 49–51
- Control-flow analysis, 25, 26
- CQual, 24
- D-method, 50
- Data Execution Prevention, 32
- Data-flow analysis, 24–26
- Decision table (Oracle), 19
- Default testing, 44
- Delimiter test definition, 128
- DEP, *see* Data Execution Prevention
- Discoverer, 40
- Distinguishing sequence, *see* D-method
- DIVEINTO, 246
- Dynamic analysis, 38, 39, 41, 46–48
- Equivalence partitioning, 43
- EXE, 47
- External monitor, 170, 175, 196, 199, 203, 204, 213
- Fault injection, 14
- FindBugs, 23
- Finite-state machine, 19, 35, 37, 40, 49–51
- FiSC, 20
- Format string, 24, 155, 232
- FSM, *see* Finite-state machine
- Fuzz, 28

- Fuzzing, 27–29, 45, 48
- Fuzzing frameworks, 28
- G-SWFIT, 16
- Generic internal monitor, 171, 196, 199, 203, 204, 213
- Goal-oriented test data generation, 47
- Golden version (Oracle), 19
- Grammatical induction, 40–42, 78, 81, 90
- Hardware fault injection, 14
- Incomplete oracle, 18
- Information disclosure, 232
- Interaction faults, 27
- ITS4, 22
- k-tail, 40
- LCLint, 25
- Lexical analysis, 22
- LibGTop, 33
- Libsafe, 33
- Linear Temporal Logic, 21
- Longest common substring, 41
- LTL, *see* Linear Temporal Logic, 21
- Manual test analysis, 17
- Mealy machine, 80, 91, 95, 96, 255, 256, 279
- MemGuard, 31
- Memory leak, 26, 33, 34, 180, 221, 229
- Metasploit, 29
- Model checking, 19
- Moore machine, 80, 91, 95, 96
- MOPS, 19
- n-way testing, *see* Pairwise testing
- Negative testing, 29, 45
- Network traces, 40–42, 79, 98, 121, 142, 144, 153, 154, 236, 265
- Oracle, 18
- Oracle problem, *see* Oracle
- Orthogonal arrays, *see* Pairwise testing
- Pairwise testing, 44, 148
- Path-oriented test data generation, 46
- PaX, 32
- PEXT, 41
- PointGuard, 32
- Polyglot, 38
- PREDATOR, 76, 126, 168, 172, 176, 215
- PREfix, 26
- Prefix tree acceptor, 40, 41, 82, 92, 143
- Prelude of the test case, 128, 140, 143, 146, 148, 166, 207
- Privileged access violation test definition, 135
- ProPolice, 31
- Prospex, 39, 41
- Protocol Informatics, 40
- Protocol language inference, 40, 81
- Protocol state machine inference, 41, 42, 90
- PTA, *see* Prefix tree acceptor
- Random testing, 45
- Repeated injection campaign with restart, 168, 179, 180, 215, 216, 221
- Resource-exhaustion, 33, 167, 176, 178, 210, 215, 229, 232, 239
- REVEAL, 172, 183, 230
- ReverX, 78, 80, 230, 246, 276
- Robustness testing, 27
- Run-time detection, 30–32
- SAGE, 48
- SDL, *see* Specification and Description Language
- Sequence alignment, 40
- Simulation-based fault injection, 15
- Single injection campaign with restart, 166, 196

- Single injection campaign without restart, Vulnerability scanners, 29
 - 196, 213
- Software aging, 34, 65, 167, 168, 177, 214
 - W-method, 51
 - Wireshark, 39
- Software fault injection, 15
- Software rejuvenation, 34
- Specialized internal monitor, 172, 175, 187, 196, 199, 203, 204, 213, 218, 230, 235, 241
- Specification, 18, 35, 49, 59, 61
- Specification and Description Language, 37, 38
- Specification languages, 18, 36, 49
- StackFences, 31
- StackGuard, 31
- Static analysis, 19, 22–25, 47, 48
- Static vulnerability analyzers, 22
- Symbolic execution, 47
- Syntax test definition, 131
- T-method, 50
- Taint analysis, 24, 38, 41
- Tcpdump, 39, 102
- Test oracle, see Oracle
- Testing message, 128, 148, 149, 166, 207
- Testing payload, 128, 132, 134, 135, 139, 140, 143, 148, 149
- Transitions tours, see T-method
- Tupni, 38
- Type checking, 23
- U-method, 51
- UIO sequence, see U-method
- Valgrind, 33
- Value test definition, 132, 135, 136, 205
- VDM-SL, see Vienna Development Method Specification Language
- Vienna Development Method Specification Language, 37
- Vulnerabilities, 56