Project no.:	**IST-FP6-STREP - 027513**
Project full title:	**Critical Utility InfrastructurAL Resilience**
Project Acronym:	**CRUTIAL**
Start date of the project:	**01/01/2006 Duration: 39 months**
Deliverable no.:	**D20**
Title of the deliverable:	**Experimental validation of architectural solutions**

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Contractual Date of Delivery to the CEC:	31/3/2009
Actual Date of Delivery to the CEC:	16/03/2009
Organisation name of lead contractor for this deliverable	CNIT
Editor(s): Giuliana Franceschinis[6]	
Author(s): Giuliana Franceschinis[6]; Eric Alata [4]; João Antunes[2]; Hakem Beitollah[5]; Alysson Neves Bessani[2]; Miguel Correia[2]; Wagner Dantas[2]; Geert Deconinck[5]; Mohamed Kaâniche[4]; Nuno Neves[2]; Vincent Nicomette [4] Paulo Sousa[2]; Paulo Verissimo[2].	
Participant(s): (2) FCUL; (4) LAAS-CNRS; (5) KUL; (6) CNIT	
Work package contributing to the deliverable:	WP5
Nature:	R
Dissemination level:	PU
Version:	004
Total number of pages:	88

Abstract

In this deliverable the experimental results carried out in four different contexts are reported. The first contribution concerns an experimental campaign performed using the AJECT (Attack inJECTion) tool able to emulate different types of attackers behaviour and to collect information on the effect of such attacks on the target system performance. This tool is also used to perform some of the experiments described in the fourth part of the deliverable.

The second contribution concerns a complementary approach using *honeypots* to capture traces of attacker behaviours, to then study and characterize them. Different kinds of honeypots were deployed in the described experiments: low-interaction and high-interaction ones, exposing different kinds of services and protocols (general purpose network services as well as SCADA specific ones).

The third and fourth contribution refer to experiments conducted on some components of the CRUTIAL architecture, namely FOSEL (Filtering with the help of Overlay Security Layer), the CIS-CS (Communication Service) and the CIS-PS (Protection Service). The experiments have been performed with the aim of evaluating the effectiveness of the proposed components from the point of view of the dependability improvement they bring, as well as the performance overhead introduced by their implementation.

Contents

List of Figures

1 Introduction

Identifying applications security related vulnerabilities and collecting real data to learn about the tools and strategies used by attackers to compromise target systems connected to the Internet is a necessary step in order to be able to build critical infrastructures and systems that are resilient to malicious threats. This is the subject of the first part of this deliverable, which presents two complementary types of experimental environments used in the context of CRUTIAL in order to fulfill these objectives. The first one concerns the development of a methodology and a tool (AJECT) for injecting attacks in order to reveal residual vulnerabilities in the applications and software components under study. This tool has been experimented in particular to locate security vulnerabilities in network servers and software components of the CRUTIAL reference architecture and information switches. The second type of experimental environment investigated in CRUTIAL and discussed in the first part of this deliverable concerns the development and the deployment of honeypots aimed at collecting data characterizing real attacks on the Internet. Such data are mainly used in the context of CRUTIAL to support the modeling activities carried out in WP2 and WP5 regarding the characterization and assessment of malicious threats. Useful feedback could be also provided to support design related activities, through the identification of common types, behaviors and scenarios of attacks observed on the Internet. Both general purpose and SCADA specific honeypots have been experimented.

Ensuring the continuity of a service, in particular in critical infrastructures, requires to deploy appropriate protection systems and strategies, able to guarantee continued service in face of accidental as well as malicious faults. In the second part of this deliverable, experimental evaluation of two components of the CRUTIAL architecture developed in WP4 is thoroughly described. The former component is Fosel, allowing to tolerate DOS attacks, while the second one includes the CIS-PS and the CIS-CS services: CIS-PS ensures that the incoming and outgoing traffic in/out of a LAN (hosting critical services) satisfies the security policy of the infrastructure. CIS-CS supports secure communication between CISs. The experiments confirmed the effectiveness of the architectural solutions developed in WP4. The presentation includes details about the experimental setup (exploiting virtualization technologies) and about the methodology followed in designing the experiments.

The structure of this Deliverable is as follows. Chapter 2 summarizes the methodology and the AJECT tool aimed at the identification of software security-related vulnerabilities and presents several experimental results illustrating the capabilities of the tool. Chapter 3 describes honeypot-based experimental environments investigated in CRUTIAL to collect and analyze real attack observed on the Internet. Chapter 4 describes the experiments performed on FOSEL under the hypothesis of a DOS attack, showing the effectiveness of the proposed approach as well as the effect of two relevant parameters which have an impact both on the introduced overhead and on the probability of succesfully delivering useful messages to their final destination. Chapter 5 describes a set of experiments that have been conducted to evaluate the Crutial Information Switch (CIS), providing both a Communication and a Protection Service (details on these services are given in deliverable D18). Finally, conclusions and ideas for future work are drawn in Section 6.

2 Software Vulnerabilities Identification Based on Attack Injection

Applications have suffered dramatic improvements in terms of the offered functionality over the years. These enhancements were achieved in many cases with bigger software projects, which can no longer be carried out by a single person or a small team. Consequently, size and complexity has increased, and software development now frequently involves several teams that need to cooperate and coordinate efforts. Additionally, to speedup the programming tasks, most projects resort to third-party software components (e.g., a cryptographic library, a PHP module, a compression library), which in many cases are poorly documented and supported. It is also very common to re-use legacy code, probably no longer maintained or supported. All these factors contribute to the presence of vulnerabilities.

However, a vulnerability per se does not cause a security hazard, and in fact it can remain dormant for many years. An intrusion is only materialized when the right attack is discovered and applied to exploit a particular vulnerability. After an intrusion, the system might or might not fail, depending on its capabilities in dealing with the errors introduced by the adversary. Sometimes the intrusion can be tolerated [47], but in the majority of the current systems, it leads almost immediately to the violation of its security properties (e.g., confidentiality or availability). Therefore, it is important to devise methods and means to remove vulnerabilities or even prevent them from appearing in the first place.

Vulnerability removal can be performed during both the development and operational phases. In the last case, besides helping to identify programming flaws, which can later be corrected, it also assists the discovery of configuration errors. Intrusion prevention, such as vulnerability removal, has been advocated because it reduces the power of the attacker [47]. In fact, even if the ultimate goal of zero vulnerabilities is never attained, vulnerability removal effectively reduces the number of entry points into the system, making the life of the adversary increasingly harder (and ideally discouraging further attacks).

In this chapter, we describe a tool called AJECT, which has been developed within the project. This tool uses an attack injection methodology to locate security vulnerabilities in software components, e.g., network servers running in the CRUTIAL Information Switches or in other interconnected machines. Some results with well-known email and DNS servers are described to validate the capabilities of the tool. The structure of this chapter is as follows: Section 2.1 starts with the identification of the architectural components to be experimentally validated; Section 2.2 describes the attack injection tool; the experimental results that illustrate the capabilities of the tool are presented in Section 2.3; and the chapter ends with a summary of the results in Section 2.4.

2.1 Identification of the Run-time Components to be Validated

The nature of the software and the reliance we place in it makes us even more vulnerable to deviations from its correct behavior. Critical infrastructures, such as the Power Grid for instance, have an important role in the normal functioning of the economy and general community, and thus they pose an even higher risk to the sustenance of the modern society pillars, such as the national and international security, governance, public health and safety, economy, and public confidence.

In recent years, these systems evolved in several aspects that greatly increased their exposure to cyber-attacks coming from the Internet. Firstly, the computers, networks and protocols in those control systems are no longer proprietary but standard PCs and networks (e.g., wired and wireless Ethernet), and the protocols are often encapsulated on top of TCP/IP. Secondly, these networks are usually connected to the Internet indirectly through the corporate network or to other networks using modems and data links. Thirdly, several infrastructures are being interconnected creating a complexity that is hard to manage [45].

Therefore, these infrastructures have a high level of exposure similar to other systems connected to the Internet, but the socio-economic impact of their failure can be tremendous. This scenario, reinforced by several recent incidents [51, 7], is generating a great concern about the security of these infrastructures, especially at government level.

The proposed reference architecture for critical infrastructures models the whole infrastructure as a WAN-of-LANS, collectively protected by some special devices called CRUTIAL Information Switches (CIS). CIS devices collectively ensure that incoming/outgoing traffic satisfies the security policy of an organization in the face of accidents and attacks. However, they are not simple firewalls but distributed protection devices based on a sophisticated access control model. Likewise, they seek perpetual and unattended correct operation, which needs to be properly evaluated and validated. It is important that components, directly or indirectly, related to the correct functioning of the critical infrastructure are identified and validated.

In this deliverable, we have focused our validation efforts on two kinds of components. We have looked into fundamental services present in most network infrastructures, namely Email and Domain Name System (DNS) services. Though these fundamental services are usually not the directly related to the final service of the network infrastructure (e.g., to provide electricity), they are nevertheless a constant presence in these networks and generally necessary to the correct functioning of the overall infrastructure. For instance, email provides a means of communication within the infrastructure, or DNS provides name resolution for other internal servers, such as email servers. Besides their ubiquity, these network components are also connected to the outside networks (i.e., the Internet) making them highly coveted targets for blackhat hackers.

Besides common network services, we have also evaluated the CIS devices since their position at the border of the protected networks makes this first line of defense the primary target of attacks (see Chapter 5).

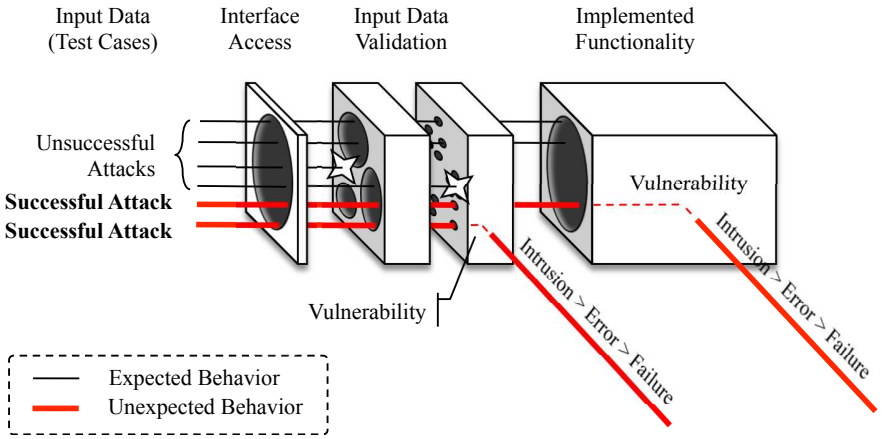


Figure 2.1: The attack process on a faulty (or vulnerable) component.

2.2 The Attack Injection Tool

Vulnerabilities are usually caused by subtle anomalies that only emerge in such unusual circumstances that were not even contemplated in test design. They also tend to elude the traditional software testing methods, mainly because 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. The typical process of manually searching for new vulnerabilities is often slow and tedious. Specifically, the source code can be carefully scrutinized for security flaws or the application can be exhaustively experimented with several kinds of input (e.g., unusual and random data, or more elaborated input based on previously known exploits) looking for problems during its execution.

Figure 2.1 shows a model of a component with existing vulnerabilities. Boxes in the figure represent the different modules or software layers that compose the component. 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.

The external access to the component is provided through a known *Interface Access*, which receives the input arriving for instance through 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 some bounds). 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 defined 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 accom-

plish some well defined objective, such as responding to client requests according to some standard network protocol.

By accessing the interface, an adversary may persistently look for vulnerabilities by stressing the component with unusual forms of interaction, such as by sending different kinds of messages or malformed files. These *attacks* are malicious interaction faults that are given to the component to process [46]. 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* along 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), and consequently, if nothing is done to handle the error, it will lead to a *failure*.

After finding a successful attack, the attacker can tweak 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 created to exploit the same vulnerability. Depending on the type of vulnerability, and on the skill of the attacker, the range of the security compromise can vary greatly, from simple denial-of-service to full control of the system.

2.2.1 The Attack Injection Methodology

The attack injection methodology adapts and extends classical fault injection techniques to look for security vulnerabilities. This methodology can be a useful asset in increasing the dependability of computer systems since it addresses an elusive class of faults and contributes for their location and removal. 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 general operation of an attack injection tool is represented in Figure 2.2.

First, several attacks are generated in order to later evaluate the target system's intended functionality (step 1). To get a higher level of confidence about the absence of vulnerabilities, the attacks have to be exhaustive and should look for as many classes of flaws 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.

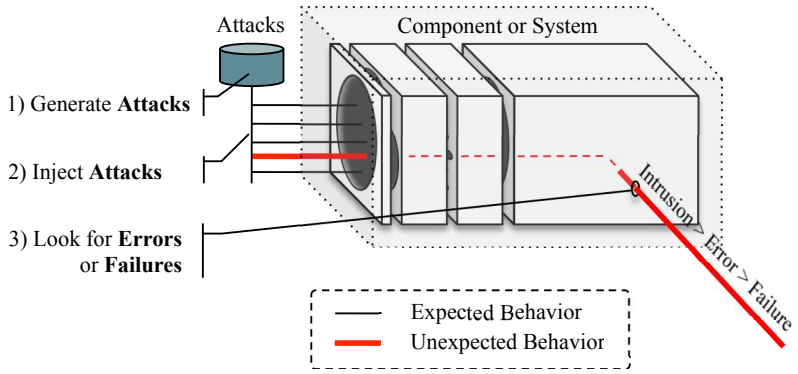


Figure 2.2: The attack injection methodology.

Ideally, one would like to build test cases that would not only exercise all reachable computer instructions but also experiment them with every possible instance of input. This goal, however, is unfeasible and intractable for most systems, due to the effort required to generate, and then execute, the various combinations of input data. The effort can be decreased by resorting to the analysis of the source code, and by manually creating good test cases. This approach requires a great deal of experience and acuteness from the test designers, and even then, many vulnerabilities can be missed altogether. In addition, because it is common practice to re-use general-purpose components developed by third parties, source code might be unavailable. To overcome these limitations and to automate the process of discovering vulnerabilities, we propose a methodology that generates a large number of test cases from a specification of the component's interface.

After creating the attacks, the attack injection tool should then execute them (step 2), while monitoring the state of the component, looking for any unexpected behavior (step 3). Depending on its monitoring capabilities, the tool could examine the target system's outputs, its allocated system resources, or even the last 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. On the other hand, if what is observed is the abnormal creation of a large file, though it might not be a vulnerability, it can eventually lead to a denial-of-service, 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 vulnerability more accurately. If locating and removing the fault is unfeasible, or a more immediate action is required, for instance if the target system is a COTS component or a fundamental 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, therefore improving the system's dependability.

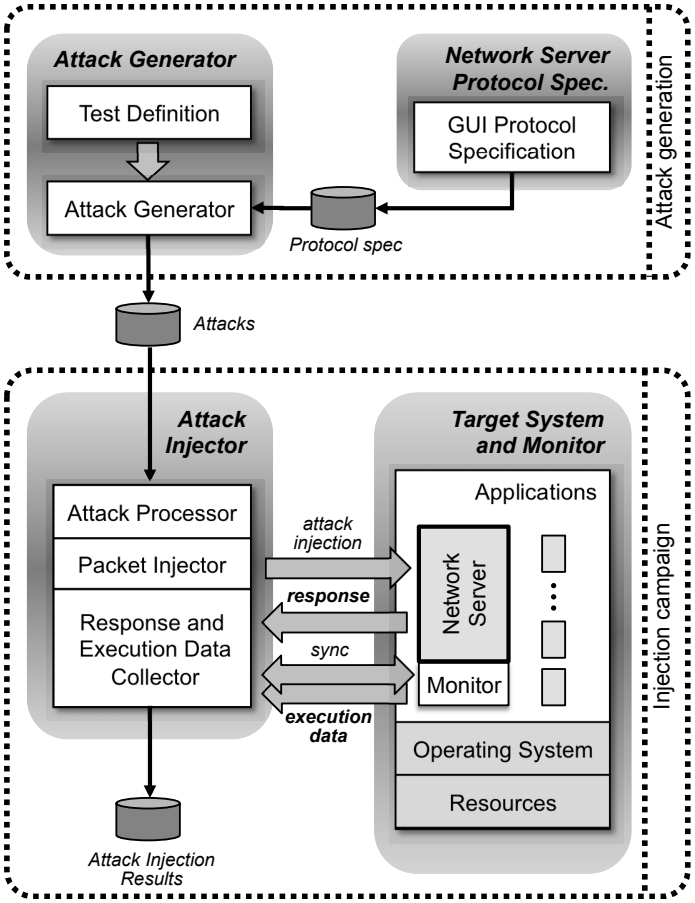


Figure 2.3: The architecture of the AJECT tool.

2.2.2 Architecture of the AJECT tool

The **A**ttack in**JEC**tion **T**ool (AJECT) is a vulnerability detection tool that implements the proposed methodology. Its architecture and main components are depicted in Figure 2.3. The architecture was developed to achieve automatic injection of attacks independently of the target servers implementation. Furthermore, it was built to be flexible regarding the classes of vulnerabilities that can be discovered and the method used to monitor the target system. Therefore, AJECT's implementation provides a framework to create and evaluate the impact of different test case generation algorithms (i.e., by supplying various *Test Definitions*) and other monitoring approaches (i.e., by implementing custom *Monitors*). In this work we have equipped AJECT with two test suites: a test definition with several malicious exploit patterns to generate malicious traffic, and a simpler attack generation traffic combined with a modified attack injection approach specifically designed to identify resource exhaustion vulnerabilities.

The *Target System* is the entire software and hardware components that comprises the target application and its execution environment, including the operating system, the software libraries, and the system resources. The *Network Server* is typically a service that can be invoked remotely from client programs (e.g., an email or FTP server). The target

application uses a well-known protocol to communicate with the clients, and these clients can carry out attacks by transmitting malicious packets. If the packets are not correctly processed, the target can suffer various kinds of errors with distinct consequences, ranging, for instance, from a slow down to a crash.

The *Network Server Protocol Specification* is a graphical user interface component that supports the specification of the communication protocol used by the server. This specification is later utilized by the *Attack Generator* to produce a large number of test cases. The *Attack Injector* is responsible for the actual execution of the attacks, by transmitting erroneous packets to the server. It also receives the responses returned by the target and the remote execution profile as collected by the *Monitor*. Some analysis on the information acquired during the attack is also performed to determine if a vulnerability was exposed (e.g., such as known fatal signals or connection error).

The overall attack injection process is carried out in two separate phases: the attack generation phase, executed only once for each different communication protocol, and the injection campaign, performed on every target system.

2.2.2.1 Attack Generation Phase

The purpose of the *attack generation phase* is to create a series of attacks to be injected in the target system. In order to be effective, the test case generation algorithm should aim to be exhaustive, fully automated, and not too simplistic. The procedure needs to be exhaustive, to be able to explore the various parts of the application code, and automated, to ensure that a large number of tests are performed with the smallest amount of human effort. Overly simple methods, based solely on random test case generation, tend to produce uninteresting results because the generated attacks do not proceed deep in the processing layers of the target application. Since we wanted to make AJECT as generic as possible, so that it could be employed with different servers (including the commercial ones), we designed it not to rely on the availability of the source code.

AJECT uses a specification of the communication protocol of the server to produce more intelligent attacks. This specification defines in practice the server's external interface. Therefore, by exploring the input space defined by the protocol, it is possible to exercise much of the intended functionality of the target, i.e., the parts of the code that are executed upon the arrival of network packets. 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). Consequently, even if the information about a particular server implementation is scarce, it is still possible to create good test cases as long as one knows its communication protocol.

To support the specification of the communication protocol, AJECT provides a graphical user interface tool called the *Network Server Protocol Specification* (see Figure 2.4 for a screenshot). The operator can, in a user-friendly way, describe the protocol states

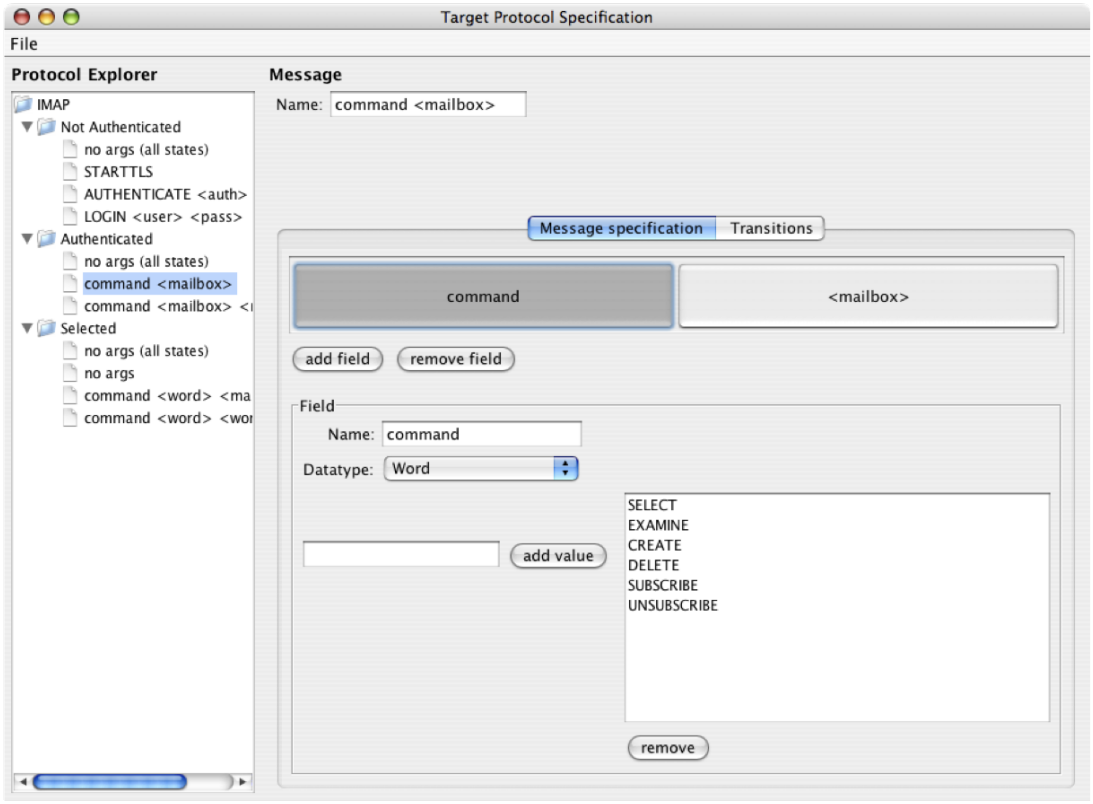


Figure 2.4: Screenshot of the AJECT protocol specification application.

and messages, and identify the data types and acceptable ranges of values of each field of a message. Messages are divided in two kinds: messages that request the execution of some specific operation (not changing the state of the protocol), and transition messages that make the protocol jump from one state to another (e.g., a login message). Using this information, AJECT can then create test cases with actual protocol messages that can take the server to a particular protocol state, and then inject some malicious data. This allows the tool to exhaust all messages in every protocol state or to generate attacks targeting a specific state of the protocol.

One aspect in which protocols may differ, is on *how* they represent data, either text-based or binary. One example of a text-based protocol is IMAP [16], while NTP [29] or Bootstrap [17] are binary protocols. Some other protocols make use of both types of data, such as the DNS protocol [30]. So, one of the challenges was to provide the means to support these both types of data representation. In AJECT, textual fields are characterized by the set of acceptable words. For example, if an operator is describing a message with two fields, a command and a parameter, the first field should be specified with all possible command strings (e.g., CREATE and DELETE), whereas for the second field the operator should define its possible parameters (or at least some of them). In addition, textual fields are normally delimited by a special character (e.g., a space) that should also be indicated (i.e., either the initial or the final delimiter). Similarly, binary fields should be described by the various possible numbers (either individually or as intervals) and their format: byte ordering, sign support, and number of bits.

The attack generation itself is dictated by the test case generation algorithm, which receives the specification of the protocol as input and outputs the attacks to be injected in the Target System.

2.2.2.2 Injection Campaign Phase

The *injection campaign phase* corresponds to the actual process of executing the previously generated test cases, i.e., the injection of the attacks in the target system (see AJECT's architecture in Figure 2.3). The *Attack Injector*, or simply the injector, performs each attack sequentially. It decomposes each test case in its corresponding network packets, such as the transition messages to change the protocol state and the actual attack message, and sends them to the network interface of the server.

During the attack injection campaign, both the server's responses and its execution data are collected by the injector. The injector resorts to a *Monitor* component, typically located in the actual target system, to inspect the execution of the server (e.g., UNIX signals, Windows exceptions, allocated resources, etc.). The monitor can provide a more accurate identification of unexpected behavior. However, its implementation requires (1) access to the low-level process and resource management functions of the target system and (2) the synchronization between the injector and the monitor for each test case execution.

Currently, our monitor can observe the target application's flow of control, while keeping a record of the amount of allocated system resources. The tracing capabilities are achieved with the PTRACE family functions, which can be employed to intercept any signal received by the server (e.g., a SIGSEGV signal, indicating a memory access violation, is frequently related to buffer overflow vulnerabilities). The LibGTop[1] library functions are utilized to fetch resource usage information, such as the memory allocated by the process (e.g., the total number of memory pages or the number of non-swapped pages) and the accumulated CPU time in user- and kernel-mode. This version of the Monitor relies in many OS-dependent facilities to supervise the target application, so it can only be used in UNIX-based systems. We have however, other more generic monitor versions that could be used in any system.

2.2.2.3 Vulnerability Detection

Vulnerabilities that result in fatal crashes are easily perceivable through 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 instance, are only detected if some illegal access is granted by the server, which can be discovered by studying both the attack (i.e., the type of protocol request) and the server's response (i.e., the authorization). AJECT currently automates some of this post-attack

[1] A public library, available at `http://directory.fsf.org/libs/LibGTop.html`.

```
TestValue
    Input: Protocol ← specification of the network protocol used by the server
    Output: Attacks

    S ← set of all states of the Protocol specification
    foreach State s ∈ S do
        M ← set of message specifications of s
        Prelude_s ← set of ordered network packets necessary to reach s
        P ← φ

        foreach MessageSpecification m ∈ M do
            foreach FieldSpecification f ∈ m do
                if f is type Numbers then
                    f' ← all boundary values, plus some intermediary illegal values, from f specification
                elseif f is type Words then
                    f' ← combination of predefined malicious tokens

                m' ← copy of m replacing f with f'
                P ← P ∪ {set of all network packets based on m' specification}

        foreach attack_packet ∈ P do
            attack ← Prelude_s ∪ {attack_packet}
            Attacks ← Attacks ∪ {attack}

    return Attacks
```

Figure 2.5: Algorithm for the Value Test generation.

analysis, by processing the resource usage logs and the returned messages by the server. However, the operator has to be involved in this analysis to determine if any abnormal behavior is actually a vulnerability.

2.2.3 Test Suites

Two different test suites were created to be used with our attack injection tool. The first one generates malicious traffic to identify general security vulnerabilities, such as buffer overflows, format strings, information disclosure, etc. It was intended to be a general attack injection test suite, suitable to discover the majority of vulnerabilities that could be triggered by outside attackers with simple network interactions, i.e., with a few packets. The second test suite was designed specifically to detect the resource exhaustion vulnerabilities, a subtle type of vulnerabilities that are usually only perceived through software aging effects. These two different test suites demonstrate the tool's versatibility in adapting to different attack injection approaches.

2.2.3.1 General Security Vulnerabilities

This test suite determines if the server can cope with messages with bad data. For this purpose, a mechanism is used to derive illegal data from the message specification, in

```
generateIllegalWords
    Input: Words ← specification of the field
    Input: Tokens ← predefined list of malicious tokens, e.g., taken from hacker exploits
    Input: Payload ← predefined list of special tokens to fill in the malicious tokens
    Input: max_combinations ← maximum number of token combinations
    Output: IllegalWords

    // step 1: expand list of tokens
    foreach t ∈ Tokens do
    if t includes keyword $(PAYLOAD) then
        foreach p ∈ Payoad do
            t' ← copy of t replacing $(PAYLOAD) with p
            Tokens ← Tokens ∪ {t'}
        Tokens ← Tokens \ {t}

    // step 2: generate combinations of tokens
    n ← 1
    while n ≤ max_combinations
        Combinations_n ← set of all combinations of exactly n elements
        IllegalWords ← IllegalWords ∪ Combinations_n

    return IllegalWords
```

Figure 2.6: Algorithm for the generation of malicious strings.

particular from each field. 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 textual content. To overcome such impossibility, a heuristic method was conceived to reduce the number values that have to be tried (see algorithm in Figure 2.5). The Value Test algorithm systematically exchanges the data of each field with illegal values. In the case of numeric fields, deriving illegal values is rather simple because they correspond to the complementary values, 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 only employs boundary values plus a subset of the complementary data (e.g., 10% coverage of the illegal numbers).

Creating illegal words, however, is a much more complex problem because there is an infinite number of character combinations that can form an arbitrary-sized string, making such an exhaustive approach impossible. Our objective was to design a method to derive potentially desirable illegal words, i.e., words that are usually seen in exploits, such as large strings, strange characters, or known pathnames (see algorithm in Figure 2.6, which is called in the underlined line of Figure 2.5). Basically, this method produces illegal words 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 example in Figure 2.7). AJECT 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 some attacks that were generated by this method, and were

used to detected known IMAP vulnerabilities ("$< A \times 10 >$", means character 'A' repeated ten times) [33]:

- A01 AUTHENTICATE $< A \times 1296 >$
- $< \%s \times 10 >$
- A01 LIST INBOX $< \%s \times 10 >$

Since the current monitor only supports UNIX-based systems, we have also developed an OS-independent monitor without resource usage or in-depth execution monitoring capabilities. This monitor *infers* the server's behavior through passive and remote observation, collecting information about the network connection between the injector and the server. Since it resides in the injector machine, it can be used with virtually any target system. After every test case execution, the monitor closes and re-opens the connection with the server, signaling the server's execution as failed if some communication error arises. This remote monitoring solution is the simplest and fastest approach when using an unknown or inaccessible target system.

2.2.3.2 Resource Exhaustion Vulnerabilities

Resource-exhaustion vulnerabilities are difficult to find because 1) they might be triggered exclusively in very special conditions, and 2) the resource leaks may only be perceived after many activations. For this reason, a specific test suite had to be developed to search for these problems.

The resource loss of the target system cannot be detected just by looking at a single snapshot of its utilization, i.e., monitoring a single attack injection. However, a continuous and careful resource tracing could perceive the overall tendency in which the resource depletion may be developing into. To achieve this, the injection of each attack is repeated several times without restarting the target server application. The monitoring data obtained is then used to plot the resource usage trend for each attack, and thus to identify the most "dangerous" ones.

In this particular test, the injector also performs a post-processing analysis on the collected data to build accurate resource usage projections allowing us to identify resource exhaustion vulnerabilities. We use regression analysis on the collected data to produce a statistical model of the actual resource consumption, i.e., the projections. These resource projections give a forecast on the amount of effort necessary for an attacker to deplete the server's resources and to cause a denial-of-service. With this trend information it is possible to recognize which attacks are more dangerous by looking for the projections with higher growth rates.

2.3 Experimental Validation

The first part of the experimental validation was done with the test suite for general security vulnerabilities from Section 2.2.3.1. 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 16 target network servers with one of the supported monitor components). The visible behavior of the e-mail server was recorded by the monitor component and stored in a log file for later analysis. If a server crashed, 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 remote monitor) and the injection campaign resumed from the next attack.

The second part of this experimental evaluation was carried out with the test suite for discovering resource exhaustion vulnerabilities from Section 2.2.3.2. In this test suite, the monitor must collect accurate resource usage information, so we only used target server applications that our monitor supported (i.e., Linux servers) running directly in the OS (i.e., with no VM images). The hardware configuration was similar, with one injector machine sending generated malicious traffic to target system machine (running the target server application and the local monitor).

At the end of the attack injection experiments, the operator analyzes the output results, looking for any unexpected behavior. Any suspicion of abnormal behavior, such as an unusual set of system calls, a large resource usage, or a bad return error code, should be further investigated. AJECT allows the operator to replay the last or latter attacks in order to reproduce the anomaly and thus confirm the existence of the vulnerability. The offending attacks can then be provided as test cases to the developers to debug the server application.

2.3.1 General Security Vulnerabilities (POP and IMAP servers)

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, fully patched to the latest stable version and with no known vulnerabilities. Most of them had gone through many revisions and are supported and continuously improved by an active development team. Since we wanted to experiment target systems with distinct characteristics, we chose servers running on different operating systems (Windows vs. Linux) and programmed under opposing philosophies regarding the availability of the source code (closed source vs. open source).

Table 2.1 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, one with each set of specific protocol attacks. If the servers run in Windows and Linux then they were analyzed in both operating

E-mail Servers (POP3/IMAP4Rev1)	OS	Version	Build Date
602LAN Suite Groupware *(602 Software)*	W	5.0.08.0403	4/8/2008
Citadel* *(Uncensored Communications Group)*	U	7.32	2/17/2008
dovecot*	U	1.1.rc3	3/9/2008
Hexamail Server Corporate	U/W	3.1.0.002	-
hMailServer	W	4.4.1	3/9/2008
IMail Server *(Ipswitch)*	W	2006,23	12/5/2007
Kerio MailServer *(Kerio Technologies)*	U/W	6.5.1	5/12/2008
Mailtraq Email Server *(Fastraq)*	W	2.12.1.2364	5/8/2008
Mdaemon Email Server *(Alt-N Technologies)*	W	9.6.5	6/19/2007
Merak Mail Server *(IceWarp)*	U/W	9.1.0	9/17/2007
NoticeWare Email Server NG	W	4.6.2	4/3/2008
Softalk Mail Server Corporate	W	8.5.1.431	10/30/2007
SurgeMail Mail Server *(NetWin)*	U/W	3.9e	4/10/2008
uw-imap* *(University of Washington)*	U	2007b	6/4/2008
WinGate Email Server *(Qbik)*	W	6.2.2	7/12/2008
xmail*	U/W	1.25	1/3/2008

* - Open source; U - Unix/Linux; W - Windows

Table 2.1: Target POP and IMAP E-mail servers.

systems.

Figure 2.7 shows the contents of the malicious tokens file used in the Value Test algorithm of Figure 2.5. The ability to generate good illegal data was of the utmost importance, i.e., values correct enough to pass through the parsing and input validation mechanisms, but sufficiently erroneous to trigger existing vulnerabilities. Therefore, picking good malicious tokens and payload data was essential. Known usernames and hostnames ("aject") were defined in these tokens, as well as path names to sensitive files (e.g., "./private/passwd"). The special keyword "$(PAYLOAD)" was expanded into various words: 256 random characters, 1000 A's, 5000 A's, a string with format string characters (i.e., "%n%p%s%x%n%p%s%x"), a string with many ASCII non-printable characters, and two strings with several relative pathnames (i.e., "../../../" and "./././"). Depending on the number of messages (and respective parameters), a different number of attacks was produced. Based on the 13 message specifications of the POP protocol, the algorithm created 35,700 test cases, whereas for IMAP, which has a larger and more complex set of messages, there were 313,076 attacks.

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 remote 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 that could eventually be exploited by malicious hackers. Table 2.2 presents a summary of the problems, including the attacks that triggered the vulnerabilities, along with a brief explanation of the unexpected behavior

```
$(PAYLOAD)
aject@$(PAYLOAD)
$(PAYLOAD)@aject
<aject<$(PAYLOAD)>
<$(PAYLOAD)<aject>
"$(PAYLOAD)"aject"
./private/passwd
D:\home\aject\private\passwd
```

(a) POP protocol

```
$(PAYLOAD)
aject@$(PAYLOAD)
$(PAYLOAD)@aject
<aject<$(PAYLOAD)>
<$(PAYLOAD)<aject>
"$(PAYLOAD)"aject"
./private/passwd
D:\home\aject\private\passwd
(\$(PAYLOAD))
"{localhost/user=\"$(PAYLOAD)}"
(FLAGS BODY[$(PAYLOAD) (DATE FROM)])
(FLAGS $(PAYLOAD))
```

(b) IMAP protocol

Figure 2.7: File with malicious tokens for the POP and IMAP protocol.

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

Table 2.2: E-mail servers with newly discovered vulnerabilities.

as seen by the monitor (see last column). All vulnerabilities were detected by some fatal condition in the server, such as a crash or a service denial. 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, such as the attacks that triggered them, so that they could reproduce and correct the problem in the following software release (third column of Table 2.2). Since all servers found vulnerable by AJECT were 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.

2.3.2 Resource Exhaustion Vulnerabilities (DNS servers)

The Domain Name System (DNS) is a network component that performs a crucial role in the Internet [30]. 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, 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.

The experimental validation was conducted with seven known DNS servers: BIND 9.4.2, MaraDNS 1.2.12.05, MyDNS 1.1.0, NSD 3.0.6, PowerDNS 2.9.21, Posadis 0.60.6, rbldnsd 0.996a. 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.

AJECT generated 19,104 different attacks from the DNS protocol specification, using a test case generation algorithm that created message variations with illegal data (but no complex malicious patterns were used here). This test suite also introduces a variation in the attack injection approach by repeating the injection of each attack 256 times without restarting the server application. This allows the resource usage leaks to add up to detectable amounts. We then repeated the injection process with 1024 repeated injections for a selected a few of the attacks with higher usage growth rates. This allowed us to identify false positives and to confirm the presence of resource exhaustion vulnerabilities.

The final resource usage projections (from the 1024 injections) are presented in Table 2.3. Four projections are highlighted in bold, 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 CPU exhaustion, i.e., the CPU has no idle times, than the remaining target systems.

Server	CPU M cycles	Processes	Memory pages
BIND-9.4.2	$\hat{\mathbf{y}} = \mathbf{0.39x + 25.31}$	$\hat{y} = 1.00$	$\hat{y} = 1251.00$
MaraDNS-1.2.12.05	$\hat{y} = 0.18x + 4.85$	$\hat{y} = 1.00$	$\hat{\mathbf{y}} = \mathbf{0.14x + 170.85}$
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	$\hat{y} = 0.19x - 19.61$	$\hat{\mathbf{y}} = \mathbf{0.01x + 7.04}$	$\hat{\mathbf{y}} = \mathbf{2.40x + 4983.49}$
Posadis-0.60.6	$\hat{y} = 0.29x - 0.02$	$\hat{y} = 2.00$	$\hat{y} = 812.00$
rbldnsd-0.996a	$\hat{y} = 0.02x + 1.50$	$\hat{y} = 1.00$	$\hat{y} = 175.00$

Table 2.3: Resource usage projections for the DNS servers.

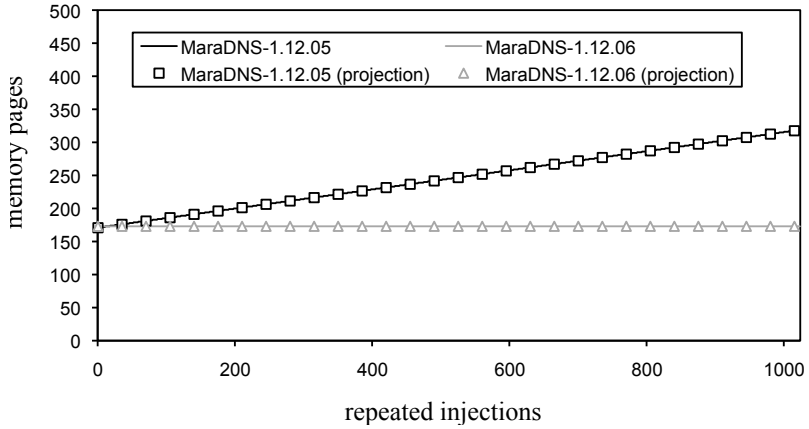


Figure 2.8: Memory consumption in MaraDNS.

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 projection 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 AIP methodology – it allows the user to tradeoff some additional time executing extra injections on a small set of attacks, with a better accuracy of the projections. In some exceptional cases, such as to confirm the attacks that potentially could exploit a vulnerability, we deliberately increased the number of injections (to over 1024) to take the server closer to the exhaustion point.

Another DNS server, MaraDNS, also showed a raising memory usage projection. 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. Being MaraDNS open source, we analyzed the server's code path of execution, which showed us that the server does not support the reverse lookup queries or non-Internet class records, abruptly stopping its processing without freeing a couple of previously allocated variables. Successively inject-

ing any of these two kinds of attacks causes the server to constantly allocate more memory. Both resource-exhaustion vulnerabilities could be exploited remotely in this way to halt the server. Figure 2.8 compares the projections for memory consumption of the vulnerable (1.12.05) and corrected (1.12.06) versions of the server.

2.4 Summary of Results

In the experimental evaluation with our first test suite, AJECT successfully found five email servers vulnerable to different types of malicious traffic. This test suite used an attack generation algorithm that combines illegal data (inferred from the protocol specification) with a set of custom malicious tokens. The malicious traffic generated from this test suite was injected in several email servers. The abnormal behavior of five of the sixteen servers while processing the malicious traffic indicated that they had some vulnerability that could be remotely exploited.

The second test suite was designed to discover resource exhaustion vulnerabilities. This type of vulnerabilities cannot be detected with a conventional testing approach. Therefore, a variation on the attack injection procedure was introduced, allowing the tool to recognize the resource consumption's trend of the target application. Even in well-established open source applications, AJECT discovered two memory leaks in one of the seven DNS servers used in the experiments.

3 Honeypot-based architecture

Besides performing controlled experiments to identify residual vulnerabilities in architectural and software components, we also need to collect data issued from real observations of attacks on the internet to improve our understanding of the behaviour of the attackers and their strategies to compromise the machines connected to the Internet. Also, such data is useful to elaborate statistical models and realistic assumptions about the occurrence of attacks, that are necessary for the evaluation of quantitative security measures. Some examples of such models are presented in deliverable D19[21].

In this section, we present honeypot-based architectures that are aimed at fulfilling this objective. This section is structured as follows. Section 3.1 presents basic background and related work about honeypots. In particular, two types of honeypots offering different levels of interaction to the attackers are discussed. The architectures corresponding to each of these types of honeypots used in the context of CRUTIAL to support data collection are described in sections 3.2 and 3.3 respectively. The honeypots described in these sections offer general services that can be targeted by the population of attackers on the Internet. Such services are not specific to SCADA systems and protocols used in electrical power infrastructures. In order to analyse malicious traffic targeting such protocols, we have deployed in addition a SCADA honeypot that is described in Section 3.4, together with the results derived from the collected data.

3.1 Background

Today, several solutions exist to monitor malicious traffic on the Internet, including viruses, worms, denial of service attacks, etc. An example of the proposed techniques consists in monitoring a very large number of unused IP address spaces by using the so called network telescopes [14, 18], blackholes [15] or Internet Motion Sensors [27]. Another approach used e.g., in the context of DShield [1] and the Internet Storm Center [3], consists in centralizing and analyzing firewall logs or intrusion detection systems alerts collected from different sources around the world. Other popular approaches that have received increasing interest in the last decade are based on honeypots. We can mention e.g., Leurre.com [37], HoneyTank [31], and many national initiatives set up in the context of the honeynet project alliance [43].

A honeypot is a machine connected to the Internet that no one is supposed to use and whose value lies in being probed, attacked or compromised [43]. In theory, no connection to or from that machine should be observed. If a connection occurs, it must be, at best an accidental error or, more likely, an attempt to attack the machine. Thus of the activities recorded should correspond to malicious traffic. This is the main advantage of using honeypots compared to other techniques that consist in collecting and analysing the data logged by firewalls or routers as in DShield [1], where the information recorded is a mixture of normal and malicious traffic.

Two types of honeypots can be distinguished depending on the level of interactivity that they offer to the attackers. Low-interaction honeypots do not implement real functional services. They emulate simple services and cannot be used to compromise the honeypot or attack other machines on the Internet. On the other hand, high-interaction honeypots offer real services to the attackers to interact with which makes them more risky than low interaction honeypots. As a matter a fact, they offer a more suitable environment to collect information on attackers activities once they manage to get the control of a target machine and try to progress in the intrusion process to get additional privileges. It is noteworthy that recently, hybrid honeypots combining the advantages of low and high interaction honeypots have been also proposed, (some examples are presented e.g., [10, 39]).

Most of the currently deployed honeypots on the Internet are low interaction honeypots that are easy to implement and do not present any risk of being used by the attackers for attacking other machines. As an example, the Leurre.com data collection platform set up by Eurecom and to which LAAS contributes is based on the deployment of identically configured honeypots at various locations on the Internet (see Section 3.2).

In the context of CRUTIAL, we are interested in the analysis and the exploitation of data collected from both, low interaction and high interaction honeypots. The first type of honeypots is well suited to easily collect a large volume of data that can be used to characterize the time occurrence of the attacks and their distribution according to their type, origin, etc. In CRUTIAL, we rely on the data collected from the Leurre.com platform to carry out such analyses. The Leurre.com data collection platform is described in Section 3.2. The second type of honeypots is needed to analyse the strategies and the behaviour of the attackers once they succeed in breaking into a target machine and try to progress in order to increase their privileges or carry out malicious actions. This requires the instrumentation of the honeypot with specific mechanisms dedicated to the capture and monitoring of attackers activities and to the control of their activities to prevent the use of the target machine as stepping stone for compromising other machines. In CRUTIAL, we have developed a specific high-interaction honeypot architecture dedicated to this purpose, which is presented in Section 3.3.

The two types of honeypots mentioned above offer general services that can be targeted by the population of attackers on the Internet. Such services are not specific to SCADA systems and protocols used in electrical power infrastructures. In order to analyse malicious traffic targeting such protocols, we have deployed in addition a SCADA honeypot that is described in Section 3.4, together with the results derived from the collected data.

3.2 Leurre.com data collection platform

The data collection environment Leurre.com is based on low-interaction honeypots using the freely available software called honeyd [38]. Since 2003, 80 honeypot platforms have been progressively deployed on the Internet at various geographical locations. As illustrated in Figure 3.1, each platform emulates three computers running Linux RedHat,

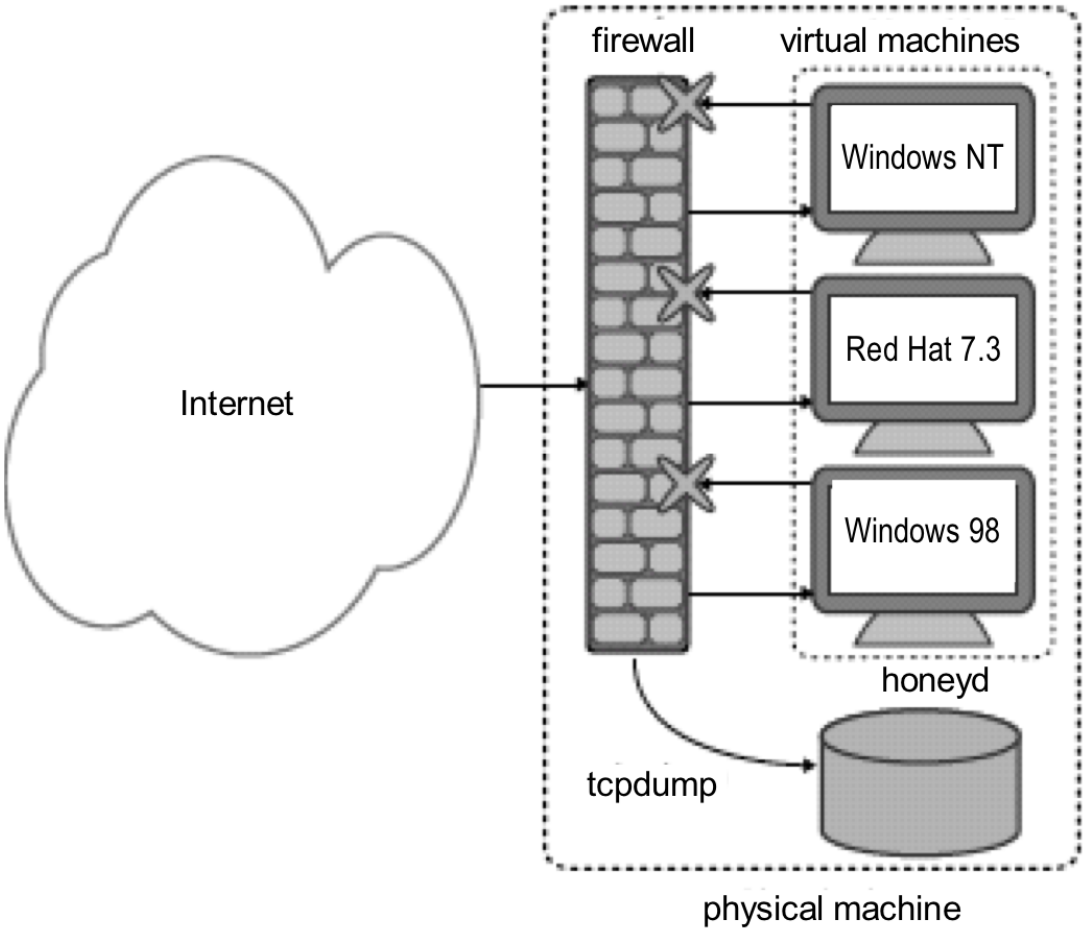


Figure 3.1: Leurre.com honeypot architecture.

Windows 98 and Windows NT, respectively, and various services such as ftp, web, etc. All traffic received by or sent from each computer is saved in tcpdump files. A firewall ensures that connections cannot be initiated from the computers, only replies to external solicitations are allowed. All the honeypot platforms are centrally managed to ensure that they have exactly the same configuration. Every day, the data gathered by each platform are securely uploaded from a trusted machine during a short period of time to a centralized database with the complete content, including payload of all packets sent to or from these honeypots and additional information to facilitate its analysis, such as the IP geographical localization of packets' source addresses, the OS of the attacking machine, the local time of the source, etc. Integrity checks are also performed to ensure that the platform has not been compromised.

The data collected from the honeypot platforms included in Leurre.com is analyzed in deliverable D19, focussing in particular on the distributions of the times betweens attacks observed on the various platforms. The data corresponds to the attacks recorded in the Leurre.com database between February 2003 and August 2007: the total number of attacks recorded is 4 873 564 attacks issued from 3 026 972 different IP addresses.

3.3 A high-interaction honeypot architecture

With low-interaction honeypots, the attackers can only scan ports and send requests to fake servers without ever succeeding in taking control over them. Thus, high-interaction honeypots are needed to allow us to learn about the behaviour of malicious attackers once they have managed to compromise and get access to a new host, and about their tools tactics and motives. We are mainly interested in observing the progress of real attack processes, and monitoring in a controlled environment the activities carried out by the attackers as they gain unauthorized access, capturing their keystrokes, recovering their tools, and learning about their motives.

The most obvious approach for building a high-interaction honeypot consists in using a physical machine and dedicating it to record and monitor attackers activities. The installation of this machine is as easy as a normal machine. Nevertheless, probes must be added to capture and store the activities. Operating in the kernel is by far the most frequent manner to do it. Sebek [39] and Uberlogger [6] operate in that way by using Linux Kernel Module (LKM) on Linux. More precisely, they launch a customised module to intercept relevant system calls in order to capture the activities of attackers. Data collected in the kernel is stored on a server through the network. Communications with the server are hidden on all installed honeypots.

Instead of deploying a physical machine that acts as a honeypot, a more cost effective and flexible approach would be to deploy a physical machine hosting several virtual machines that act as honeypots. Usually, VMware,User Mode Linux (UML) or, more recently, Qemu virtualisation and emulation software are used to set up such virtual honeypots. Some examples of virtual honeypots are presented in [39].

In CRUTIAL, we have decided to build our own virtual high-interaction honeypot that can be easily customized to our needs and experiments. The design choices and the architecture of our honeypot are detailed in the following sections.

3.3.1 Objectives and design needs

Our objective is to set up and deploy an instrumented environment that offers some possibilities to attackers to break into a target system under strict control, and includes mechanisms to log their activities. In particular, we are interested in capturing: 1) the communication traffic going through the honeypot over the network, 2) the keystrokes of the attackers on their terminals, 3) the logins and passwords use, and 4) the programs and tools executed on the honeypot.

The vulnerability to be exploited by the attacker to get access to the honeypot is not as crucial as the activity they carry out once they have broken into the host. That's why we chose for a first implementation to use a traditional vulnerability: weak passwords for ssh user accounts. Our honeypot should not be particularly hardened for two reasons. First,

we are interested in analyzing the behavior of the attackers even when they exploit a buffer overflow and become root. So, if we use some kernel patch such as Pax [4], our system will be more secure but it will be impossible to observe some behavior. Secondly, if the system is too hardened, the intruders may suspect something abnormal and then give up.

In our setup, only ssh connections to the virtual host should be authorized so that the attacker can exploit this vulnerability. On the other hand, any connection from the virtual host to the Internet should be blocked to avoid that intruders attack remote machines from the honeypot. This does not prevent the intruder from downloading code, using the ssh connection[1]. Forbidding outgoing connections is needed for liability reasons. This limitation precludes the possibility of observing complete attack scenarios. Moreover, it might also have an impact on attacker behavior: attackers might stop their attack and decide to never use again the honeypot for future malicious activities. To address this problem, some implementations limit the number of outgoing connections from the honeypot through the use of "rate limiting" mechanisms. Although this solution allows more information about the attack process to be captured, it does not address the liability concerns. A possible solution that we have investigated is to redirect outgoing connections to a local machine, while making the attackers believe that they are able to bounce from the honeypot. This solution is detailed in [5].

As regards the mechanisms that need to be included in the honeypot to log attackers activities, capturing network traces using e.g., tcpdump as usually done in the case of low interaction honeypots would not be enough. We also need to record the activities carried out by the attackers on their terminal and the logins and passwords tried. This requires the modification of some OS calls as well as the ssh software. Additional mechanisms are also needed to regularly archive the information logged in a secured way.

In the next section, we describe the architecture of the proposed honeypot and describe the mechanisms that have been implemented for logging and archiving the activities of the attackers.

3.3.2 Architecture and implementation description

To fulfill the objectives listed in Section 3.3.1, an open source implementation of the target operating system is needed. It is also important to be familiar with, and to have a deep knowledge of the selected operating system in order to be able to keep the activities of the attackers under strict control. For these reasons, we have decided to use GNU/Linux. As regards the implementation of the virtual machines, our choice was for a virtualisation software such as VMware or Qemu. Compared to VMware, Qemu presents the advantages of being freely distributed and open source. Indeed, we have developed a first implementation based on VMware that does not include the redirection mechanism. This implementation was then upgraded at a second stage to include the redirection mechanism, using Qemu. For

[1] We have sometimes authorized http connections for a short time, by checking that the attackers were not trying to attack other remote hosts.

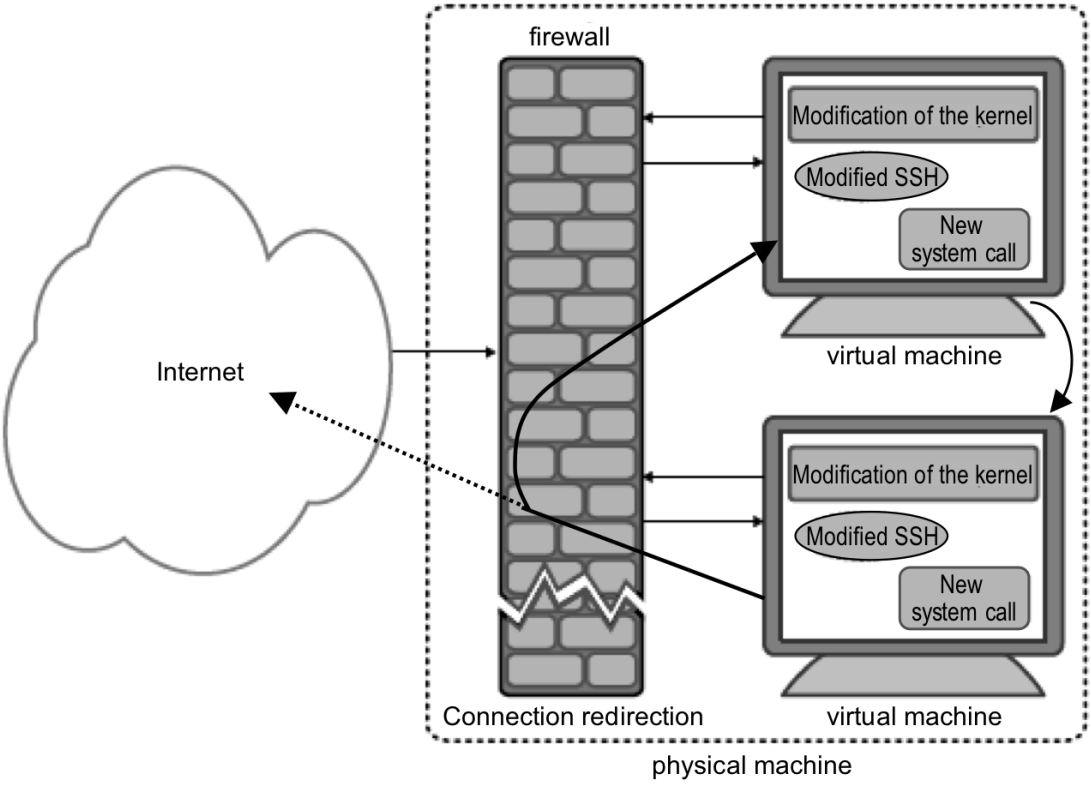


Figure 3.2: Overview of the high-interaction honeypot architecture.

both implementations, the honeypot was developed using a standard Gnu/Linux installation with kernel 2.6 with the usual binary tools. No additional software was installed except the http apache server. This kernel was modified as explained in the next subsection.

An overview of the general architecture of the honeypot is presented in Figure 3.2. The mechanisms implemented for capturing the attackers activities are highlighted. These mechanisms are described briefly in the following.

3.3.2.1 Data collection mechanisms

In order to log what the attackers do on the honeypot, we modified some drivers functions (`tty_read` and `tty_write`), as well as the `exec` system call in the Linux kernel. The modifications of `tty_read` and `tty_write` enable us to intercept the activity on all the terminals of the system. The modification of the `exec` system call enables us to record the system calls used by the intruder. These functions are modified in such a way that the captured information is logged directly into a buffer of the kernel memory of the honeypot itself. This means that the activity of the attacker is logged on the kernel memory of the honeypot itself. This approach is not common: in most of the approaches we have studied, the information collected is directly sent to a remote host through the network. The advantage of our approach is that logging through the kernel is difficult to detect by

the attacker (more difficult at least than detecting a network connection). It is noteworthy that the logging activity is executed on the real host not on the virtual, thus it is not easily detectable by the intruder (he cannot find anything suspicious in the list of processes for example). Furthermore, the data is compressed using the LZRW1 algorithm before being logged into the kernel memory.

Moreover, in order to record all the logins and passwords tried by the attackers to break into the honeypot we added a new system call into the kernel of the virtual operating system and we modified the source code of the ssh server so that it uses this new system call. The logins and passwords are logged in the kernel memory, in the same buffer as the information related to the commands used by the attackers. As the whole buffer is regularly stored on the hard disk of the real host, we do not have to add other mechanisms to record these logins and passwords.

The activities of the intruder logged by the honeypot are preprocessed and then stored into an SQL database. The raw data are automatically processed to extract relevant information for further analyses, mainly: i) the IP address of the attacking machine, ii) the login and the password tested, iii) the date of the connection, iv) the terminal associated (tty) to each connection, and v) each command used by the attacker.

3.3.2.2 Connection redirection mechanism

As indicated in Section 3.3.1, this mechanism is aimed at automatically and dynamically redirecting outgoing Internet connections from the honeypot to other local machines. The goal is to make the attacker believe he can connect from the honeypot to hosts on the Internet, whereas in reality, the connections are simply redirected towards another honeypot. The main idea is illustrated by the example presented in Figure 3.3.

In this example, *b, c* and *d* are honeypots and *a, e, f* and *g* are machines on the Internet. An attacker from Internet host *a* breaks into honeypot *b* (connection 1). From this honeypot, the attacker then tries to break into Internet host *e* thanks to connection 2. This connection is blocked by our mechanism. The attacker then tries another connection 3 towards Internet host *f*. This connection is accepted and automatically redirected towards honeypot *c*. The attacker is under the illusion that his connection to *f* has succeeded, whereas it has merely been redirected to another honeypot. The attacker tries to establish another connection 4 towards Internet host *g*. Similar to connection 3, this connection is accepted and automatically redirected towards honeypot *d*. The attacker finally initiates another connection (5) to Internet host *g* from host *f* (in reality, from host *c*). This connection is also accepted and is redirected towards honeypot *d*.

This mechanism allows the observation of attackers activity on different hosts. In general, a honeypot allows the activity of the attacker to be observed at only one side of the connection. The other connection end is the machine that interacts with the honeypot. For all redirected connections, we can observe an attacker on both connection ends.

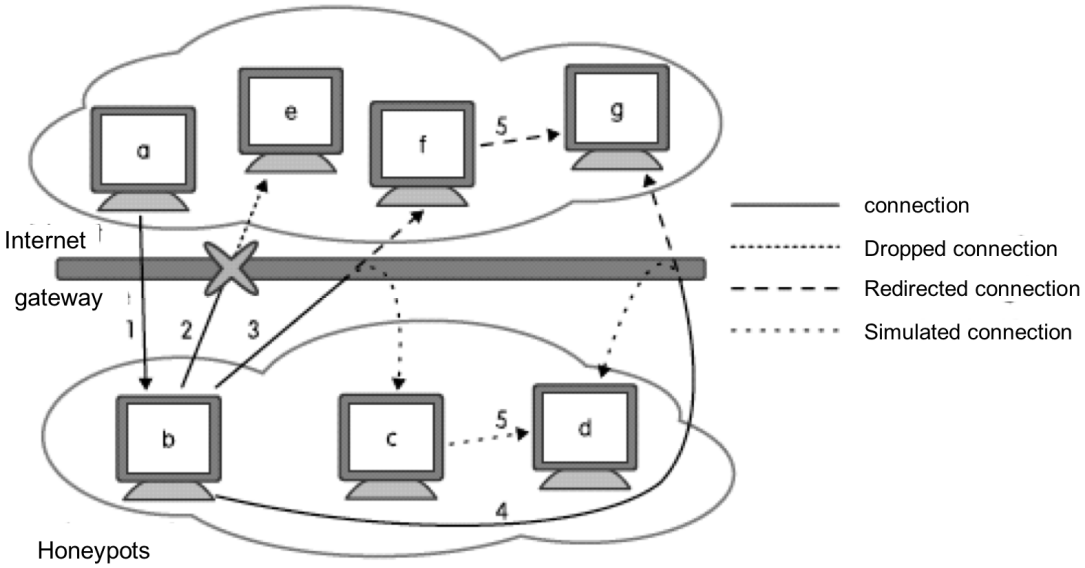


Figure 3.3: Connection redirection principles: example.

On the other hand, it is possible for a clever attacker to see through the hoax. For example, in Figure 3.3, suppose the attacker already controls the machines *a, e* and *f*. He can then check, after establishing connection 3, if the machine he is connected to really is machine *f*. This limitation does exist; however we believe that many attackers will not systematically do such checks, in particular if the attack is carried out by non-sophisticated automatic scripts. Just as low-interaction honeypots provide some useful albeit limited information, more attack information would be gleaned from systems that implement our redirection mechanism than those that do not.

The dynamic redirection mechanism has been implemented in the Gnu/Linux operating system through the NETFILTER firewall of the kernel. This firewall allows the interception and the modification of the packets flowing through the IP stack. As illustrated in Figure 3.4, the mechanism includes three components:

- the *redirection module* (inside the kernel) extracts the received packets.
- the *dialog_handler* decides whether the extracted packets must be redirected or not. Several algorithms can be used for this purpose and for the distribution of the redirected connections among the local honeypots.
- the *dialog_tracker* maintains the link between the redirection module and the dialog_handler. This way, the implementation of the dialog_handler can be totally independent of the architecture and the operating system. In particular, the dialog_handler and the dialog_tracker could be run on different machines.

The implementation of the three components is described in detail in [5], with experimental results showing that the redirection mechanism does not lead to a significant

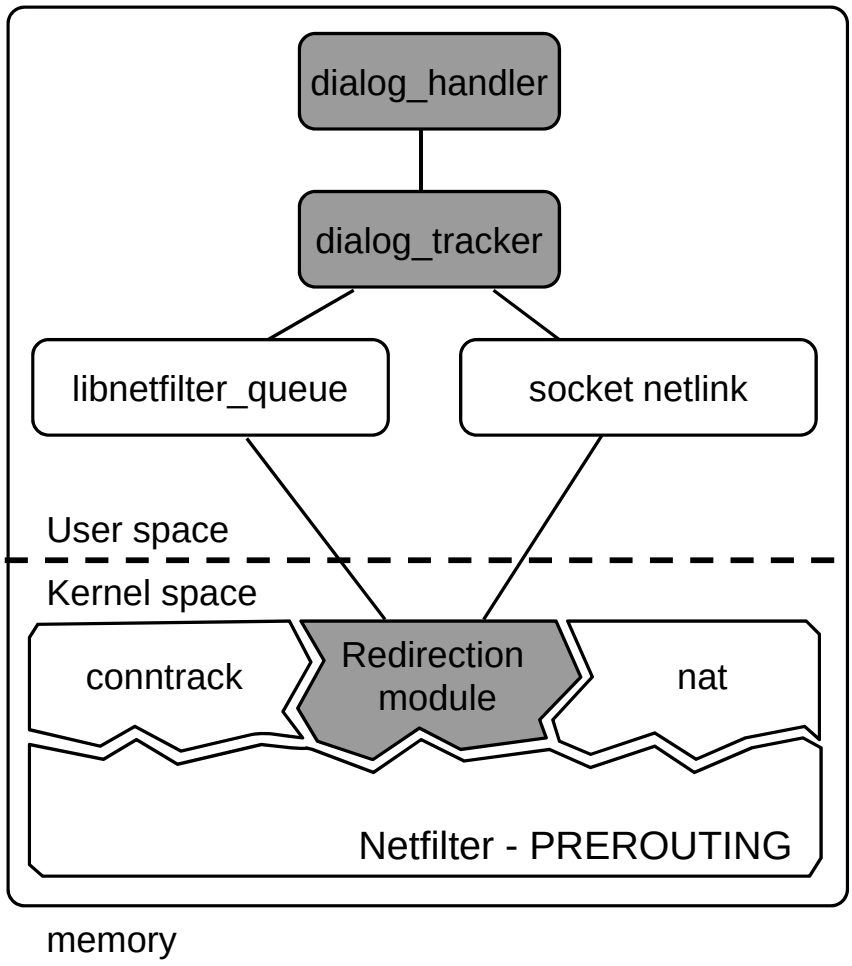


Figure 3.4: Redirection mechanism implementation through NETFILTER.

latency due to the interception and redirection of connections. It is important to ensure that the overhead is sufficiently low as to prevent detection of the redirection mechanism by the attacker.

3.3.3 Deployment

In order to collect real data about attackers activities and to validate our set up, we have deployed our virtual high-interaction honeypot on the Internet. Figure 3.5 gives an overview of the most recent version of the honeypot. Three Gnu/Linux virtual machines M1, M2 and M3 have been set up. Each machine includes usual desktop software (Compiler, Text editors, etc.). Only M1 and M2 are accessible from the Internet. M3 is accessible from the other virtual machines. The only input connections authorized by the firewall are those targeting the ssh service. Concerning output connections from the honeypot, two virtual machines R1 and R2 are used by the redirection mechanism.

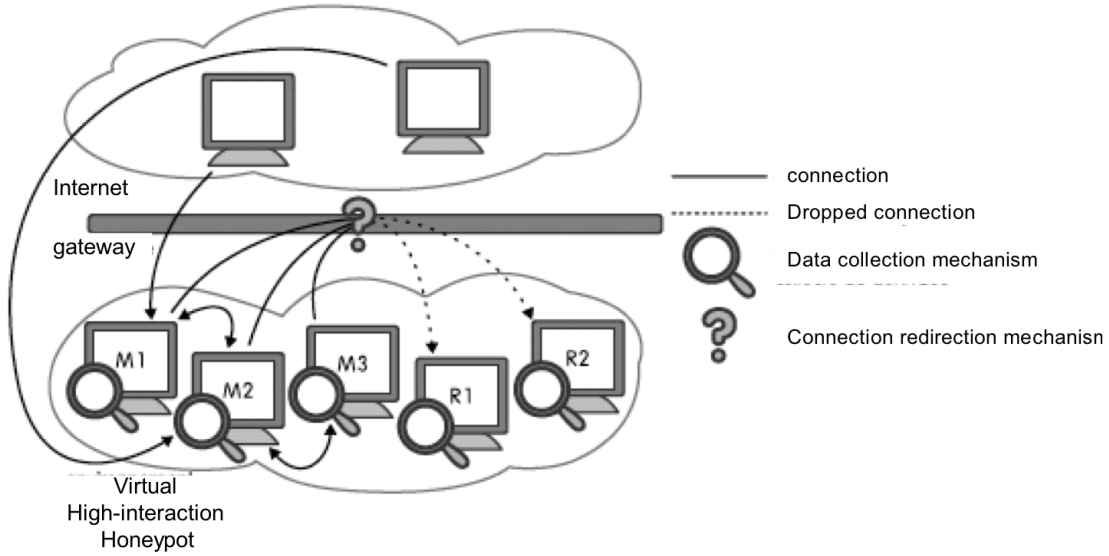


Figure 3.5: Overview of the most recently deployed version of the honeypot.

The deployment was carried out in two stages. The first deployed version did not include the redirection mechanism and the virtual machines have been set up using VMware. This setup was then upgraded using Qemu and included the redirection mechanism.

In the beginning of the experiment (approximately one and a half month), we deployed a machine with a ssh server, offering no weak account and password. We have taken advantage of this observation period to determine which accounts were mostly tried by automated scripts. Using this acquired knowledge, we have created 17 user accounts and we started looking for successful intrusions. Some of the created accounts were among the most attacked ones and others not. As we already explained in previous sections, we have deliberately created user accounts with weak passwords (except for the root account).

3.3.4 Overview and high-level analysis of the data

As illustrated on Figure 3.6, we can distinguish two main steps for the activities recorded for each user account, identified by the durations $\tau 1$ and $\tau 2$. The first one measures the duration between the creation of the account and the first successful connection to this account, and the second one measures the duration between the first successful connection and the first real intrusion (i.e., a successful connection with commands). Table 3.1 summarizes these durations (*UAi* means *User Account i*).

The second column indicates that there is usually a gap of several days between the time when a user account is successfully found and the time when someone logs into the system with this account to issue some commands on the now compromised host. This is somehow surprising. The particular case of the *UA5* account is explained as follows: an intruder succeeded in breaking the *UA4* account. This intruder looked at the contents of the

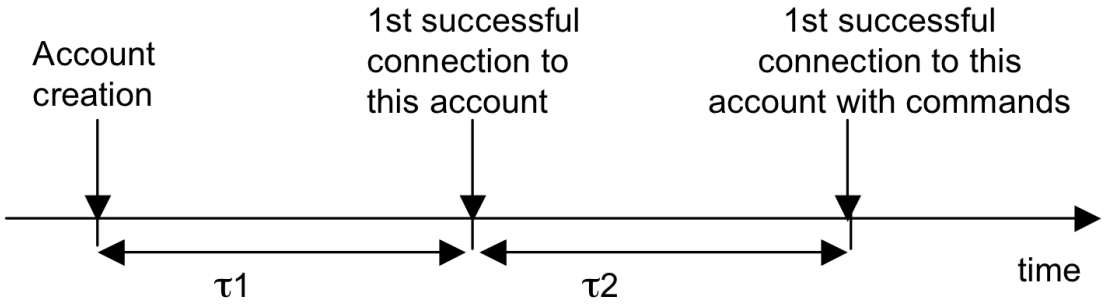


Figure 3.6: Definitions of $\tau 1$ and $\tau 2$.

`/etc/passwd` file in order to see the list of user accounts for this machine. He immediately decided to try to break the *UA5* account, the first successful connection is also the first intrusion.

User Acount	$\tau 1$	$\tau 2$
UA1	1 day	4 days
UA2	1.5 day	4 minutes
UA3	15 days	1 day
UA4	5 days	10 days
UA5	5 days	0
UA6	1 day	4 days
UA7	5 days	8 days
UA8	1 day	9 days
UA9	1 day	12 days
UA10	3 days	2 minutes
UA11	7 days	4 days
UA12	1 day	8 days
UA13	5 days	17 days
UA14	5 days	13 days
UA15	9 days	7 days
UA16	1 day	14 days
UA17	1 day	12 days

Table 3.1: $\tau 1$ and $\tau 2$ values for each user account

As regards the data collected during the experiment, the number of ssh connection attempts to the honeypot that we have recorded is 552362 (we do not consider here the scans on the ssh port). This represents about 1318 connection attempts a day. Among these 552362 connection attempts, only 299 were successful. The total number of accounts tested is 98347 and the number of different IP addresses observed on the honeypot was 654. This represents a significantly large volume of data on which statistical analyses can be carried out to extract relevant information about the observed attack processes.

The detailed analysis of the high interaction honeypot data is presented in [34]. In

the following, we outline some of the conclusions derived from the analysis. The statistical modelling of the observed times between attacks distribution is presented in deliverable D19.

As illustrated in Figure 3.6, two main steps of the attack process can be distinguished: 1) the first one generally carried out by means of automatic tools, concerns brute-force dictionary attacks aimed at gaining access to the honeypot, and 2) the second step concerns the activities carried out by the attackers once they succeed in breaking into the system (i.e., intrusions). The methodology described in [34], based on the analysis of the keystrokes hit by the attackers on the terminal and the transmission mode of the data exchanged with the attackers (per character, or per block) led us to conclude that the intrusions have been generally performed by human beings. Also, the comparison of the IP addresses that performed the dictionary attacks and those associated to the intrusions revealed that the intersection between the two sets is empty. Moreover, none of these IP addresses has been observed on the low interaction honeypots deployed in the Leurre.com platform. Thus, it is likely that the different types of attacks are carried out by different communities, using different sets of machines, each one specifically dedicated to a specific type of attack.

The analysis of the data characterizing the dictionary attacks has been focused on the analysis of the pairs (username/password) tried by the attackers and the identification of the dictionaries used to perform these attacks. It appears that the attackers are using and sharing a few dictionaries, including a large number of username/password pairs that are commonly used in Unix and Windows systems, as well pairs with identical user names and passwords. Moreover, the attackers do not seem to adapt their dictionaries to the geographic location of their victim.

Of course, our experiment at this stage is not sufficient to derive general conclusions.We would need to deploy the honeypot at other different locations. Also, it would be worth running the same experiment by opening other vulnerabilities into the system and verifying if the identified activities remain the same. This is something that we plan to investigate in the future.

3.4 SCADA honeypot

In this section we present a low interaction honeypot architecture exposing specific services and ports that are characteristic of SCADA systems, used e.g., in process control electrical power infrastructures. The objective is to analyse malicious traffic targeting such services and ports and compare the observed trends with the data collected from the low and high interaction honeypots described in the previous sections.

The SCADA honeypot architecture is outlined in Section 3.4.1 and the results derived from the collected data are analyzed in Section 3.4.2. Finally, Section 3.4.3 discusses the relevance of the results obtained in our experiment.

3.4.1 SCADA honeypot architecture

The development of honeypot architectures that are designed to emulate specific services used in SCADA systems has been investigated recently in order to better understand the threats targeting critical infrastructures. Two main implementations of SCADA honeypots have been identified. The first one developed by Matt Franz and Venkat Pothamsetty from the CISCO Critical Infrastructures Assurance Group (CIAG)[2] is based on a low-interaction honeypot solution using honeyd like scripts. The idea consists in simulating a whole SCADA network, including the devices, protocols, and applications in a single Linux box, using multiple scripts. An open-source implementation has been made available. This implementation includes python scripts for a honeyd low interaction honeypot simulating FTP, HTTP and TELENET servers and a partial implementation of the MODBUS server (port 502) corresponding to the Modbus protocol used in some SCADA systems. The honeypot implements responses to top queries of the Modbus protocol (code 1, code 8 and 16) and returns "error codes" to others.

The second implementation of SCADA honeypots has been proposed by Digital Bond[3]. It is based on a high-interaction honeypot using VMware virtual machines. The SCADA honeypot is designed to appear as a PLC running ModBus TCP (port 502), that is configured with realistic sample data from the ten largest electrical power utilities in the US. The implementation uses open source applications including Linux as an operating systems, and the HTTP, FTP, TELNET and SNMP services in addition to Modbus TCP.

In the context of CRUTIAL, we have decided to deploy a SCADA honeypot that is based on the first implementation. Indeed, this implementation offers the disadvantage of being easy to integrate into our honeyd based low interaction honeypot described in Section 3.2, and therefore easy to deploy in a short time to be able to collect data before the end of the project.

The architecture that we have set up is described in Figure 3.7. It is built as an extension of the low-interaction honeypot architecture presented in Figure 8. Besides the initial virtual machines M1, M2 and M3, a virtual machine implementing the SCADA related honeyd script is included. Additionally, we have monitored a number of ports that are used in other protocols implemented in SCADA services and devices. These include for example port 2404 used in IEC 60870-6, 60870-5-104 and 61850 protocols, port 2020 used in the DNP3 protocol, and ports 50, 51 and 500 used in the IPsec protocol. The IPsec ports are included in the analysis because this protocol is generally used for securing the communications accesses to the control center. The list of monitored ports is also determined based on the analysis of published vulnerabilities targeting SCADA systems. As an illustration, Table 3.2 lists some examples of SCADA-related vulnerabilities published at the CVE website (http://www.mitre.org), with an indication of the associated TCP port or protocol.

[2] http://scadahoneynet.sourceforge.net

[3] http://www.digitalbond.com/index.php/2006/08/23/scada-honeynet

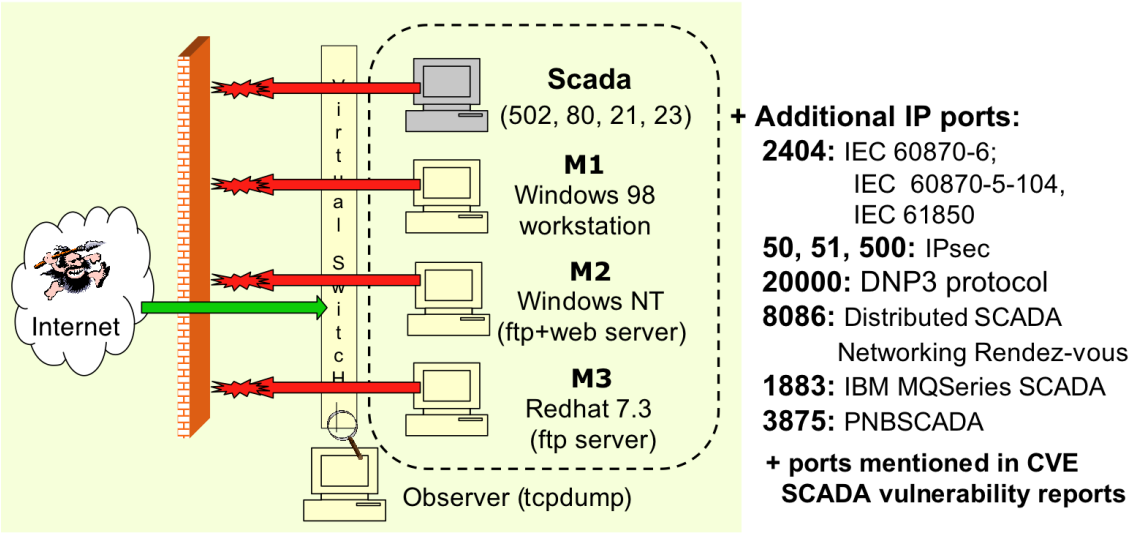


Figure 3.7: SCADA honeypot architecture.

Vulnerability identifier	Description
CVE-2008-4322	Stack-based buffer overflow in RealFlex Technologies Ltd. RealWin Server 2.0, as distributed by DATAC, allows remote attackers to execute arbitrary code via a crafted FC_INFOTAG/SET_CONTROL packet (**TCP port 910**).
CVE-2008-2639	Stack-based buffer overflow in the ODBC server service in Citect CitectSCADA 6 and 7, and CitectFacilities 7, allows remote attackers to execute arbitrary code via a long string in the second application packet in a TCP session on **port 20222**
CVE-2008-2474	Buffer overflow in x87 before 3.5.5 in ABB Process Communication Unit 400 (PCU400) 4.4 through 4.6 allows remote attackers to execute arbitrary code via a crafted packet using the (1) IEC60870-5-101 or (2) IEC60870-5-104 communication protocol to the X87 web interface (**TCP port 2404**)
CVE-2008-2005	The SuiteLink Service (aka slssvc.exe) in WonderWare SuiteLink before 2.0 Patch 01, as used in WonderWare InTouch 8.0, allows remote attackers to cause a denial of service (NULL pointer dereference and service shutdown) and possibly execute arbitrary code via a large length value in a Registration packet to **TCP port 5413**, which causes a memory allocation failure.
CVE-2008-0176	Heap-based buffer overflow in w32rtr.exe in GE Fanuc CIMPLICITY HMI SCADA system 7.0 before 7.0 SIM 9, and earlier versions before 6.1 SP6 Hot fix - 010708_162517_6106, allow remote attackers to execute arbitrary code via unknown vectors. (**TCP port 32000**)

Table 3.2: Examples of CVE SCADA related vulnerability reports

3.4.2 Deployment and data analysis

We have deployed the SCADA honeypot on the Internet starting on 1st April 2008. The data analyzed in this section correspond to the attacks observed on the honeypot until December 8, 2008, corresponding to 9 month observation period.

Table 3.3 summarizes the activity observed on different ports of four virtual machines of the honeypot including the SCADA. Besides SCADA related ports, we also provide the activities on traditional http and FTP ports to have an idea about the amount of traffic received by the honeypot.

	Modbus 502	IEC 60870 2404	IPSEC 50,51,500	DNP 20000	32000	910	8086	HTTP 80	FTP 21
SCADA	0	0	0	36	129	1	0	17895	705222
M1	0	0	0	46	133	0	2	2351	2690191
M2	0	0	0	46	133	0	2	77849	20977724
M3	0	0	0	46	133	0	2	8484	1918628

Table 3.3: Activity observed on the virtual machines for different ports (number of packets exchanged)

The first observation is that the ports associated to Modbus (that is used in several implementations of SCADA systems) and to the IEC protocols 60870-6, 60870-5-104 and 61850 (supporting the communication between the control Centers and the substations in the CRUTIAL architecture) did not exhibit any activity during the data collection period, on any of the four virtual machines of the honeypots. The same observation applies to the ports associated to the IPsec protocol used for securing the communications. Most of the activities that could be associated to SCADA related vulnerabilities and protocols have been observed on port 20000 typically used in the DNP3 protocol and to port 32000[4] mentioned e.g., in the CVE-2008-0176 vulnerability report (see Table 3.2).

Looking more in detail at the activities that targeted the SCADA related ports at one of the four virtual machines, these activities originated from 60 IP addresses, among which only 18 IP addresses did not target the SCADA virtual machine. Generally, the attackers scanned simultaneously the four virtual machines targetting a single port (e.g., 32000) or several ports successively (e.g., 20000 and 10000). Overall, we have observed 125 attack sessions, among which only 47 sessions (i.e., 38%) did not scan all the virtual machines. An attack session gathers the activity received from an IP address such that the time between the first packet and the last packet does not exceed one hour.

Another noteworthy observation is that mostly all the addresses that targeted port 32000 did not scan other ports (80%). The situation is different concerning port 20000. In

[4]It is noteworthy that this port is also used as the default port for several services and protocols, e.g., the Mercur Mail server, in remote administration, Java wrapper server, the IceWarp Mail and the Merak Mail Server.

particular, we have noticed that several IP addresses scanned the sequence 10000-20000, or the sequence of ports: 10000-10001-20000-20001. Such behaviour has been also observed by the SANS Internet Strom Center[5].

3.4.3 Discussion

The data that we have collected so far from the SCADA honeypot provide some preliminary evidence about the existence of malicious traffic targeting specific ports used in SCADA related protocols such as DNP, or for which some SCADA related vulnerabilities have been published. However, the corresponding data is still scarce and is not sufficient to derive more solid conclusions about the existence of real threats targeting SCADA systems. On one hand, uncertainty remains concerning the potential correlation between the fact that some activity has been observed on the corresponding ports and the conclusion that by doing so the attackers were willing to perform SCADA related attacks. Indeed, some of the ports that we have monitored (e.g., 32000) are also used in other services that are not necessarily related to SCADA systems. More evidence would be needed to conclude about the real motivation of the attackers. In particular, in the short term, it is important to continue the analysis of the activities recorded by the SCADA honeypot considering a longer period of time. It would be interesting to investigate if the other ports (e.g., MODBUS and IEC 60870-6, 60870-5-104 and 61850) for which we did not observe any activity still remain inactive. At a longer term, it would be relevant to develop and deploy a high-interaction SCADA honeypot that would offer real services to the attackers so that we can observe their activities and have better insights about their motivation.

[5]See e.g., http://isc.sans.org/diary.html?storyid=2067.

4 Experimental Evaluation of the Fosel Architecture

This chapter presents a set of experiments that we have conducted to verify the ability of the Fosel architecture (Filtering with the help of Overlay Security Layer) [12, 22] to mitigate the effects of DoS attacks. As explained in CRUTIAL deliverable D18 [22], the goal of Fosel architecture is to design an efficient and well-suited filter to drop malicious traffic effectively. Two main characteristics of Fosel architecture are a) precise identification of legitimate traffic and independent from DoS attack types b) reduce the processing time.

There are two classes of DoS attacks: infrastructure-level and application-level attacks [50]. In the former, attackers directly attack the resources of the service infrastructure, such as the networks and hosts of the application services; for example, attackers send floods of network traffic to saturate the target network. In contrast, application-level attacks are through the application interface; for example, attackers overload an application by sending it abusive workload, or malicious requests which crash the application. This work focuses on infrastructure-level DoS attack. However, application-level attack can be solved by authenticating the received traffic (techniques such as IPsec).

Figure 4.1 shows a high level overview of the implemented test-bed. First some points:

- Attackers know the IP (location) of the application sites (targets) and also IPs of the overlay nodes.
- Attackers do not know the location of green nodes in the Fosel architecture.
- All machines in the experiments ran Windows XP.
- We use socket programming on Windows for communication.
- All communication between target and legitimate clients (other application sites) are encrypted by DES encryption algorithm. The symmetric key, K, has already been shared among them.

In the following we describe the key components used in the empirical study and the resources used in the experiments.

4.1 Software Environment

The experiments use four key software components: the selective filter, network traffic monitor, a client interface to send any type of request to the server (an application site) and a DoS attack toolkit.

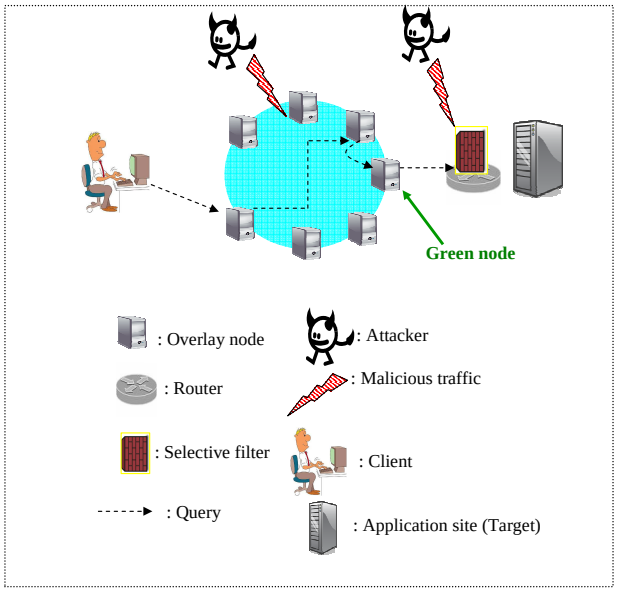


Figure 4.1: High-level overview of implemented testbed

4.1.1 Selective Filter

The selective filter has been implemented in C++ and installed on the server (the target). The server ran Windows XP. The selective filter can receive data over both TCP and UDP protocols.

4.1.2 Network Traffic Monitoring

We have implemented a network traffic monitoring software to calculate the total number of arrived packets. When the arriving rate per second is above a given threshold, the software alarms the application site (target) that it is under DoS attack. We use Raw socket concepts to implement network traffic monitoring software.

4.1.3 User Interface

We have designed a simple toolkit to generate user requests. This toolkit generates any type of requests (http request, FTP request, and etc.) and then measures the response time for each of the requests. This allows us to evaluate user access and collect statistics which characterize user experience performance.

4.1.4 DoS Attack Toolkit

UDPFlood [25] is a UDP packet sender. It sends out a flood of UDP packets to the specified IP and port at a controllable rate. Packets can be made from a typed text string, a given number of random bytes or data from a file. Figure 4.2 shows UDPFlood software interface.

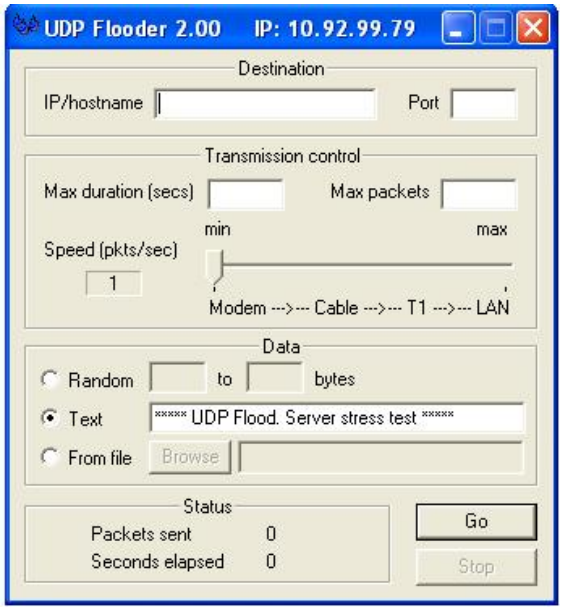


Figure 4.2: UDPFlooder software interface

4.2 Physical Resources

To evaluate the Fosel architecture, we use a test-bed consisting of Emulab machines [2] located at various sites in the continental US. Emulab is a network testbed, which allows you to specify an arbitrary network topology, giving you a controllable, predictable, and repeatable environment, including PC nodes on which you have full "root" access, running an operating system of your choice. We use the GT-ITM topology generator to generate the Internet topology. The GT-ITM generator can generate a random graph based on input parameters closely resembles the Internet Topology. Figure 4.3 shows our experimental topology with 69 nodes.

The experimental topology includes 11 LAN (Local Area Network), 60 PC nodes and 9 routers. Each LAN contains $4 \sim 7$ nodes. We use 60 PC nodes to create the Chord overlay network [44] at the top of the experimental topology.

All PC machines run Windows XP. Some PC machines run on 850 MHZ Intel Pentium III processors with 512 MB RAM; while others run on 1.7 GHZ Pentium IV with 512 MB RAM and a few of them run on 600 MHZ Intel Pentium III with 256 MB RAM. All

machines are connected by 100 Mbps Ethernet switches and routers. Using these fairly distributed machines, we have been constructing our network of overlay nodes by running small proxy software on each of the participating machines (see Chord network implementation).

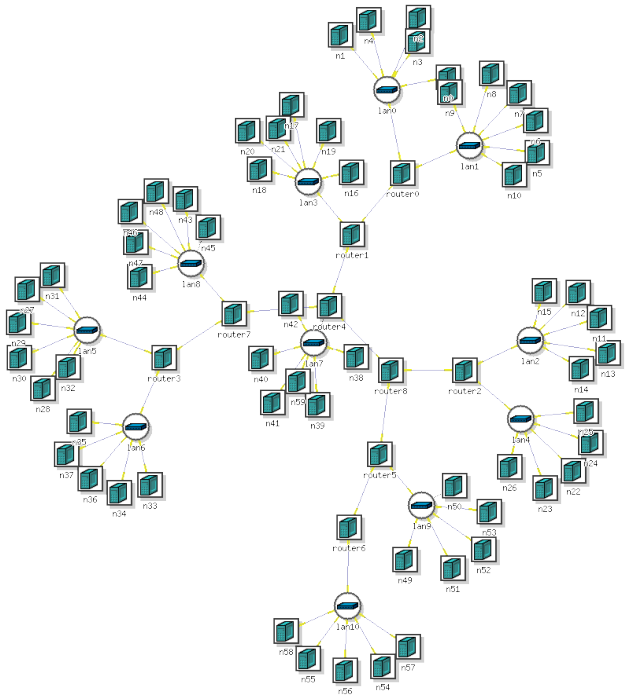


Figure 4.3: Experimental topology

A client and server run on two nodes 1.7 GHZ Pentium IV with 512 MB RAM on a separated topology in the same area as overlay network located (continental US). Attackers run on at least one node and at most 40 nodes based on the experiments. All attackers are located in Belgium. All attackers' machines run Windows XP.

4.2.1 Chord Overlay Network Implementation

The Chord overlay network is composed of proxy nodes. Proxies are software programs, each one installed on a separate node in the Emulab. Each proxy has a unique 160-bit Chord node identifier, produce by a SHA hash of the node's IP address. We have implemented the Chord network in an iterative style [44]. Each proxy listens to user connection requests. Requests are verified through a firewall (Emulab firewalls that can be augmented with user-specified rules) and then legitimated requests routed through the Chord to the green nodes of destination.

Routing legitimate packets through Chord overlay network

In the Chord network, each node is assigned a ID in the range $[0, 2^m]$, the amount of m depends on the total number of the overlay nodes (in our case $m = 6$ because we have 60 overlay nodes). In the Emulab as IPs are in the range [155.98.0.0, 155.98.255.255], we use last two bytes of IP addresses for nodes' ID in our implementation (of course in real world SHA-1 hash function of IP address is used for ID). The nodes in the overlay are ordered by these identifiers. The ordering can be viewed conceptually as a circle, where the next node in the ordering is the next node along the circle in the clockwise direction. Each node maintains a table that stores the identity of m other overlay nodes (this table is called finger table). The ith entry in the table is the node whose identifier x equals or, in relation to all other nodes, most immediately follows $x + 2^{i-1} mod\ n^m$.

When overlay node x receives a packet destined for ID y, it forwards the packet to the overlay node in its finger table whose ID precedes y by the smallest amount. For example, suppose in the figure 4.4, ID of the green node (the final desired node in the overlay) is 56. The node with ID-8 receives a packet from an application site and it should deliver it to the node ID-56. The node ID-8 routes the packet based on its finger table to node ID-42. The node ID-42 routes the packet to the node ID-51 according to its finger table and then node ID-51 routes the packet to the node ID-56 (the green node). Chord routes packets around the overlay "circle" progressively getting closer and closer to the desired node, visiting log(N) nodes.

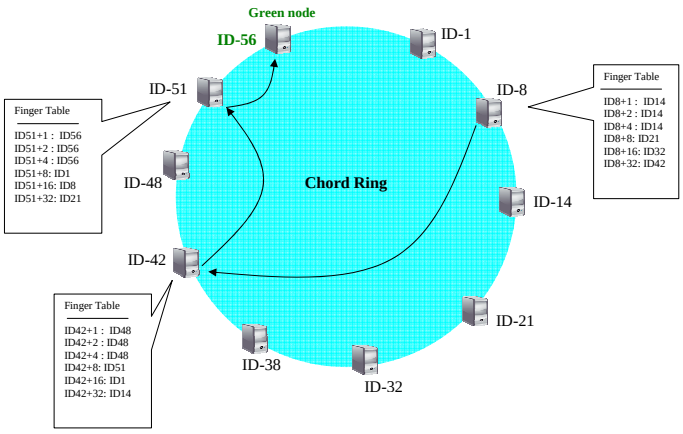


Figure 4.4: Chord overlay network routing protocol

Stabilization protocol

The Chord overlay network runs stabilization protocol every 500 ms. In fact every node of Chord runs stabilization protocol periodically (500 ms in our case) to learn about newly joined nodes and update its finger table (routing table) as well. In stabilization period live nodes remove dead nodes (attacked nodes) from their finger table and add newly joined

nodes to their finger table.

4.3 Experiments and Results

To understand and evaluate the basic performance of the Fosel architecture, we compare performance metrics for direct connection access and the Fosel architecture. In direct connection access, clients access an application site directly and without any interface such as overlay network (no green nodes are used). Direct connection access utilizes a straightforward filter such as commercial filters to drop DoS packets. In summary, the straightforward filter in direct application access verifies the signature of the packet (RSA with 2048 bit) and in case of wrong signature drops the packet.

There are two possibilities to attack the system: a) attack against an application site (target) and b) attack against the overlay network.

4.3.1 Attack against an Application Site

In these experiments, a client tries to download a pdf file (size=671 KB) via the FTP protocol. We study the performance of the system by two metrics: response time and failure rate.

Let us recall that, when under DoS attack, Fosel on one hand replicates the legitimate messages to be delivered C times, on the other hand the selective filter drops messages (both malicious and legitimate) *without processing them*, with a given probability (1-P). The interested reader can find further details in Deliverable D18, Section 3.2.3.2.

Figure 4.5 shows the average response time for a client to download a file of given size (671KB). The x-axis is the probability of message acceptance (P). The attack rate is fixed at 500 packets/second. The Fosel architecture has faster response time than direct application access. Results show that the Fosel architecture improves response time more than 40% (in average) compared to direct connection access in case of DoS attacks. The result shows that by selecting a small value for P, the response time is decreased.

A client's message may fail to be processed by the target due to two reasons: a) the message may be lost due to huge amount of traffic. In our experiments if a message was not accepted by the receiver's socket within 21 seconds, it is dropped automatically. We call this, network loss. b) Any message also is dropped with probability of (1-P) where P is the packet acceptance probability (due to the Fosel policy).

Figure 4.6 shows failure rate for different values of C (number of message copies) when attack rate varies along the x-axis. In this experiment we hold P at 0.25 (it means 1/4 of packets are processed and 3/4 of packets are dropped without any processing). The experimental result shows that when the attack rate is increased the failure rate increases

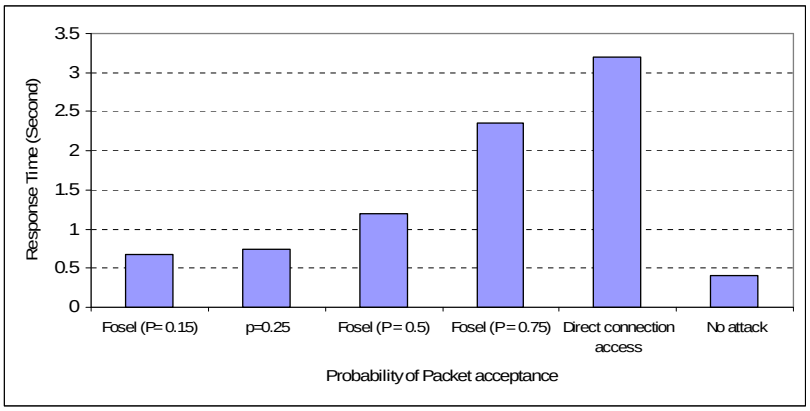


Figure 4.5: Average response time to download a file (size=671KB)

as well. This is evident because traffic is increased and more messages are lost by the socket timeout (network loss). Failure rate in the Fosel architecture is far less than direct connection access. In addition by choosing bigger C, failure rate is reduced more in the Fosel architecture.

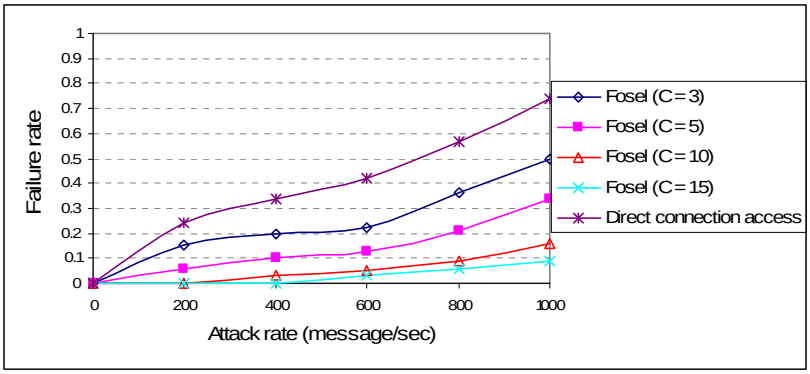


Figure 4.6: Failure rate vs. attack rate

Figure 4.7 shows response time (downloading a 671KB pdf file) for different values of P when attack rate varies along x-axis. Response time is increased when attack rate is increased because more malicious packets are processed before the legitimate request is processed. It shows that response time is decreased when a smaller P is selected because less malicious packets are processed.

So by selecting a suitable values for P and C, we can have an efficient and fast filter against DoS attacks.

4.3.2 DoS Attacks against the Overlay Network

In this section we discuss DoS attacks against the overlay network, one of the main element of the Fosel architecture.

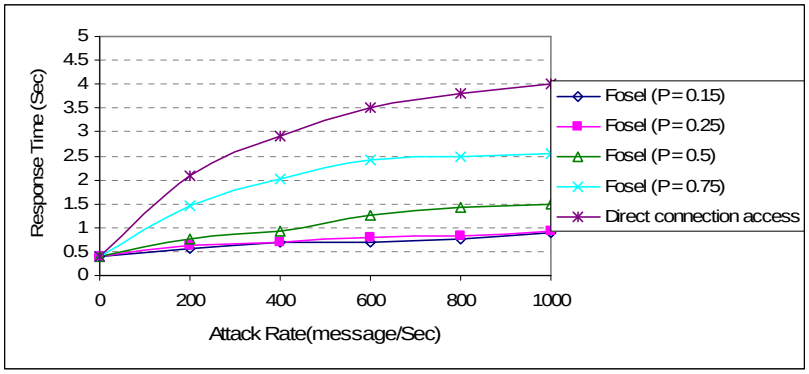


Figure 4.7: Response time vs. attack rate

Dynamic attack and recovery

Let's first, look at the attack-recovery process. Attackers attack the overlay nodes and bring down n_a nodes. When an overlay node identifies the attack, the node is removed from the overlay. When an attacker identifies that a node it is attacking no longer resides in the overlay, it stops the attack on that node and redirects its attack toward a new node that resides in the overlay. On the other hand, when an attack on a node was terminated, that node is immediately brought back into the overlay. We assume that bringing back a node to the overlay is done at the same speed as redirecting the attack to a new overlay node. As you can see, there is quite dynamic environment, where some nodes are removed from the overlay (due to attack) while some other nodes join the overlay again after termination of attack.

The dynamic nature of the overlay network (in our case Chord network) tolerates DoS attacks. However, attackers by attacking large fraction of overlay nodes can stop routing action of legitimate packets towards the green nodes. There is a possibility that an overlay node that has just received the legitimate packet to route it toward the green nodes, is attacked. Hence the node is forced to remove itself from the overlay and consequently the legitimate packet is lost and does not deliver to the green node. We call this loss rate.

Figure 4.8 shows the loss rate when we vary the number of overlay nodes that are under attack (node failure in the figure). If we suppose loss rate below 10% is acceptable, then attackers can bring down the overlay network by attacking only 20% of total overlay nodes (in our case 12 nodes out of 60 nodes). Of course, when we have large overlay network, say 2000 nodes, it is not easy for attackers to bring down 400 nodes simultaneously. Attackers need thousands of Zombie machines to attack this fraction of overlay nodes.

We can improve the overlay's ability to thwart attacks by packet replication (redundancy) by the sender (the sender application site). To increase delivery chance of legitimate packets to the green nodes, senders can send legitimate packets simultaneously via multiple-path in the overlay. In fact by simply duplicating packets (i.e., simultaneously sending the same packet through two or more different overlay nodes), we can guarantee packet delivery with high probability to the green nodes.

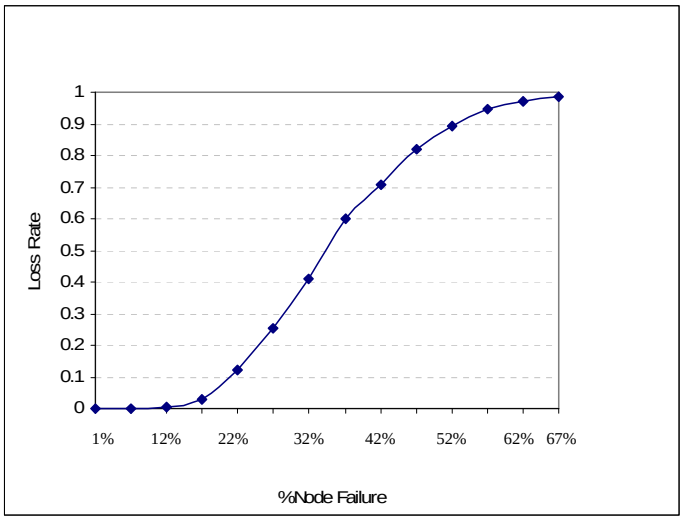


Figure 4.8: Loss rate vs. number of attacked nodes

Figure 4.9 presents loss rate when we vary both node failure (the number of overlay nodes that are under attack) and the packet replication factor. In the figure, 1-replication means that sender will replicate all packets once, effectively sending twice the amount of traffic simultaneously over different paths. For 3-replication, it means sender will send a packet simultaneously via 4 different paths. However, 0-replication means that sender only sends packets via one overlay path. Note that packet replication and routing via multiple-path toward the green nodes is different from delivery of multiple copies of legitimate packets from a green node to the target (the receiver application site).

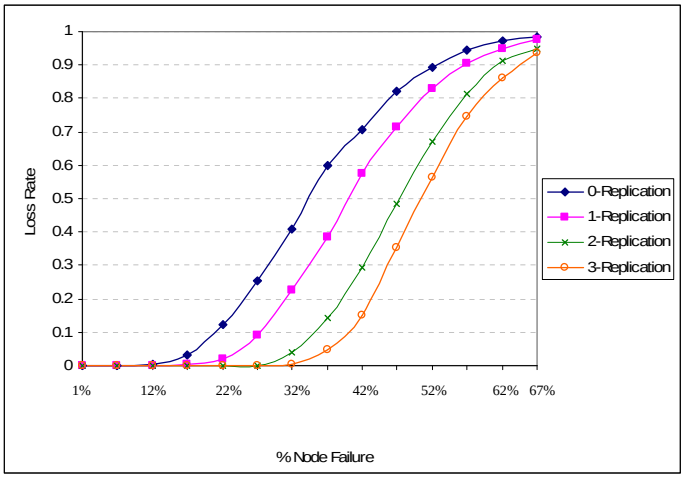


Figure 4.9: Loss rate vs. number of attacked nodes for different packet replication

Figure 4.9 shows that for 3-replication we can sustain attacks up to 40% of the overlay nodes (in our case 24 nodes out of 60 nodes). It means that when attackers are able to attack more than 40%, the loss rate is less than 10%. Extrapolating, imagine we have 2000 nodes, attackers should bring down 800 nodes simultaneously to make serious obstacles against packet delivery. For such attacks, attackers should catch hundreds thousands of Zombie

machines, which is not easy.

We use a throughput metric to show the effect of attacks on the overlay networks as well. To measure throughput, we calculate the amount of connectivity between the application site sender and the application site receiver. Figure 4.10 shows throughput results in KB (kilo bit) per second when attackers attack the overlay nodes. The experiment shows that attackers should subvert or suppress more than 40% of overlay nodes before the system becomes unusable for all application sites.

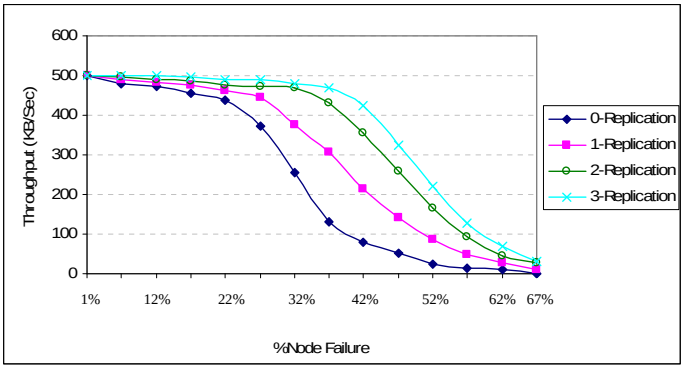


Figure 4.10: Throughput (KB/Sec) vs. number of attacked nodes for different packet replication

End-to-end Latency

One expects that using overlay networks will impose a performance penalty. Indeed, when we use overlay network between senders and receivers, an undesired latency is created. In our case, end-to-end latency increases by a factor 1.5 in the worst case. Although a large overlay network improves resilience against DoS attacks, it makes the latency worse. Figure 4.11 shows end-to-end latency when numbers of overlay nodes are varied a long x-axis. In the figure T_o indicates T(overlay) and T_d indicates T(direct connection). In our experiments an application site sender can reach the target (the receiver application site) within 200 ms directly. While when we use an overlay network with 60 nodes this latency reaches 300 ms in the worst case (Chord routing protocol).

Attacking large fraction of overlay nodes increases latency up to 4.0 times more than an attack-free system. The main reason behind this increase is that when an overlay node is attacked, the node is forced to remove itself from the overlay. Perhaps such removed node would have been the desired node for the next hop of routing process. In other words, perhaps the removed node is the candidate node in the finger table of current packet holder node. However, the node that is responsible for routing the packet now should wait for stabilization period to update its finger table.

However, this increase drops to the low levels when we use packet replication. In figure 4.12 we present our latency results when we vary both the number of overlay nodes

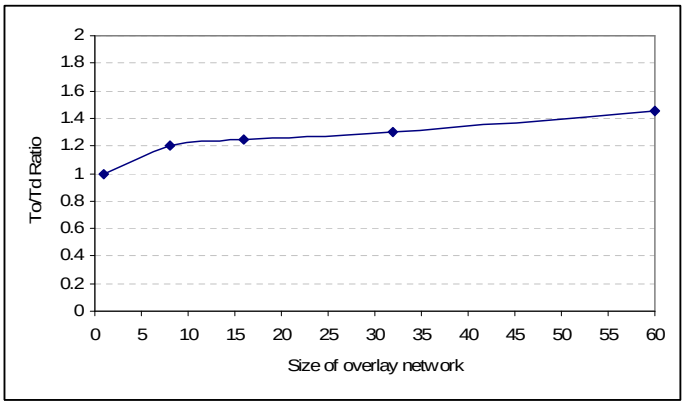


Figure 4.11: End-to-end latency ratio vs. size of overlay network

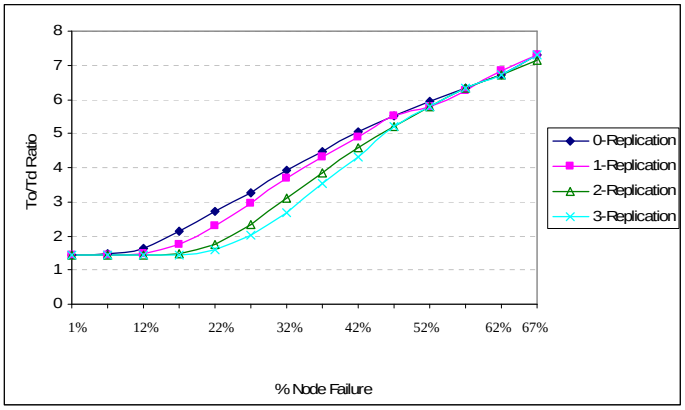


Figure 4.12: End-to-end latency ratio vs. percentage of node failure

that are under attack (node failure) and the packet replication factor. End-to-end latency ratio is about 2.5 when 40% of overlay nodes are attacked in compare to attack-free system. This ratio is about 4.0 in compare to direct connection access. Remember that according to the above explanation, the probability that more than 40% overlay nodes are attacked simultaneously is negligible.

As we increase the replication factor, we get a better average latency. However, when number of attacked nodes reaches 50% of total overlay nodes; surprisingly the latency of different amounts of packet replication is converging. This is true, certainly some nodes that are candidate for the next hop routing via path "A" are candidate for path "B" as well. Probably these nodes are candidate for path "C" too. Let us call these nodes "common nodes". However, when attackers paralyze 50% of overlay nodes, this common set of candidate overlay nodes is becoming large. Now if a significant fraction of these common nodes is removed from the overlay, the latency of different paths is converged.

Although the overlay network creates undesired latency, we prefer it because Fosel architecture provides a good resilience against DoS attacks.

Table 4.1: Correlation of simulation and empirical results

Number of message copies / Pearson Coefficient	$C = 3$	$C = 5$	$C = 10$	$C = 15$
r_{xy}	0.854876	0.935327	0.813114	0.871521

4.4 Comparison of empirical and simulation results

Fosel architecture was evaluated by simulation as well [12, 22]. Simulation results and empirical results confirm each other.

Let us to show how empirical results correlate with simulation results. In probability theory and statistics, correlation, indicates the strength and direction of a linear relationship between two random variables. A correlation coefficient indicates how much two variables are close together. Pearson correlation coefficient is one of best known coefficients that we use here.

The correlation coefficient $\rho_{X,Y}$ between two random variables X and Y with expected values μ_X and μ_Y and standard deviations σ_X and σ_Y is defined as:

$$\rho_{X,Y} = \frac{COV(X,Y)}{\sigma_X \sigma_Y} = \frac{E((X-\mu_x)(Y-\mu_Y))}{\sigma_X \sigma_Y} \tag{4.1}$$

Where E is the expected value operator and COV means covariance.

If we have a series of n measurements of X and Y written as x_i and y_i where $i = 1, 2, ..., n$, then the Pearson correlation coefficient can be used to estimate the correlation of X and Y. The Pearson correlation coefficient is written:

$$r_{xy} = \frac{n\Sigma x_i y_i - \Sigma x_i \Sigma y_i}{\sqrt{n\Sigma x_i^2 - (\Sigma x_i)^2}\sqrt{n\Sigma y_i^2 - (\Sigma y_i)^2}} \tag{4.2}$$

Table 4.1 shows Pearson correlation coefficient for empirical and simulation results. Elements of table are correlation coefficients where simulation results are indicated by X set and experimental results are indicated by Y set (r_xy = correlation coefficient between simulation value and experimental value).

Table 4.1 shows correlation of simulation and empirical results (experiment: failure rate is versus attack rate for different numbers of message copies). In fact table 4.1 shows that there is strong covariance between experimental and simulation results. Table 4.1 indicates that in all cases, Pearson correlation coefficient is more than 80% which means that empirical results confirm simulation results and vice versa.

5 Evaluation of the CIS

This chapter describes a set of experiments we have conducted to evaluate the Crutial Information Switch (CIS). As explained in CRUTIAL deliverable D18, the CIS provides two basic services: the *Protection Service (PS)* and the *Communication Service (CS)*. CIS-PS ensures that the incoming and outgoing traffic in/out of a LAN satisfies the security policy of the infrastructure. CIS-CS supports secure communication between CISs and, ultimately, between LANs.

5.1 CIS Protection Service

This section presents the evaluation results of the CIS Protection service (CIS-PS). As explained in CRUTIAL deliverable D18, we defined two versions of the CIS-PS with different properties and requirements. To make the present deliverable self-contained, we start this section with a brief description of the CIS-PS and of its two versions.

5.1.1 CIS-PS Versions

To understand the *rationale of the design* of the CIS-PS, consider the problem of implementing a replicated firewall between a non-trusted WAN (or LAN) and the trusted LAN that we want to protect. Further assume that we wish to ensure that only the correct messages (according to the deployed policy) go from the non-trusted side, through the CIS-PS, to the computers and/or devices in the protected LAN. A first problem is that the traffic has to be received by all n replicas, instead of only 1 (as in a normal firewall), so that they can perform their active replication mechanisms. A second problem is that up to f replicas can be faulty and behave maliciously, both towards other replicas and towards the station computers.

Our solution to the first problem is to use a device (e.g., an Ethernet hub) to broadcast the traffic to all replicas. These verify whether the messages comply with the security policy and do a vote, approving the messages if and only if at least $f + 1$ different replicas vote in favor. A message approved by the CIS-PS is then forwarded to its destination by a distinguished replica, the *leader*, so there is no unnecessary traffic multiplication inside the LAN. The attentive reader will identify three problems, addressed below: dealing with omissions in the broadcast; ensuring that there is always one and only one leader; and that it is correct.

The existence of faulty replicas is usually addressed with masking protocols of the Byzantine type, which extract the correct result from the n replicas, despite f maliciously faulty: only messages approved by $f + 1$ replicas should go through (one of which must be correct since at most f can be faulty). Since the result must be sent to the computers in

the protected LAN, either it is consolidated at the source, or at the destination.

The simplest and most usual approach is to implement a front-end in the destination host that accepts a message if: (1) $f + 1$ different replicas send it; or (2) the message has a certificate showing that $f+1$ replicas approve it; or (3) the message has a signature generated by $f + 1$ replicas using a threshold cryptography scheme. This would imply modifying the hosts's software. However, modifying the software of the SCADA/PCS system can be complicated, and the traffic inside the protected LAN would be multiplied by n in certain cases (every replica would send the message to the LAN), so this solution is undesirable.

So we should turn ourselves to consolidating at the source, and sending *only one, but correct, forwarded message*. However, what is innovative here is that source-consolidation mechanisms should be transparent to the (standard) computers/devices in the protected LAN. Moreover, a faulty replica (leader or not) has access to the protected LAN so it can send incorrect traffic to the protected computers, which typically can not distinguish faulty from correct replicas. This makes consolidation at the source a hard problem.

The solution to the second problem lies on using IPSec, a set of standard protocols that are expected to be generalized in SCADA/PCS systems, according to best practice recommendations from expert organizations and governments. Henceforth, we assume that the IPSec Authentication Header (AH) protocol [24] runs both in the protected computers and in the CIS-PS replicas. The basic idea is that the protected computers will only accept messages with a valid IPSec/AH *Message Authentication Code* (MAC), which can only be produced if the message is approved by $f + 1$ different replicas. However, IPSec/AH MACs are generated using a shared key[1] K and a hash function, so it is not possible to use threshold cryptography[2]. As the attentive reader will note, the shared key storage becomes a vulnerability point that can not be overlooked in a high resilience design, therefore, *there must be some secure component that stores the shared key and produces MACs for messages approved by $f + 1$ replicas.*

The requirement in the previous paragraph, for a secure component in an otherwise Byzantine-on-failure environment, call for a hybrid architecture that includes *a secure wormhole* [49]. This wormhole can be deployed as a set of local trustworthy components (one per replica).

5.1.1.1 CIS-PS-IT: intrusion-tolerant CIS-PS

CIS-PS-IT requires $2f + 1$ machines in order to tolerate f intrusions. Thus, the example configuration presented in Figure 5.1 is able to tolerate one intrusion. Notice that

[1] We assume that IPSec/AH is used with manual key management.

[2] Threshold cryptography is a cryptographic scheme that allows to split a key among a set of parties and to define the minimum number of parties that are required to make useful things with the key (see [41]). Unfortunately, it can only be used in combination with public key cryptography schemes where different keys are used to encrypt and to decrypt, whereas IPSec/AH uses the same key to produce and verify MACs.

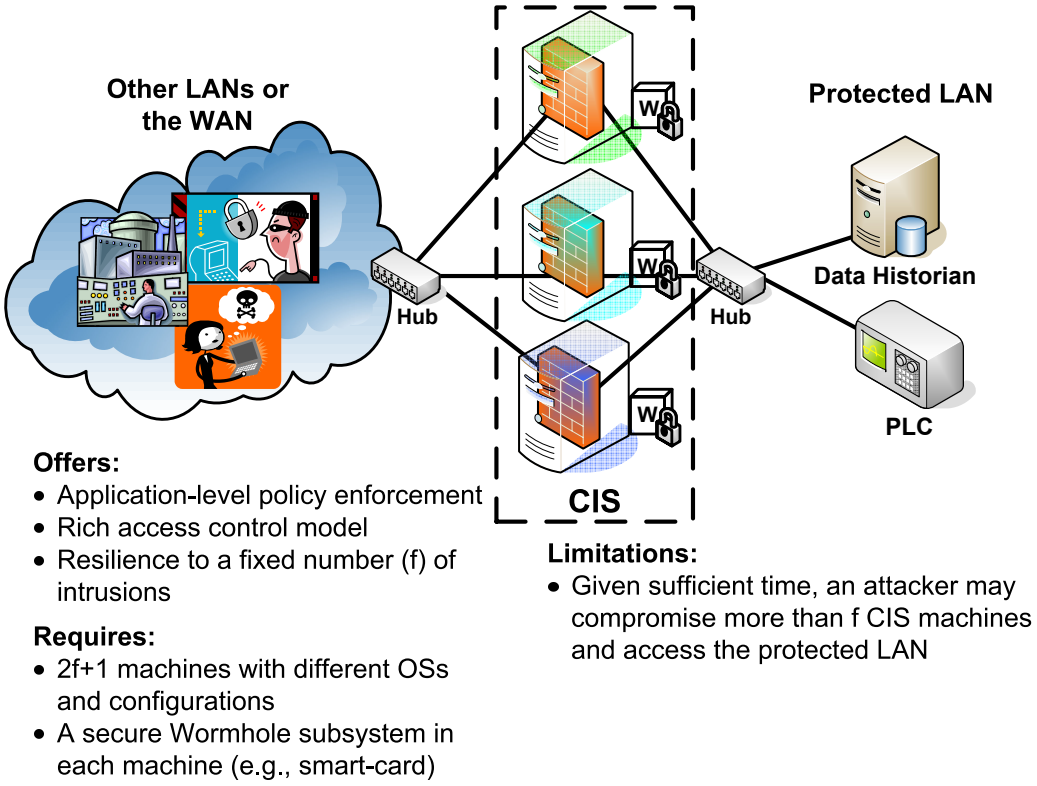


Figure 5.1: Intrusion-tolerant CIS-PS version (CIS-PS-IT).

such a design only makes sense if the different machines cannot be attacked in the same way, i.e., they must not share the same set of vulnerabilities. In order to achieve this goal, each CIS-PS-IT replica is deployed in a different operating system (e.g., Linux, FreeBSD, Solaris), and the operating systems are configured to use different passwords and different services. In addition, each CIS-PS-IT replica is enhanced with a secure wormhole subsystem that stores a secret key and this key is used to produce a MAC for messages that are approved by at least $f + 1$ replicas. Given that the wormhole is secure, no malicious replica can force it to sign an unapproved message.

5.1.1.2 CIS-PS-SH: intrusion-tolerant and self-healing CIS-PS

The most resilient CIS-PS design combines intrusion tolerance with self-healing mechanisms. Self-healing is provided by a proactive-reactive recovery service that combines time-triggered periodic rejuvenations with event-triggered rejuvenations when something bad is detected or suspected.

Proactive recoveries are triggered periodically in every replica even if it is not compromised. The goal is to remove the effects of malicious attacks/faults even if the attacker remains dormant. Otherwise, if an attacker compromises a replica and makes an action that can be detected (e.g., sending a message not signed with the shared key K), the compro-

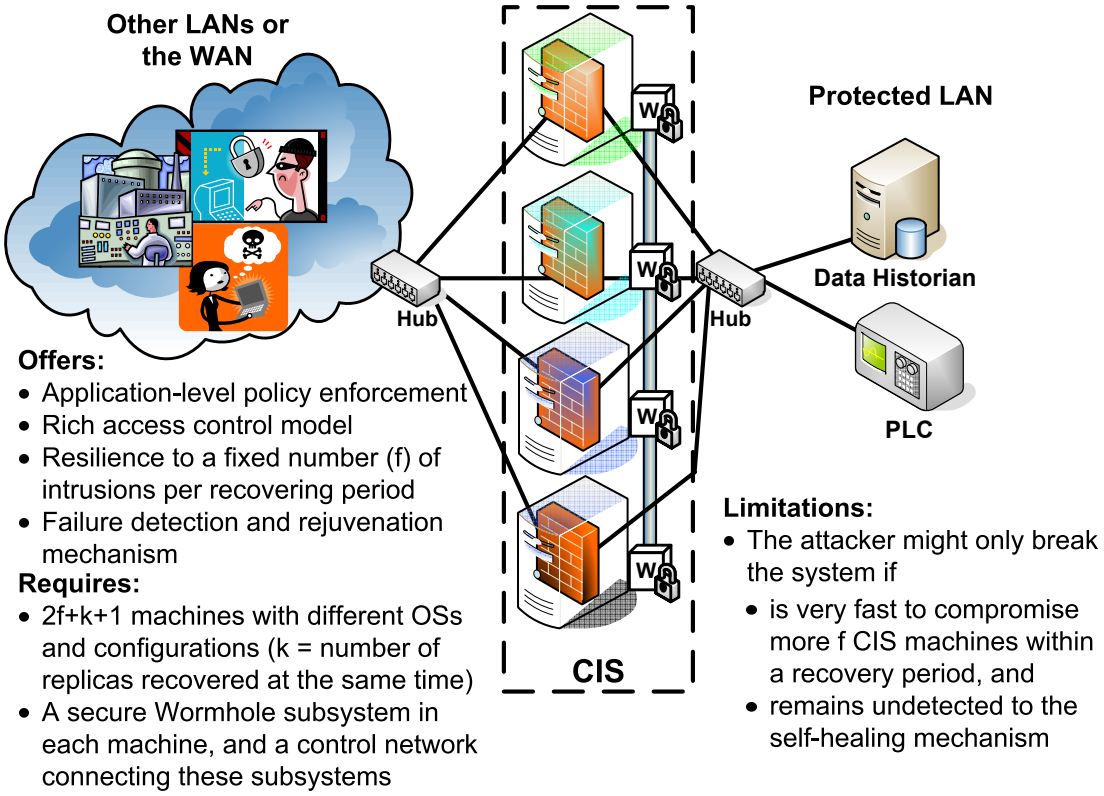


Figure 5.2: Intrusion-tolerant and Self-healing CIS-PS version (CIS-PS-SH).

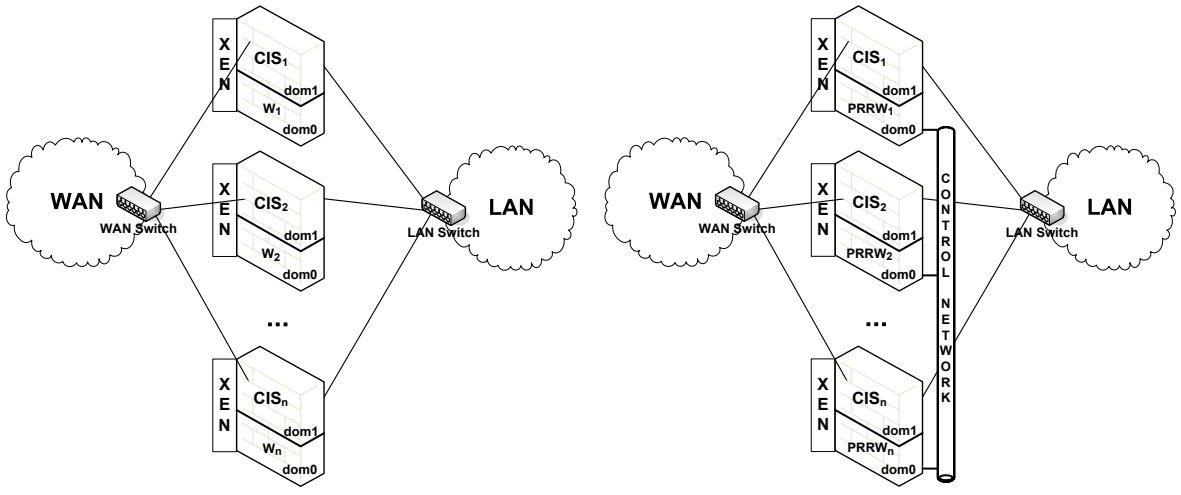
mised replica is rejuvenated through a reactive recovery. Moreover, the rejuvenation process of a (proactive or reactive) recovery does not simply restore replicas to known states, since this would allow an attacker to exploit the same vulnerabilities as before. To address this issue, the rejuvenation process itself introduces some degree of diversity to restored replicas (change operating system, use memory obfuscation techniques, change passwords, etc.), so that attackers will have to find other vulnerabilities in order to compromise a replica. Figure 5.2 depicts the design of an intrusion-tolerant CIS with self-healing mechanisms.

This CIS-PS design requires $2f + k + 1$ machines in order to tolerate f intrusions per recovering period (dictated by the periodicity of the proactive recoveries). The new parameter k represents the number of replicas that recover at the same time and its value is typically one. If this parameter was not included in the calculation of the total number of required machines, the CIS-PS-SH could become unavailable during recoveries. Thus, the configuration presented in Figure 5.2 composed of 4 replicas is able to tolerate one intrusion per recovering period.

5.1.2 CIS-PS Prototypes

We implemented a prototype of each CIS-PS version (CIS-PS-IT and CIS-PS-SH) described in the previous section. Both prototypes use the XEN virtual machine monitor [11] with the Linux operating system to deploy the hybrid architecture (different system parts, payload and wormhole, with distinct characteristics and fault assumptions) that is required by the two CIS-PS versions. A XEN system has multiple layers, the lowest and most privileged of which is XEN itself. XEN may host multiple guest operating systems, every one executed within an isolated virtual machine (VM) or, in XEN terminology, a *domain*. Domains are scheduled by XEN to make effective use of the available physical resources (e.g., CPUs). Each guest OS manages its own applications. The first domain, *dom0*, is created automatically when the system boots and has special management privileges. Domain *dom0* builds other domains (*dom1, dom2, dom3, ...*) and manages their virtual devices. It also performs administrative tasks such as suspending and resuming other VMs, and it can be configured to execute with higher priority than the remaining VMs.

The architecture of the prototypes is depicted in Figure 5.3. Every replica uses XEN to isolate the payload from the wormhole part. Local wormholes run in replicas' domain *dom0*, and the CIS-PS executes in replicas' domain *dom1*. In the case of CIS-PS-IT (Figure 5.3(a)), the wormhole part does not require a control networks given that it only implements local operations as explained in Section 5.1.1. On the other hand, CIS-PS-SH (Figure 5.3(b)) requires an isolated control network to connect the local PRRWs in each replica. In this case, domain *dom0* is configured to execute with higher priority than domain *dom1* in every replica, in order to emulate the real-time behavior required by the PRRW service.



(a) CIS-PS-IT version.

(b) CIS-PS-SH version.

Figure 5.3: CIS-PS prototypes architecture.

The recovery actions executed by the (CIS-PS-SH) PRRW make use of XEN system calls *xm destroy* and *xm create* to, respectively, shutdown and boot a CIS replica. Note

that *xm destroy* corresponds to a virtual power off and it is almost instantaneous, whereas a normal shutdown could take several seconds. We avoid the delay of a normal shutdown because we assume that the running OS may have been compromised and thus it cannot be reutilized.

In order to provide a virtual global clock to each (CIS-PS-SH) local PRRW, we implemented a clock synchronization protocol. There are many clock synchronization algorithms available in the literature suitable to synchronous environments with crash faults [48]. The protocol we implemented combines techniques proposed on some of these works and it is now briefly explained. Local PRRWs are initially launched by any order as long as local PRRW 1 is the last one to begin execution. When local PRRW 1 starts executing, it broadcasts a small synchronization message to all PRRWs (including itself) and this message is used by every local PRRW to define the global time instant 0. Then, in order to maintain the virtual global clocks with a bounded precision, each local PRRW broadcasts a synchronization message exactly when a proactive recovery is triggered. Given that all local PRRWs know when proactive recoveries should be triggered, they can adjust their clocks accordingly. Both mechanisms assume that the broadcast reception instant is practically the same everywhere in the control network, which is substantiated by the fact that the control network is provided by a switch used only by the local PRRWs.

To simplify the design and to avoid changes in the kernel, the CIS-PS prototypes operate at the UDP level, instead of IP level as most firewalls do. Therefore, there was no need to implement packet interception because packets are sent directly to the CIS-PS. Moreover, authentication is not done at the IP level (as when using IPSec/AH), but in alternative the HMAC[3] of the payload UDP packet is calculated, and then the two are concatenated. Notice that this type of authentication implies the same type of overhead of IP authentication. Given that the CIS-PS prototypes operate at the UDP level, we employ IP multicast to enforce broadcast in WAN and LAN communication. Therefore, we do not need physical traffic replication devices in our prototypes[4]. Moreover, the LAN switch uses access control lists to prevent replicas from spoofing their MAC addresses, and each replica stores a table with the MAC address of each other replica.

Regarding the CIS-PS-SH prototype, periodic and reactive recoveries reset the state and restore the code of a replica. While this is useful to restore the correctness of the replica, it would be interesting if we were able to introduce diversity in the recovery process. For instance, each recovery could randomize the address space of the replica (e.g., using PaX[5]) in order to minimize the usefulness of the knowledge obtained in the past to increase the chances of future attacks. The Xen and PaX communities are currently making efforts to release a joint distribution, and we plan to integrate this mechanism in the prototype when it is available. Nevertheless, the current version of the prototype already incorporates some diversity mechanisms: we maintain a pool of OS images with different configurations (e.g., different root passwords, different kernel versions) and each recovery (proactive or reactive)

[3]HMAC is a standard for calculating MACs, and in the prototype we used the SHA-1 hash function [32].

[4]Nevertheless, if the CIS-PS would operate at the IP level, we could configure the WAN and LAN switches to use the failopen mode in order to force broadcast.

[5]Available at http://pax.grsecurity.net/.

randomly chooses and boots one of these images. Moreover, domain *dom*0 executes with higher priority than domain *dom*1, but since it is based on a normal Linux OS, it provides no strict real-time guarantees. Currently, only modified versions of Linux, NetBSD, OpenBSD and Solaris can be run on *dom*0. However, XEN is continuously being improved and we expect that in the near future a real-time OS can be used.

Physical vs VM-based replication. In order to evaluate the tradeoff of using physical or virtual replicas, each of the two prototypes described above was evaluated using two different setups: physical replication and VM-based replication. With physical replication, each replica (composed of the payload and wormhole part) is deployed in its own physical host. With VM-based replication, all replicas are deployed in the same physical host, using XEN virtual machines to isolate the different runtime environments, preventing intrusions from propagating from one replica to the others. Physical replication is suitable to CIS-PS deployments where the LAN to be protected is highly critical, given that it offers tolerance to both hardware and software faults. VM-based replication is vulnerable to hardware faults, but it offers a more cost effective solution taking into account that price is always a major concern of power grid operators and that a typical critical infrastructure has hundreds of station computers and other hosts to be protected[6].

5.1.3 CIS-PS Experimental Evaluation

On the of the worst possible types of attacks that the CIS-PS can suffer is a denial-of-service (DoS) attack coming from the WAN. Nowadays, such an attack is quite easy to trigger and it may slow down CIS-PS operation and affect the timeliness of SCADA/PCS supervision and/or control mechanisms that are normally deployed in critical infrastructures. Therefore, and following the reasoning presented in [13], the first and largest set of experiments that we have conducted had the goal of evaluating the CIS-PS performance (latency, throughput, and loss rate) when in the presence of external DoS attacks. We also conducted additional experiments with CIS-PS-SH to assess the impact and overhead of recoveries, and to observe how attacks from compromised replicas are solved in few seconds. Before presenting the results of the experiments, we start by describing the experimental setup.

5.1.3.1 Setup

A total of ten server PCs were used in the experiments. CIS-PS replicas were deployed in a number of servers ranging from one to six, depending on the version (CIS-PS-IT or CIS-PS-SH) and replication type (physical or VM-based). These servers were connected to the

[6]For example, as stated in the CRUTIAL deliverable D2, the Italian Power System contains about 1000 station computers. This means that using three machines (CIS-PS-IT version) and two hubs to protect every one of these computers would require 3000 and 2000 extra machines and hubs, respectively.

networks represented in the prototype architecture: LAN, WAN, and the control network in the case of CIS-PS-SH (see Figure 5.3). These networks were defined as separated VLANs configured on two Dell Gigabit switches. The LAN and control networks shared the same switch, whereas the WAN network was deployed in a different switch. The LAN and WAN were configured as 100 Mbps networks while the control network operated at 1 Gbps.

In addition to the server PCs where the CIS-PS replicas were deployed, we used three more PCs in the experiments. One PC was connected to the LAN emulating a station computer and, in the WAN side, two PCs were deployed: a *good* sender trying to transmit legal traffic to the station computer, and a *malicious* sender sending illegal messages to the LAN (equivalent to a DoS attack). Figure 5.4 depicts this setup. CIS-PS is represented by a centralized component to simplify presentation.

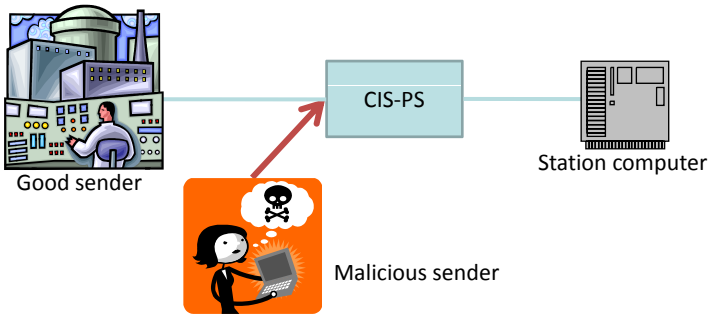


Figure 5.4: CIS-PS experimental setup.

Nine of the ten server PCs of our setup were a 2.8 GHz Pentium 4 PC with 2 GB RAM running Fedora Core 6 with Linux 2.6.18, and Xen 3.0.3 to manage the virtual machines. The remaining server was a 2.3 GHz 64-bit quad-core Xeon processor with 8 GB of RAM, running Fedora Core 8 with Linux 2.6.21 and Xen 3.1.0. This last server was used to deploy the prototypes when using VM-based replication given that this type of replication is more demanding in terms of processor speed and memory size.

Independently of the prototype version and replication type, the payload part of each replica was deployed on a virtual machine with 1.5 GB of RAM, and the wormhole part used 512 MB of RAM.

In all experiments, the CIS-PS security policy is very simple, since it just checks if certain strings are present in the received packets.

5.1.3.2 Performance under external DoS attacks

Methodology. The goal of external DoS experiments was to to evaluate how much legal traffic each version of the CIS-PS using the different replication types can deliver while it is being attacked by an outsider. During these experiments, the malicious sender is constantly injecting illegal traffic at a certain rate. The AJECT tool described in Chapter 2 was

employed to perform this task. The good sender transmits legal packets (of 1400 bytes) at a different rate depending on the experiment type. In the throughput and loss rate experiments, legal packets are sent at a rate of 500 packets/second and the obtained results correspond to the worst case values (lowest throughput and highest loss rate) that were observed during the transmission of 10,000 legal packets. Notice that this is a very high traffic for a SCADA system, since it represents, for instance, 500 MMS commands being sent to a device per second. In the latency experiment, a legal packet is transmitted after the arrival of an acknowledgement (ACK) of the previous packet. The obtained results correspond to the average round-trip of 10,000 legal packets (ACKs are not verified by the CIS-PS), after excluding the 5% most distant from the average. The standard deviation value was at most 10% of the average. In all experiments presented in this section, recoveries (when supported) are disabled.

A substantial number of experiments was performed to achieve the results that will be presented in this section. Each experiment is identified by the following tuple:

$\langle type, version, replication, DoS\,rate, f \rangle$, where

- *type* specifies latency or throughput/loss rate. Throughput and loss rate we calculated at the same time.
- *version* specifies the prototype version: CIS-PS-IT or CIS-PS-SH.
- *replication* specifies the replication type: physical or VM-based.
- *DoS rate* specifies the rate at which the malicious sender injects illegal traffic using AJECT. The following rates (in megabits per second (Mbps)) were tested: $\{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$.
- f specifies the number of replicas that are allowed to be faulty/compromised. This value has an impact on the total number of replicas needed to deploy the CIS-PS. We tested two values: $f = 1$ and $f = 2$.

We executed every possible combination of the above parameters, which resulted in a total of 1660 different experiments. Moreover, each experiment was run five times, resulting in an overall total of 8300 experiments.

Latency results. Figures 5.5 and 5.6 present the average latency of the two versions of the CIS-PS in the presence of DoS attacks launched at different rates. The graphs show that two versions have similar latency values for DoS attack rates up to 70 Mbps, and that the physically replicated prototypes start to degrade performance when DoS attacks rates are close to 100 Mbps. However, CIS-PS-IT with physical replication does not degrade so much as CIS-PS-SH. The worst case average latency of CIS-PS-IT is less than 14 milliseconds, while CIS-PS-SH obtained latency values in the order of seconds (not represented in graph due to this reason).

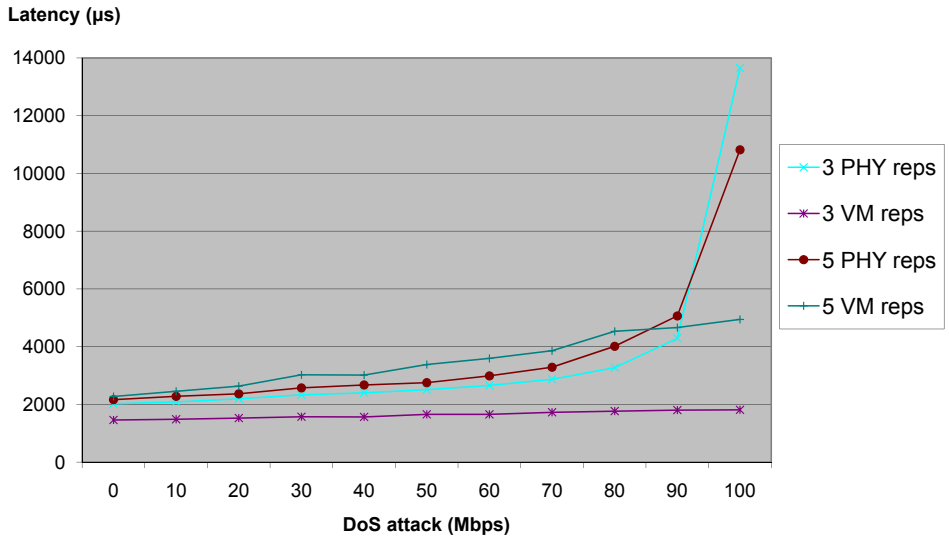


Figure 5.5: CIS-PS-IT average latency with 3 ($f = 1$) and 5 ($f = 2$) replicas, and when using physical and VM-based replication.

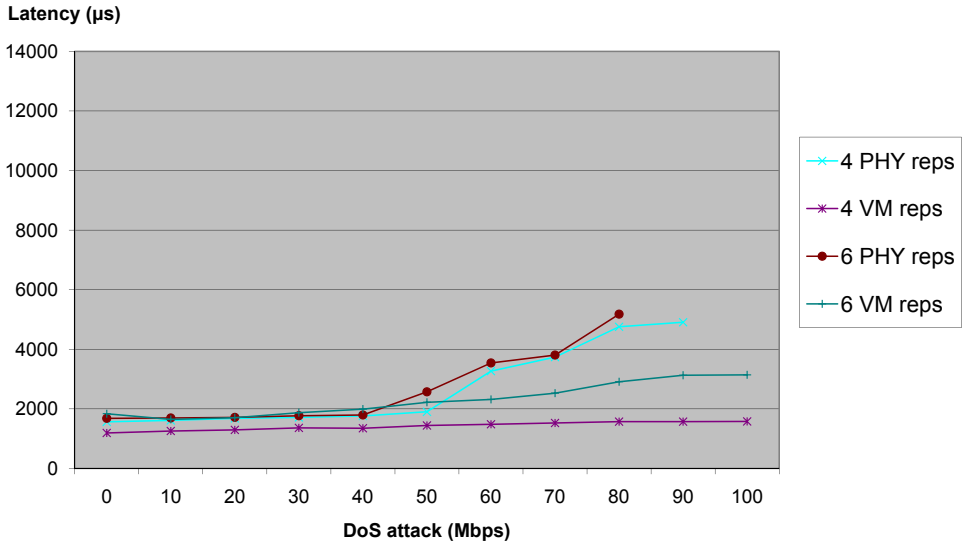


Figure 5.6: CIS-PS-SH average latency with 4 ($f = 1$) and 6 ($f = 2$) replicas, and when using physical and VM-based replication.

These results suggest that, independently of the replication type, both versions of the CIS-PS offer modest latency (less than 4 milliseconds) even with a reasonably loaded network (up to 70 Mbps of illegal/mallicious traffic). To cope with significant DoS attacks ($\geq$ 70 Mbps) coming from the WAN, complimentary mechanisms may be employed, given that CIS-PS processing latency has a huge increase, namely when using physical replication.

Note that, in the context of power systems, the scenarios presented in CRUTIAL deliverable D2 do not require latency values lower than 200 milliseconds. Both versions of

the CIS-PS satisfy this requirement offering latencies one order of magnitude smaller, under any DoS attack rate. The unique exception is the physically replicated CIS-PS-SH in the presence of DoS attack rates greater than 80 Mbps.

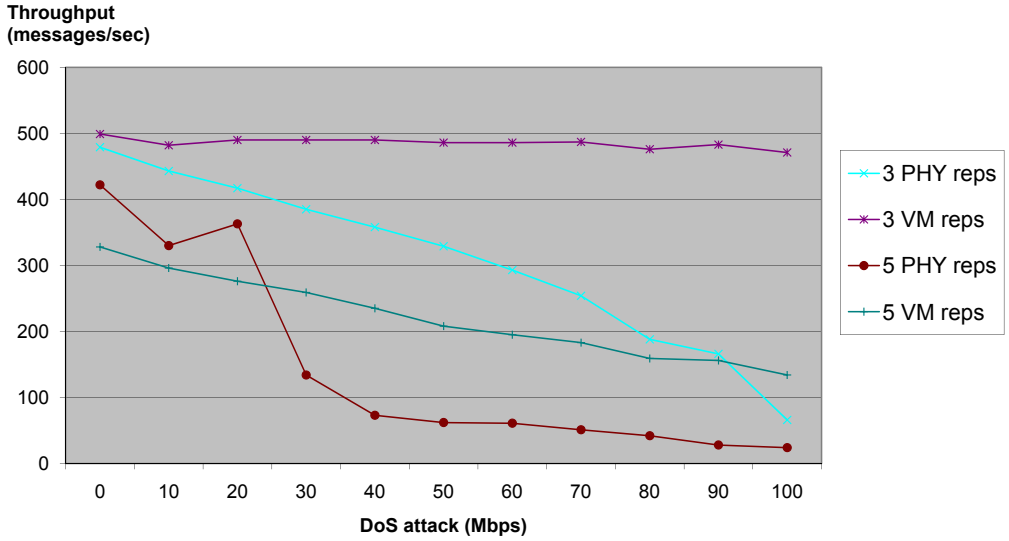


Figure 5.7: CIS-PS-IT maximum throughput with 3 ($f = 1$) and 5 ($f = 2$) replicas, and when using physical and VM-based replication.

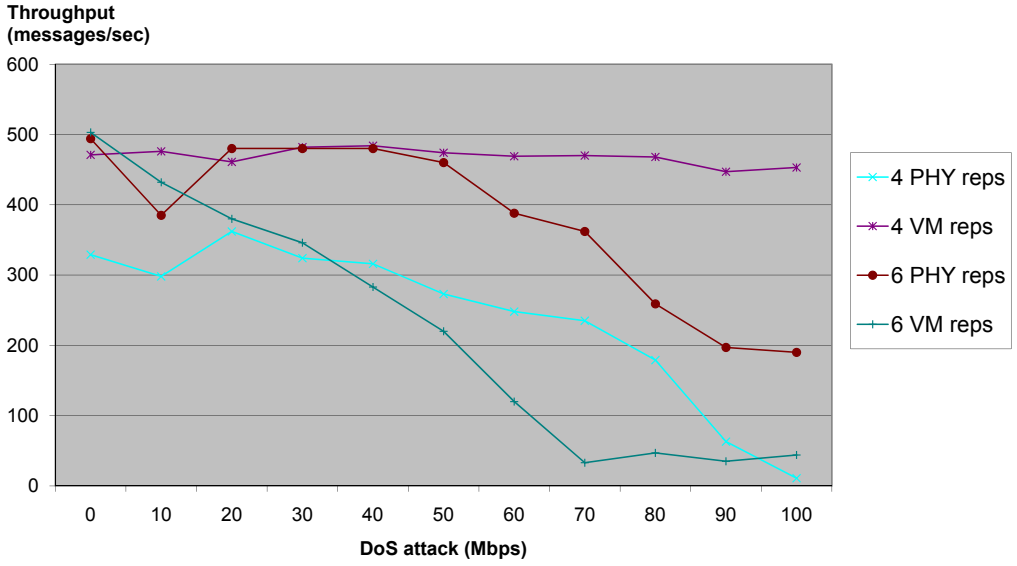


Figure 5.8: CIS-PS-SH maximum throughput with 4 ($f = 1$) and 6 ($f = 2$) replicas, and when using physical and VM-based replication.

Throughput and loss rate results. Figures 5.7, 5.8, 5.9, and 5.10 present the maximum throughput and maximum loss rate of the two versions of the CIS-PS in the presence of

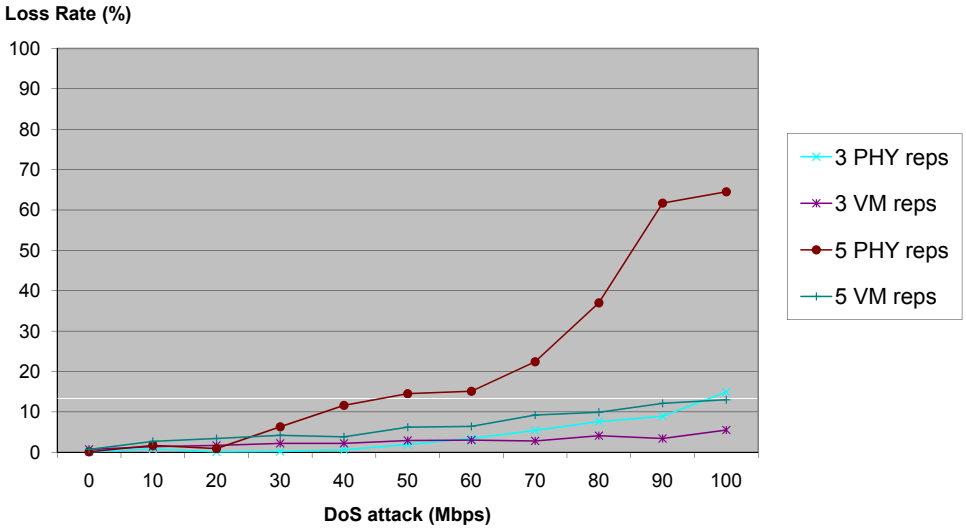


Figure 5.9: CIS-PS-IT maximum loss rate with 3 ($f = 1$) and 5 ($f = 2$) replicas, and when using physical and VM-based replication.

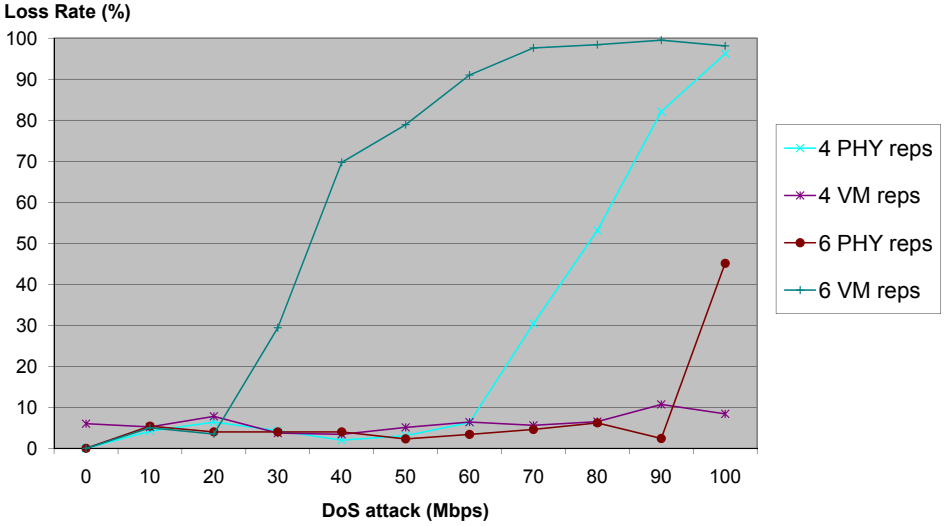


Figure 5.10: CIS-PS-SH maximum loss rate with 4 ($f = 1$) and 6 ($f = 2$) replicas, and when using physical and VM-based replication.

DoS attacks launched at different rates. The graphs show many interesting features of the various prototypes.

The first conclusion is that VM-based replication offers better throughput and loss rate than physical replication. The exception is CIS-PS-SH with 6 replicas ($f = 2$) that offers lower throughput (Figure 5.8) and higher loss rate (Figure 5.10) when using VM-based replication. The superiority of VM-based replication in most scenarios is quite surprisingly and is due mainly to implementation and deployment reasons. Our prototype was implemented in Java and we observed that it runs considerably faster in the 64-bit processor

of the physical machine used to deploy VM-based prototypes. This performance gain is attenuated when the number of replicas increases (because more virtual machines need to be run on the same physical machine) and it drops considerably with 6 VM replicas, i.e., when using CIS-PS-SH with $f = 2$. Note that this is a direct consequence of the specific equipment that we used in our experiments. We think that CIS-PS-SH would run fine in a physical machine with more resources, namely higher CPU power and increased memory size.

Secondly, the two versions of the CIS-PS present a different behaviour in the transition from $f = 1$ to $f = 2$. CIS-PS-IT with 3 replicas ($f = 1$) offers higher throughput (Figure 5.7) and lower loss rate (Figure 5.9) than CIS-PS-IT with 5 replicas ($f = 2$), independently of the replication type. On the other hand, CIS-PS-SH with 4 physical replicas ($f = 1$) offers lower throughput (Figure 5.8) and higher loss rate (Figure 5.10) than CIS-PS-SH with 6 physical replicas ($f = 2$). In this case the replication type matters because, as explained before, CIS-PS-SH with 6 VM replicas has poor performance. The different behaviour of CIS-PS-IT and CIS-PS-SH can be explained by their internal design. CIS-PS-IT does not have a control channel connecting the local wormholes (see Figure 5.1), and consequently, message voting is done using the same network that is being (DoS) attacked by the malicious sender. When the number of replicas increases, a higher number of protocol messages need to be sent to approve each incoming message, which increases the overhead of message approval. CIS-PS-SH avoids this issue because message voting is done through the isolated control channel connecting the local PRRWs (see Figure 5.2). In this case, a higher number of replicas is better because it increases the chances of each incoming message being approved more quickly. A minimum of $f + 1$ votes (from different replicas) are required to approve a message. This means that with $f = 1$ CIS-PS-SH needs 2 out of 4 replicas to vote on each message, and that CIS-PS-SH with $f = 2$ needs 3 out of 6 replicas. In the former case, there are 6 possible combinations of replicas that may approve a message, while in the latter case there 60 possible combinations.

Overall, if we exclude the results of CIS-PS-IT with 5 physical replicas and of CIS-PS-SH with 6 VM replicas, the two CIS-PS versions offer reasonably good throughput (> 200 messages/sec) and loss rate (< 10 %) in the presence of DoS attacks up to 60 Mbps. As mentioned before, complimentary mechanisms may be employed to cope with significant DoS attacks (≥ 70 Mbps) coming from the WAN.

5.1.3.3 Overhead and Benefits of Self-Healing

Methodology. The goal of the experiments presented in this section was to assess the impact of recoveries in the behaviour of a specific configuration of the CIS-PS-SH in scenarios with and without attacks. The configuration chosen was the one using 4 physical replicas given that it combines resilience to hardware faults with a reasonable cost.

The first set of experiments evaluated the time a recovery takes; the second set of experiments evaluated the impact of proactive recoveries and crash faults in CIS-PS-SH

throughput; and the last set of experiments measured the resilience of the CIS-PS-SH to attacks launched by compromised replicas. In the second and third experiments, there is a good sender transmitting legal messages at a rate of 5 Mbps. No external malicious sender was used in any of the experiments presented in this section. Instead, in the third experiment, there is an internal malicious sender (in a compromised replica) flooding the protected network with packets of 1470 bytes sent at a rate of 90 Mbps. The AJECT tool described in Chapter 2 was again employed to perform this task.

Performance of recoveries. We measured the time needed for each recovery task in a total of 300 recovery procedures executed during CIS-PS-SH operation. Table 5.1 shows the average, standard deviation, and maximum time for each recovery task: replica shutdown, replica rejuvenation by restoring its disk with a clean image randomly selected from a set of predefined images with different configurations, and the reboot of this new image. All disk images used in this experiment had sizes of approximately 1.7 GB.

	Shutdown	**Rejuvenation**	**Reboot**	**Total**
Average	0.6	72.2	70.1	144.6
Std. Deviation	0.5	1.2	0.3	0.9
Maximum	1.0	74.0	71.0	146.0

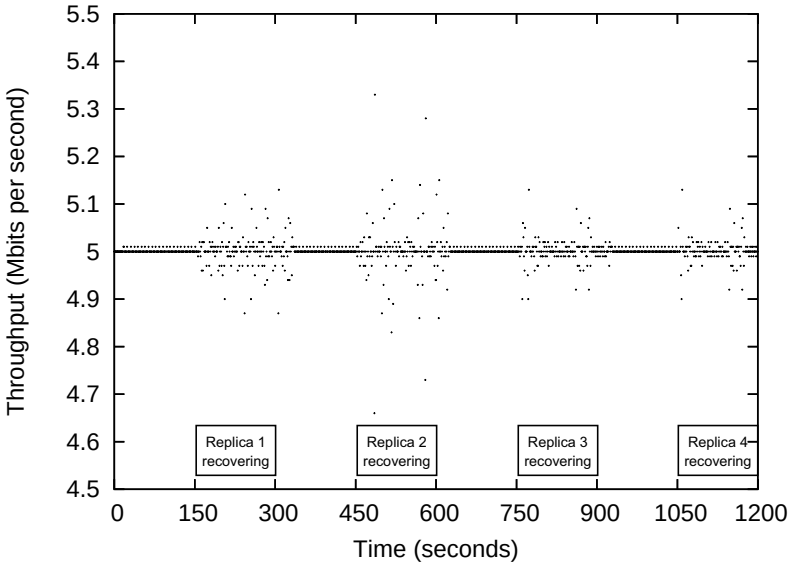
Table 5.1: Time needed (in seconds) for the several steps of a replica recovery (1.7 GB OS images).

From Table 5.1 one can see that a maximum of 146 seconds ($\sim$ 2.5 minutes) are needed in order to completely recover a virtual machine in our environment, being most of this time spent on two tasks: (1.) copying a clean pre-configured disk image from a local repository; and (2.) starting this new image (including starting the CIS protection service). These tasks could have their time lowered if we were able to build smaller images, which was not possible with the current Linux distribution we are using (Fedora Core 6).

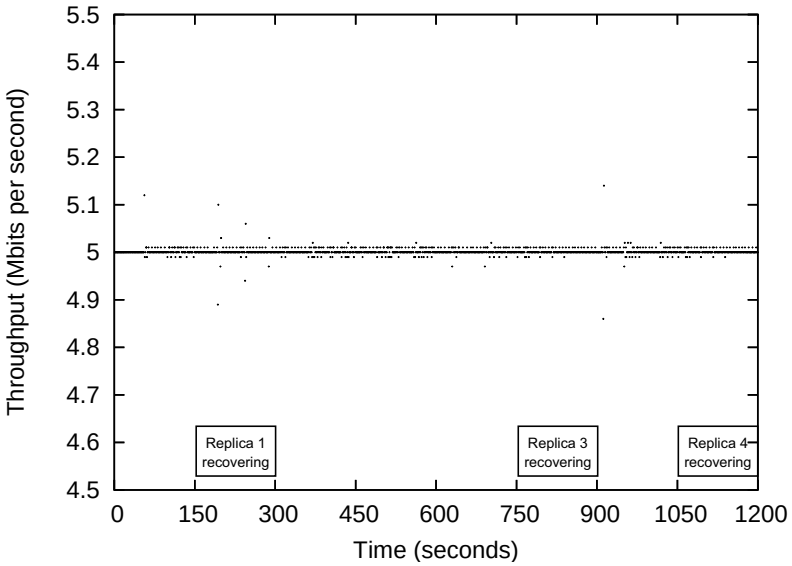
The results from the first experiment allowed to define the maximum recovery time (T_D) as 150 seconds for the remaining experiments described below. Considering that we had $n = 4$ replicas to tolerate $f = 1$ faults and $k = 1$ simultaneous recoveries, we used the expressions defined in Section 4.1.3 of CRUTIAL deliverable D18 to calculate the maximum time between two recoveries of an individual replica as $T_P = 1200$ seconds (20 minutes). By applying these values to the Proactive Resilience model described in the same deliverable, we conclude that a malicious adversary has at most $T_P + T_D = 22.5$ minutes to compromise more than f replicas and to harm the safety (i.e., make the CIS-PS-SH approve an illegal message) of our experimental prototype.

Throughput during recoveries and in the presence of crash faults. The second set of experiments evaluated the impact of proactive recovery and crash faults in the overall system throughput. Figure 5.11 presents two time diagrams that show the throughput of the CIS during a complete recovery period (20 minutes) without faults and with one crashed

(silent) replica. In the latter experiment, replica 2 is the one crashed, and it stays crashed until the end of the recovery period.



(a) No faults.



(b) One faulty replica (replica 2).

Figure 5.11: Throughput of the CIS-PS-SH during a complete recovery period (20 minutes) with $n = 4$ ($f = 1$ and $k = 1$), with and without crash faults.

The time diagrams of Figure 5.11 lead to the following conclusions. First, it can be seen that, without faults, recoveries do not have a substantial impact on the perceived throughput of the system (Figure 5.11(a)). The minimum observed throughput during recoveries was 4.6 Mbits/second, which represents a 8% drop in comparison with the expected throughput of 5 Mbits/second. This can be explained by the fact that proactive recoveries are executed with higher priority than voting procedures and thus may delay their execution

during the recovering periods. Second, the occurrence of crash faults also does not affect substantially the throughput of the system, even during periodic recoveries (Figure 5.11(b)). This happens because we use $k = 1$ extra replicas to ensure that the system is always available: even with one fault and one recovery, there are still two correct replicas ($f + 1$) to vote and approve messages. Note that without these k extra replicas, the system would become completely unavailable in the recovering periods of Figure 5.11(b). Third, by comparing the observed throughput with and without crash faults, one can conclude that the impact of proactive recoveries is smaller when there is a crashed replica during the entire execution. This happens because 3 replicas generate less vote messages in the wormhole control channel than 4 replicas. This reduction is sufficient to minimize the impact of the higher priority proactive recoveries on the vote processing time.

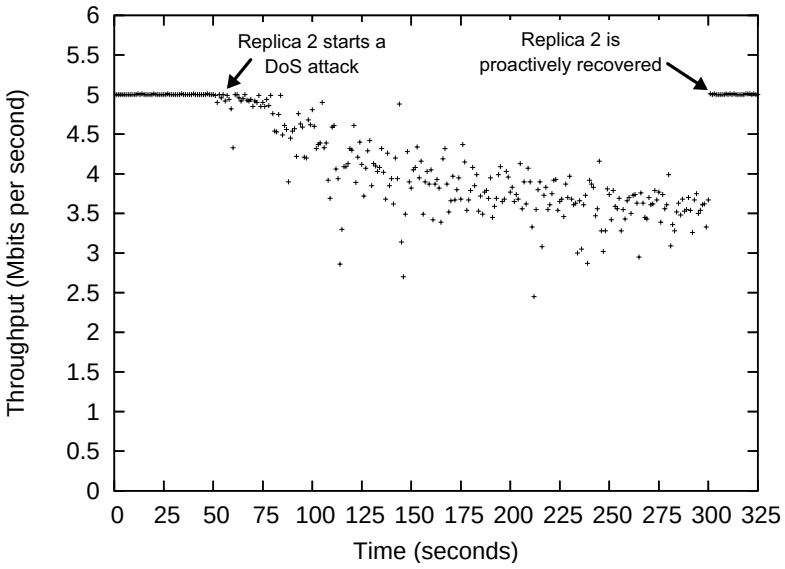
Throughput under a DoS attack from a compromised replica. Finally, the fourth set of experiments measured the resilience of the CIS-PS-SH against Byzantine faults, i.e., in the presence of up to f compromised replicas. Given that the CIS-PS-SH algorithms already tolerate up to f Byzantine faults, we choose a malicious behavior orthogonal to the algorithms logic that could nevertheless endanger the quality of the service provided by the CIS-PS-SH. In this way, we configured one of the CIS-PS-SH replicas (replica 2) to deploy a DoS attack 50 seconds after the beginning of the service execution. This DoS attack floods the LAN with packets of 1470 bytes sent at a rate of 90 Mbps. We observed how the throughput is affected during this attack and until the recovery of the replica. In order to show the effectiveness of proactive and reactive recoveries, we compared what happens when only proactive recoveries are used, and when they are combined with reactive recoveries. The results are presented in Figure 5.12.

Figure 5.12(a) shows that the CIS-PS-SH throughput is affected during the DoS attack from replica 2 when only proactive recovery is used. The throughput decreases during the attack and reaches a minimum value of 2.45 Mbps (half of the expected throughput). The attack is stopped when the local PRRW of replica 2 triggers a proactive recovery after 300 seconds of the initial time instant.

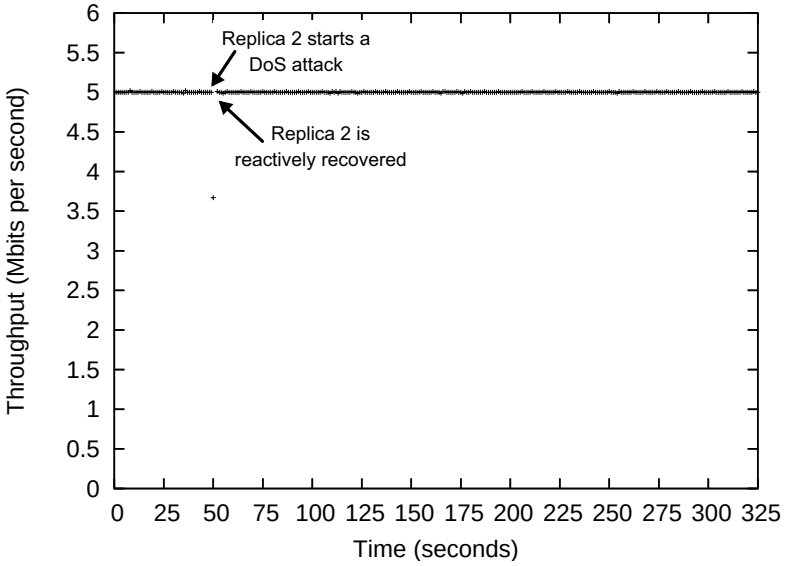
The utility and effectiveness of combining proactive and reactive recoveries is illustrated by Figure 5.12(b), which shows that the CIS-PS-SH throughput is minimally affected by the DoS attack from replica 2. This attack is detected by the remaining replicas given that invalid traffic is being sent to the LAN and consequently they all agree that replica 2 needs to be reactively recovered. The reaction is so fast that the throughput drops to 3.67 Mbps just during one second and then it gets back to the normal values.

5.1.4 Summary of Results

The experimental results allow to conclude that, with the exception of two specific configurations, both versions of the CIS-PS offer reasonable latency (< 4 milliseconds), throughput (> 200 messages per second), and loss rate (< 10 %), in the presence of powerful



(a) With proactive recovery only.



(b) With proactive and reactive recovery.

Figure 5.12: Throughput of the CIS-PS-SH during 325 seconds with $n = 4$ ($f = 1$ and $k = 1$). Replica 2 is malicious and launches a DoS attack to the LAN after 50 seconds. We present graphs without (a) and with (b) reactive recovery.

external DoS attacks of up to 60 Mbps. These results are even better when analyzed in the context of power systems, given that the scenarios presented in CRUTIAL deliverable D2 are less demanding than what CIS-PS is able to offer.

We have also shown than VM-based replication is a viable, and sometimes better, alternative to physical replication, at the price of not tolerating hardware faults.

Finally, the self-healing mechanisms used by CIS-PS-SH reduce the time window

an adversary has to compromise more than f replicas, and the experimental results show that self-healing allows instantaneous reaction when attacks from compromised replicas are detected.

5.2 CIS Communication Service

Here, we present the results of the evaluation of the CIS-CS. These results give an initial insight of the benefits provided by the CIS-CS solution to achieve communication with strong timeliness requirements over a large-scale network as expected to be found in the CI context. At the same time, this evaluation intends to shed some light on the trade-offs of using our solution in this same context.

This section first outlines the methodology of the evaluation. After presenting the setup used, we analyse the data obtained in the several evaluation cases. At the end of the section, we summarise and discuss the main results obtained.

5.2.1 Methodology

In order to evaluate the proposed solution, we assess the cost-benefit of the CIS-CS channel selection strategy –the Calm-Paranoid algorithm (CP)– in comparison with other existing similar approaches. Such similar solutions (including network flooding as well) are potential candidates to improve timeliness communication in CI control operations, whether overlay or not. The evaluation is conducted using simulation. We are led to opt for simulation rather than network execution for two fundamental reasons. The first one resides in the fact that it is reasonably unlikely to get proper access to the network resources used by a CI production environment to manage experiments. Furthermore, simply running experiments over the Internet does not match our goal, since the Internet is not representative of the CI network environment. To support simulation modeling and execution, we resort to the JSim simulation environment [36].

JSim is a component-based open-source software platform written in Java which provides common features to incrementally build, run and manage a variety of network simulation scenarios based on the packet-switching model. In particular, JSim natively implements several Internet basic protocols, what is consistent with the purpose of this work. Further, as the behaviour of simulated distributed applications can be encapsulated into functional individual JSim components (based on the JSim Autonomous Component Architecture [35]), the future effort of migrating our solution to a real-world setup is also facilitated.

Our evaluation intends to answer two immediate questions:

1. Given a set of messages with different deadlines transmitted by a certain strategy, at

what extent are these messages received in time?

2. What are the transmission costs associated with using the CP algorithm in relation to other strategies?

These questions are related to the feasibility of using the CP algorithm to support timely communication in CI control applications, namely if the network is under the effect of accidental failures and DoS attacks.

We simulated all strategies shown in Table 5.2. We denote a *simulation episode* as the simulation of a communication strategy during a time window equivalent to 5 hours in a simulated network environment setup, which considers a predefined control traffic source, under some faultload configuration (fault-free, accidental faults, and accidental plus malicious faults). The next section describes all simulation setup attributes and how such attributes were defined.

Approach	Overlay?	Basic idea	Reference
Best-Path (BP)	Yes	Send message through the best overlay channel. In case of failures, retransmit message at most 3 times using a fixed timeout of 3 seconds, all using the best overlay channel.	RON [8]
Calm-Only (C)	Yes	Our strategy with calm operation mode only (i.e., without auxiliary channels).	This work
Calm-Paranoid (CP)	Yes	Our strategy in 1-paranoid operation mode (i.e., with 1 auxiliary channel). Whenever possible, calm and paranoid channels employ distinct ISP access links.	This work
Flooding (F)	Yes	Send message using all available overlay channels	N/A
Multi-Path (MP)	Yes	Send message through 2 overlay channels: the direct path and a randomly chosen overlay channel (direct or not).	Mesh-routing [42]. Implementation as indicated in [9].
Hybrid (SOSR)	Yes	Send message through one random direct path. If failure, send message via 4 randomly chosen overlay channels (direct or not).	SOSR [23]
Round-Robin (RR)	No	Send message in a circular fashion alternating among all existent access links. Retransmit 3 times at most using a timeout of 3 seconds.	N/A

Primary-Backup (PB)	No	Send message always through one specific access link until it fails. In case of failure and if there is redundant links, pick another one. Retransmission scheme as RR.	Current in the Italian backbone

Table 5.2: List of all communication strategies evaluated

5.2.2 Simulation Setup

Network environment. We modelled our simulated network environment having as starting point an Italian ISP IP backbone topology presented at a report of the IRRIIS Project [40]. That topology (Figure 5.13(a)) is composed by 31 routers and 51 IP paths. Each router is capable of pushing data at the rate of 1 Gbps, and network paths provide a propagation delay of 50ms. We replicate that backbone for representing two fully decoupled ISP backbones across a virtual Italian territory, the whole set representing the large-scale network of the simulation (WAN).

On the underlying network topology, we deploy 17 nodes (the polygons in Figure 5.13(c)), each of which representing a CIS entry point of a different utility local subnetwork virtually situated in some of the 17 out of 20 distinct Italian autonomous regions (Figure 5.13(b)). A utility subnetwork provides local links with propagation delay of 10ms, and a single node representing a CIS device sends data at the transmission rate of 100Mbps. Each CIS is located at a control centre (CC) or a substation (SS), but these facilities are not represented explicitly in Figure 5.13(b).

CIS nodes run CIS-CS instances that allow communication between any pair of them. For each simulation episode, the CIS nodes use a single strategy. In the simulation of overlay strategies, CIS nodes use a subset of 7 out of 17 overlay nodes as transit nodes (the filled polygons in Figure 5.13(c)) responsible for relaying information upon request of the others. This means that each CIS-CS has access to 7 other CIS-CS nodes that can be used to relay messages[7]. A CIS node accesses the large-scale network by using a multihoming scheme composed of two redundant links, each one provided by one of the two existent ISPs.

Four special CIS-CS nodes (the circles in Figure 5.13(c)) are located in the main cities in distinct Italian regions. They are in charge of utilities with regional remote controller stations, thus being responsible for passing along traffic relative to them (e.g., traffic initiated by CC or receiving traffic from monitored SS). The remainder CIS-CS nodes (indicated as

[7]We elected a portion of CIS-CS nodes as relays by picking the ones already responsible for CC traffic (4 nodes) and others deemed at strategic locations across the ISP backbone (3 nodes). This made possible to minimise the network traffic load during simulation without compromising overlay network coverage.

(a) Italian ISP topology [40] (b) Italy by regions (c) JSim Simulation ISP topology

Figure 5.13: ISP network topology: from reality to simulation

squares in the same figure) are in charge of traffic relative to SS (e.g., signalling traffic sent by monitored SS to the respective CC).

Workload We separately generated the traffic of each CC and SS nodes involved in the setup. In total, 17 different traffic sources with the same duration of a simulation episode (i.e., 5 hours) were created, totalising 90,134 unique transmissions. We aggregate such amount of traffic sources to our simulated network environment, so the same set of traffic sources is utilised as input for all simulation episodes. We note that the resulting amount of communication over the simulated large-scale network from the sources of control traffic particularly depends on the strategy applied.

Traffic can be periodic or sporadic according to the type of operation involved. Message deadlines vary also as the type of operation in question. The information listed below is partially based on the Crutial Project Deliverable 9 [20]. Other information consists in data collected from partners of the project during the last 8th Technical Meeting of the Crutial Project held at October 2008. The result is that we identify 5 distinct types of traffic patterns:

- CC-STATE: CC periodically notifies its state to the other CCs with a message of deadline of 4 seconds. The message period is of 4 seconds;
- SS-STATE: SS notities its state to its associated CC with a message of deadline of 1 second. The message period is of 2 seconds;
- SS-ALARM: SS may send an alarm message to its corresponding CC. This message is sporadic and has a deadline of 1 second;

- CC-CMD: CC may remotely send a command message to one of its SSs with deadline of 2 seconds. This message is also sporadic;
- CC-HPCMD: CC may remotely send a high priority command message to one of its SSs with a shorter deadline of 1 second. As the command message, this message is sporadic.

For the periodic events, we initially set up a waiting random number r_i (proportional to the total number of CIS-CS nodes) before launching the sequence of periodic sends. Each periodic send does not necessarily occur at a precise instant, it happens at an instant that is a result of the known event period plus a random seed r_p (a randomly generated fraction of the period).

Faultload We generated and progressively applied at independent moments three different faultloads to the basic simulation setup (i.e., simulated network environment plus set of control traffic sources). The faultloads differ from each other by severity level. They represent the following failure scenarios:

- **Faultload 1 (fault-free)**: an ideal scenario with no failures in the large-scale network. Trivial case where no faultload source was really used;
- **Faultload 2 (accidental faults)**: a scenario where the large-scale network has accidental ISP network failures. Modelled using a combination of two specific network failure models encountered in the literature [19, 26, 28];
- **Faultoad 3 (accidental plus malicious faults)**: a more stringent scenario where the large-scale network is subject to accidental plus malicious failures. The idea is to simulate the effect of failures originated by DoS attacks directed onto the same large-scale network the CI local subnetworks are connected to. For it, a modification of the model of the faultload 2 was used.

The strategy we followed to simulate accidental and accidental plus malicious network failures in our setup consists in generating a faultload source for the whole network environment considering a simulated time window of 5 hours (the same we used to generate traffic). The intention is to have a faultload as an approximation of the impact of the respective failure scenario during the aforementioned simulation time period.

To obtain a faultload, we first specified a total number of failures f to be injected across all network components. For each network component class, we defined a specific number f_c corresponding to a certain portion of f such that the sum of each class-specific number f_c is equal or very close to f. The attribution of f_c is determined by some predefined classification of failures in terms of network component sets. The number of failures per network category f_c closely follows the distribution of unplanned ISP IP backbone failures shown in Markopolou et al. [28]. For all faultloads generated, we set $f = 74$ to each

simulated ISP backbone (i.e., a total of 148 failures for the 2 ISPs infrastructures). **We applied failures to backbone routers, links and CIS-CS node network interfaces**. In particular, a failure of the CIS-CS node network interface is the way of simulating the effects of failures on ISP backbone access routers used by such CIS-CS node.

The essential failure generation process is indicated as follows. Any failure has a time to start over the simulation time interval and, once started, it lasts a certain time interval. We simulated the effect of failures by disabling the functionality of the affected network component during some time, which represents the time-to-repair of the failure. Therefore, to inject a unique failure, **we have first to define some basic failure properties according to some network failure characterisation**. These properties are the failure starting and duration times, as well as the failures localisation per network component.

Accidental fault approximation. To generate the faultload for accidental failures, we based our failure injection on a blend of some network failure models present in the literature. To capture failure duration, we based on the models described in [19, 26], to capture failure starting time and localisation per network component, we used the model in [28]. In practice, the resulting model states that *(1)* the starting time of failures are randomly picked following the Weibull distribution over the network-wide simulated time window [28]; *(2)* 30% of all failures last more than 30 seconds according to a Pareto truncated distribution [19], while the remainder ones last up to 30 seconds following an Exponential distribution [26]; *(3)* for some network category, an individual element is selected according to a Power-Law based distribution [28].

Despite we are simulating the effect of failures from realistic models, the modelling approach here has limitations that may lead to bias. A first limitation is that we are merging two separate models that partially furnish the complete failure formalisation required into a common one in order to have **all** the necessary information to simulate failure behaviour in the network environment of interest. Such measure was necessary, since we just had complementary failure aspects explicitly formalised in each related work. Implicitly, we assume that both descriptions were obtained from observing the same large-scale network environment. We note, however, that while [28] entirely observed failures in an ISP backbone environment, which is closer to our environment here, [26, 19] took into account a composition of our own and third-party datasets, in both cases obtained from distinct experiments run over the Internet (i.e., considering an interdomain context with multiple ISPs). Finally, the second problem is that the models described by these works inherently represent the behaviour of accidental failures.

Crisis situation (accidental plus malicious faults) approximation. To approximate a scenario of accidental and malicious failures, we followed a simple reasoning that, at the same time, takes advantage of the formal descriptions available in the accidental network failure models mentioned before. Given that failures as a consequence of DoS attacks usually present more amplified effects on network components over the attack duration than their accidental counterparts, we could modify some aspect of those models such that we would produce some failures with increased power. While we do not claim that our modified model is an accurate characterisation of network failures caused by DoS

attacks, we indeed expect that the combined effect of these generated failures on the CI control application end-to-end communication can be stronger than the perceived impact of the accidental failures produced from the original models. With such synthetic faultload model, we intend to model a crisis situation.

The altered model for accidental plus malicious faults states that *(1)* a single failure initial time is obtained by uniformly selecting a random number within the simulation interval, thereby better fitting over all interval the starting times of all failures to be applied; *(2)* 80% of all failures last more than 30 seconds according to a Pareto truncated distribution, and the remainder 20% last up to 30 seconds following an Exponential distribution (i.e., altering the model in [19, 26] to give more room to the occurence of failures with longer duration); *(3)* for some network component class, an element is uniformly chosen, replacing a Power-law based distribution used in [28], thereby equally distributing the probability to fail over all elements of some network component set, instead of concentrating a higher incidence of failures on a restricted set of elements.

5.2.3 Cases of Analysis

We organise our evaluation in three distinct cases. A *case* corresponds to a particular analytic focus on the results of simulation episodes done under some faultload scenario: *(i)* failure-free scenario; *(ii)* accidental fault scenario; *(iii)* crisis scenario (accidental plus malicious faults). For each case, we subdivide the focus into two different aspects of analysis. They are described next in more detail.

Accounting deadlines missed CI remote process control operations are very sensitive to network instabilities whose impact ultimately leads to periods of untimely communication. Once the interaction between applications participating of such control needs to happen on a real-time basis, and the encompassing CI control operation is mission-critical, the network communication steps involved in such interaction also have to keep expected timeliness characteristics. Otherwise, CI control applications may experience failures from their communication as long as application message deadlines are not complied as supposed to be.

In this first case of analysis, we explore this notion of control application failure and observe the relative ability of CP to provide timely communication. We measure this feature by accounting the number of deadlines missed. Accordingly, we do the same for the other observed communication strategies.

Looking at the costs The application of mechanisms to increase timeliness for end-to-end control application communication requires an additional costs that can be observed in several dimensions. One possible is to examine the overhead originated from the interaction between any two communicating parts executing the same mechanism.

Here, we examine communication overhead via measuring the amount of extra messages sent. This metric enables us to reason about the degree of additional costs imposed on all individual network components involved in an end-to-end communication. Obviously, it includes the own endpoint CIS local processing cost of preparing and primarily putting messages on the large-scale network, what makes this applied measure of extra cost specially important to evaluate not only the potential of additional burden on the CIS-CS behaviour, also on the whole CIS device operation in a real situation where each observed strategy is applied.

From our analysis, the main results we observe are:

- Under faulty network conditions, CP was capable to provide more guarantees than other candidate solutions. Under harsher faulty condition, CP further manages to offer a benefit close to an optimal but extreme solution, flooding;
- CP presented a reasonable cost in relation to other observed solutions and demonstrated a significant lower cost than the same optimal (and very expensive) solution.

5.2.3.1 Failure-free Scenario

No deadlines were missed when the network environment is free of failures for all strategies simulated, so we do not provide a graph with these failures by strategy. On the other hand, extra costs were observed. The Figure 5.14 depicts the overhead incurred during control communication for each simulation episode. The y-axis of the graphic represents for all strategies the normalised amount of extra messages sent with respect to the most costly solution.

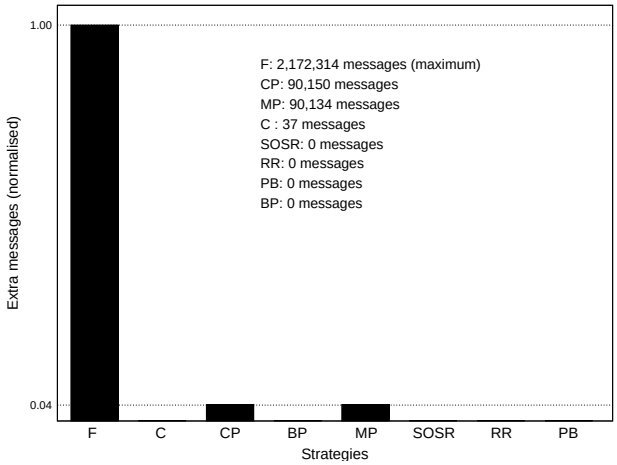


Figure 5.14: Extra messages (normalised values): no faults

At a first glance, we see on the leftmost side of the figure that the most expensive solution is undoubtedly the network flooding (scheme F). This result is not surprising. Just

as it was expected to have flooding the ideal case as of minimising the number of deadlines missed, we also expected to have flooding the worst case of overhead. It is a solution which offers both best-benefit and highest-cost. Particularly, the cost provided by flooding is significantly higher than the ones achieved by our solution (schemes C and CP).

When our solution (schemes C and CP) are compared with the other strategies, we conclude that applying one additional auxiliary paranoid channel implied one or more orders of magnitude of overhead. CP unadvisedly duplicates each message transmitted by always exploring the paranoid channel (like MP). It led CP to be a quite costly strategy. Without paranoid channel (scheme C), our solution presented a very negligible cost.

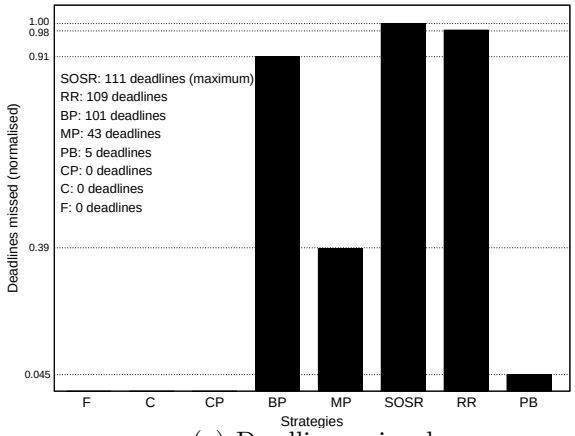
5.2.3.2 Accidental Faults Scenario

The Figure 5.15(a) summarises the relative number of message deadlines that were missed for the simulation episodes with accidental faults. Conversely, the Figure 5.15(b) shows the relative extra costs for the same simulation episodes. In both cases, the numbers presented are normalised with the worst benefit (most costly) solution observed.

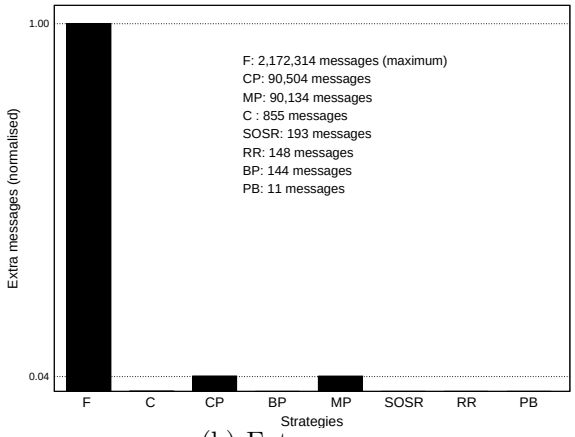
In this failure case, we observed in the Figure 5.15(a) that the scheme SOSR presented the highest amount of deadlines missed among all strategies examined, although in fact only a very small number of 111 of 90,134 deadlines were missed (less than 1%). The other schemes performed relatively better than SOSR. Sending a message through all available virtual channels (direct or not) leads to the best outcome among all examined strategies. However, the same can be achieved via schemes C and CP at much less additional cost. Similarly to the failure-free case, network flooding continued to be the most expensive solution while still offering an optimal benefit.

Surprisingly, the non-overlay scheme PB overperformed some candidate overlay strategies (all except C, CP and flooding) allowing for a lower number of deadlines missed. Such evidence suggests that PB (usually employed in CIs to improve control communication) is not an unsuitable solution for the purpose of timely control communication regarding a scenario of accidental failures. As observed, PB really worked out as long as it could offer a near optimal outcome (very close to what was provided by our solution and flooding) at a very low overhead, specially in relation to the same optimal-benefit strategies.

Similar to the failure-free case, CP in 1-paranoid mode has a larger absolute cost than its counterparts (except flooding). Nevertheless, starting to inject some faults into the network environment, we started to observe an interesting trend: we could early notice that CP is turned from a strategy that is more expensive than other schemes in a fault-free scenario (i.e., all strategies excluding C and F) into one with better benefits than those same schemes, having, at most, the same increasing cost in orders of magnitude. It is rather noteworthy that, as the environment starts to get harsher, CP was capable to circumvent **all** the failures which otherwise prevented deadlines from being accomplished for those same other candidate strategies. This is clear to see in our graphics by accompayning in paralell



(a) Deadlines missed



(b) Extra messages

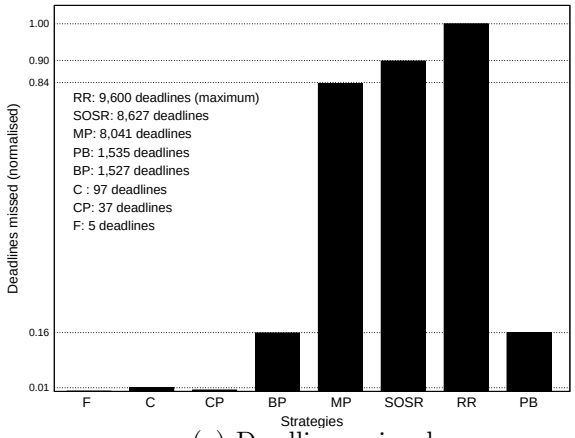
Figure 5.15: Cost-benefit under accidental Failures (normalised values)

both the progress of the costs for all the referred strategies in the fault-free and accidental scenarios (Figures 5.14 and 5.15(b), respectively) and their correlated benefits, with special attention to the accidental case (Figure 5.15(a)).

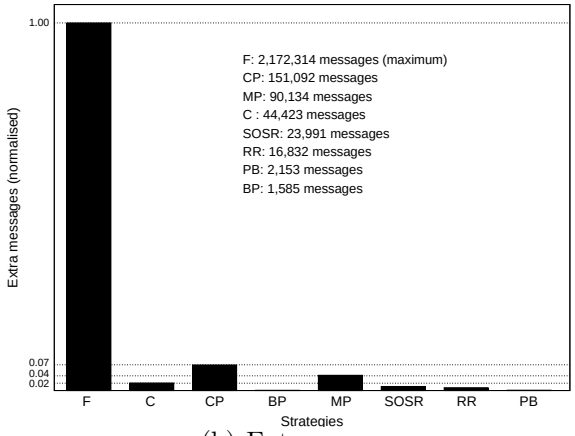
On the other hand, the cost of the non-paranoid case (scheme C) lays in the same order of magnitude of the overlay strategies MP and SOSR, and one order of magnitude more than the overlay scheme BP. This is because, although the scheme C does not consider auxiliary channels (like BP and on the contrary to MP and SOSR, which apply 1 and 4 auxiliary channels, respectively), C could perform more retransmissions. We observed that the most part of the control traffic flows in our simulation setup referred to messages with more extended deadlines. BP, MP and SOSR always explored a fixed number of retransmissions (3 retransmissions for BP and 1 retransmission for SOSR), even when the deadlines were larger, rendering less extra messages sent. MP did not retransmit messages at all. Similarly, the non-overlay strategies simulated (PB and RR) performed 3 retransmissions at most in their implementations, also leading to lower costs than the scheme C.

5.2.3.3 Crisis Scenario (accidental plus malicious faults)

The Figures 5.16(a) and 5.16(b) respectively refer to the number of deadlines missed and the overhead experienced by control communication using all strategies in a failure scenario that simulates a crisis situation. Like the other cases, the y-axis of both graphics here also show normalised numbers relative to the highest values obtained.



(a) Deadlines missed



(b) Extra messages

Figure 5.16: Cost-benefit in a crisis situation (normalised values)

As expected, the number of deadlines missed in this scenario was greater than the corresponding one in the accidental scenario. This observation matches when we look at the results of both all strategies and each specific strategy. In this failure scenario, the non-overlay round-robin scheme RR exhibited the highest amount of deadlines missed. RR reported 9,600 of 90,134 missed (about 10% of all messages sent). The other schemes performed relatively better than RR as can be seen in the results of the Figure 5.16(a).

Once the environment is more severe than the accidental scenario, the scheme PB is no longer near optimal. Even with its performance degraded, PB still had less extra cost than SOSR and MP while exhibiting a cost-benefit comparable to BP (respectively about

1.7% and 2% of all transmissions for both strategies).

A quite small number of 5 deadlines is missed when flooding (scheme F) is employed. These misses happened in situations where it was impossible to mask all failures, even flooding the network. When our solution was employed without auxiliary channels (scheme C), it missed 97 deadlines. This corresponds to a fraction of about 1% of all deadlines missed in comparison with the scheme RR (the worst performing strategy) and 6% of all missed by the scheme BP (the better strategy excluding our solution and flooding). This percentages decrease as one paranoid channel is added to the scheme C: about 0.4% of all deadlines missed in relation to RR and 2.4% with respect to BP.

The benefit provided by scheme C differs from the one of scheme F in one order of magnitude, but the former just uses one virtual channel per transmission and the latter explores all virtual channels available (a total of 26 channels in our setting). CP sounds promising, since we can decrease such difference between the non-paranoid approach and flooding by augmenting the scheme C with more auxiliary virtual channels while keeping a lower cost. In our results, we can see an initial trend towards it. In fact, the difference fell down much when the 1-paranoid scheme (i.e., one additional auxiliary channel) is applied, reporting from 97 to 37 deadlines missed, against 5 deadlines for flooding. We believe that this trend is maintained as we increase the number of paranoid channels, yet using a much lower amount of auxiliary channels than that one one of flooding.

Just as observed in the accidental case, CP had more overhead than other strategies (except flooding). However, again, we could observe here new changes of the cost-benefit trade-offs favoring 1-paranoid CP in relation to its counterparts (again, excluding schemes C and F). This special trend was reported earlier in the Section 5.2.3.2 when we started to inject accidental network failures into the testbed. Here, we noted that when the faultload is amplified to simulate more severe failures as the result of DoS attacks, 1-paranoid CP was capable to provide much more overall improvement than its counterparts. CP experienced less deadlines missed and, at least, the same incremental cost in order of magnitude. Interestingly, even exploring spatial redundancy as CP does, MP and SOSR could not exhibit the same progress that CP did. The basic difference comes from the overlay channel selection strategy employed by each strategy.

5.2.4 Discussion

The results presented allow us to clearly assess the main benefits and drawbacks of our solution with respect to other similar candidate approaches (overlay-based or not).

We showed that, in general, CP can provide an improvement of timeliness properties as required for SCADA/PCS communication that operates over a large-scale network environment. To quantify benefit, it was used a single measure of control application failure: the number of deadlines missed. By assessing this metric it was possible to assess the extent of timely guarantees that each candidate strategy can provide to control communication

when it is operating under three special faultload scenarios: *(i)* ideal, failure-free (Section 5.2.3.1); *(ii)* more common, with accidental failures (Section 5.2.3.2); *(iii)* crisis situation, under harsher failure effects of network DoS attacks (Section 5.2.3.3).

Moreover, CP can deliver a communication service at a reasonable incremental overhead. Overheads are defined here in terms of extra messages sent through the large-scale network. Such measure encapsulates other kinds of costs involved in an end-to-end communication, including the CIS local processing costs. CP shows a reasonable cost-benefit offering a much lower relative overhead when compared with network flooding, the ultimate but highly expensive solution. Besides, CP can offer more guarantees for timely communication at equivalent increasing cost than a traditional solution applied in practice in CIs (scheme PB). When auxiliary channels are disabled (non-paranoid CP), we found extra costs equivalent to almost all other solutions while providing more assurance for timely communication than them. Even with larger cost, paranoid use was justifiable: it could keep timely communication up, circumventing much more failures that caught other strategies that do not employ network flooding as long as the large-scale communication environment gets harsher and harsher, even failures affecting the non-paranoid CP.

6 Conclusions

This deliverable comprises two parts; in the first part two complementary experimental environments have been presented that are aimed to support the activities carried out in CRUTIAL in order: 1) to identify security-related vulnerabilities in software components and servers used in the CRUTIAL architecture (AJECT), and 2) to collect real data representative of attacks typically observed on the Internet that will be useful to build models characterizing the times between attacks distribution, and to understand malicious threats targeting SCADA related systems (honeypots). The thorough experimental results complete the preliminary ones presented in the interim report D26.

In the second part the Fosel component, used to mitigate the effects of DoS attacks, and the CIS-PS and CIS-CS components are tested in different configurations. Configuration parameters concern both the components deployment and the type and intensity of threats to which they are exposed. The CIS-PS experiments take advantage of the AJECT tool presented in the first part of the deliverable. The experiments had the aim of evaluating the effectiveness of these important CRUTIAL architecture components in critical scenarios. Most experiments have been performed on controlled, real environments, often exploiting virtualization technologies. Some results have been obtained by means of simulations.

Vulnerabilities identification and characterization of malicious user behavior

Overall, the results show that AJECT can be very useful in discovering vulnerabilities even in fully developed and tested software. Even though developers of the systems for which vulnerabilities have been discovered by AJECT were not forthcoming into disclosing the details of the vulnerabilities, which is understandable because all servers were commercial applications, most of them actually 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.

Concerning the collection and analysis of attack data based on honeypots, three complementary types of honeypots have been investigated . The first two types of honeypots are dedicated to the observation of malicious traffic targeting traditional IT systems connected on the Internet and supporting general services used in most computing based infrastructures, including critical infrastructures used e.g. in electrical power systems. The third type of honeypots is typically dedicated to the analysis of malicious traffic targeting specific ports used in SCADA systems or for which vulnerabilities have been published. The data collected from these honeypots provide useful insights considering different perspectives (depending on the level of interaction offered to the attackers and the services exposed by the honeypots). Additional insights are provided in deliverable D19 that addresses the statistical modelling of the times between attacks distribution based on the data collected

from the low and high interaction honeypots. As discussed in Section 3.4, the SCADA honeypot provide some preliminary indication the potential existence of real threats targeting SCADA related systems. This study is still at a preliminary stage and need to be strengthened in the future by: 1) collecting data during a longer period of time, 2) deploying our high-interaction honeypot and the SCADA honeypot at other locations to investigate if new trends and activities appear, depending on the location where the honeypots are deployed, and finally 3) enhancing the capabilities offered by the current implementations, by including additional vulnerabilities that can be used by potential attackers to compromise systems and servers connected to the Internet, including SCADA based systems.

Experimental evaluation of Fosel, CIS-PS and CIS-CS

The experimental setting for Fosel evaluation consisted of an Emulab testbed (involving 11 LANs, 60 PC nodes and 9 routers). Both attacks directed towards an application (FTP) and towards the overlay network have been considered. The experiments on application target show the effect on response time and failure rate as functions of the two parameters (P, probability of discarding packets, and C, number of forwarded copies of good messages) of the method. In the experiments concerning a scenario of attack to the overlay network, throughput as well as end-to-end latency have been evaluated, as functions of the number of attacked nodes in the overlay network. The experimental results show the effectiveness of the proposed method, moreover they validate the results previously obtained through simulation.

CIS-PS and CIS-CS evaluation experiments have been performed using two different methodologies. The former set of experiments have been performed by actually measuring performance indices in a real environment; besides evaluating the effectiveness of the protection mechanism, the viability of using virtualization techniques to achieve replication of components (as an alternative to replication on different physical machines) has been evaluated. AJECT has been used to emulate attacks for the evaluation of the CIS-PS. The Self-healing variant of CIS-PS has also been evaluated, showing how the combination of proactive-reactive strategies can be used to limit the probability of experiencing more simultaneous faults than the given replication schema can tolerate. The experimental results allow to conclude that, with the exception of a few specific configurations, both versions of the CIS-PS offer reasonable latency (< 4 milliseconds), throughput (> 200 messages per second), and loss rate (< 10 %), in the presence of powerful external DoS attacks of up to 60 Mbps.
CIS-CS has been evaluated by using a simulation environment (based on J-Sim). The results allow to conclude that the proposed technique can provide an improvement of timeliness properties as required for SCADA/PCS communication that operates over a large-scale network environment. To quantify benefit, it was used a single measure of control application failure: the number of deadlines missed. By evaluating this metric it was possible to assess the extent of timely guarantees that each candidate strategy can provide to control communication when it is operating under given faultload scenarios. Moreover, the proposed

technique can deliver a communication service at a reasonable incremental overhead (defined here in terms of extra messages sent through the large-scale network). Such measure encapsulates other kinds of costs involved in an end-to-end communication, including the CIS local processing costs. The extensive experiments showed that the proposed method as reasonable cost-benefit ratio, offering a much lower relative overhead when compared with network flooding, the ultimate but highly expensive solution.

Final remarks

Overall the experimental results have demonstrated the relevance of the CRUTIAL project outcome, both from the point of view of the auxiliary tools developed for assessing the vulnerability of existing systems (including those specifically built for protecting critical infrastructures), and in studying and modelling the shape of malicious attacks (also in SCADA-specific environment).

Moreover the thorough experimental analysis conducted on Fosel and on the CIS-CS and CIS-PS allowed us to reach encouraging conclusions about the effectiveness of the CRUTIAL architecture components developed within the project.

Last but not least, the methodological framework that was set up to perform systematic and extensive experiments, and to relate the obtained results to the requirements for the specific type of critical systems considered in the project, is per-se a valuable contribution that can be exploited also in other contexts.

Bibliography

[1] Dshield; cooperative network security community – internet security. Adresse Internet : http://www.dshield.org.

[2] *Emulab total network testbed.* http://www.emulab.net/.

[3] The sans institude, internet storm center. Adresse Internet : http://isc.sans.org.

[4] Pax, the pax team, 2007. Internet Address : http://pax.grsecurity.net.

[5] E. Alata, I. Alberdi, V. Nicomette, P. Owezarski, and M. Kaaniche. Internet Attacks Monitoring with Dynamic Connection Redirection Mechanisms. *Journal in Computer Virologgy*, 4(2):127–136, 2007.

[6] Ion Alberdi, Jean Gabès, and Emilien L Jamtel. UberLogger : un observatoire niveau noyau pour la lutte informati que défensive. In *Symposium sur la Sécurité des Technologies de l'Inform ation et des Communications 2005*, 2005.

[7] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Scaling byzantine faulttolerant replication to wide area networks. In *Proc. Int. Conf. on Dependable Systems and Networks*, pages 105–114, 2006.

[8] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 131–145, 2001.

[9] David G. Andersen, Alex C. Snoeren, and Hari Balakrishnan. Best-path vs. multi-path overlay routing. In *IMC '03: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement*, pages 91–100, 2003.

[10] Hassan Artail, Haidar Safa, Malek Sraj, Iyad Kuwatly, and Zaid Al Masri. A hybrid honeypot framework for improving intrusion detection systems in protecting organizational networks. *Computers & Security*, 25(4):274–288, 2006.

[11] Paul Barham, Boris Dragovic, Keir Fraiser, Steve Hand, Tim Harris, Alex Ho, Rolf Neugebaurer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles - SOSP'03*, October 2003.

[12] H. Beitollahi and G. Deconinck. Fosel: Filtering by helping an overlay secure layer to mitigate dos attacks. In *Proceedings of the 7th IEEE Int. Symp. on Network Computing and Applications (NCA'08)*, pages 19–28, Cambridge, MA, USA, 10-12 July 2008.

[13] Eric Byres, Dan Hoffman, and Nathan Kube. The special needs of SCADA/PCN firewalls: Architectures and test results. In *Proc. of the 10th IEEE Int. Conf. on Emerging Technologies and Factory Automation*, 2005.

[14] CAIDA. The cooperative association for internet data analysis. Adresse Internet : http://www.caida.org.

[15] E. Cook, M. Bailey, Z. Morley Mao, D. Watson, F. Jahanian, and D. McPherson. Toward Understanding Distributed Blackhole Placement. In *2004 ACM Workshop on Rapid Malcode (WORM'04)*, pages 54–64, New york, 2004. ACM Press.

[16] M. Crispin. Internet Message Access Protocol - Version 4rev1. RFC 3501 (Proposed Standard), March 2003. Updated by RFCs 4466, 4469, 4551, 5032, 5182.

[17] W.J. Croft and J. Gilmore. Bootstrap Protocol. RFC 951 (Draft Standard), September 1985. Updated by RFCs 1395, 1497, 1532, 1542.

[18] G. M. Voelker D. Moore and S. Savage. Inferring Internet Denial of Service Activity. In *10th USENIX Security Symposium*, 2001.

[19] Michael Dahlin, Bharat Baddepudi V. Chandra, Lei Gao, and Amol Nayate. End-to-end wan service availability. *IEEE/ACM Trans. Netw.*, 11(2):300–313, 2003.

[20] G. Deconinck, H. Beitollahi, G. Dondossola, F. Garrone, and T. Rigole. Testbed deployment of representative control algorithms, January 2008. Project CRUTIAL EC IST-FP6-STREP 027513 Deliverable D9.

[21] Susanna Donatelli6(Editor), Eric Alata4, Andrea Bondavalli3, Davide Cerotti6, Alessandro Daidone3, Silvano Chiaradonna3, Felicita DiGiandomenico3, Ossama Hamouda, Mohamed Kaniche, Vincent Nicomette, Francesco Romani, and Luca Simoncini. Model-based Evaluation of the Middleware Services and Protocols and Architectural Patterns. Technical Report Deliverable D19, Critial Utility InfrastructuAL Resilience, Project co-funded by the European Commission within the Sixth Framework Programme, 2009.

[22] N. Neves et al. *Architecture, Services and Protocols for CRUTIAL (D18)*, 2009. WP4.

[23] Krishna P. Gummadi, Harsha V. Madhyastha, Steven D. Gribble, Henry M. Levy, and David Wetherall. Improving the reliability of internet paths with one-hop source routing. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 13–13, 2004.

[24] S. Kent. IP Authentication Header. RFC 4302 (Proposed Standard), December 2005.

[25] R. Kier. *UDPFlood 2.00.* www.foundstone.com/us/resources/proddesc/udpflood.htm. visited at July 2008.

[26] Zhi Li, Lihua Yuan, Prasant Mohapatra, and Chen-Nee Chuah. On the analysis of overlay failure detection and recovery. *Computer Networks*, 51(13):3828–3843, 2007.

[27] F. Jahanian M. Bailey, E. Cooke and J. Nazario. The Internet Motion Sensor - A Distributed Blackhole Monitoring System. In *Network and Distributed Systems Security Symposium (NDSS-2005)*, San Diego, CA, USA, 2005.

[28] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, Yashar Ganjali, and Christophe Diot. Characterization of failures in an operational ip backbone network. *IEEE/ACM Transactions on Networking*, 16(4):749–762, 2008.

[29] D.L. Mills. Network Time Protocol (NTP). RFC 958, September 1985. Obsoleted by RFCs 1059, 1119, 1305.

[30] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343.

[31] O. Bonaventure N. Vanderavero, X. Brouckaert and B. Le Charlier. The Honeytank: A Scalable Approach to Collect Malicious Internet Traffic. In *International Infrastructure Survivability Workshop (IISW'04)*, Lisbon, Portugal, 2004.

[32] National Institute of Standards and Technology. Secure Hash Standard. Federal Information Processing Standards Publication 180-2, August 2002.

[33] Nuno Neves, Joo Antunes, Miguel Correia, Paulo Verissimo, and Rui Neves. Using attack injection to discover new vulnerabilities. In *Proceedings of the International Conference on Dependable Systems and Networks*, Philadelphia, USA, June 2006. DSN.

[34] V. Nicomette, M. Kaniche, and E. Alata. Attackers Behavior: Experiment and Results. Technical Report Report, LAAS-CNRS, 2009.

[35] JSim Web Page. The autonomous component architecture. Available at <`http://www.j-sim.org/whitepapers/aca.html`>, 2005.

[36] JSim Web Page. Jsim: Java-, component-based, compositional simulation environment. Available at <`http://www.j-sim.org/`>, 2008.

[37] Fabien Pouget, Marc Dacier, and Van Hau Pham. Leurre.com: on the advantages of deploying a large scale distributed honeypot platform. In *E-Crime and Computer Conference (*Ecce *'05)*, Monaco, 2005.

[38] Niels Provos. A Virtual Honeypot Framework. In *13th* Usenix *Security Symposium*, San Diego, Ca, 2004.

[39] Niels Provos and Thorsten Holz. *Viirtual Honeypots: From Botnet Tracking to Intrusion D etection*. Addison Wesley, 2007.

[40] Vittorio Rosato and et al. IRRIIS Project, Deliverable D2.1.2: Final report on analysis and modelling of LCCI topology, vulnerability and decentralised recovery strategies. Available at <`http://www.irriis.org/File.aspx?lang=2&oiid=9135&pid=572`>, 2007.

[41] Fred B. Schneider and Lidong Zhou. Implementing trustworthy services using replicate state machines. *IEEE Security & Privacy*, 3(5):34–43, September 2005.

[42] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using xml. *SIGOPS*, 35(5):160–173, 2001.

[43] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley Professional, 2002.

[44] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer communications (SIGCOMM'01)*, pages 149– 160, San Diego, CA, USA, August 2001.

[45] M. van Eeten, E. Roe, P. Schulman, and M. de Bruijne. The enemy within: System complexity and organizational surprises. *International CIIP Handbook*, 2:89–110, 2006.

[46] P. Verissimo, N. Neves, C. Cachin, J. Poritz, D. Powell, Y. Deswarte, R. Stroud, and I. Welch. Intrusion-tolerant middleware: The road to automatic security. *IEEE Security and Privacy*, 4(4):54–62, July/August 1996.

[47] P. Verissimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 3–36. Springer-Verlag, 2003.

[48] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, 2001.

[49] Paulo Verissimo. Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1), 2006, http://www.navigators.di.fc.ul.pt/docs/abstracts/ver06travel.html.

[50] J. Wang, L. Lu, and A. Chien. Tolerating denial-of-service attacks using overlay networks-impact of topology. In *Proceedings of the ACM Workshop on Survivable and Self-Regenerative Systems*, 2003.

[51] C. Wilson. Terrorist capabilities for cyber-attack. *International CIIP Handbook*, 2:69–88, 2006.