DiveInto: Supporting Diversity in Intrusion-Tolerant Systems

João Antunes Nuno Neves
LASIGE, Departament of Informatics,
Faculty of Sciences, University of Lisboa, Portugal
{jantunes,nuno}@di.fc.ul.pt

***Abstract*—Intrusion tolerant services are usually implemented as replicated systems. If replicas execute identical software, then they share the same vulnerabilities and the whole system can be easily compromised if a single flaw is found. One solution to this problem is to introduce diversity by using different server implementations, but this increases the chances of incompatibility between replicas. This paper studies various kinds of incompatibilities and presents a new methodology to evaluate the compliance of diverse server replicas. The methodology collects network traces to identify syntax and semantic violations, and to assist in their resolution. A tool called DiveInto was developed based on the methodology and was applied to three replication scenarios. The experiments demonstrate that DiveInto is capable of discovering various sorts of violations, including problems related with nondeterministic execution.**

***Keywords*-service replication, diversity, intrusion tolerance;**

I. Introduction

Intrusion tolerance is a security and dependability paradigm that has been gaining momentum over the past decade bacause it lets system designers address both accidental faults and attacks in a seamless manner [1]. An usual way to deploy an intrusion tolerant (IT) service is through a middleware that implements Byzantine state machine replication [2], [3]. Server replicas execute the same requests from the clients, and rely on the middleware protocols to carry out the coordination actions. The purpose of using replication is to keep the overall service correct even if some of the replicas are compromised. Typically, a system with n servers can tolerate up to f corrupted replicas, with $n \geq 3f+1$[1]. Malicious replica behavior is usually addressed by running the same request operation in all replicas, and then by selecting at the client the result that has more than a predetermined minimum number of votes ($f+1$ equal votes).

Replication is normally carried out with the same software (e.g., operating system and server application) at each (virtual) machine, so that correct servers produce identical results. However, in part due to their complexity, systems can remain to some extent faulty and/or vulnerable. Therefore, if an attacker is able to compromise one of the replicas, he or she can easily attack the remaining servers, defeating the original purpose of having replication. In the past, this issue has been mainly considered an orthogonal problem to the design of Byzantine replication protocols [5], [4], [6], [7], [8]. However, once the actual deployment of systems is considered, it becomes a fundamental problem for which there are very few practical approaches.

[1]Recently, some authors have looked for particular solutions with lower bounds, e.g., $n \leq 2f+1$, starting with the work of [4].

One potential solution is to take advantage of the inherent diversity provided by different software products that implement the same functionality, with the expectation that they are compromised in different ways. For example, at the operating system level, one can employ distinct flavors of UNIX and Windows [9]. Other good candidates are application level servers. For instance, there are several IMAP applications freely available on the Internet that could be utilized to build an IT email service. However, distinct software implementations, even if complying to a formal specification (e.g., an IETF protocol specification), may behave differently and/or respond with small variations due to specification gaps, incorrect implementation or misconfiguration. Moreover, diversity also increases the difficulty of ensuring determinism on the execution of the replicas, an important requirement to keep the various servers with consistent states. All these issues can have an impact on the replica responses, by causing them to look different, and thus preventing the required quorum on the results from being reached. In this case, the client is left with a number of responses from which it can not distinguish the correct ones from the malicious, consequently stopping the progress of the computation.

This paper studies various types of violations based on the content of the server replies, and presents a new methodology to evaluate the compliance of diverse IT replica configurations. The methodology employs network traces with the messages exchanged among the clients and the servers to identify syntax and semantic violations, and to assist developers in their resolution. We also developed a tool called DiveInto, based on our methodology, and used it to evaluate three different replication scenarios. In each of these scenarios, we resorted to a diverse set of servers (we selected FTP, SMTP, and POP because at this moment we are concentrating our research in IETF protocols). The experiments demonstrate that DiveInto is capable of discovering various kinds of violations, some of them related to syntax and semantic errors, and others to nondeterminism in the executions.

II. Overview of an IT System

An IT replicated system offers a service to a group of clients under very weak failure assumptions (see Figure 1). Typically, it is assumed that the adversary can perform various

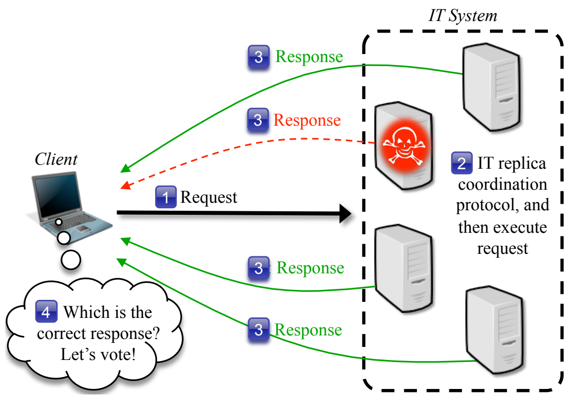
IT System
3 Response
3 Response
Client
2 IT replica coordination protocol, and then execute request
1 Request
3 Response
3 Response
4 Which is the correct response? Let's vote!

Figure 1. IT system architecture.

attacks on the network (e.g., replay, delay, re-order, corrupt messages), and that he or she controls an undetermined number of clients and up to f servers (for a total of $n \geq 3f+1$ replicas). Nevertheless, even under these challenging conditions, the system needs to provide correct responses to the good clients.

The design and implementation of an IT system has to address two main concerns. First, the protection of the communication between the client and the servers, where it is necessary that all correct replicas execute the request (step 1 of the figure) and then that the client is able to select a correct response (step 3). By employing re-transmissions and cryptographic methods, it is possible to secure the messages from the network attacks and ensure their delivery. The selection of the response is a bit more delicate because some of the replies might be produced by malicious replicas, and therefore contain erroneous data. Consequently, the usual procedure is to wait for $f+1$ **equal** responses, and only then choose this answer (this ensures that at least one correct replica vouches for the reply). If no response has such a quorum, then the client is unable to use the service because there is no generic mechanism to distinguish correct from wrong answers[2].

Second, all correct servers should start to execute from an identical initial state, and then they should evolve through the equivalent states as they process the requests [3]. This is achieved by running a Byzantine replication protocol among the servers (step 2), which associates a processing order number to each request and makes sure the correct servers carry out the same requests. More formally, a server can be modeled by two functions: $f_t : S \times I \rightarrow S$ is the *transition function* that maps the current state and input to the corresponding next state; and $f_o : S \times I \rightarrow O$ the *output function* that maps the current state and input to the corresponding output, with S being the set of finite states, I being the input messages that a server can receive (i.e., the requests) and O being the respective output (i.e., the responses). The IT implementation should guarantee that correct servers always exhibit an equivalent behavior by returning similar replies, which implies that the state transition and output functions among the correct replicas should be indistinguishable.

[2]The client could for instance carry out validity checks on the data to exclude the wrong answers. However, in general this sort of solutions is ineffective because we assume that the malicious replicas are controlled by a human that can produce data to evade the checks.

III. Classification of Violations

A. Syntax and semantic violations

Servers offer a service accordingly to some specification. Whenever a replica's output differs from what is expected, we call it a *syntax violation*. More specifically, we denote $f'_o : S \times I \rightarrow O'$ with output $O' \neq O$ as a non compliant output function that returns a different response from the other replicas for the same current state and input. If a given replica has an output function $f'_o(s,i) \neq f_o(s,i)$ for some $s \in S$ and $i \in I$, we say it has a syntax violation in state s for input i.

Another type of violation concerns with the meaning of the message, as opposed to its actual format. If the meaning of a message is different from the expected, we say that a *semantic violation* has occurred. Semantic violations are more serious because they typically reflect a deviation in the internal state of the replica. Hence, a semantic violation is usually, but not necessarily, also accompanied by a syntax violation. We denote semantic violations when the replica's transition function f'_t is different from f_t, i.e., when $f'_t(s,i) \neq f_t(s,i)$ for some $s \in S$ and $i \in I$.

Naturally, the cause for violations is linked to the actual implementation or configuration of the servers. Errors or differences in messages can be attributed to several reasons, from which some examples are:

- Incompatible configurations: an option to refuse a username without the domain (*syntax violation*) or to support some particular feature (*semantic violation*);
- Undefined behavior by the specification: an additional explanation of the response (*syntax violation*) or an optional state in the implementation (*semantic violation*);
- Incorrect implementation: a misspelled word (*syntax violation*) or an error that leads the server into an incorrect state (*semantic violation*);
- Nondeterminism: a message field with the date or time (*syntax violation*) or a random number that leads the servers to divergent states (*semantic violation*).

B. Mitigation

Both syntax and semantic violations have an impact on the correctness of the IT system and ultimately they can prevent clients from utilizing the service because the required quorum on the answers might not be reached. Our solution is aimed at identifying violations between replicas and providing the necessary additional information to eliminate them. Below, we discuss the main approaches to remove the violations:

Eliminate at the source: Removing a syntax or semantic violation at its source requires specific knowledge about the server execution, configuration, or actual implementation. If

$f_o : S \times I \to O$ is the output function that models the desired responses of the replicas and $f'_o : S \times I \to O'$ the output function of an invalid replica that produces a different output, removing the syntax violation at the source means changing the actual replica so that its output function f'_o conforms to f_o. For example, a welcome banner returned by a replica may be the source of a syntax violation if a server name is different from the expected. Many times, this problem can be corrected by changing the server configuration value to the right name. A semantic violation, such as a missing feature, could be removed by enabling the corresponding functionality in the configuration of the server or by implementing it from scratch, which is the equivalent of correcting the replica's non compliant transition function f'_t into f_t.

Normalization: Another approach to resolve a violation is to change the output of the invalid replica, in order to normalize the replies so that they conform to the other replicas. Being f'_o the incorrect output function of a replica that produces invalid output $O' \neq O$ for the same input I, this approach consists in creating a normalizer function $norm : O' \to O$ that transforms invalid responses into correct ones, and thus $norm(f'_o(s,i)) = f_o(s,i)$, $\forall s \in S, i \in I$. However, one must take into account that since the normalizer uses the output of the invalid replica, $f'_o(s,i)$, it must get the information required to construct $f_o(s,i)$. For example, if the invalid response, $f'_o(s,i)$, does not contain an expected identifier, then it may be difficult for the normalizer to produce $f_o(s,i)$.

Circumventing: A violation can also be prevented from occurring. If the request types that the replica is failing to process correctly are optional or not essential to the overall replicated service, the violation can be removed by blocking those requests of all replicas. A firewall component in front of the replicas could provide ingress filtering to ensure that only the requests that can be correctly processed by all servers are allowed. This firewall can be modeled as limiting the input set to $J \subset I$, so that $f'_o(s,j) = f_o(s,j)$ and $f'_t(s,j) = f_t(s,j)$, $\forall s \in S, j \in J$.

IV. Methodology

There is a reasonable amount of research dedicated to the analysis of software components. One way to study the behavior of a replica is to look for instance at its internal execution and state, the contents of the files it reads or writes, and the resources it is using. However, performing *whitebox* analysis on a replicated system with diversity can be extremely difficult. For one, as the number of replicas increases, so does the cost to analyze their behavior. Additionally, distinct server implementations may not be directly comparable. For instance, one server might use a single large file to maintain some of its state, while another server uses a hierarchy of several files, and still another server resorts to a database system.

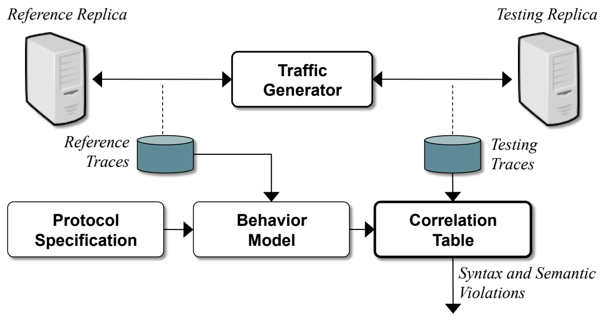

Figure 2. Methodology overview.

Another way to study the behavior of a software component is to look at it as a *blackbox*, and examine its input and output interfaces. Since the blackbox approach can be built around the specification of the component under analysis, it is well suited for modeling the behavior of the replicas without having to scrutinize their internal states and configurations.

The approach we propose examines the behavior of the replicas by inspecting and comparing their state transition and output as they process the same input. If the servers deviate from the expected state, eventually this will be reflected in their behavior and thus in their output. Our methodology is depicted in Figure 2, and in general it consists in capturing the behavior of a reference replica and in verifying if other replicas conform to it.

A. A methodology for detecting violations

Our methodology resorts to a specification of the protocol that the replicated system uses to communicate with the clients and to network traces that were collected from the replicas (see Figure 2). *Protocol Specification* formally defines the format of the messages (protocol language) and the rules for exchanging those messages between the clients and the servers (protocol state machine). A Mealy machine $(S, I, O, f_t, f_o, s_0, F)$ is a well suited formal representation for the protocol specification:

S	is a finite, non-empty set of states,
I	is the input alphabet (finite set of requests),
O	is the output alphabet (finite set of responses),
f_t	is the transition function: $f_t : S \times I \to S$,
f_o	is the output function: $f_o : S \times I \to O$,
s_0	is the initial state, and
F	is the set of final states.

This specification can be obtained in several ways. If the replicas use an open protocol and its standard is available (e.g., RFC), then it can be translated into a finite-state machine (FSM). For closed protocols, however, an approximate specification must be inferred, using for instance reverse engineering techniques. In the next section, we will show that an approximate specification can also be used in our approach.

The protocol specification is an *incompletely defined* FSM because it models the behavior of the client and the server

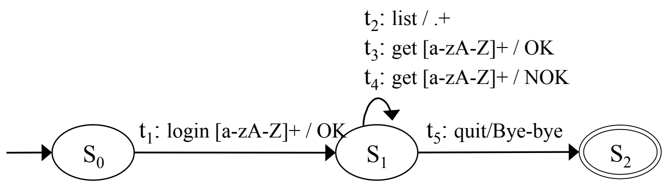

Figure 3. Protocol specification.

for only a relevant set of transitions that are necessary to understand the protocol operation and to create compliant implementations—it does not necessarily define for every state and input, a next state and an output. Some parts of the protocol execution are usually loosely described or omitted, giving the developers some liberty for the implementation. For instance, some specifications may not be clear about how a server should reply to malformed protocol messages, and therefore, some implementations may choose to return an error message while others may opt not to respond. Another example is when the specification defines message types with optional descriptions that are server-dependent, such as welcome banners or the FTP protocol replies.

Let us consider a simple example. Figure 3 shows the Mealy machine of a protocol specification. Each transition defines an input and an output. For instance, transition t_1 in state s_0 accepts requests of the type "login [a-zA-Z]+"[3] and responses of the type "OK". If successful, the protocol jumps to state s_1 where it expects three different transitions for the commands "list" and "get". The command "get" in particular can take a parameter of the type "[a-zA-Z]+" and two types of responses: "OK" or "NOK". This specification is an incompletely defined machine because it does not define transitions for the entire input and state space. For instance, the behavior is undefined if the client sends a "get" command without parameters or in state s_0.

A server that implements a protocol specification, on the other hand, is a *completely defined* machine because its behavior is fully realized by its particular implementation and configuration—for every state and input, it defines a next state and an output (which may be null). Our methodology obtains the *Behavior Model* of a reference replica by creating a completely defined machine based on the FSM of the protocol specification, with the difference that it redefines the original transition and output functions (f_t and f_o) to accept only the messages from the reference replica.

The behavior model is constructed using network traces containing client-server interactions together with the protocol specification. Therefore, the coverage of this model depends on the network traces. A trace containing *all* possible client-server interactions would yield a completely defined machine with the entire protocol coverage. However, this is unpractical (if not impossible) and it may not even be desirable. In fact, in order to limit the exposure of the replicas, we may wish to restrict the service provided by the IT system to a subset of the servers' functionality. In this case, we would also like to limit the evaluation of the diverse replicas to the respective subset of the protocol specification.

[3]We resort to regular expressions to represent message fields that can take different values depending on the specific message. In this case, this field corresponds to the *username* of the *login* command.

Table I
REFERENCE AND TESTING TRACES.

nth	Req_n	$Resp_n$	$Resp'_n$
1	login user1	OK	OK
2	list	file1	file1
3	quit	Bye-bye	Good bye
4	login user2	OK	OK
5	list	file2	file1,file2
6	get file2	OK	OK
7	quit	Bye-bye	Good bye
8	login user1	OK	OK
9	get file2	NOK no file	NOK no file
10	quit	Bye-bye	Good bye

Returning to the example, the reference trace from Table I (second and third columns) and the protocol specification can be used to build a behavior model that completely defines the transition and output functions for the entire trace. The behavior model is shown in the second column of the correlation table in Table II. One can see, for instance, that the behavior model defined transitions t_{11} and t_{12} at state s_0 based on the original transition t_1 of the protocol specification. These transitions accept respectively the first, the fourth, and the eighth requests/responses from the traces (see also column *nth* in both tables), and take the protocol to state s_1. The behavior model thus defines more restrictive transitions to accept *only* the input/output present in the reference trace.

A *Traffic Generator* is used to exercise the replicas into performing equivalent kinds of tasks that are expected to be executed on the IT system. Although the traffic generator acts as a single-threaded client, this is just for executing and analyzing each protocol task and transition individually, which is unrelated to the IT system support for multiple and concurrent connections. The generated requests and the observed responses are saved as the network traces. Our approach is flexible enough to control the degree of protocol coverage that will be evaluated. Therefore, this component must create the protocol requests that cover the desired part of the protocol specification, allowing the evaluation to focus on a subset of the replicas' functionality.

The requests produced by the traffic generator are used as input to both the *Reference Replica* and the *Testing Replica*. The messages exchanged with the reference replica are stored as the *Reference Trace*, and then they are utilized to construct the behavior model. Messages transmitted between the traffic generator and the testing replica are saved as the *Testing Trace*, and they are used to correlate the behavior of the testing replica with the behavior model. By giving the same input on both replicas, we ensure that their behavior can be directly compared—if their transition and output functions are

Table II
CORRELATION TABLE.

Protocol Specification	Behavior Model	nth	$Resp'_n$	Accepted
$t_1(s_0$, "login [a-zA-Z]+/OK"$) = s_1$	$t_{11}(s_0$,"login user1/OK"$) = s_1$	1,8	"OK", "OK"	true
	$t_{12}(s_0$,"login user2/OK"$) = s_1$	4	"OK"	true
$t_2(s_1$,"list/.+"$) = s_1$	$t_{21}(s_1$,"list/file1"$) = s_1$	2	"file1"	true
	$t_{22}(s_1$,"list/**file2**"$) = s_1$	5	**"file1,file2"**	**false**
$t_3(s_1$,"get [a-zA-Z]+/OK"$) = s_1$	$t_{31}(s_1$,"get file2/OK"$) = s_1$	6	"OK"	true
$t_4(s_1$,"get [a-zA-Z]+/NOK"$) = s_1$	$t_{41}(s_1$,"get file2/NOK ..."$) = s_1$	9	"NOK ..."	true
$t_5(s_1$,"quit/Bye-bye"$) = s_2$	$t_{51}(s_1$,"quit/**Bye-bye**"$) = s_2$	3,7,10	**"Good bye","Good bye","Good bye"**	**false**

equivalent, then if they start in the same state and receive the same input, their output and next state (which is confirmed by the subsequent behavior) should also be identical.

Our approach to determine if the testing replica conforms to reference replica consists in comparing the responses of the testing replica with the behavior model. To compare the responses, we resort to a *Correlation Table* that correlates the order in which the responses appear in the traces (which is the same for both replicas since the requests were sent in the same order) with the expected transitions of the behavior model and the protocol specification. Thus, in every state, the *nth* request/response of the reference replica should be identical to the *nth* request/response of the testing replica. If the behavior model rejects the respective messages from the testing replica, we have a violation and we mark the transition of the protocol specification. Marked transitions mean that the testing replica is not equivalent to the reference replica for that part of the protocol.

In the previous example, we use the order of the request/response (third column of Table II) to obtain the transition of the behavior model (second column) that is expected to accept the corresponding response of the testing trace (fourth column). If the transition rejects the response, which is basically a matter of comparing the response of the testing replica with the one from reference replica, it means that the testing replica is not fully complying with the behavior of the reference replica for the corresponding transition of the protocol specification (first column). For instance, the fifth response from the testing trace (“file1,file2”) is rejected by the transition t_{22}, which was expecting a message identical to the fifth response of the reference replica (“file2”). This violation indicates that the testing replica is not complying to the reference replica’s execution of that transition of the protocol specification, i.e., in the command “list”, thus we mark transition t_2 (column five).

In addition, protocol states that are only reached by marked transitions are also considered to be in violation in the testing replica. Therefore, we also mark any transitions of the protocol specification leaving these states.

The methodology is able to detect all syntax violations automatically. To find out semantic violations, one has to look at the violations to understand if the messages differ because of a semantic difference. For instance, the violations in transition t_5 of the protocol specification are related to the behavior model expecting a response of the type “Bye-bye” and getting the response “Good bye” from the testing replica (last row of Table II). This is just a syntax violation because messages are semantically expressing same thing—the state of both replicas is the same. On the other hand, the violation triggered in the 5th request/response shows that the behavior model was expecting an output of “file2” and it got instead a “file1,file2”. Here, not only is the text different, but it also indicates that both replicas have semantically different views at what should be the same state—one replica just sees file2, while the other sees the files file1 and file2. Hence, this is also a semantic violation.

The methodology can also discover nondeterminism issues in responses, such as dates, times, or random numbers. To find out this type of violations, we get additional versions of the testing trace from the same replica at a different date and time. Each version consists of the same sequence of protocol requests produced by the traffic generator and should also have the same sequence of responses. In this case, we add extra columns to the correlation table (one for each additional version of the responses of the testing replica) and compare the different versions. The comparison of responses from distinct trace versions allows the detection of discrepancies that are caused by nondeterminism. If the responses have some date or time field, or even a value derived from a random number or event, they will differ among the various versions.

The detected violations can then be resolved (i.e., eliminated at the source, normalized, or circumvented) by the IT architect or administrator, by looking at the form of violation, the message type, and the protocol state in which they appear.

B. The DiveInto tool

We developed a tool in Java, called DiveInto, that implements the methodology to detect inconsistencies among diverse replicas. The tool automatically discovers syntax violations and provides additional information to help a human operator to identify semantic violations and to devise ways to mitigate them. This section provides an overview of the implementation details, giving some focus to the traffic generator and the inference of the protocol specification.

The traffic generator is implemented by a specific scripted client for a given protocol. A scripted client is an automated program that performs a sequence of predetermined tasks in order to interact with a server in a deterministic way. For instance, we developed a scripted client that executes a sequence of FTP-related tasks on a server, from navigating in the remote file system to downloading a particular file. While the scripted client interacts with each replica, a packet capture tool is used to collect and store the entire communication.

Currently, we are focusing our research in text-based protocols from the IETF. Hence, we resort to a reverse engineering technique to infer an approximate specification of the protocol used by the replicas. DiveInto can derive automatically a protocol specification based only on the reference trace. For protocols that we are unable to infer at this moment, we have developed a graphical tool that supports the manual specification of the protocol, by creating a FSM that recognizes the protocol language and state machine.

DiveInto inference techniques have evolved from ReverX [10], a protocol reverse engineering tool that uses network traces to obtain an approximate protocol specification. First, the tool iteratively builds two FSMs to accept respectively every protocol request and response. Then, the tool analyzes each state of these FSMs to identify transitions that should be generalized in order to make the FSM accept the same types of messages with different values. If the ratio of the number of symbols recognized by a state over the total frequency of that state is above a certain threshold, the tool considers that part of the message as a parameter field. Transitions in the language FSM that are related to a parameter field are merged into a single transition with a regular expression that accepts any value of the same type (i.e., letters, digits, or special characters). This results in two language FSMs that express the different *message formats* recognized by the protocol—each path in one of these FSMs corresponds to a sequence of message fields.

DiveInto then infers the protocol state machine by creating a FSM that accepts the sequences of message formats observed in the reference trace (e.g., a login message, followed by a list message, and then by a quit message). First, the tool extracts individual application sessions from the trace (e.g., from the moment the client connects with the server until it disconnects). Then, it converts each protocol message into the corresponding message format (obtained from the language FSMs) and creates a new FSM that accepts every sequence of message formats. In particular, the tool builds a Mealy machine, so that each transition actually defines a protocol request and its respective response. Finally, DiveInto identifies and merges similar protocol states.

After obtaining the protocol specification (inferred or manually), DiveInto derives the behavior model from the reference trace and uses the testing trace to detect violations as described in the methodology. In the end, the tool produces a report with the transitions of the protocol specification that it marked with violations and asks the operator of the tool to identify semantic violations. For every marked transition, the tool shows two regular expressions, one created from the messages of the reference replica and another from the messages of the testing replica. This helps the operator to determine in which state the violation occurred and if there was a syntax or/and semantic violation. Listing 1 and Figure 4 in Section V shows an excerpt of the tool's output. This information is complemented with a trace of the requests that allowed the discovery of the violation, so that it can be reproduced if needed.

Table III
HARDWARE AND SOFTWARE CONFIGURATION.

OS	FTP	SMTP	POP
Fedora 14	wu-ftp	Sendmail	UW IMAP
Ubuntu server 10.04	Pure-FTPd	Postfix	dovecot
Windows XP SP2	IIS5	Surgemail	Surgemail
Windows Server 2003 SP2 Enterprise Edition	IIS6	Exchange Server 2003	Exchange Server 2003

V. Experimental Evaluation

To evaluate the methodology, we used DiveInto in three replication scenarios with the protocols: FTP, SMTP, and POP. These protocols are open standards from the IETF that have a large user base, but are quite difficult to replicate in an IT system. One of the challenges is that their specification is vague and undefined at times. For instance, most server replies contain a variable-length text field where the developers can further detail the reason of the response. Also, some protocol replies can be composed of an unspecified number of messages that must begin with the same reply code. This incompletely specified behavior creates an additional barrier when trying to implement a replication solution because the behavior of each replica must be directly comparable in order to detect intrusions.

The experimental procedure for each replication scenario follows our methodology: we generate and collect network traffic from a pair of replicas, which is used to derive the behavior model of the reference replica and to detect violations of the testing replica. In addition, the tool also infers an approximate protocol specification from the network traces of the reference replica. Each of the four replicas was evaluated both as a potential reference replica and as a testing replica.

A. Replication Scenarios

Each replication scenario simulates an IT system with four diverse replicas, with different OS and server implementations. Every replica is executed in a virtual image using VirtualBox[4] and was configured with the minimal hardware and software requirements to run the respective server (FTP, SMTP, or POP). Table III shows the various configurations of the experimental scenarios. Despite the different software,

[4]http://www.virtualbox.org/

Table IV
VIOLATIONS DETECTED ON THE DIFFERENT REPLICATION SCENARIOS.

(a) FTP servers.

Reference		IIS5	IIS6	Pure-FTPd	wu-ftp
IIS5	transitions	–	40	40	40
	syntax	–	4	38	20
	semantic	–	4	3	4
	nondeterm.	–	1	4	3
	rejected msgs	–	72/1580	858/1580	1133/2395
IIS6	transitions	40	–	40	40
	syntax	4	–	38	17
	semantic	4	–	1	1
	nondeterm.	1	–	4	3
	rejected msgs	72/1580	–	858/1580	1105/2395
Pure-FTPd	transitions	47	47	–	47
	syntax	45	45	–	45
	semantic	3	1	–	1
	nondeterm.	1	1	–	3
	rejected msgs	858/1580	858/1580	–	1717/2395
wu-ftp	transitions	38	38	38	–
	syntax	20	17	36	–
	semantic	4	1	1	–
	nondeterm.	1	1	5	–
	rejected msgs	318/1580	290/1580	902/1580	–

(b) SMTP servers.

Reference		Exchange	Postfix	Sendmail	Surgemail
Exchange	transitions	–	31	31	31
	syntax	–	31	31	31
	semantic	–	6	5	6
	nondeterm.	–	1	5	5
	rejected msgs	–	218/396	218/396	218/396
Postfix	transitions	25	–	25	25
	syntax	25	–	25	25
	semantic	6	–	3	2
	nondeterm.	2	–	2	5
	rejected msgs	218/396	–	218/396	218/396
Sendmail	transitions	33	33	–	33
	syntax	33	33	–	33
	semantic	5	3	–	3
	nondeterm.	7	1	–	5
	rejected msgs	218/396	218/396	–	218/396
Surgemail	transitions	29	29	29	–
	syntax	29	29	29	–
	semantic	6	2	3	–
	nondeterm.	2	1	2	–
	rejected msgs	218/396	218/396	218/396	–

(c) POP servers.

Reference		Dovecot	Exchange	Surgemail	Uw-imap
Dovecot	transitions	–	22	22	22
	syntax	–	20	21	21
	semantic	–	1	0	1
	nondeterm.	–	10	10	10
	rejected msgs	–	80/190	104/190	104/190
Exchange	transitions	23	–	23	23
	syntax	21	–	22	22
	semantic	1	–	1	0
	nondeterm.	12	–	12	12
	rejected msgs	80/190	–	104/190	104/190
Surgemail	transitions	22	22	–	22
	syntax	21	21	–	21
	semantic	0	1	–	1
	nondeterm.	6	6	–	6
	rejected msgs	104/190	104/190	–	104/190
Uw-imap	transitions	23	23	23	–
	syntax	22	22	22	–
	semantic	1	0	1	–
	nondeterm.	12	12	12	–
	rejected msgs	104/190	104/190	104/190	–

each replica was started with the same initial state, by configuring every server as similar as possible (same authentication scheme, user accounts, user files, email messages, etc.).

The first scenario was set with four diverse FTP server replicas. File Transfer Protocol (FTP) is a communication protocol for exchanging files between a client machine and a server [11]. This service allows clients to log into the server by connecting to a known port and by providing their credentials through the USER and PASS commands. Several other commands are then available to the clients allowing them to remotely manipulate files and directories on the server. The communication is typically done in two separate TCP connections, a control connection where the clients and server exchange commands and status replies, and a data connection that the server uses to send data through. The data connection is established by either the client (active mode) or the server (passive mode). The second scenario involved four replicas with different SMTP implementations. The Simple Mail Transfer Protocol (SMTP) is an IETF standard for outgoing electronic mail transport [12]. Email clients can send emails by connecting to the destination SMTP server or to a local or intermediary server that will relay the email message to the destination SMTP server. And finally, the third scenario corresponds to a replicated POP service. The Post Office Protocol (POP) is a standard protocol for retrieving email messages from a remote server [13]. Email clients can access their electronic mailbox by connecting to the POP server and by providing their credentials.

One scripted client was created for each replication scenario to generate the necessary traffic. The traffic generator scripts and the network traces can be found at http://homepages.lasige.di.fc.ul.pt/~jantunes/diveinto. The FTP scripted client executes 36 distinct FTP sessions using the most representative FTP commands, such as successful and unsuccessful login attempts, logouts, directory listings, traversing the remote file system, getting information from existing and non-existing files, and manipulating files and directories, which results in 354 FTP requests. To identify the FTP requests more likely of being used in a replicated system, we analyzed networks traces from 320 public FTP servers[5] and selected the 15 different FTP commands that accounted for more than 99% of the requests. To detect nondeterminism, we collected traces of each replica at two distinct dates and times, totaling 10,954 FTP packets.

The SMTP traffic generator creates 20 SMTP sessions with the most relevant commands, ranging from simple connections to the composition of malformed email messages, and includes the successful delivery of 7 emails. Unfortunately, we did not find any public traces with the full SMTP payload, so we analyzed the most common behavior of email clients

[5]http://ee.lbl.gov/anonymized-traces.html

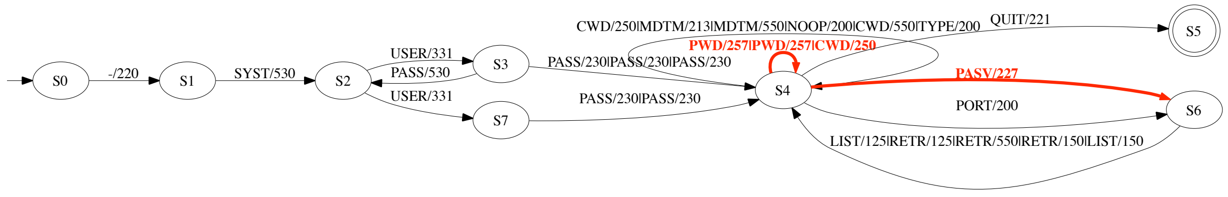
S0
-/220
S1
SYST/530
S2
USER/331
PASS/530
USER/331
S3
S7
PASS/230|PASS/230|PASS/230
PASS/230|PASS/230
CWD/250|MDTM/213|MDTM/550|NOOP/200|CWD/550|TYPE/200
PWD/257|PWD/257|CWD/250
QUIT/221
S4
S5
PASV/227
PORT/200
S6
LIST/125|RETR/125|RETR/550|RETR/150|LIST/150

Figure 4. Behavior model for FTP server IIS6 with the identified conformance violations from IIS5 (bold).

of our laboratory while interacting with SMTP servers. We found the most representative SMTP commands to be: HELO, EHLO, MAIL, RCPT, DATA, RSET, VRFY, EXPN, HELP, NOOP, and QUIT. As with the FTP scenario, SMTP traffic was collected from each replica at two different days to support the identification nondeterminism violations. A total of 1640 packets were obtained from the SMTP replicas.

The POP scripted client produces 14 different sessions employing the mostly used POP commands (10 out of 12 commands), taken from an similar analysis of the most common commands of email clients (e.g., retrieve, delete, list). Traffic data for each replica was collected at two different dates and times. The network traces accounted for a total of 900 POP packets.

B. Results

Table IV shows a summary of the violations identified by DiveInto between each pair of replicas in the three replication scenarios. Each server acts a reference replica in one subset of the experiments. The table provides values for the total number of distinct transitions of the inferred protocol specification (transitions row), and on how many of those transitions were marked with violations (syntax, semantic, nondeterministic rows). In addition, we provide the number of request/response messages that were rejected over the total count of messages that were exchanged between a pair of replicas (rejected msgs row).

The information about the number of rejected messages gives evidence of one of the benefits of our methodology. Without the methodology, an operator would normally have to look at each one of these messages to understand why and where they differ from the expected. For instance, the evaluation between the IIS5 server (reference replica) and the wu-ftp (testing replica) yielded 1133 different request/response messages between the replicas, which corresponds to 20 syntax violations. DiveInto automatically clusters the 2395 messages in the 40 inferred transitions, where it detects inconsistencies in 20 of them. This approach allows an operator to look at a manageable number of clusters of messages, which pertain to a single state and type of message, instead of hundreds of messages without any additional information.

Table IVa demonstrates that DiveInto found several violations in every pair of FTP servers. The tool correlates the identified violations with the state and transition of the protocol specification, and provides regular expressions to help a human operator to determine the existence of semantic

```
 1 State  S4;  Transition  PASV/227
 2    iis6 :  227 Entering  Passive  Mode  (10,10,5,163,4,.*) .\r\n
 3    iis5 :  227 Entering  Passive  Mode  (10,10,5,31,19,1.*) .\r\n
 4 (Non−determinism detected!)
 5
 6 State  S4;  Transition  CWD/250
 7    iis6 :  250 CWD command successful.\r\n
 8    iis5 :  550 /priv :  The system cannot  find  the  file   specified .\r\n
 9
10 State  S4;  Transition  PWD/257
11    iis6 :  257  "/"  is  current  directory .\r\n
12    iis5 :  257  "/.*"  is  current  directory .\r\n
13
14 State  S4;  Transition  PWD/257
15    iis6 :  257  "/p.*"  is  current  directory .\r\n
16    iis5 :  257  "/.*"  is  current  directory .\r\n
```

Listing 1. Detected violations between IIS6 and IIS5.

violations (and eventually to mitigate them). As an example, let us look more closely to a specific pair of FTP servers, the Internet Information Services version 6 (IIS6) and the Internet Information Services version 5 (IIS5). Figure 4 shows the IIS6 inferred protocol specification with the violation transitions (drawn in darker) that were found when comparing with the IIS5 replica's testing trace. As expected, DiveInto found most of the protocol execution to be identical because replicas were running similar server implementations. However, it detected some violations related with the FTP commands PASV, CWD, and PWD.

Listing 1 exhibits DiveInto's output when detecting the violations between IIS6 and IIS5. The majority of the discovered problems were syntax violations, mostly due to responses that conform to the transition function, but do not comply to the output function. This type of violations are typically easier to resolve, however, we will focus on semantic violations because are typically more difficult to analyze and to mitigate.

The tool found the first violation in State S4 (lines 1–4), for transitions that accept PASV requests and replies with response code 227[6]. Although the regular expressions created from the responses of both replicas (lines 2 and 3) show that they correspond to the same message type (i.e., 227), they still reveal different payloads and nondeterministic behavior. The PASV command is a special command to instruct the server to transfer data in *passive mode*. In this mode, the server replies with a free port from which it will send the data. Since the port number is not defined by the RFC specification, a server can choose any free port that with high probability will not be the same across the replicas. Therefore, this violation is

[6]For the sake of readability, the label of the transitions is trimmed to the first word of the request and response.

also a semantic violation because it affects the state of the server.

In the second violation (lines 6–8) the responses of each replica correspond to different types of messages (i.e., 250 and 550). Since both responses are syntactically and semantically different, the operator can safely determine that this is also a semantic violation. The remaining two problems are also semantic violations, although the type of the response messages is the same (response code 257). By inspecting the part of the messages in which they differ (i.e., "/" and "/.*"), one can conclude that they display different the path names, which implies conflicting states (e.g., some file or directory is missing or the current directory is different). As it turns out, the last three semantic violations (i.e., lines 6–16) are the result of the lack of the *user isolation* feature[7] present in IIS servers since version 6.

DiveInto also found problems on the other replication scenarios, some of them related to malformed protocol requests. For instance, several SMTP servers refused RCPT or MAIL requests with non-existent email addresses, while other servers accepted those requests. Sendmail was the only server to accept RCPT commands with an email address from unknown servers, and all but the Exchange Server rejected RCPT commands with an email address of an non-existent local user. Additionally, the Exchange Server was the only one to allow SMTP HELO/EHLO requests without the server address (response with 250 message code), and Dovecot and Surgemail recognized a POP USER command without the username parameter, while the other two servers did not.

An interesting nondeterminism violation was discovered in Pure-FTPd. DiveInto detected a syntax violation in one of the FTP commands that was due to an *easter egg* – a hidden message – introduced by the developers. The message randomly appears in replies to the RETR command: *"150-Connecting to port 5000\r\n150 The computer is your friend. Trust the computer\r\n226 file successfully transferred\r\n"*. This is an example of an unspecified behavior that could not have been detected by looking at the FTP protocol specification or by inspecting the configuration of the server.

VI. Related Work

Diversity is a reasonably well studied approach to build systems capable of handling (accidental) design faults (see [14] for a survey). Techniques, such as n-version programming [15], recovery blocks [16], and n-self-checking programming [17], use alternative implementations of the same component in order to identify errors and/or tolerate them. Diversity can also be provided in other forms, for example, by expressing inputs in a syntactically different but logically equivalent way, attempting to force disparate executions [18].

[7]FTP user isolation prevents users from viewing or overwriting other users' web content by restricting users to their own directories. Users cannot navigate higher up the directory tree because the top-level directory appears as the root of the FTP service.

In the area of security there has been much less research with diversity, but over the years people have argued about the possibility of its use [19], [20]. A few works have embraced this concept and prototyped it: an intrusion detection system was built by comparing the output of diverse COTS web servers [21]; an implementation of the HACQIT architecture was built to tolerate intrusions in a pair of diverse web servers [22].

Our work aims at supporting diversity in intrusion tolerance systems, in particular at detecting syntax and semantic violations between server implementations. Related to these ideas, conformance testing emerged from the need to check and ensure the compliance of a given implementation with a predefined specification [23]. Testing the conformance of an implementation can be carried out by applying special test sequences that traverse all transitions of the state machine of the specification and by comparing its output with the expected one. Other approaches use passive testing to extract a set of invariants from the specification and to check them against the observed traces of an implementation [24], [25], [26]. However, these works test the conformance of a given implementation with respect to a specification. Since the specification is an incompletely defined machine, several implementations can be developed in conformance with it and still behave differently (and are therefore incompatible to be used in an intrusion tolerant system).

Our tool uses network traces to infer a protocol specification of the servers. The problem of automata inference has been tackled in different research areas in the past, from natural languages to software component behavior [27], [28]. Typically, a prefix tree acceptor is first built from the training set, accepting all events. Then, similar states are merged according to their local behavior (e.g., states with the same transitions or states that accept the same k consecutive events) [28], [29]. Other approaches resort to taint analysis to obtain execution traces for each execution session, which are then used to build an acceptor machine [30]. Another approach by Trifilo et al. proposes a protocol reverse engineering solution that resorts to the statistical analysis of network traces [31].

VII. Limitations of Our Current Approach

Diversity aims at minimizing the common set of vulnerabilities between replicas, however, vulnerabilities or other types of faults can still exist and allow individual replicas to be compromised and to behave inconsistently. Our approach is not focused on detecting violations due to such problems, but rather due to differences between the implementations or configurations of the replicas while they process normal requests.

Additionally, DiveInto's capability to infer the protocol specification is bound by the limitations of the employed reverse engineering technique, and therefore, it currently lacks the support for some types of protocols, such as binary-based protocols.

VIII. Conclusion

Introducing software diversity in IT systems is a challenging and difficult problem because replicas must behave similarly and execute in a deterministic way. The paper presents a new methodology for the automatic discovery of inconsistencies among diverse replicas, and to assist on their correction. We also developed a tool, DiveInto, that is able to infer the behavior of each replica by analyzing network traces collected from their execution, which is then utilized to identify syntax and semantic violations. We used DiveInto to evaluate different server implementations in three replication scenarios (FTP, SMTP, and POP), where it was able to automatically detect a large number of violations.

Acknowledgements*:* This work was partially supported by the EC through project FP7-257475 (MASSIF) and by the FCT through the Multi-annual and the CMU-Portugal Programmes, and the project PTDC/EIA-EIA/100894/2008 (DIVERSE).

References

[1] P. E. Verissimo, N. Neves, and M. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems*. Springer-Verlag, 2003, pp. 3–36.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, 1978.

[3] F. B. Schneider, "Implementing faul-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[4] M. Correia, N. F. Neves, and P. Verissimo, "How to tolerate half less one byzantine nodes in practical distributed systems," in *Proc. of the IEEE Int. Symp. on Reliable Distributed Systems*, 2004, pp. 174–183.

[5] M. Castro and B. Liskov, "Practical Byzantine fault-tolerance and proactive recovery," *ACM TOCS*, vol. 20, no. 4, pp. 398–461, 2002.

[6] R. Kotla, L. Alvisi, M. Dahlina, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerant replication," in *Proc. of the Symp. on Operating Systems Principles*, 2007.

[7] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: a hybrid quorum protocol for byzantine fault tolerance," in *Proc. of the Symp. on Operating Systems Design and Implementation*, 2006, pp. 177–190.

[8] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," in *Proc. of the ACM Symp. on Operating Systems Principles*, 2005, pp. 59–74.

[9] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro, "OS diversity for intrusion tolerance: Myth or reality?" in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2011.

[10] J. Antunes, N. Neves, and P. Verissimo, "ReverX: Reverse engineering of protocols," Faculdade de Ciências da Universidade de Lisboa, Tech. Rep. TR-2011-01, 2010.

[11] J. Postel and J. Reynolds, "File transfer protocol," RFC 959, 1985.

[12] J. Klensin, "Simple Mail Transfer Protocol," RFC 5321 (Draft Standard), 2008.

[13] J. Myers and M. Rose, "Post Office Protocol - Version 3," RFC 1939 (Standard), 1996, updated by RFCs 1957, 2449.

[14] B. Littlewood, P. Popov, and L. Strigini, "Modeling software design diversity: a review," *ACM Computing Surveys*, vol. 33, no. 2, pp. 177–208, 2001.

[15] A. Avizienis, "The n-version approach to fault-tolerant software," *IEEE Trans. Softw. Eng.*, vol. 11, pp. 1491–1501, 1985.

[16] B. Randell, "System structure for software fault tolerance," in *Proc. of the Int. Conf. on Reliable Software*, 1975, pp. 437–449.

[17] J. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware-and software-fault-tolerant architectures," *IEEE Trans. on Computers*, vol. 23, no. 7, pp. 39–51, 1990.

[18] P. Ammann and J. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Trans. on Computers*, vol. 31, no. 4, pp. 418–425, 1988.

[19] Y. Deswarte, K. Kanoun, and J. Laprie, "Diversity against accidental and deliberate faults," in *Proc. of the Conf. on Computer Security, Dependability, and Assurance: From Needs to Solutions*, 1998, pp. 171–181.

[20] B. Littlewood and L. Strigini, "Redundancy and diversity in security," in *Proc. of the European Symp. on Research in Computer Security*, 2004, pp. 423–438.

[21] E. Totel, F. Majorczyk, and L. Me, "Cots diversity based intrusion detection and application to web servers," in *Recent Advances in Intrusion Detection*. Springer, 2006, pp. 43–62.

[22] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt, "The design and implementation of an intrusion tolerant system," in *Proc. of the Int. Conf. on Dependable Systems and Networks*, 2002, pp. 285–290.

[23] R. Lai, "A survey of communication protocol testing," *Journal of Systems and Software*, vol. 62, no. 1, pp. 21–46, 2002.

[24] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an extended finite state machine specification," *Information and Software Technology*, vol. 45, no. 12, pp. 837 – 852, 2003.

[25] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi, "A passive testing approach based on invariants: Application to the WAP," *Computer Networks*, vol. 48, no. 2, pp. 247 – 266, 2005.

[26] F. Zaidi, E. Bayse, and A. Cavalli, "Network protocol interoperability testing based on contextual signatures and passive testing," in *Proc. of the Symp. on Applied Computing*, 2009.

[27] C. de la Higuera, *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[28] A. Biermann and J. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE Trans. on Computers*, vol. 21, no. 6, pp. 592–597, 1972.

[29] D. Lo, L. Mariani, and M. Pezzè, "Automatic steering of behavioral model inference," in *Int. Symp. on Foundations of Software Engineering*, 2009, pp. 345–354.

[30] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *IEEE Security and Privacy*, 2009.

[31] A. Trifilò, S. Burschka, and E. Biersack, "Traffic to protocol reverse engineering," in *Proc. of the Int. Conf. on Computational Intelligence for Security and Defense Applications*, 2009.