ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **18**, Number 1, 2015, 54–68

IxFIZZ: Integrated Functional and Fuzz Testing Framework based on Sulley and SPIN

Lucian PETRICĂ[1,2], Laura VASILESCU[1],
Ana ION[3], Octavian RADU[3]

[1]University POLITEHNICA Bucharest
[2]IMT Bucharest
[3]Ixia LLC, IxNovation Department, Bucharest
E-mail: lucian.petrica@upb.ro

Abstract. Fuzzing has long been established as a way to automate negative testing of software components. While effective, existing fuzzing frameworks lack the necessary features to test stateful protocols in-depth. We propose using the modelling language Promela, and its interpreter SPIN, as an intuitive and generic way to describe protocol state machines, allowing the automatic generation of stateful fuzzing scripts for the popular Sulley fuzzing framework. Our approach involves the simulation of the Promela description in order for a set of valid protocol conversation sequences to be extracted. These sequences are then automatically modified by IxFIZZ, which inserts erroneous messages in the protocol conversation according to a set of heuristics. This approach also enables automatic analysis of test results against the protocol model and a tight integration of fuzzing with existing test-driven methodologies. We evaluated IxFIZZ against Linphone, a popular multi-platform SIP phone, to demonstrate the effectiveness of this approach, and compared the results to PROTOS, an established fuzzing framework for stateful network protocols. Our results indicate that IxFIZZ is able to detect more defects in the target software.

1. Introduction

Security and robustness are critical properties which can determine the success or failure of a given software application. It is a known fact that a significant portion of the time-to-market and resources of a given software project are spent testing the software before deployment [1]. The importance of testing is reflected in modern agile, test-driven development methodologies [2, 3], but these are sometimes impractical to implement because of the increased overhead of continually writing tests for each feature added [4]. In this context, it is essential to develop a testing methodology which is effective and does not add much complexity or time to the development cycle. Any such methodology must feature a seamless integration of functional and security testing.

A proven method for security testing is fuzzing [5]. When testing a software component, fuzzers generate pseudo-random traffic across the component interfaces, called trust boundaries, in the hope of finding unexpected sequences of data which trigger a fault. This type of testing is often called negative testing, and its effectiveness has been well documented, but the approach has been criticized for its shallow scope, since fuzzers are rarely able to test beyond the peripheral layers of the target application. This has led to the development of increasingly sophisticated fuzzers, which are able to conform to interface syntax and even learn communication patterns in an attempt to improve penetration rates [6, 7, 8, 9]. Despite any criticism, fuzzing remains a useful tool for security and robustness testing and an increasing number of software developers use fuzzing at one stage or another.

On the other end of the test spectrum is formal verification through model checking [10]. Model checking attempts to prove the correctness of the implementation of a given function in hardware or software by using a mathematical model of the desired functionality and of the implemented functionality. These models are then exhaustively checked for equivalence. SPIN [11] is one the most popular model checkers, along with its associated modelling language, Promela [12]. A system designer can specify a protocol in Promela and then check the protocol for unwanted properties such as deadlocks and non-progress cycles. The Promela description also serves as an executable golden model of the system which the designers can use when checking the functionality of the final implementation.

This paper presents an integrated test methodology which combines fuzzing with functional testing. We introduce IxFIZZ, a novel framework for automatic generation of fuzzing scripts from a functional description and a syntax specification of a system under test. IxFIZZ tests explore functional scenarios, fuzzing at each step along the way and ensuring compliance with the functional description as well as system robustness and security. This paper contributes to the state of the art in two ways. First, this is the first stateful protocol fuzzer which uses Promela and SPIN for state machine exploration of the target protocol and Sulley as a fuzzer backend. Although other work has been carried out on test generation using a model-checker, none has used a fuzzer backend for the generation of pseudo-random attacks on the protocol state machine. We consider Promela and SPIN to be the best choices among model checking alternatives, because of their open-source character and wide adoption in the

scientific community. Secondly, we present a novel approach for test result analysis based on the simulation of the device under test (DUT) model against a functional model of the actual fuzzer.

2. Related Work

Fuzzing is an active research field and numerous fuzzers have been proposed with varying strengths and weaknesses and tailored to fuzzing specific types of software systems. Open-source fuzzing frameworks generally target network protocols and sometimes file interfaces. Among these are GPF, Peach, Antiparser, Autodafe [13, 14, 15, 16] and other fuzzers which present the user with a custom fuzzer development language used to describe the tested protocols. SPIKE [17] is one of the most popular open-source network fuzzers because of its block-based approach to constructing protocol data units. This approach allows the developer to split the protocol data unit (PDU) into blocks which are then handled by SPIKE's own fuzzing heuristics. This description method along with a helper API in C make for a powerful fuzzer, but SPIKE lacks features for describing stateful protocols.

Sulley [18] is a network protocol fuzzer developed in the spirit of SPIKE. Sulley exposes a Python API for describing protocol data units in a block-based fashion and handles transformations of these blocks automatically. While still lacking explicit support for fuzzing stateful protocols, Sulley does offer a framework to link several requests together in a session and allows callback functions and user-specified data encoders to modify the fuzzing process. Sulley also suffers from its static session construction algorithm, which does not allow the developer to pick which request to send based on previous replies from the DUT.

Several fuzzers have focused specifically on stateful network protocols. The most established of these has been developed as part of PROTOS [6], a security testing project at the University of Oulu. PROTOS uses a complex system of Java and TCL descriptions to model a protocol and extract a test scenario. Fuzzing transformations are applied to trigger specific vulnerabilities such as buffer overflows, and DUT aliveness is tested by following up the fuzzed request with a valid request for which the normal reply is known. The extent to which stateful fuzzers can be tested with the PROTOS fuzzer is unclear, since the framwork was not released to the public. The released test-cases are simplistic and the description method appears to be overly complex.

Another stateful fuzzer is SNOOZE [7], which also makes use of a method for functional description of a protocol in XML. The custom language used by SNOOZE models the PDU as well as the protocol state machine. SNOOZE also exposes a Python API which allows the developer to query the protocol description in order to extract valid messages and replies. The developer uses this API to manually construct a test scenario consisting of send and receive events, and to guide the fuzzing of primitives within sent PDUs. SNOOZE is notable for its unified description language which allows complete specification of a protocol. Its main drawback is the requirement that a test scenario be written by hand, leaving protocol functional

coverage up to the imagination and skill of the scenario developer. SNOOZE also uses a single type of attack to test the state machine.

A different approach to stateful protocol fuzzing is KiF [8], which does not use a protocol description but instead relies on captured traffic, which is then modified and replayed to the DUT. While this approach has been used by other fuzzers, AutoFuzz [9] being the most notable, KiF adds fuzzing primitives specifically suitable for state machine testing, such as message reordering, repeating and other attacks. AutoFuzz and KiF alike rely on capturing a complete state machine diagram from protocol messages sent and received by the DUT in normal conversations. This approach is very unlikely to uncover corner cases and cannot test a protocol implementation for completeness. KiF is also restricted to fuzzing the SIP protocol [19].

Recent research has gone into automatic generation of test scripts from functional simulation in SPIN. In [20], the authors demonstrate a framework for generating functional tests in the TTCN-3 scripting language [21] for network protocol security testing. A target system is first described in Promela and simulated in SPIN, and the test generator uses counter-examples generated by the functional simulation to create scripts in TTCN-3 for directed testing of the network protocol implementation. We consider this approach to be promising, but limiting testing to only those cases where the protocol model fails functional tests may miss the majority of software defects which occur because of a bad implementation of a good protocol model. The work in [22] describes a method for the generation of test scripts from a software model, through model analysis and simulation. The generated test scripts are able to drop protocol messages or insert valid messages in order to simulate an attacker, and monitor the status of protocol agents. The scripts do not involve any mechanism for the controlled modification of message fields in order to explore the robustness of the protocol implementation to this type of attack.

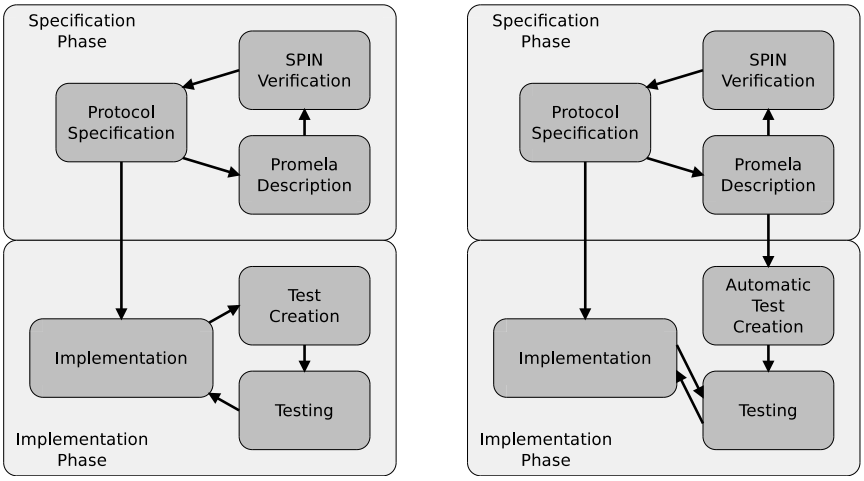
3. IxFIZZ Motivation and Concept

The main objectives of IxFIZZ are to integrate fuzzing and functional testing, to automate stateful fuzzer generation and to provide a method for automatic test analysis. Integration of security and functional testing is important from a productivity point of view, and an automatic test generator enables testing of a software component much earlier in the design cycle, improving time-to-market. Figure 1a illustrates an example of a traditional design cycle for a software system, whereby a protocol specification is developed, a formal model for the system is written in a modelling language and a model checker is used to verify the validity of the functional specification. When the specification is considered sufficiently mature, work on the implementation begins. Tests are usually written for individual components while they are being developed, and called unit tests. When all components are complete and pass unit testing, the system is assembled and the design is tested as a whole. Implementation may, generally, start before the specification is completed, but this requires that both the implementation and the tests be updated every time the specification is amended.

Unit tests are usually written by the software developers themselves while system-

level testing may be handled by quality assurance engineers. Having the developer write tests for his own code is advantageous because the developer has inside information on where complexity is concentrated and where problems may lie. However, systematic problems may occur such as the developer and quality assurance engineer understanding the specification differently, which is an issue that cannot be solved by unit testing. Another problem is that a defect may require a complex set of circumstances to occur, which the developer fails to simulate in their tests, but which persistent attackers eventually identify and exploit. Unit testing at function or class level has been made more efficient by the advent of unit testing frameworks [23]. We consider that there is a need for a testing framework which is specification-aware, mostly automatic and can generate unit tests at the protocol agent level.

The proposed improvement to the traditional design cycle is to automatically generate test scripts from the modelling language description of the system, as Fig. 1b illustrates. These can augment or even replace manually written test cases, and are available immediately after the specification is amended. Thus, the programmer is free to work on the actual protocol implementation. Furthermore, tests are guaranteed to comply to the protocol specification.



(a) Manual Test Creation (b) Automatic Test Creation

Fig. 1. Typical Design Phases.

4. IxFIZZ Implementation

We propose a test framework based on Promela and SPIN. The open-source Sulley fuzzing framework is used as a back-end for its fuzzing heuristics and test harness capabilities. Figure 2 presents the data flow diagram of the proposed fuzzer framework. A set of files represent the protocol description, comprising of the PDU description and the executable state machine description. These are inputs to the fuzzing script

generator, which makes use of the state machine simulator. The generator outputs a fuzzing script and an executable description of the fuzzer. The fuzzing script is then executed against the system under test, and its executable description is also simulated. The analysis block compares real and simulated conversations between the system under test and the fuzzer and reports mismatches, which may signal faults in the protocol implementation.

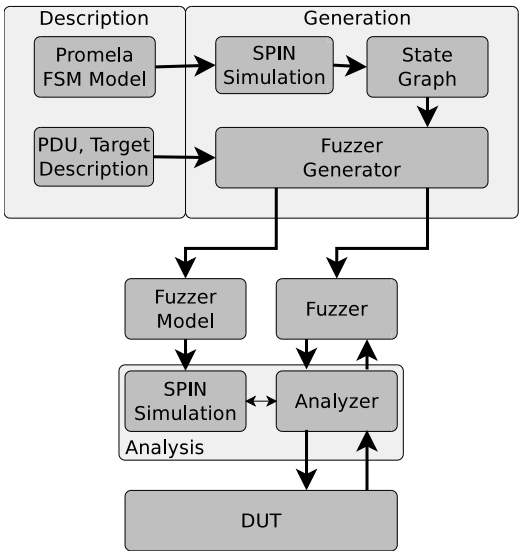


Fig. 2. IxFIZZ Architecture.

4.1. Protocol Description

Like PROTOS and SNOOZE, we choose to start from a protocol state machine description, which IxFIZZ uses to extract test scenarios, the protocol state machine, to generate fuzzing scripts and to analyze test results. There are several languages used for modeling systems and protocols. Graphical formal modelling languages include SDL [24] and UML [25]. Non-graphical modelling languages include Promela [12] and Microsoft's AsmL [26]. We chose Promela as a modelling language because it is free, open-source and has a mature interpreter, SPIN. Promela and SPIN have already been used in [20] as a base for building a functional test generator.

The Promela description must adhere to a set of structural rules, as well as contain the above-mentioned meta-comments, in order to properly link up with the PDU description. First, each protocol agent must be represented by a Promela process. Communication between processes must occur through channels, and each channel may transfer a single type of message. If, for example, there are two types of messages, a request and a response, then a channel must be declared to carry requests and a separate channel to carry responses. This set-up is sufficiently expressive to model most message-passing protocols, and therefore most network protocols.

Promela messages are symbolic representations of the protocol data unit. Modelling the whole PDU in Promela would be impractical and would provide little benefit. Instead, the Promela messages consist only of those PDU fields which carry meaning for the protocol state machine. Figure 3 illustrates such a set-up. The code is extracted from a Promela description for the SIP protocol. The description is minimal and contains two protocol agents, a User Agent Client (UAC) and a User Agent Server (UAS). Each is described using a Promela process, and the two communicate using the channels depicted in the figure. The UAC only issues SIP requests, and the UAS only issues SIP responses, so only two channels are required for communication. Channel names must be in the form *SenderProcessReceiverProcess_PDUTYPE* in order for the fuzzer generator to properly process the description.

```
/* SIP Channels */
/* SName {method, cseqv, cseqm, callid} */
chan p1p2_REQUEST = [1] of {mtype, int, mtype, int};
/* SName {code, cseqv, cseqm, callid} */
chan p1p2_RESPONSE = [1] of {int, int, mtype, int};
```

Fig. 3. Promela Metacomments.

```
s_initialize("REQUEST")
...
s_string(method, fuzzable=True)
...
s_int(callid, format="ascii", fuzzable=True)
...
s_int(cseqv, format="ascii", fuzzable=True)
...
s_string(cseqm, fuzzable=True)


s_initialize("RESPONSE")
...
s_int(code, fuzzable=True)
...
s_int(callid, format="ascii", fuzzable=True)
...
s_int(cseqv, format="ascii", fuzzable=True)
...
s_string(cseqm, fuzzable=True)
```

Fig. 4. Sulley PDU Description.

The actual protocol data units must be described in the Sulley blocks Python API. Figure 4 lists selections of the corresponding Sulley blocks description for the protocol PDU. Sulley blocks begin with *s_initialize("PDUTYPE")* which is a command to specify that we are starting a block describing a PDU of a given type. The block itself consists of a series of statements, each describing a field in the PDU. Fields can be of numeric or string types, and each can have several attributes: a value or name,

an encoding type and a third parameter which specifies if Sulley may apply its fuzzing heuristics on the field. Fields relevant to the protocol state machine are specified with a name instead of a value. The encoding type is protocol-dependent.

The linkage between the PDU description in Sulley blocks and the FSM description in Promela is done through a set of meta-comments which specify how messages exchanged through channels by Promela are mapped to actual fields in the PDU. The meta-comment is signalled by the presence of the keyword *SName*, followed by a list of ordered PDU field names. These names map one to one with the components of the message relayed in the channel declared on the line immediately following the meta-comment.

Finally, a test harness description in the Sulley API specifies the IP address of the target protocol agent and fuzzer, the ports, and the transport protocol utilized for communication. This harness description is presented in Fig. 5. Sulley currently supports only TCP and UDP as transport protocols, but transport over HTTP or other higher-level protocols may also be implemented.

```
sess = sessions.session(proto="udp", bind=("192.168.1.135",5060))
target = sessions.target("192.168.1.118",5060)
sess.add_target(target)
```

Fig. 5. Sulley Target Description.

4.2. Fuzzer Generation

Fuzzer generation focuses on each protocol agent in turn. We call the agent currently under focus the test target. The generator creates a Python test script which simulates the environment in which the target is designed to function, including all protocol agents which exchange protocol messages with the target. To gather information on the nature and structure of the conversation between the target and the protocol environment, the generator simulates the Promela model and records all messages passed along channels to and from the target process. The messages are utilized to construct the protocol conversation state graph of the target protocol agent, as Fig. 6 illustrates. In this graph there are two types of states: send states Si which emit a protocol message and receive states Ri which absorb a message.

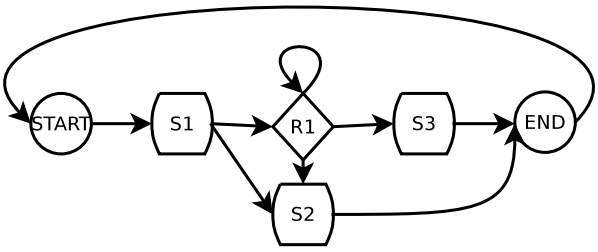


Fig. 6. Conversation State Graph.

Promela is a non-deterministic language so the simulation behaviour depends on the initial seed of the random number generator in SPIN. The FSM model is simulated extensively by repeatedly running SPIN with different initial seeds. Figure 7 represents sample output of the SPIN simulation. The fuzzer generator utilizes the graph and PDU description to generate fuzzing scripts, which consist of Sulley sessions and blocks, as well as callback functions and encoders required to set FSM-related fields and do graph traversal. The fuzzer generator outputs the fuzzer itself as a Python script as well as a fuzzer model in Promela.

```
proc  1 (UAC) sip.pml:24    Send  OPTIONS,20,OPTIONS,1  -> queue 1  (p1p2_REQUEST)
proc  2 (UAS) sip.pml:97    Recv  OPTIONS,20,OPTIONS,1  <- queue 1  (p1p2_REQUEST)
proc  2 (UAS) sip.pml:106   Send  200,20,OPTIONS,1      -> queue 2  (p2p1_RESPONSE)
proc  1 (UAC) sip.pml:59    Send  CANCEL,20,OPTIONS,1   -> queue 1  (p1p2_REQUEST)
proc  2 (UAS) sip.pml:142   Recv  CANCEL,20,OPTIONS,1   <- queue 1  (p1p2_REQUEST)
proc  1 (UAC) sip.pml:82    Recv  200,20,OPTIONS,1      <- queue 2  (p2p1_RESPONSE)
proc  2 (UAS) sip.pml:129   Send  400,20,OPTIONS,1      -> queue 2  (p2p1_RESPONSE)
```

Fig. 7. Sample SPIN Output.

4.2.1. Stateful Fuzzing Attacks

When fuzzing fields in a PDU, a fuzzing framework like Sulley utilizes type-dependent heuristics to generate relevant attacks. For example, Sulley attacks integer fields by loading into them the maximum value, zero or other corner-case values. String fields, on the other hand, are attacked by e.g., long strings, malformed ASCII, insertion of metacharacters. Fuzzing the state machine, however, requires a different set of attacks which are most likely to trigger a fault in the protocol implementation. While traversing the state graph, the fuzzer may choose to apply any one of the available attacks in each state. We identify the following minimal set of state machine attacks, which our fuzzer must implement:

Skip Receive This attack skips the receive state without waiting for a reply from the DUT. The expected reply may be en-route, in which case the attack may trigger a race condition.

Skip Send This attack skips emitting a message in a sequence and violates FSM ordering. All subsequent emitted messages are unexpected at the DUT and should be handled accordingly.

Random Fields When this attack is selected for the currently emitted message, the PDU fields related to state machine functionality, as specified in the Promela model metacomments, randomly take values within the allowable ranges specified in the model. That is, integer fields may randomly assume any value in their range and strings may assume one of the Promela mtype values. This attack generates a malformed message which should be detected correctly by the DUT.

Replay Resend a sequence of one or more PDUs which were previously sent to the DUT. This attack tests how the implementation handles resent messages as well as possibly triggering a race condition.

Fast Forward This attack suspends transmission of messages and instead stores them in a queue. Subsequent additions of messages to the queue triggers all available messages to be transmitted in rapid succession. This tests if the DUT is able to handle fast sequences of messages correctly.

4.2.2. Graph Traversal

The fuzzer scripts have to navigate the conversation state graph in order to mimic valid functionality and fuzz deep within the protocol, applying the above-mentioned state machine attacks. The attacks fall into two categories, which are emission attacks like Fast Forward and Random Fields, and traversal attacks. The traversal attacks can be implemented by adding supplementary edges to the original graph, representing functionally illegal (fuzz) transitions. The sole restriction is that such a transition must not end in a receive state. Emission attacks and fuzz transitions have associated counters which decrement each time the transition is taken or the attack is applied, thus ensuring the test completes in finite time. Legal transitions do not have an associated counter but must be taken at least once.

The traversal algorithm is the following. If it is in a send state, IxFIZZ emits the required PDU, randomly choosing to apply a PDU attack or not. If an attack is applied, the corresponding counter is decremented. Emission attacks can no longer be selected after their associated counter reaches zero. After PDU emission, IxFIZZ transitions to the next state. If transitions to multiple states are possible, one is chosen randomly. In a receive state, IxFIZZ waits until it receives a specified message and proceeds to the next state accordingly. Each time a fuzz transition is taken, its counter is decremented. If the counter reaches zero, the transition is deleted. When all attack counters in each state reach zero, and no more fuzz transitions are present, IxFIZZ does one more pass over the conversation graph and declares the test complete. Figure 6 presents an example conversation state graph. Its corresponding fuzzed version, with fuzz transitions represented in dotted lines, is presented in Fig. 8.

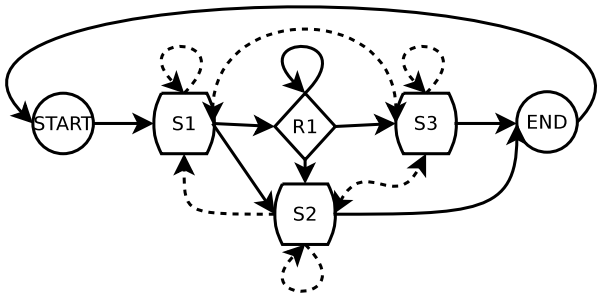


Fig. 8. Fuzzed Conversation State Graph.

4.3. Analysis Engine

One of the goals of IxFIZZ is to automatically analyze DUT behaviour to identify functional mismatches between the specification and the implementation. The specified behaviour of the fuzzing target is already known as it is encoded in the Promela process which corresponds to the target. In order to simulate the whole test environment, the fuzzer generator outputs a Promela process which models the fuzzer itself, and connects it to the target process, as illustrated by Fig. 9. Here a protocol system consisting of four agents is originally modelled in Promela. Process D is the test target, so the fuzzer generation engine outputs a Promela process which can receive and transmit messages through all channels connected to process D.

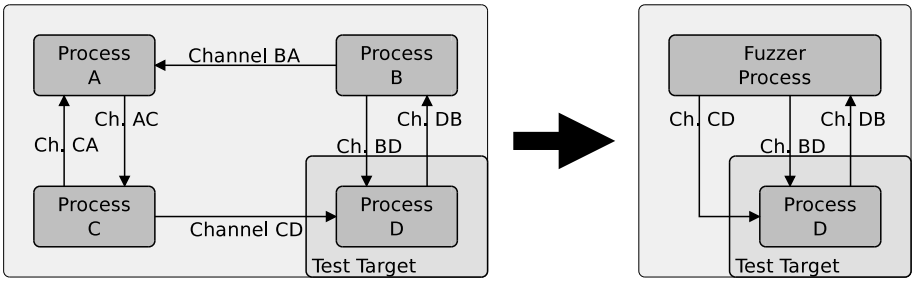


Fig. 9. Construction of the Analysis Model.

The analyzer is an adaptation layer between the fuzzer, the DUT and the simulation. It inspects packets exchanged between the fuzzer and the DUT during the testing process. The simulation behaviour of the system is then compared with actual conversations between the fuzzer and the DUT. The analyzer reports any mismatch to the user. Identified problems can either be caused by fuzzer action or can be functional mismatches between the protocol specification and the implementation. The analysis engine is essential for detecting subtle faults which do not cause the DUT to crash or hang.

5. Evaluation

We evaluated the proposed fuzzing framework by testing an implementation of the Session Initiation Protocol. We chose SIP because it is widely-used in Voice-Over-IP services, and is a stateful protocol. Secondly, there has already been research on fuzzers for SIP, which have been discussed in Section 2. The PROTOS fuzzing suite serves as a benchmark against which to measure the proposed fuzzer.

The SIP implementation under test is Linphone, a free and open-source soft phone. Linphone was chosen because of its popularity and multi-platform nature. We tested two Linux versions of Linphone, 3.3.2 and 3.5.2, which were at the time of testing the default version in Ubuntu 11.10 Linux and the latest Linphone version, respectively.

A protocol description for SIP was written from RFC3261 [19]. The description was partially modeled from previous work on SIP model checking in Promela [27]. The

Augmented Backus-Naur Form (ABNF) [28] description of the protocol syntax, as well as SIP packets captured with Wireshark [29] were utilized to extract PDU structure and write the Sulley blocks. Default values for PDU fields were manually inserted into the blocks. The Promela description of the SIP state machine was written taking into account mainly "MUST" and "MUST NOT" clauses in the RFC. Clauses like "SHOULD" were ignored, and "MAY" clauses were selectively implemented. This approach was chosen because we wanted the generated fuzzer to be constrained as little as possible, thereby increasing the likelihood of finding bugs in any particular implementation of the protocol.

We implemented a minimal set of SIP functionality: the INVITE dialog, which includes ACK, BYE and CANCEL requests, and two session-independent requests, OPTIONS and INFO. Also, system structure was minimal, comprising only of one UAC and one UAS. The description comprises about 100 lines of code, excluding comments. The fuzzer generator was utilized to generate Sulley scripts for testing the UAS. It found seven unique state-machine traversal scenarios, which together cover the full functionality of the modelled SIP system. The resulting conversation state graph is presented in Fig. 10. The fuzzer script was executed against each Linphone version, with a total of over ten thousand individual test cases that cover both parser and state machine attacks.

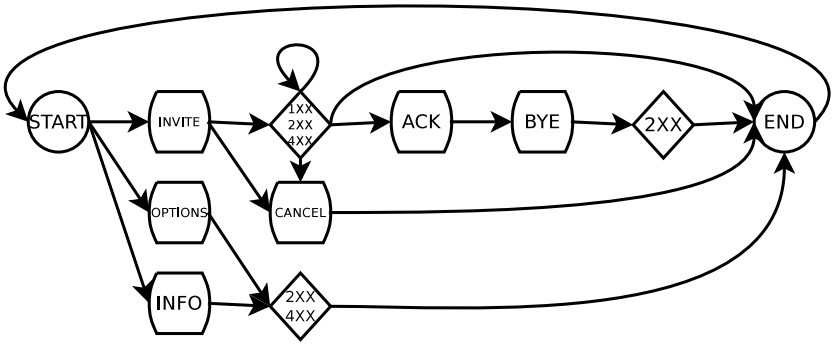


Fig. 10. SIP Conversation Graph.

The test was carried out on two Ubuntu 11.10 virtual machines connected together through a virtual network. The generated fuzzing script was executed on one machine and Linphone acted as a SIP UAS on the second machine. Both Linphone versions were started in auto-answer mode, to provide the stateful fuzzer with an opportunity to test DUT behaviour beyond the call request phase. During testing we found several issues, listed in Table 1. We identified multiple defects in the Linphone parser, as well as three state machine bugs. All of the parser bugs consisted of segmentation faults caused by improper syntax in the SIP PDU. Most commonly, crashes were caused by lacking CRLFs between headers and also malformed header names. Header names were also found to be vulnerable to buffer overflow attacks. The SIP version number in the request line was also vulnerable to attack. All bugs were found on both Linphone versions under test and have been submitted to the developers. Some of the parser bugs were also found by testing both Linphone versions against the PROTOS fuzzer.

Table 1. Identified Linphone Defects

Type	Trigger
Segmentation Fault	Malformed Request-URI
	Replace space before SIP Version Number with semicolon
	Missing SIP Version Number
	Missing CRLF before Via and From headers
	Replace header name with long pattern
	Missing colon after Via and From header names
	CR immediately after Via and From header names
	INFO request with Cseq method field set to INVITE
Unresponsiveness	Repeated INVITE and CANCEL sequences
Unrequested Response	Malformed Request-URI

State machine tests against the SIP state machine in Linphone yielded a lower number of defects, but we nevertheless found several issues, including one which caused Linphone to crash. The first was a potential information leak vulnerability triggered by invalid URIs in the SIP invite request, which caused Linphone to send data to a random port on the UAC. This data was non-null and it did not consist of a valid SIP reply, and therefore we must conclude an unwanted information leak occurred. Another issue was a denial-of-service condition in Linphone 3.3.2, caused by repeaded invites and canceling of invites. After a number of such conversation sequences, Linphone 3.3.2 stops sending provisional responses to invites, which leaves the UAC unable to determine the liveness status of Linphone. The third issue is a crash caused by the transmission of a malformed INFO request from the UAC while Linphone is hanging up an existing call. The INFO request has the PDU field *cseq method* set to "INVITE". Both Linphone versions were affected by this bug but we found that the specific timing of requests influences Linphone's vulnerability to this bug. Linphone 3.5.2 was relatively more resistant. These issues were not detected by the PROTOS fuzzing suite and have also been reported to the developers.

As we can see, not all the defects we found caused crashes. Arguably the most potential for a hacker exploit lies with the data leak vulnerability, which did not cause a crash and did not interfere with the otherwise normal functionality of Linphone. Being able to monitor traffic between the UAS and UAC and compare it to traffic generated by the executable model in SPIN allowed us to detect such subtle functional faults. In this respect we have improved on previous work such as the PROTOS fuzzer, which was only able to trigger parser faults.

6. Conclusion and Future Work

In this paper we have demonstrated IxFIZZ, a fuzzer development environment which utilizes Promela for the specification of protocol functionality and a combination of SPIN and Sulley for the automatic generation of fuzzing scripts and analysis of results. The system under test is described in Promela and must conform to a set of structural rules and utilize metacomments in order to enable the fuzzer generator

to link up the state machine description with the PDU description. Each protocol agent is described as a Promela process and the generator can focus on any of the protocol agents as the target of the fuzzer generation process.

We have also expressed a number of state machine attacks and formulated a fuzzing algorithm to implement said attacks. The evaluation results demonstrate the effectiveness of our approach at triggering faults in the selected target. The automatically generated fuzzing scripts were able to exercise the target protocol agent, which was Linphone functioning as a SIP UAS on Ubuntu Linux. Several software bugs were identified. Some of these, related to PDU structure, were also identified by PROTOS. Other bugs were only identified through the use of stateful fuzzing heuristics introduced by IxFIZZ. One of the bugs identified did not lead to a crash of Linphone, but was a data leak issue spotted by the automatic test analysis engine.

Future work should focus on eliminating the task of manually writing Sulley blocks to describe protocol PDUs. This process is time-consuming and could be replaced by a method of automatic generation of PDU descriptions from, for example, ABNF descriptions. The SIP specification does come with ABNF PDU descriptions, and ABNF is becoming more popular as a way of describing protocol syntax. The automatic generation of PDU Sulley blocks would reduce test turn-around time in those situations where the PDU structure is under development and frequently modified.

Acknowledgement. This work has been jointly funded by Ixia LLC and the Sectoral Operational Programme Human Resources Development 2007- 2013 of the Romanian Ministry of European Funds through the Financial Agreement POSDRU/159/1.5/S/132397.

References

[1] MYERS G., SANDLER C., BADGETT T., *The Art of Software Testing, 3rd Edition*, Wiley, 2011.

[2] BECK K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.

[3] BECK K., *Test-Driven Development: By Example*, Addison-Wesley, 2002.

[4] GEORGE B., WILLIAMS L., *A Structured Experiment of Test-Driven Development*, Information and Software Technology, Volume **46**, Issue 5, pp. 337–342, 2004.

[5] MILLER B.P., FREDRIKSEN L., SO B., *An Empirical Study of the Reliability of UNIX Utilities*, Communication of the ACM, Volume **33**, Number 12, pp. 32–44, 1990.

[6] WIESER C., LAAKSO M., SCHULZRINNE H., *Security testing of SIP implementations*, Technical Report, Columbia University, Department of Computer Science, 2003.

[7] BANKS G., COVA M., FELMETSGER V., ALMEROTH K., KEMMERER R., VIGNA G., *SNOOZE: Toward a Stateful NetwOrk prOtocol fuzZEr*, Information Security, pp. 343–358, 2006.

[8] ABDELNUR A., STATE R., FESTOR O., *KiF: A Stateful SIP Fuzzer*, Proceedings of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications, pp. 47–56, 2007.

[9] GORBUNOV S., ROSENBLOOM A., *AutoFuzz: Automated Network Protocol Fuzzing*, International Journal of Computer Science and Network Security, Volume **10**, 2010.

[10] BERARD B., BIDOIT M., FINKEL A., LAROUSSINIE F., PETIT A., PETRUCCI L., SCHNOEBELEN P., *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer, 2010.

[11] HOLZMANN G.J., *The Model Checker SPIN*, IEEE Transactions on Software Engineering, Volume **23**, Number 5, pp. 279–295, 1997.

[12] HOLZMANN G.J., *Design and Validation of Computer Protocols*, Prentice Hall, 1991.

[13] General Purpose Fuzzer, URL: `www.vdalabs.com/tools/efs_gpf.html`

[14] Peach Fuzzer, URL: `www.peachfuzzer.com`

[15] Antiparser Fuzzer, URL: `antiparser.sourceforge.net`

[16] Autodafe Fuzzer, URL: `autodafe.sourceforge.net`

[17] AITEL D., *An Introduction to SPIKE, the Fuzzer Creation Kit*, Immunity Inc. White Paper, 2004.

[18] AMINI P., PORTNOY A., *Sulley Fuzzing Framework*, (`www.fuzzing.org/wp-content/SulleyManual.pdf`), 2010.

[19] ROSENBERG J., SCHULZRINNE H., CAMARILLO G., JOHNSTON A., PETERSON J., SPARKS R., HANDLEY M., SCHOOLER E., *SIP: Session Initiation Protocol*, IETF Network Working Group Request for Comments 3261, 2002.

[20] ZHOU L., YIN X., WANG Z., *Protocol Security Testing with SPIN and TTCN-3*, Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation, pp. 511–519, 2011.

[21] GRABOWSKI J., HOGREFE D., RETHY G., SCHIEFERDECKER I., WILES A., WILLCOCK C., *An Introduction to the Testing and Test Control Notation (TTCN-3)*, Computer Networks, Volume **42**, Number 3, pp. 375–403, 2003.

[22] JURJENS J., *Model-based security testing using UMLsec: A case study*, Electronic Notes in Theoretical Computer Science, Volume **220**, Number 1, pp. 93–104, 2008.

[23] List of Unit Testing Frameworks, URL: `http://en.wikipedia.org/wiki/Unit_test`, Accessed 7/15/2012.

[24] ELLSBERGER J., HOGREFE D., SARMA A., *SDL: Formal Object-Oriented Language for Communicating Systems*, Prentice Hall, 1997.

[25] FRANCE R., EVANS A., LANO K., RUMPE B., *The UML as a Formal Modeling Notation*, Computer Standards and Interfaces, Volume **19**, Number 7, pp. 325–334, 1998.

[26] GUREVICH Y., ROSSMAN B., SCHULTE W., *Semantic Essence of AsmL*, Formal Methods for Components and Objects, Springer Lecture Notes in Computer Science, Volume **3188**, pp. 240–259, 2004.

[27] ZAVE P., *Understanding SIP through Model-Checking*, Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks, Springer Lecture Notes in Computer Science, Volume **5310**, pp. 256–279, 2008.

[28] Augmented Backus-Naur Form, URL: `http://www.ietf.org/rfc/rfc2234.txt`

[29] Wireshark Network Protocol Analyzer, URL: `www.wireshark.org`