# In-depth fuzz testing of IKE implementations

**Author(s):**
Tsankov, Petar; Torabi Dashti, Mohammad; Basin, David A.

**Publication Date:**
2011

**Permanent Link:**
https://doi.org/10.3929/ethz-a-007313874 →

# In-depth fuzz testing of IKE implementations

Petar Tsankov, Mohammad Torabi Dashti, David Basin
Institute of Information Security, ETH Zürich

December 23, 2011

## 1 Introduction

The correctness of security protocols can be proved using formal models. Typically, a security protocol is proven correct at a high level of abstraction, hiding away the implementation details. To execute the protocol on a real system, software engineers take up the task to implement the protocol. Programming is an error-prone activity: despite the fact that a security protocol is proven correct, its implementation may have vulnerabilities, which can be exploited by attackers. In this work we consider the problem of testing security protocol implementations for vulnerabilities, assuming that a formal model of the protocol is given. Finding security vulnerabilities is nontrivial. Even if a system correctly implements the model and behaves as expected when it performs intended tasks, i.e. tasks that are specified in the model, it is difficult to check that the implementation does not exhibit any additional behaviors, i.e. behaviors that are not specified in the model. Behaviors that are unintentionally implemented in the system can be dangerous and may introduce vulnerabilities in the system.

When programmers implement the protocol, they need to consider how the system handles unexpected events, e.g. how the system reacts when the disk is full or how to handle malformed input. Such unexpected events are the low-level details that are typically abstracted away in the formal model. We can virtually split the program into two parts: (1) *functional part* which handles the expected behavior of the system, and (2) *error handling part* which implements how the program handles unexpected events. The top 4 most dangerous software errors listed at the Common Weakness Enumeration database[1] are due to lack of or improper handling of unexpected inputs [6]. Typically, the functional part of the program is well specified in the model and it is less likely that the programmers will implement it incorrectly. We believe that due to lack of specification on how the implementation

[1]`http://cwe.mitre.org`

should handle unexpected events, the error handling part of the program is often implemented in an ad-hoc manner or it may not be implemented at all. Moreover, when the implementation is handed to the users, faults in the functional part of the program can be quickly revealed; the users serve as testers reporting any observed system failures. Conversely, trivial faults in the error handling part of the code may remain undetected for a long time, simply because the faulty code is executed infrequently.

In model based testing an implementation under test is tested for compliance with a model that describes the required behavior of the implementation [16]. That is, the implementation is executed against inputs specified in the model and then the system's behavior is compared to the expected behavior specified in the model. In contrast to model based testing, we propose a testing method that checks the system's behavior when executed against unexpected inputs, i.e. inputs that are not specified in the model. For instance, suppose that the model described that the system receives a message $a$ followed by another message $b$. We can send the message $b$ before the message $a$, or send a malformed message $a$ in order to check how the system handles this input. Intuitively, this forces the implementation to execute the error handling part of the code and allows us to check for the presence of low-level vulnerabilities.

Fuzz testing is a well known testing method for finding software failures [13]. In fuzz testing, a well-formed input is mutated using a set of fuzz operators. The mutated input is executed against the implementation and the behavior of system is checked for failures. Model-based fuzz testing is a form of fuzz testing in which a formal model of the system is used to generate well-formed inputs. This method is particularly useful for fuzz testing of protocol implementations, where knowledge of the underlying protocol is required for the generation of valid message sequences. We propose that the fuzz operators should be defined such that the mutated inputs do not conform to the protocol while a prefix of the mutated input remains well-formed. The goal is that the well-formed prefix of the mutated input will first take the protocol implementation to some state and then the unexpected input will guide the execution towards the error handling part of the implementation, where we can check for difficult-to-find vulnerabilities.

To investigate the feasibility of our approach, we conducted a preliminary case study without using a formal model of the protocol. Instead, we use an implementation of the protocol as a low-level model. The approach of using an implementation for input generation is practically useful for testing of two-party security protocols, because we have the implementations of the end-points which can construct valid messages. However, we could not extend this approach to testing of services, for instance, because the client implementations may not be available.

**Contributions.** We define a set of fuzz operators for mutating the inputs to a security protocol implementation. The fuzz operators take a well-formed sequence of messages and mutate it to an input that does not conform to the protocol. We conducted a case study on the Internet Key Exchange protocol and evaluated the approach on Openswan, a mature IPsec implementation for Linux. Our case study reveals a previously unknown vulnerability in Openswan, which remained undetected for more than 5 years and was missed by a number of other fuzz testers [7]. Additionally, the case study reveals another known vulnerability in a previous version of Openswan.

One of the problems in fuzz testing of security protocol implementations is encrypted messages. This poses a challenge to the fuzz tester because it cannot apply most its fuzz operators on encrypted messages. We present a testing setup in which the fuzz tester has access to the encryption keys, initialization vectors, and type of encryption algorithm; this allows the fuzz tester to decrypt a message, if needed, before applying the fuzz operators. After mutating the input, the fuzzed message is encrypted before it is sent to the implementation under test. Our testing setup allows this in a modular way, without any modifications on the implementation used for generation of well-formed message sequences or the implementation under test.

**Roadmap** The remainder of the paper is organized as follows. In Section 2 we present our approach for testing of security protocol implementations. We present the case study on the Internet Key Exchange protocol in Section 3. In Section 4 we discuss related work and in Section 5 we draw conclusions.

## 2 Fuzz testing of security protocols

In this section we describe the fuzz operators that we define for mutating the inputs to security protocol implementations. We also present the testing setup for fuzz testing of protocol implementations and explain how the fuzz tester is executed in this setup so that it can monitor, modify, and decrypt the messages generated by one of the end-point.

### 2.1 Fuzz operators

Each fuzz operator takes a message sequence as input and outputs a mutated message sequence. The goal of a fuzz operator is to output a message sequence that does not conform to the protocol. In order to test the implementation under test *in* *depth*, it is required that a prefix of the mutated input is well-formed. It is infeasible to test the protocol implementation against all inputs not conforming to the protocol. To overcome this, we define fuzz operators that are likely to reveal the most common and severe types of faults. Our fuzz operators are motivated by the vulnerabilities reported at the *Common Vulnerabilities and Exposures*[2] database. We looked at the most common types of faults and summarized the types of inputs that can expose these faults. We then defined a set of fuzz operators that output such inputs.

Network protocols strictly define the structure of the messages exchanged by the end-points. Messages are often constructed using several payloads, chained consecutively in a list. Payloads are typically specified as a set of fields. Each field has a type and we categorize them into numerical fields and byte fields. We distinguish three fuzz operator categories: (1) fuzzing messages, (2) fuzzing payloads, and (3) fuzzing fields. The first category, fuzzing messages, refers to operators that mutate the sequence of messages, fuzz operators that mutate the list of payloads that form a message belong to the fuzzing payloads category, and fuzzing fields refers to mutating individual fields of the payload. Below we define fuzz operators for each of these categories.

**Fuzzing messages** We define one operator for fuzzing messages:

1. *Insert random message*. This operator inserts a random, but well-formed message at a random position in the sequence of messages. The goal is to send an unexpected message to the implementation.

**Fuzzing payloads** We define three operators for fuzzing payloads:

1. *Duplicate payload*. This fuzz operator duplicates a randomly chosen payload in the message.
2. *Remove payload*. It removes a randomly chosen payload in the message.
3. *Insert random payload*. This operator chooses a random payload and inserts it at a random position in the list of payloads.

**Fuzzing fields** We define two operators for mutating numerical fields:

1. *Set to zero*. This operator set the field to zero.

[2]http://cve.mitre.org/

2. *Set to random number*. It sets the field to a randomly chosen number.

We define four operators for fuzzing byte fields:

1. *Append random bytes*. This operator appends a sequence of random bytes to the byte field.

2. *Empty field*. It sets the field to the string of length zero.

3. *Modify random byte*. This operator replaces a randomly chosen byte in the sequence of bytes with a random byte.

4. *Insert string termination*. A string termination character is inserted at a random position in the sequence of bytes.

When fuzzing a field, all data dependent fields are updated accordingly. For instance, appending a long sequence of bytes to a byte field requires that the field holding the payload's length is set to the new length of the payload. However, if the length field is fuzzed, then the payload's size would not match the value in the length field, as expected.

A fuzzer, or fuzz tester, is the program that implements the fuzz operators. In our preliminary case study, we use an implementation of the protocol as a low-level model for input generation. The implementation used to generate inputs does not provide an entire message sequence for a single protocol execution, it generates individual messages one by one. The fuzzer reads each message generated by the system and applies the fuzz operators on-demand. Therefore, when the protocol is initiated, the fuzzer first chooses a position at which the input will be mutated. It forwards all messages before this position unmodified and applies a fuzz operator when reaching the position for mutating the input.

## 2.2 Experimental setup

We consider a security protocol that is executed between an initiator and a responder, which we call end-points. The two end-points communicate via unidirectional channels, each channel is characterized by the IP address of the end-point and a port number. In our setup, the initiator is used to generate inputs and we test the responder for vulnerabilities.

Figure 1 illustrates our testing setup. $I$ and $R$ are the initiator and the responder, respectively, and $F$ is the fuzzer. The figure shows three communication channels: a channel from the responder to the initiator, a channel from the fuzzer to the responder, and a *broken* channel from the initiator to the responder. The broken channel indicates that the responder is not listening for messages on that channel;

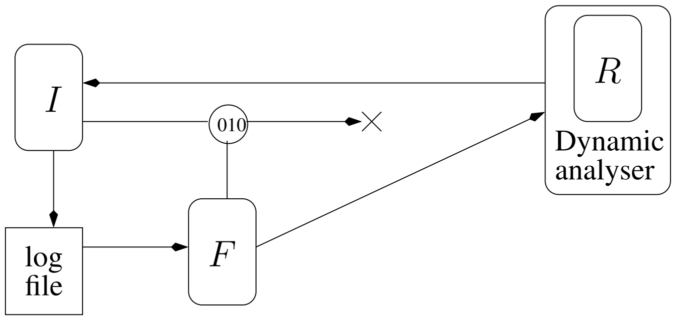

Figure 1: Experimental setup: I and R are the initiator and the responder, F is the fuzzer.

as a consequence, messages sent over the channel are not delivered at their destination. The initiator writes the protocol execution details to a log file. We assume that the fuzzer has physical access to the initiator-responder channel and can read the messages sent over this channel; furthermore, it has access to the initiator's log file. This is easily achieved by executing the fuzzer and the initiator on the same machine and granting the fuzzer with the necessary permissions to read the log file and listen for messages on the network interface. The responder executes within a dynamic analyzer, e.g. a memory error detector, which monitors the execution for failures. We remark that our experimental setup is modular as we do not need to modify the initiator or the responder implementations.

The fuzz testing process proceeds in consecutive protocol executions. The algorithm executed by the fuzzer is given in Algorithm 1. The fuzzer reads messages sent to the responder and stores them in a list. The fuzzer detects when a new session of the protocol is initiated and randomly chooses a fuzz operator for this session and a random position at which the input will be mutated. Note that we apply a single fuzz operator per protocol execution. When the fuzzing position is reached, the fuzzer applies the fuzz operator to the input. All messages received before the fuzzing position are sent to the responder intact.

## 2.3 Encryption

In security protocols, the end-points typically exchange encrypted messages. Encryption poses a challenge to fuzzing because an encrypted message appears as a random sequence of bits and we cannot apply most of the fuzz operators defined above. For instance, fuzzing the internal structure of the message, e.g. a payload or a field, is not possible. We address this problem by allowing the fuzzer to

**Algorithm 1** Fuzzer algorithm. $n_{\mathsf{max}}$ is the maximum number of messages exchanged in a single protocol execution.

```
1:  $i \leftarrow [\,]$
2:  $n \leftarrow 0$
3:  loop
4:     if $n \leq \mathsf{size}(i)$ then
5:        $m \leftarrow \mathsf{read\_message}()$
6:        $\mathsf{append}(i, m)$
7:     if $\mathsf{is\_new\_session}(i[n])$ then
8:        $f \leftarrow \mathsf{choose\_fuzz\_operator}()$
9:        $pos \leftarrow \mathsf{random}(0, n_{\mathsf{max}})$
10:       $i \leftarrow [m]$
11:       $n \leftarrow 0$
12:    if $pos = n$ then
13:       $i \leftarrow f(i)$
14:    $\mathsf{send}(i[n])$
15:    $n \leftarrow n + 1$
```

decrypt messages before applying a fuzz operator. To do so, the fuzzer needs to know the encryption algorithm, the encryption key, and the initialization vector if the encryption algorithm uses a block cipher.

In our experimental setup, see Figure 1, the fuzzer has access to the log file of the implementation used for generating messages. The information relevant for decrypting messages is typically output to the log file when the implementation is configured in debugging mode. The fuzzer determines whether it needs to decrypt a message by checking the flag, typically the flag is a single bit in the message, indicating whether encryption is used. Before applying a fuzz operator on an encrypted message, the fuzzer reads the log file in order to find about the encryption algorithm, the encryption key and the initialization vector, if used. Then the fuzzer decrypts the message, applies the fuzz operator, and re-encrypts the message using the same algorithm, key, and initialization vector used to decrypt the message.

## 3 The IKE case study

The goal of our approach is to find vulnerabilities in real world security protocol implementations. To investigate the usefulness of the proposed method we conducted a case study on the Internet Key Exchange (IKE) protocol [10].

## 3.1 IKE overview

The IKE protocol is used to set up security associations for the IPsec protocol suite. It is widely used in practice and there are a number of open source implementations of the protocol, which makes IKE a good candidate for our case study. Concretely, we decided to conduct our experiments using Openswan[3], which is a mature IPsec implementation for Linux. The National vulnerability database[4] reports several vulnerabilities in Openswan; this allows us to test older versions of the implementation and check whether our method can find these problems.

The IKE protocol negotiates an IPsec tunnel between two end-points. The protocol proceeds in two phases - phase 1 and phase 2. The purpose of phase 1 is to establish a secure authentic channel between the end-points. It achieves this by generating a secret key using the Diffie-Hellman key exchange algorithm and authenticating the end-points using pre-shared secret key, signatures, or public key encryption. Phase 1 has two modes - main mode and aggressive mode. Main mode requires the exchange of 6 messages and the authentication of the end-points is protected, i.e. the IDs are sent encrypted, whereas aggressive mode requires only 3 messages but the end-points' identities are not protected. In phase 2, the end-points negotiate a security association using the already established secure authentic channel and the state established after phase 1.

## 3.2 IKE fuzzer

We implemented a fuzzer that implements all fuzz operators defined in § 2. Our fuzzer is implemented in Python and uses the Scapy library[5]. Scapy is a packet manipulation library that allows decoding of a wide number of protocols, including the IKE protocol. For capturing messages sent to the responder, the fuzzer uses tcpdump[6], which is popular Linux tool for dumping traffic sent over a network interface.

## 3.3 Dynamic analysis

We execute the responder within MEMCHECK, a memory error detector for C and C++ binary programs based on Valgrind [14]. The tool can detect a broad range of memory access problems: access of unallocated memory or memory that has been freed, using uninitialized memory, double freeing heap blocks, and memory leaks. MEMCHECK monitors each memory access of the responder. In case of a failure,

[3]http://www.openswan.org

[4]http://nvd.nist.gov/

[5]http://www.secdev.org/projects/scapy

[6]http://www.tcpdump.org/

Valgrind reports a stack trace which can be used to backtrack to the source of the problem.

## 3.4 Results

We performed two experiments on Openswan.

**Experiment 1** For the first experiment we used Openswan version 2.6.20, a single laptop Thinkpad T60 with a dual-core Intel T2500 processor and 2GB of RAM, Fedora 15 operating system using kernel version 2.6.40.4. The duration of the experiment was 10 hours.

The national vulnerability database reports a vulnerability in the ASN.1 parser in all Openswan versions before 2.6.22 [5]. It allows an attacker to cause a denial of service attack via an X.509 crafted certificate with crafted relative distinguished names, crafted UTC time string, or crafted generalized time string. Our first experiment successfully reveals the failure. MEMCHECK reports over a hundred invalid memory access violations in two of the ASN.1 parser functions. The security patch released by the Openswan team makes changes to the same two functions in order to fix the vulnerability.

**Experiment 2** For the second experiment we used Openswan version 2.6.35 (the latest version at the time of conducting the experiment), 26 ThinkCenter machines equipped with quad-core Intel i5-2400 processors and 8GB of RAM, RedHat 6 operating system using kernel version 2.6.32-131. The duration of the experiment was 10 hours. Our second experiment reveals a serious use-after-free vulnerability in the latest version of Openswan. The vulnerability allows authenticated gateways to cause denial of service [7], it is network exploitable and requires authentication.

After completion of phase 1, the end-points have established an IKE security association. In phase 2, the end-points negotiate an IPsec security association. The key and initialization vector for the new IPsec security association are computed using the state established after phase 1. Cryptographic operations are time consuming and Openswan uses separate threads (called cryptographic helpers) for their execution. In phase 2, Openswan assigns to a cryptographic helper the task to compute the encryption key and the initialization vector for a new IPsec security association. The computation of the initialization vector, for instance, uses phase 1 state information; this information is handed to the thread by passing a pointer to the data structure holding the phase 1 information. There is a race condition when the cryptographic helper accesses the phase 1 data: it does not check if the phase 1 session is still open. If the IKE session is closed, the phase 1 data is freed and the cryptographic thread accesses freed memory. This vulnerability is known as use-after-free vulnerability[7].

We reported the vulnerability to the developers of Openswan, who promptly responded and released a security patch. We assisted the developers by testing the proposed patches and our IKE fuzzer demonstrated its ability to discover failures by finding a null pointer dereference introduced by one of the candidate patch. More details on the vulnerability and the released patch are given in CVE-2011-4073[8].

## 4 Related work

Fuzz testing was first introduced by Barton Miller as an automated testing method [13]. The original method has shown that it can successfully reveal software failures. In contrast to other security testing methods, which often focus on testing for specific known vulnerabilities, fuzz testing is able to expose unknown vulnerabilities. Practically, this is highly important as attackers continuously discover new attack vectors and securing a system against known attacks is helpful, but not sufficient for security critical systems and systems that are expensive to update.

**Random fuzzing** Random fuzz testing is the oldest and the simplest form of fuzzing. In random fuzzing, the implementation under tests is fed with random inputs and the system's behavior is checked for failures. The main advantage of this type of fuzzing is its simplicity. The fuzz tester does not need any knowledge about the implementation under test, which implies that a random fuzz tester can be used for testing of any system. It is most effective when used for testing of stateless request-response systems. When applied to stateful systems, it is generally limited to testing only the initial state of the system.

**Feedback-driven fuzzing** Feedback-driven fuzzing assumes no knowledge of the source code or the specification of the system under test. Instead of executing the system against purely random input, the fuzz tester obtains feedback from each execution of the system, e.g. basic block coverage information, and based on the received feedback it determines the next input. Typically, feedback-driven fuzzers [15] use taint tracing in order to infer how each bit of the input influences the execution of the program. Starting with a well-formed input, such tools dynamically learn how to explore a program.

[7]http://cwe.mitre.org/data/definitions/416.html

[8]`http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2011-4073`

In [8], the authors improved the effectiveness of feedback-driven fuzzing by introducing sub-instruction profiling. In contrast to using basic block coverage feedback, sub-instruction profiling allows the fuzz tester to obtain more precise coverage information. For instance, if the program compares two 32-bit sized memory chunks, sub-instruction profiling assigns a score corresponding to the size of the longest common prefix. Using this technique, the fuzzing tool can reconstruct specific fields in the input such as CRC checksums.

**White box fuzzing** White box fuzz testing leverages the recent advances in symbolic execution and dynamic test generation [4, 3]. This approach records the execution of the program on a well-formed input, symbolically evaluates the recorded trace and collects constrains on the input [9]. The set of collected constrains are negated one by one. Each new set of constraints is then fed into a constraint solver, which in turn generates a concrete input that satisfies the set of constrains, or returns that it could not find a satisfiable solution. The program exercises different control paths when executed on the newly generated inputs. The generated inputs are then fuzzed and executed against the implementation.

**Model-driven fuzzing** Model-driven fuzz methods [12, 1] leverage the knowledge of the underlying protocol in order to generate inputs more intelligently. PROTOS targets at fuzz testing of network protocols [12]. It starts from the protocol specification and generates a set of inputs which can be executed against a system implementing the protocol. Using the protocol specification, PROTOS ensures that the initial messages are well-formed and take the system into a state which will accept some message $m$. Message $m$ is fuzzed using heuristics and aims to exploit common vulnerabilities such as buffer overflows and format string vulnerabilities. Although the PROTOS fuzzer is not publicly available, the authors have published test suites for several popular protocols, including IKE, which have been used to discover vulnerabilities in a number of different implementation.

SNOOZE is a tool for building flexible, security-oriented, network protocol fuzzers [2]. It allows a tester to describe the stateful operation of a protocol and the messages that need to be generated in each state. Additionally, SNOOZE provides attack-specific fuzzing primitives which can be used to exercise the system for common vulnerabilities. It requires that the tester manually specifies fuzzing scenarios, which describe the fuzzing activities to be performed.

In [11], the authors consider the problem of detecting implementation flaws using model-based fuzz testing without a formal model of the protocol. Their approach first synthesizes an abstract behavioral model from the protocol implementation and then uses this model to guide the testing process for detecting flaws.

The proposed method is particularly useful when testing proprietary protocols that do not have any specification.

## 5 Conclusions

In this work we presented a fuzz testing method for finding vulnerabilities in security protocol implementations. Despite the fact that security protocols are proven correct, their protocol implementations have low-level vulnerabilities. Typically, vulnerabilities are introduced in the error handling part of the protocol implementation. Our testing method focuses on this hypothesis and guides the execution towards the code that handles unexpected events. We presented a general testing setup for fuzz testing of security protocol implementations and a concrete implementation for the Internet Key Exchange protocol. Finally, we demonstrated that our testing method can find real vulnerabilities by conducting a case study on the IKE protocol and reporting a new vulnerability in a popular IPsec implementation for Linux.

## References

[1] Dave Aitel. The advantages of block-based protocol analysis for security testing. Technical report, 2002.

[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin C. Almeroth, Richard A. Kemmerer, and Giovanni Vigna. Snooze: Toward a stateful network protocol fuzzer. In Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel, editors, *ISC*, volume 4176 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2006.

[3] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee : Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, page 209224, 2008.

[4] Cristian Cadar, Paul Twohey, Vijay Ganesh, and Dawson Engler. *EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution*, pages 1–20. Citeseer, 2006.

[5] The MITRE Corporation. CVE-2009-2185. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2185`, June 2009.

[6] The MITRE Corporation. 2011 CWE/SANS top 25 most dangerous software errors. `http://cwe.mitre.org/top25/index.html`, September 2011.

[7] The MITRE Corporation. CVE-2011-4073. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2011-4073`, October 2011.

[8] Will Drewry and Tavis Ormandy. Flayer: exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1:1–1:9, Berkeley, CA, USA, 2007. USENIX Association.

[9] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.

[10] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 (Proposed Standard), November 1998. Obsoleted by RFC 4306, updated by RFC 4109.

[11] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. *2008 IEEE International Conference on Network Protocols*, pages 114–123, 2008.

[12] Rauli Kaksonen, M. Laakso, and A. Takanen. System security assessment through specification mutations and fault injection. In Ralf Steinmetz, Jana Dittmann, and Martin Steinebach, editors, *Communications and Multimedia Security*, volume 192 of *IFIP Conference Proceedings*. Kluwer, 2001.

[13] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[14] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 89–100. ACM, 2007.

[15] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, 2005.

[16] Jan Tretmans. Model Based Testing with Labelled Transition Systems. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, chapter 1, pages 1–38. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.