Transforming Existing Stateful Network Protocol Applications into Effective Protocol-Specific Security Testing Tools

by

Vasily Rudchenko

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

Transforming Existing Stateful Network Protocol Applications into Effective Protocol-Specific Security Testing Tools

Vasily Rudchenko
Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto
2019

To discover low-level security vulnerabilities in target applications that implement stateful network protocols, security researchers usually employ manually-constructed protocol-specific testing tools. Due to the complexity and heterogeneity of network protocols, however, building such tools for every network protocol used by target programs is a challenging task. We propose a semi-automated technique for creating protocol-specific testing tools, which leverages an existing application that implements a network protocol (a source), and converts it into a testing tool that can be used for bug discovery in any target program that "speaks" the same protocol. We call our implementation of this method InSource, and, by measuring coverage gain, show that InSource can test a target just as well a manually-created protocol-specific testing tool. Furthermore, we show that InSource is effective at testing applications that use different protocols, as long as a source application for each protocol is available.

Acknowledgements

I would like to sincerely thank Professor David Lie for continuously providing keen insight and advice throughout the course of my project, as well as for giving invaluable feedback on the various drafts of my thesis. Furthermore, I would like to thank him for creating and tirelessly maintaining the lab's computer infrastructure on which this work so greatly relies.

I would also like to thank Dr. Ivan Pustogarov for providing knowledgeable insights into the tools and techniques used in security testing, and for giving me feedback throughout the course of the project.

Finally, I am grateful for the financial support provided by the University of Toronto towards both my studies and this research.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Securing software from attackers is a difficult task, as it relies on the ability to find and fix software vulnerabilities before these vulnerabilities are exploited by attackers. To discover and fix vulnerable code before it is exploited, organizations rely on both manual approaches, such as crowd-sourcing of efforts from white-hat attackers through sites like HackerOne, where users reported a total of 78,275 security bugs [38] to software companies in 2017 alone, as well as automated efforts, such as the large-scale continuous security testing performed by Google, which automatically discovered 16,000 bugs [49] in the codebase of the Chrome browser since the project's launch in 2016.

The difficulty of securing software from bugs is exacerbated by the continued wide-spread use of low-level language like C and C++, which give programmers direct access to memory, treat integers as raw bits, and contain undefined behavior. On GitHub, C and C++ are among the top 10 most-used programming languages in 2018 [36]. The effect of this persisting popularity of low-level languages in both industry and open-source projects has resulted in 1 in 5 security vulnerabilities listed on the MITRE CVE database [93] being directly caused by memory corruption and overflows. These kinds of low-level security vulnerabilities are issues that are linked specifically to low-level languages, as they are often mitigated by modern programming paradigms.

While such low-level vulnerabilities can affect the security of any software that contains them, their effect is particularly vicious when they occur in software that implements network communication. Networked software is frequently connected to the internet by design, so vulnerabilities in this software can be easily exploited when anyone in the world can send network messages to a vulnerable application. A famous example is the Heartbleed vulnerability in OpenSSL [24] that was notoriously easy to exploit to perform out-of-bound memory reads, and was left exposed to anyone with an internet connection. One of the detected exploits of this vulnerability was the unauthorized access to 900 Canadian social insurance numbers [32] via a government website that used a vulnerable OpenSSL server. More recently, in 2019, Netflix researchers were able to identify how to exploit the Linux TCP implementation [56] to compromise availability of a service by simply sending a sequence of messages that could cause a kernel panic due to an integer overflow. If left unfixed, this bug would allow attackers to compromise the reliability of cloud servers relying on Linux for TCP interactions. In the same year, a buffer overflow vulnerability in the VoIP stack of the WhatsApp secure messaging application was discovered [25]. The vulnerability allowed attackers to compromise a target user's message confidentiality, effectively negating the secure messaging feature provided by WhatsApp [51]. Because all of these vulnerabilities were caused by bugs in code that implemented network protocols, attackers were able to exploit them without any cooperation from the software user, as is common with vulnerabilities that target other kinds of applications.

Despite how exploitable some of these bugs in network protocols are, finding them is often challenging. Both the applications that implement TLS and TCP protocols as well as the VoIP network stack tend to be very large, and encompass a lot of functionality and complexity. To be able to find the bug that caused the Heartbleed vulnerability, for example, one would need to be able to know how to generate an *invalid* "heartbeat" message and then try sending it to an OpenSSL server. An invalid message is one that does not follow the correct format of a protocol message. In this case, a heartbeat message that can trigger the bug holds a length field that doesn't reflect the actual length of the message, while the rest of the message fields are specified correctly. Because the target application erroneously processes this invalid message and then uses its contents to generate a response, the bug occurs.

Although some bugs are revealed through construction of invalid messages, the TCP bug discussed earlier does not require this ability. Instead, the bug is triggered by sending valid SACK messages in a *non-standard order*. By using a special order of messages that is not anticipated by the engineers who built the server, while also keeping each individual message in the sequence valid, the server accepts and processes each message fully. The server keeps accepting and processing the TCP SACK requests and incrementing an internal variable with each received message, until an integer overflow occurs. Thus, to test the nuances of a TCP network protocol implementation, a tester has to not only be able to send individual invalid messages, but should be able to test the server with valid messages sent in a non-standard order. Unlike a *standard order*, which is the order of messages produced by a well-behaved TCP peer, a non-standard order of messages does not necessarily follow protocol conventions about how messages should be ordered, and is thus able to place the target into a particular state in which a bug occurs.

From the three bugs discussed previously, the one affecting WhatsApp's VoIP implementation is probably the most impactful in its severity, but also the most difficult to find. In order to discover this bug, one has to be able to both execute messages in a specific order, and to carefully modify the contents of one of the messages. Specifically, the message has to be sent after a SIP handshake is performed, and the contents of the SRTCP packet, which are both authenticated and encrypted, have to be correct, in order for the packet not to be discarded by the application. For this combination of events to occur, a tester should be able to both send valid messages in a specific order to put the target in a particular state and then change individual fields in one of the messages in order to trigger the bug.

The defining feature of these bugs is that they are triggered by either completely valid messages or invalid messages that are only subtly wrong: to discover the bugs, a tester needs to correctly follow most of a protocol's structure, while only making small changes to the messages or the sequence of messages sent. If the tester deviates too far from correctness, deep bugs in each application are left undiscovered, since the applications pessimistically reject most invalid messages before the messages are used in any processing code. The most effective current approach for generating such messages is the use of *protocol-specific* testing tools. By employing knowledge about a specific protocol, a tester implements a working version of the protocol that can send valid network messages in a standard order, and then modifies both the contents and the order gradually, so as to find subtle bugs in the application they are targeting. Indeed, this approach has been effectively used by security testers for both low-level software vulnerability discovery [30][70][84] and while checking programs for other kinds of implementation bugs in network protocol applications [26][47][52].

Indeed, the combination of the effectiveness of protocol-specific testing tools and the manual effort required to build them makes them a valuable commodity. Companies that provide meticulously crafted protocol-specific testing tools for a variety of network protocols include Synopsys, Beyond Security, and Peach Teach, each of which offers its own assortment of commercial protocol-specific testing tools [8][27][66], for testing both simple and complex network protocol implementations. On the other hand, the open-source world has mostly lagged behind commercial efforts. While many general frameworks for making protocol testing tools have been made over the years [1][5][60][65][68][87][91], the testing tools made using these frameworks usually target simpler protocols. The total number of supported network protocols that can be tested using open-source tools is slim in comparison to the amount of protocols that commercial tools [28] support.

The difficulty of building protocol-specific testing tools is two-fold. First, many protocols are *complex*, so creating a testing tool that can effectively generate both valid and invalid messages requires re-implementation of the many protocol features that are present in the application that they are trying to target. Second, network protocols are *heterogeneous*, with many different protocols being used across many applications; the manual effort of protocol re-implementation has to be repeated for every protocol which is to be tested in a protocol-specific way. We present examples of how these properties of network protocols make effectively testing network protocol applications for low-level software vulnerabilities a difficult task.

1.1 Protocol complexity

Protocols generally fall into two categories: stateless and stateful. Stateless protocols are used by sending individual messages between two applications, with each message being independent from the previous ones. State is not maintained by the code responsible for implementing the protocol, meaning that the protocol implementation is only responsible for parsing message contents and providing the results to a higher layer in the network stack. Although stateless protocols perform a limited role in a network stack, the message format of individual messages is often complex enough such that multiple protocol RFCs describe features that are implemented in a network protocol application. Therefore, to test the implementation of individual features in a stateless protocol application, a protocol-specific testing tool has to be extended with support for every new message format feature that an application implements.

Stateful protocols, on the other hand, require a protocol implementation to maintain state, contain inter-message dependencies between messages, and use more complex fields in message construction, such as fields that are encrypted or hashed, or fields that depend on data from previous messages. All of these properties make them difficult to test; however, the most challenging property that a testing tool has to tackle, is the presence of state in the program being tested. To test all of the code associated with an implementation of a protocol, a protocol-specific testing tool has to be able to generate multiple message sequences that can "toggle" different states in the application. For example, Figure 1.1 shows how the state variable `s->state` in an OpenSSL server dictates which message the application is ready to accept next. In the switch statement to the left, a server first uses the state variable to pick which message type it accepts. It then receives the message and processes it. In order to test the message processing code associated with each state, a testing tool must first be able to toggle the state in which the processing code runs, and then vary the message contents in order to exercise the code and trigger

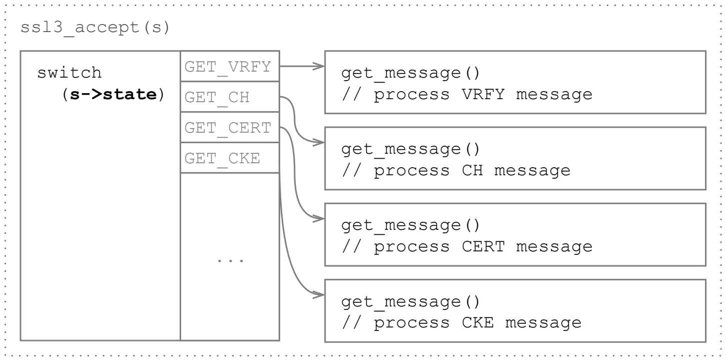

ssl3_accept(s)
switch (s->state)
GET_VRFY
GET_CH
GET_CERT
GET_CKE
...
get_message() // process VRFY message
get_message() // process CH message
get_message() // process CERT message
get_message() // process CKE message

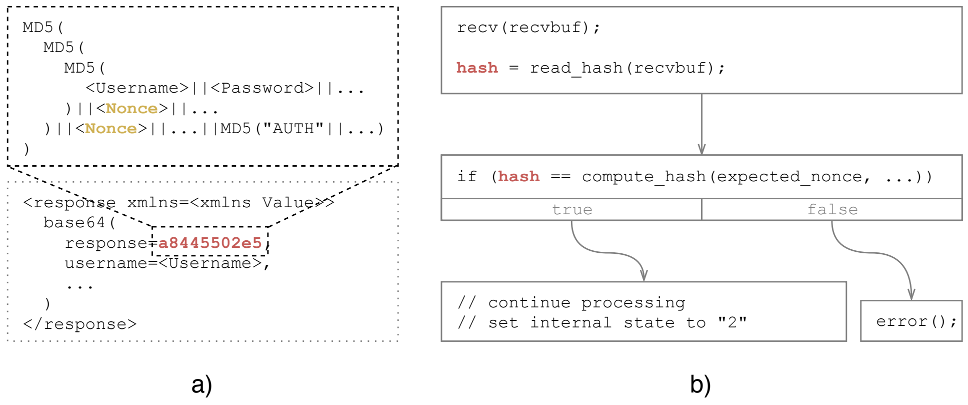
Figure 1.1: Code in the function `ssl3_accept` in the OpenSSL library demonstrates the importance of state when testing stateful network protocols. (The code is simplified for brevity)

bugs. For example, the bug found in the WhatsApp application is only triggered once the WhatsApp code reaches a specific state.

Furthermore, the state of an application is usually defined by multiple state variables; indeed, in the OpenSSL server, the `s->state` variable is only one of many variables that help define the current state of the application. Because it is difficult for developers of protocol-based applications to reason about complex interactions of different state variables, protocol applications often also contain extraneous states [26][33][86], which are not defined by protocol specifications. To test both the states that are defined by the protocol and the extraneous states that are inadvertently introduced in the development process of an application, a testing tool must be able to generate multiple different message sequences, including non-standard message sequences that can reveal unusual states in the application. Furthermore, as demonstrated by the vulnerability in the Linux TCP implementation, vulnerabilities can be linked to the incorrect handling of internal state, further necessitating that testing tools implement the ability to generate non-standard message sequences.

Generating message sequences that trigger new states in stateful protocol applications is difficult because individual messages in these sequences have to be valid enough to not be rejected by the application. For example, the "Response" message which gets sent by an XMPP client to an XMPP server during authentication is difficult to generate correctly, such that it gets processed by a target server and changes the application's state. This is because, as shown in Figure 1.2.a, the message contains a hash of a "nonce" field. The nonce is a value that is provided to the client by the server, which the client must use when computing the "hash" field. The server always provides a different nonce to the client every time a sequence of messages is sent. As shown in part b) of the figure, if the server sees that the hash value in the message does not match its own computation of the field, the message is discarded, and the server does not continue processing the message, and does not iterate its internal state. As seen in the Figure 1.2.b, an invalid hash results in the application aborting its interaction with its peer. To make the server accept the message and change its internal state, a message with a valid hash must be sent. Since the nonce is changed during every handshake, the testing tool cannot simply replay messages from previous executions of the XMPP client.

In addition to implementation complexity associated with protocol state, stateful programs are also difficult to test for bugs because of the complex format of individual messages. Figure 1.3.a shows the structure of a TLS record message, which is difficult to construct in a way that effectively tests the program code. The data is first compressed (upper-left corner). The compressed value is then used



```
MD5(
  MD5(
    MD5(
      <Username>||<Password>||...
    )||<Nonce>||...
  )||<Nonce>||...||MD5("AUTH"||...)
)
```

```
<response xmlns=<xmlns Value>>
  base64(
    response=a8445502e5
    username=<Username>,
    ...
  )
</response>
```

```
recv(recvbuf);

hash = read_hash(recvbuf);
```

```
if (hash == compute_hash(expected_nonce, ...))
```

```
// continue processing
// set internal state to "2"
```

```
error();
```

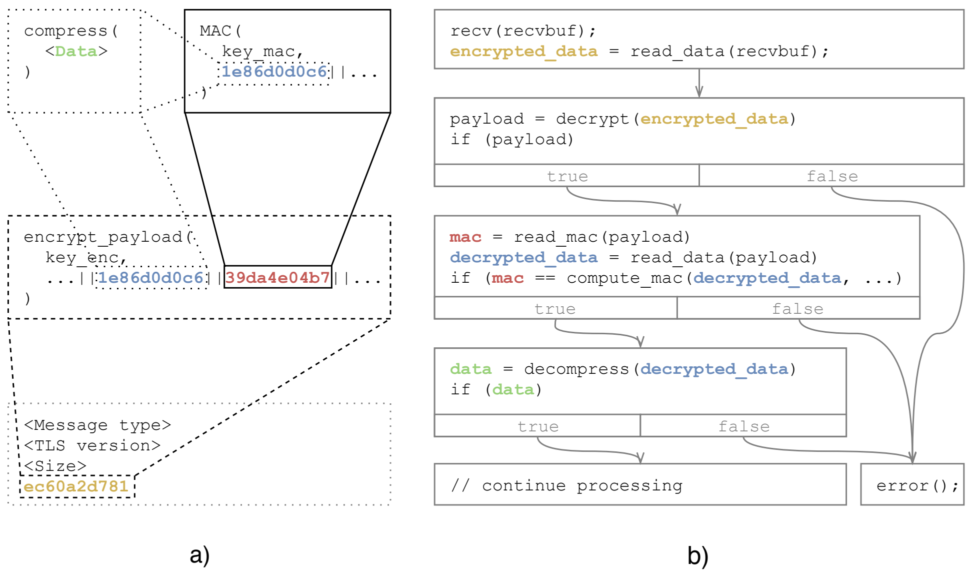
Figure 1.2: a) The structure of the "response" message sent by an XMPP client. The "response" field in the message is constructed by computing a hash of various subfields, including the "nonce." b) Code in the function `QXmppSaslServerDigestMd5::respond` in the QXMPP library (parts of the code are omitted and simplified for brevity).

to calculate a MAC value that helps a TLS server verify message integrity. Both the MAC and the compressed data are then encrypted using a key that is unique every time a client and a server execute the protocol; finally, the entire message is prefixed with a header field. When the server receives the message, it has to reverse the process (as shown in Figure 1.3.b): the message body is first decrypted, the MAC of the message is checked, and the compressed data is decompressed. If, at any stage the server is unable to perform the operation (which is possible if the keys used by the client and server don't match, or if the message is too mangled by the testing tool), the server aborts the operation. To, for example, generate valid record messages that get correctly decrypted and used by the target application, a testing tool must be able to generate valid TLS record messages that pass all steps shown in the figure, but contain a mutated "data" field. Even just to test the decompression code (i.e., the `decompress` function call in the figure), a testing tool must be able to generate a message that is only invalid in the compressed data field, but for which the encryption and the MAC are computed correctly.

The final complexity in designing protocol-specific testing tools for testing stateful network protocol application is the presence of numerous message types in each stateful protocol. To test an application, an effective testing tool must be aware of different message types, and be able to both know how to construct them in a valid way, and be able to make use of them when generating different message sequences used to find and test new states in the application. Testing tools that implement a larger number of messages can both discover more states in the application being tested, and better exercise message-processing code associated with each state.

1.2 Protocol heterogeneity

The heterogeneity of network protocols results in the need to re-implement protocol-specific testing tools for every new protocol. This heterogeneity of protocols reveals itself in the diversity of details used to implement each protocol, such as message structure, data encoding, encryption schemes used, and the methods of handling state. These differences mostly stem from the differences in each protocol's requirements, as some protocols, like TLS, provide strong confidentiality guarantees, while others, like



```
compress(
  <Data>
)

MAC(
  key_mac,
  1e86d0d0c6||...
)

encrypt_payload(
  key_enc,
  ...||1e86d0d0c6||39da4e04b7||...
)

<Message type>
<TLS version>
<Size>
ec60a2d781
```

```
recv(recvbuf);
encrypted_data = read_data(recvbuf);

payload = decrypt(encrypted_data)
if (payload)

mac = read_mac(payload)
decrypted_data = read_data(payload)
if (mac == compute_mac(decrypted_data, ...)

data = decompress(decrypted_data)
if (data)

// continue processing

error();
```

Figure 1.3: a) The structure of a record message sent by a TLS client. The message body is constructed by first compressing, then taking a MAC of, and finally encrypting the data. b) Code in the function `ssl3_get_record` in the OpenSSL library (parts of the code are omitted and simplified for brevity).

SIP, are primarily used for quickly establishing a persistent session between network nodes. A protocol like XMPP is built with extensibility in mind, while a protocol as simple as UDP is unlikely to ever change in design.

Furthermore, protocols are frequently used together with each other in multiple layers, meaning that a fully-functional network stack requires multiple protocols to be implemented, each with a different purpose. Even the commonly used HTTPS stack requires two separate persistent session to be established between nodes (one for TCP and one for TLS). Because the layered model encourages protocol isolation, applications in the stack can usually be tested in isolation from one another. However, sometimes protocols aren't well-separated into layers and are jointly implemented by an application, so while a testing tool for the simpler SDP protocol can be developed for when an application only uses SDP, once SDP is integrated with SIP, the testing tool has to also be modified to account for the tight coupling that occurs between the two protocols. In a context where multiple nodes are present in a functional network infrastructure, such as LTE, the total number of protocols used grows further, with every node using a different stack and "speaking" a different protocol depending on which other node it is currently interacting with.

These differences in protocols and how they are used mean that those wishing to extensively test these protocols for security vulnerabilities have to spend considerable manual effort implementing protocol-specific functionality anew, for every new protocol that they wish to test. Unfortunately, this task is likely to get more difficult over time. As technology evolves and requirements change, new protocols spring into existence. A recent study [88], for example, showed that a larger number of wireless protocols are being used for implementing healthcare-related devices and systems in automotive vehicles, while the growth in the number of IoT devices and the diversity of specialized industrial control systems have both contributed to the diversity of protocols used. As the number of internet-connected devices is projected to grow [89], the number of use-cases for both existing and new network protocols is likely

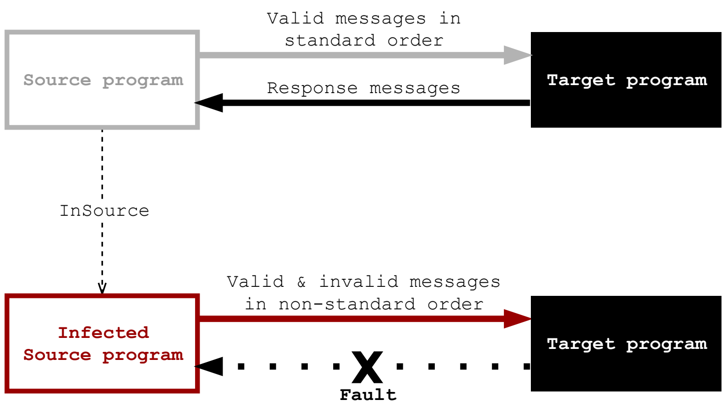


Figure 1.4: An overview of how InSource converts a source application into a protocol-specific testing tool that can discover bugs in a target program.

to increase. Indeed, with the recent introduction of 5G, protocols used for cellular communication are already evolving [71] to adapt to new underlying technologies and user needs.

1.3 Generating protocol-specific testing tools

We strive to eliminate most of the manual effort currently required to create effective protocol-specific testing tools. To automate parts of this process we rely on one key insight: instead of leveraging the knowledge of a human tester (as is done by existing tools), we can leverage existing valid implementations of network protocols to be able to generate effective testcases that can discover low-level security vulnerabilities in a target application. A program implementing a protocol can be used as a protocol-specific testing tool in the following way:

1. An existing implementation of a protocol can be executed to generate valid protocol-specific messages in standard message order.

2. Then, non-standard message orders can be generated by modifying the behavior of the existing implementation. We insert instrumentation at compile-time that allows us to change the behavior of the program while it is running. The same instrumentation is inserted for every source program, allowing us to guide the source program itself towards generating different protocol-specific message sequences.

3. Finally, different valid and invalid variations of valid messages can be generated by modifying individual message fields while the message is being built by the protocol implementation, or even by mutating the contents of the whole message before it gets sent.

Thus, instead of investing manual effort in writing a new protocol-specific testing tool for every new protocol, we create a general method that can re-wire existing implementations of network protocols to act as protocol-specific testing tools. We call the software that we implement and use for transforming existing protocol implementations into protocol-specific testing tools "InSource." We call our tool InSource because it is able to "`Infect`" a `Source` application and force it to behave differently for InSource's own gain.

In the way described above, InSource uses an existing implementation of a protocol to generate messages that can be used to test another application. We call the program that InSource augments (i.e., “infects”) in this way the *source* program, since this program is used (with some help from InSource) to generated and send different valid messages in non-standard message order. The *target* program is the one that is being tested for bugs: it receives the messages generated by the source. If a bug is triggered in the target, the bug is detected by InSource. The source and the target don’t have to be the same program; instead, a source has to be an application that implements the same network protocol as the one implemented by the target. Figure 1.4 shows, at a high level, how InSource modifies the default behavior of a source application in order to find bugs in a target.

By evaluating InSource by testing real programs, we find that our approach effectively addresses the two challenges that make manual construction of protocol-specific tools a difficult task. First, InSource effectively handles test generation for complex stateful network protocols, making it able to compete with manually-constructed protocol-specific testing tools. Second, InSource is able to adapt to different network protocols, by leveraging existing implementations of each protocol.

Chapter 2

Background

As mentioned in Chapter 1, protocol-specific testing tools are an effective and commonly used method for finding bugs in applications that implement network protocols. Note that the methods for discovering security vulnerabilities are often the same as methods used for finding bugs. This is because only a subset of bugs are security vulnerabilities and a testing tool must first discover bugs in a target application before a user can determine which bugs are exploitable by attackers.

There are various techniques that these tools use to discover bugs in target applications, including methods for generating *invalid* messages, methods for making different *valid* messages, methods for combining messages in *non-standard* message orders, as well as methods for detecting when bugs occur in the target application. Valid messages are those that, in their format, completely follow the specifications for some message type in a network protocol. InSource and many other protocol-specific testing tools do not strive to differentiate between whether they generate valid or invalid messages (although some tools do have this ability [9]). Instead, a tool's ability to generate more different valid messages is usually indicative of its ability to effectively test a target application, as will be shown in the examples in this chapter. When we evaluate InSource in Chapter 6, we will not directly measure how many messages that it generates are valid, but will instead evaluate what effect techniques for generating more valid messages have on our ability to test the target application.

2.1 Fuzzing protocol messages

One effective technique often used by protocol-specific testing tools is *fuzzing*. In its most basic form, fuzzing is actually not a protocol-specific testing method. Fuzzing is a general technique for testing a target application by sending it many test inputs that are generated by performing random mutations to an initial test input, known as the *seed*. Mutations include operations like bit flips, arithmetic operations of individual bytes, insertions of known "tricky values" like `NULL` and `INT_MAX`, deletion and reordering of different parts of the input, as well as insertion of very long strings into various parts of the input. Fuzzing is known to be good at discovering parsing errors in target applications since the mutations that are applied to a seed help a fuzzer trigger edge-cases in the target's code. Figure 2.1 illustrates the basic operating principle of a fuzzer. First, a seed is provided to the fuzzer (in this case, "`Hello World`"). This seed is usually a valid input to the program, chosen by a human tester, but the seed may be any string of bytes. The fuzzer then performs mutations on the seed and generates multiple new inputs to

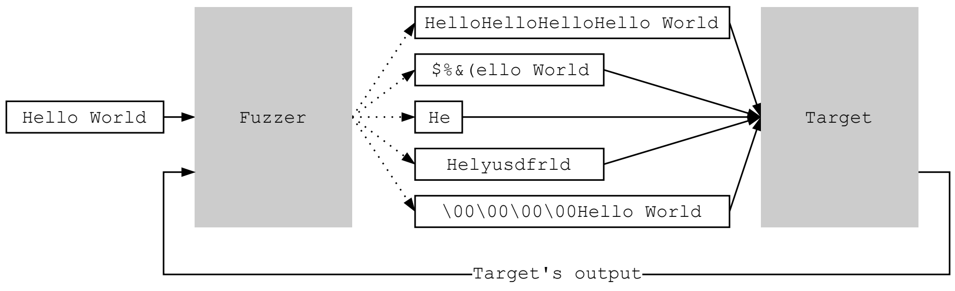
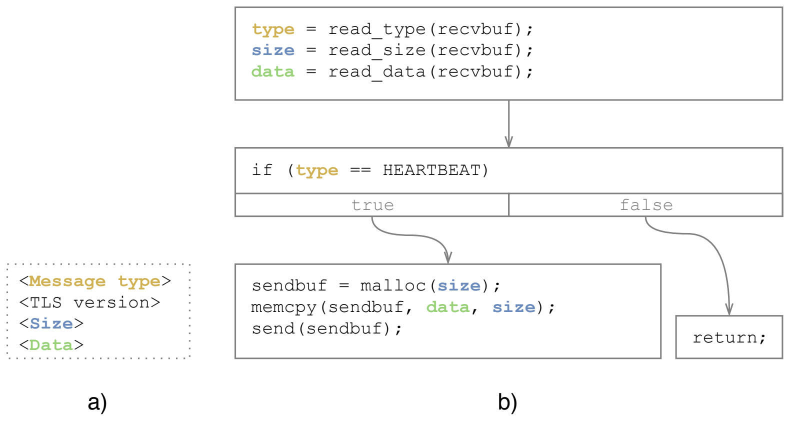


Figure 2.1: An example of how a fuzzer operates. A human tester provides the fuzzer with an initial input (seed). The fuzzer then performs various mutations on the seed input and generates mutated inputs to the target (shown in the center). Then either the human tester or the fuzzer itself sends the inputs to the target being tested, one after another. The fuzzer observes the output of the target program after each input is sent, to determine if a bug has occurred.

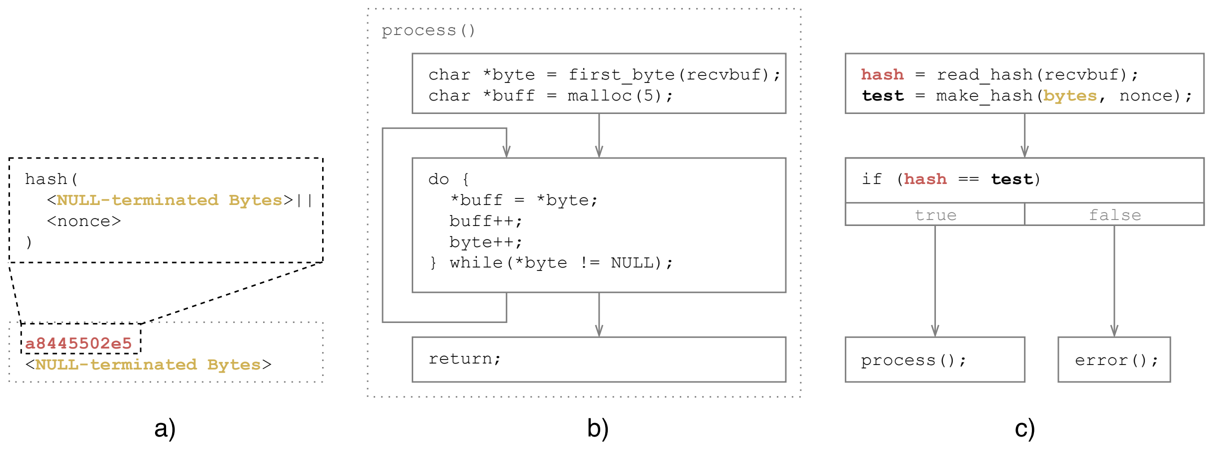


```
type = read_type(recvbuf);
size = read_size(recvbuf);
data = read_data(recvbuf);

if (type == HEARTBEAT)

sendbuf = malloc(size);
memcpy(sendbuf, data, size);
send(sendbuf);

return;
```

Figure 2.2: a) Heartbeat message format. b) Message parsing of the Heartbeat message. The bottom-left block in b) contains a bug. The code is simplified for brevity.

the target program, such as "`He`," "`Helyusdfrld`," etc. The mutated inputs are then sent by the fuzzer to the target. The fuzzer then observes the target's behavior and uses it to deduce when a bug has occurred (this is also called fault detection, and will be discussed in more detail shortly). If the fuzzer observes a bug, it notifies the tester. The main reason why fuzzers perform well is because they are able to generate many inputs quickly, so by sheer volume of mutations made, interesting bugs related to edge-cases in the code can be uncovered in the target.

In this basic form, however, fuzzing is a blind method of testing: it does not take into account the format of the message when performing mutations. In other words, the fuzzer mutates different parts of the message indiscriminately. Thus, when the fuzzer makes modifications to a seed that is a whole valid message, the mutations that it makes mostly produce invalid messages. Still, fuzzing whole messages can be a powerful technique. Consider Figure 2.2, which shows the code for the Heartbleed bug [24]. Part a) shows the format of the Heartbeat message. Part b) shows how the message is parsed and used by the target application. The `size` field is never checked by the application and so the value used in `memcpy` may be larger than the size of `data`. If this is the case, a buffer over-read occurs. In order to find this bug, a fuzzer starts with an initial valid heartbeat message as a seed, and generates many variations of this message that are invalid. If one of the variations that it generates contains a size field that is



```
hash(
  <NULL-terminated Bytes>||
  <nonce>
)

a8445502e5
<NULL-terminated Bytes>
```

```
process()

char *byte = first_byte(recvbuf);
char *buff = malloc(5);

do {
  *buff = *byte;
  buff++;
  byte++;
} while(*byte != NULL);

return;
```

```
hash = read_hash(recvbuf);
test = make_hash(bytes, nonce);

if (hash == test)

process();

error();
```

Figure 2.3: Example of a message that is difficult to generate correctly. a) Message format. b) Message parsing code. This code contains a bug. c) The code responsible for first checking that the hash is valid.

larger than the data field, the bug is triggered. While a fuzzer is good at generating many different invalid messages, and is sometimes also able to generate similar valid messages, a fuzzer is unlikely to generate entirely *different* valid messages through random mutations alone. In fact, if a message format also contains complex message fields (an example of this will be shown in Section 2.2.1), a blind fuzzer is unlikely to ever generate a valid message of this type.

2.2 Generating different valid messages

Unlike blind fuzzers, protocol-specific testing tools are built to be able to generate valid protocol messages. Such tools usually define an “alphabet” of messages that they are able to generate. An implementation of an alphabet consists of functions for generating each message in the alphabet. Sometimes, tools also define an alphabet of message *fields* (a.k.a. message “components,” or “blocks” in a message) and rules for combining individual fields in ways that generate valid messages. Being able to generate valid protocol messages helps protocol-specific testing tools trigger more bugs.

One of the simplest ways in which protocol-specific testing tools use the ability to generate valid messages, is to generate different seeds for a fuzzer, in order to make the fuzzer generate many invalid variations of a particular valid message, as has been described in the previous section. For example, by giving a valid “Heartbeat” message for a fuzzer to use as a seed (instead of, for example, a “Client Hello” message), a testing tool can increase the chances that a fuzzer generates a message that triggers the bug in Figure 2.2.

2.2.1 Fuzzing message fields

While generic fuzzing tools are good at generating variations of a message by fuzzing the whole message, for some types of messages this only ever leads to the generation of invalid messages. To find some bugs, a protocol-specific testing tool must be able to generate different valid variations of a message as well. To see why this is important, consider Figure 2.3. Part a) of the figure shows an example message format that is difficult to generate correctly. One field of the message contains a string of bytes, and

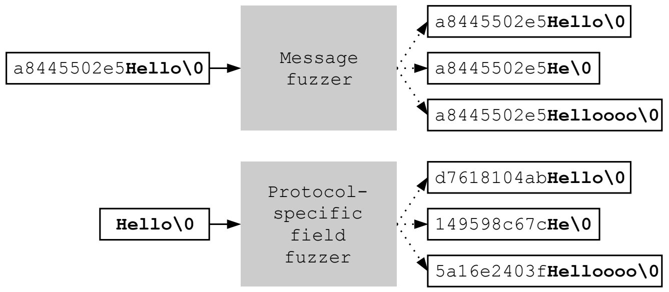


Figure 2.4: Different between how whole-message fuzzing (top) and protocol-specific fuzzing (bottom) generate messages to test a target. The "bytes" field is highlighted for readability.

another field contains a hash of the bytes. The hash is taken using a nonce value that is computed anew every time the target program executes. The target application sends a new nonce to the testing tool every time it is executed. Part b) shows the target program's message processing code, which reads the "bytes" field into a buffer `buff`. This processing code contains a basic buffer overflow vulnerability: if the "bytes" field is not `NULL`-terminated, the target will keep reading the data in the packet, and will overwrite past the end of `buff`. Part c) shows how this `process` function is called in the code. First, the target checks the hash in the message with the expected value `test`. If the hash matches, the target processes the message data using the vulnerable function.

In this example, only a valid message can trigger the bug, since only a valid message can pass the hash check. For example, the message `a8445502e5Hello\0` is valid, since the hash value contains the word `Hello\0` as well as the latest nonce. The message `5a16e2403fHelloooo\0`, according to the specification in part a) of Figure 2.3, is still valid, since the hash is updated to reflect both the new value and the new nonce (as is evident from the different hash). However, the fact that the "bytes" field is longer than 5 bytes (i.e., is `Helloooo\0`) causes the buffer to overflow. Indeed, an effective protocol-specific testing tool should be able to generate different valid messages, which deviate from the seed, but which can trigger bugs in the code after the message's validity (here, the hash check) is confirmed by the target.

Thus, to generate different valid messages, some protocol-specific tools [18][68][84] first fuzz individual message fields and then reconstruct valid messages around the mutated values before sending the valid messages to the target. This approach helps these tools test code that executes after a message is checked for validity. The top of Figure 2.4 shows an example of how whole-message fuzzing fails to generate messages that pass the check in the target, since the hash is never updated to a valid value. On the other hand, fuzzing individual fields and re-generating a message on each execution, as in the bottom of Figure 2.4, helps generate valid messages that are different only in their contents, but are otherwise valid in their form. Other situations for when this fuzzing approach helps generate valid messages include encryption and compression, both of which are also frequently used in stateful network protocols.

Of course fuzzing message fields, too, can sometimes result in the generation of invalid messages. For example, the bug in Figure 2.2 can also be triggered by a message that was generated by only fuzzing the length field of the Heartbeat message, while keeping the rest of the message the same. It is up to a developer of a testing tool to specify how far away from validity the messages that it generates can deviate, and, usually, testing tools generate both valid and invalid messages when fuzzing message fields.

2.2.2 Reordering messages

Finally, protocol-specific testing tools also use the ability to generate valid messages to change the order of messages sent to a target that uses a stateful protocol. The reason why protocol-specific testing tools implement this ability is to be able to test different states in a target application. As shown in Figure 1.1 in the previous chapter, some code in a target application can only be tested when a target is in a particular state.

One way in which testing tools make use of the ability to generate valid messages, is by generating these messages in non-standard order to systematically find states in a target application [26][33][86]. In this approach, a testing tool sends different messages from its alphabet to the target in order to deduce, based on the target's responses, the state machine of the target's behavior. Each new non-standard message sequence that such a tool generates is constructed so as to test its understanding of the target's state machine and refine its learned model. This approach is used to find all possible extraneous states in a protocol implementation, given some input alphabet of messages. These tools, however, are mostly used to discover bugs in the implementation that are not low-level security vulnerabilities.

The second approach used by testing tools is to generate valid messages in *random* non-standard order so as to test how a protocol application responds to different combinations of messages [52][84]. This is a less systematic version of the previous approach, since the target's responses are not analyzed to build a complete description of the target's behavior, and since message sequences are constructed randomly from an alphabet of known messages. Instead of finding all possible states, this approach helps discover subtle bugs that are due to how a target application's state changes when multiple messages are received. Furthermore, this approach helps protocol-specific testing tools find new states in the target that they can then test through the fuzzing approaches discussed earlier. By combining the ability to generate messages in different order with the ability to fuzz messages and message fields, testing tools can find bugs in a target's code that only executes when the target is in a specific state.

2.3 Detecting faults in network applications

A challenge that all protocol-specific testing tools face is detecting when a bug in the target application has occurred, which is also known as "fault detection." While many fault detection techniques are available for programs that are under the control of the tester (some of which will be discussed in Chapter 3), the most general ways for fault detection when testing network protocol applications rely on observing the network behavior of the targeted program [18]. For example, by observing the status of an underlying stateful network layer like TCP (while testing the implementation of a protocol at a layer above TCP), a connection that is closed unexpectedly (rather than through the proper close sequence) is indicative of a potential bug in the target application. If the target stops responding to new connection requests it is likely that it has crashed outright. If an underlying protocol does not maintain state (e.g. UDP or HTTP) a target can be observed by sending "heartbeat" requests after every exchange of messages between the testing tool and the target. If the target is unresponsive, it is, again, an indication that there might be a bug in its implementation. The "heartbeat" request can be any message that invokes a response from the target, and does not have to be a dedicated "heartbeat" message from the protocol being tested. Although some targets can be easily restarted to reset state after every re-execution, targets which are slow to restart can be restarted only occasionally, and targets that are not under the control of the tester have to be left running until a fault occurs. In this case, all the logs of messages that have been sent by the testing tool to the target since the last re-execution of the target can be kept to improve detection of the root cause of the bug.

Some protocol-specific testing tools further improve fault-detection rates by checking response values from the target. One manual way that tools achieve this goal is by defining conditions that must hold true when certain messages are sent [47][84], which usually allows testing tools to verify how closely implementations follow important or ambiguous parts of the network protocol specifications. Another method for fault detection in network protocols is called "differential testing." Multiple targets are tested simultaneously with the same message sequences, and, if the targets respond differently to the same messages, a tester investigates this unusual behavior [12][21][26][83] to see if it may be indicative of a bug in the implementation. It is not one of the goals of our work to automatically generate protocol-specific fault detection techniques. Rather, we use generic fault detection methods such as the network monitoring method described above, as well as other known fault-detection methods used with targets whose binary or source code is available. We will discuss existing fault detection methods that are available to us, in Chapter 3.

2.4 Types of network application targets

There are three general kinds of target applications that a protocol-specific testing tool can encounter: black-box, gray-box, and white-box targets. Since the usage of these terms varies, we define these terms here, and then further discuss testing techniques applicable to each target type in the next chapter.

Black-box targets are target programs that can be reached only by sending network messages but nothing is known about their internals. Sometimes black-box targets can be restarted by the tester (e.g. by physical interaction or through an alternative interface), but sometimes the targets that protocol-specific testing tools target are completely inaccessible to the tester [47]. *Gray-box targets*, on the other hand, are programs whose binary is available during testing. These targets are frequently easier to test. For example, a binary that is meant to run on a general-purpose OS and is compatible with the CPU architecture of the tester's machine, can be easily run, observed, and analyzed by a testing tool. If a binary is compiled for an architecture that is supported by an emulation utility [7] or by a dynamic binary instrumentation tool [13][57][63], the behavior of the target program can be more closely monitored or even augmented to produce additional output, improving a testing tool's understanding of the target's behavior for purposes of fault detection and input generation. Finally, *white-box targets* are those for which the source-code is available. White-box targets give a tester all the information necessary to apply techniques used for black-box and gray-box targets, as well as to use methods that rely on analyzing source code or by augmenting the target application during compilation.

We built InSource to be able to target any black-box, gray-box, and white-box network protocol applications that are running and can be communicated with. However, we do make use of some additional known techniques that become available when either the binary or the source-code of the target application is available. The only hard constraint that is faced by our testing technique is that the *source* program that is used for test generation must be white-box, to allow InSource to augment it during compilation.

Chapter 3

Related Work

Manually-constructed protocol-specific testing tools are built to be able to test all targets (i.e., black-box, gray-box, and white-box), so to make InSource a viable alternative to manually-constructed tools, we build it with the same goal in mind. However, if a protocol application's binary or source code is available, improvements can be made to how a protocol-specific testing tool generates tests for the target application. Thus, we discuss how our testing methods compare to various existing approaches that are used for testing each kind of target, and discuss how each approach can be (or is already) used in context of testing network protocol applications.

3.1 Testing black-box targets

Black-box targets are those for which the tested application can be executed and communicated with by the testing tool, but for which no information is known about the internals of the application's implementation. Blind fuzzing, which has been introduced in Section 2.1 is a generally-applicable technique for testing black-box targets. Despite the coarseness of their approach, blind fuzzers like Radamsa [43], which don't even take into account the structure of inputs that they generate, are able to discover numerous security bugs in real software. However, in the context of testing black-box network protocol applications, these fuzzers are less effective. Specifically, these fuzzers are not able to generate any valid messages, if the message structure is complex. For example, a blind fuzzer would fail to generated any messages that are accepted by the target code shown in Figure 2.3.c. As a result of this limitation, these fuzzers are also not able to generate non-standard sequences of valid messages in a stateful network protocol.

3.1.1 Protocol-specific black-box testing tools

Because of the limitations of blind fuzzers, protocol-specific testing tools are used to test applications that implement network protocols. Protocol-specific testing tools take into account the structure of individual messages and are able to mutate individual fields. They are also capable of sending messages in non-standard order to a target, and each tool can generate any valid messages that are available in its alphabet of messages. Some protocol-specific testing tools are built by taking existing network testing frameworks, such as Peach [65] or boofuzz [68], and describing network protocol structure and message format using a protocol "grammar." Once a grammar is used to describe message structure, the framework can mutate individual fields (e.g., through fuzzing), as well as add, remove, and reorder the fields while preserving each individual field's correctness. For example, Peach's grammars are described in configuration files that define the structure of network messages using primitives provided by the framework, while boofuzz provides an API of standard message components that a human tester has to call in order to inform boofuzz about what message fields are used by a protocol.

The idea of using grammars for testing network protocols likely originates in the use of formal grammars to test programming language compilers, an idea which was first applied in the 1970s [41], and which was more recently used to find many bugs hidden deep in the functionality of C compilers [100]. The intuition used for testing programming language compilers and testing network protocol applications is the same: constructing valid messages can help prevent the target from rejecting inputs, and thus allow a testing tool to find bugs that are related to how valid inputs are handled in the application. However, while grammar-based tools in other domains frequently rely on "formal" grammars to describe inputs due to their ability to characterize recursive structures, such grammars are not well suited for the unique complexities of network protocol messages, which can include hashed/encrypted fields, inter-message dependencies, and multiple chained messages, but which don't frequently include long recursive constructs. Thus, frameworks for building protocol-specific testing tools using grammars rely on special grammar formats that aren't frequently seen in other domains. Furthermore, protocol testing frameworks are able to generate sequences of messages and test the target with different non-standard message orders in addition to being able to test individual messages.

Despite the abundance of grammar-based network protocol testing frameworks (such as Peach [65], boofuzz [68], and others [1][5][60][87][91]), common limitations of these frameworks prompt testers to implement protocol-specific testing tools by hand. Specifically, the general primitives provided in each tool's grammar format are not sufficient for describing unusual fields present in more complex network protocols, and often don't give enough flexibility that would allow testers to describe more complex protocol interactions. For example, all of the tools that we are aware of for testing TLS targets, implement the TLS protocol from scratch [9][26][46][84]. While it would be intuitive to suppose that using an API of an existing protocol library for development of protocol-specific testing tools would speed up the development process, the coarseness of the APIs provided by network protocol libraries make it difficult for developers of protocol-specific testing tools to leverage them for fine-grained testing, such as mutation of individual message fields or reordering of messages during the TLS handshake. If testing tool developers do take the route of leveraging an existing implementation, they find themselves making significant changes to the implementation by hand. For example, LTEFuzz [47] is created by manually analyzing a 90k LOC[1] open-source implementation of the LTE protocol stack and inserting 3.5k lines of code that allow it to generate messages in non-standard order. GSMFuzz [92] takes the same approach but relies on a different open-source application in order to be able to generate variations of valid messages for testing nodes in a GSM network. Sometimes, a combination of approach is taken to build a protocol-specific testing tool. Fuzzers like AIM-SDN [30] and DELTA [52] augment the behavior of network nodes in an SDN by calling APIs exposed by the nodes but have to resort to intercepting and rewriting individual message fields on-the-wire while they propagate through the network, requiring them to implement protocol-specific parsing code that can be used to parse and recompile each message.

While there are multiple existing ways to approach the problem of creating protocol-specific testing tools, they are all marred by the need for substantial manual effort to be put in by the tools' developers,

[1]Lines of Code (LOC)

which is repeated for every new network protocol that they target. We built InSource to alleviate the need for manual effort when creating protocol-specific testing tools for complex, stateful network protocols. Our approach takes inspiration from the existing manual approaches for testing black-box network protocol applications. For example, just as tools like LTEFuzz and AIM-SDN manually convert existing network protocol applications into protocol-specific testing tools by leveraging their existing ability to generate valid messages, we *automatically* apply compile-time instrumentation to an existing source applications in order to allow a tester to use it to test a target in a protocol-specific way. Similarly to how grammar-based testing tools mutate individual message fields through fuzzing, we use existing fuzzing techniques at the granularity of message fields while the message is being constructed by the source application, allowing us to generate many different valid messages for the target. Furthermore, we borrow mutation techniques used by blind black-box fuzzers like Radamsa [43] in order to fuzz each individual message field; however, instead of using a purely black-box fuzzer, we leverage the fuzzing functionality of AFL [102], which is more capable at making use of information provided by gray-box and white-box targets (which will be discussed later in the section), but works just as well as Radamsa in black-box settings. To generate new messages and modify message order, we change the behavior of the source application such that it executes different message-generating code, instead of requiring a tester to manually implement functions that describe how new messages can be constructed. Unlike AIM-SDN and DELTA, we don't leverage APIs exposed by a source application in order to generate messages for a target because, while APIs allow the generation of some interesting standard message sequences, they are usually too coarse-grained to, for example, reorder messages that are sent during a protocol handshake.

3.1.2 Automating the process of testing network protocol applications

While testing tools described in the previous section either manually augment existing network protocol applications, manually build the protocol functionality from scratch, or manually define grammars that can be used by frameworks to test a target application, some effort has been made to automate parts of the process of building protocol-specific testing tools. For example, in order to reduce the amount of manual effort required to generate grammars that could be fed to grammar-based protocol testing frameworks such as Peach, researchers have sought to develop methods to automatically reverse-engineer the grammar of network protocol messages by leveraging existing network protocol applications, which either generate or parse network messages. PI [6] takes the simplest approach of extracting the possible values of message fields by overlaying different network traces generated by a real network protocol implementation and observing which substrings in the messages overlap. Discoverer [22] takes a similar approach, but uses knowledge about patterns commonly seen in network protocols (e.g. length fields) to deduce message structure from network traces. Pulsar [35] observes network traces of messages exchanged between network nodes to construct a state machine describing protocol message order. Among these tools, Pulsar is the only one that attempts to automatically reconstruct properties of a *stateful* protocol. Polyglot [14] takes the unique approach of observing the binary of a program that can parse protocol messages in order to deduce how different parts of the inputs are deconstructed. AutoFormat [55] and Tupni [23] further refine the heuristics proposed by Polyglot.

While most of these tools work well for reverse-engineering grammars of simpler protocols, they don't meet the challenges presented by messages that are used in more complex stateful network protocols, which can include encryption and hashing, inter-message data dependencies, or fields that change every time a protocol application is executed. Indeed, neither of these tools would be able to automatically create grammars that could be used to generate valid messages for the code shown in Figures 1.2 and 1.3. Since generating exact grammar descriptions for complex stateful protocols is a difficult task, instead of relying on an intermediate grammar or state machine representations of a protocol, we directly augment a real open-source application that implements the protocol and use it for fuzzing a target. This allows us to side-step the complexities involved in effectively employing grammar-based fuzzers and instead allows InSource to start testing a target application without first having to understand the intricacies of the protocol that it uses. Still, when no open-source implementation of a network protocol application is available, and when the target being tested does not implement a complex protocol, tools for reverse-engineering grammars are a viable alternative to InSource.

Only several other tools have taken the approach similar to ours, which relies on directly modifying the behavior of a source application to test a target. IoTFuzzer [18] is a tool that shares the most similarity with our work. This tool targets IoT devices by leveraging their "companion apps," which are Android application that communicate with the IoT device using a shared protocol. Instead of reverse-engineering the network protocol or creating manual testing tools for every IoT target, IoTFuzzer identifies inputs to functions in the companion apps that can be mutated before a message is constructed and sent from the app to the IoT device. The main benefit that the authors highlight from taking this approach is being able to overcome encryption performed on message payloads, allowing the encrypted data to be accepted and correctly decrypted by the targeted IoT devices and thus be used for testing further processing code. Because of the effectiveness demonstrated by IoTFuzzer's approach of generating valid messages (as well as invalid messages that only slightly differ from valid messages), we incorporate an adaptation of this tool's technique when implementing message field fuzzing in InSource. Thus, InSource also fuzzes the function arguments in the source application, allowing the source application to then construct messages using the fuzzed values and send them to a target, akin to how a "companion app" used by IoTFuzzer generates messages that get sent to an IoT device. Both IoTFuzzer and InSource require some human input to identify the function arguments to fuzz, but the effort needed in both tools is not substantial. While IoTFuzzer can effectively generate valid and invalid messages by using a source Android application to test a black-box IoT device target, it does not test implementations of stateful network protocols. Instead, IoTFuzzer targets the application-level code that is executed regardless of the application's state, and for which encryption and hashing checks are the only obstacle. On the other hand, InSource is able to test stateful network protocol applications and can use a source application to send messages in non-standard order and thus toggle different states in the target. In context of testing IoT devices, the approaches used by both tools would be complementary to one another. IoTFuzzer could be used to test application-level code on the IoT device, for which the companion app is the only other application that implements the protocol. InSource could then be used to test all the other network protocols in the target's stack, for which white-box source programs are available.

Another tool that takes a similar approach of modifying a source application to test a target in a format-aware way is MutaGen [45]. While MutaGen is not explicitly built to test network protocol applications, this tool also leverages a source application in order to test a target: it identifies a slice of a source program that is responsible for generating an output and mutates the instructions in this slice in order to influence the output generated by the source program. The source program's output is then fed as an input to the target application. This approach allows MutaGen to generate many more valid variations of messages, compared to a blind fuzzer. MutaGen's modifications to the source program are substantially different from ours: MutaGen intentionally tries to keep the control-flow of the source program intact, while only influencing the output string that the source program produces. On the other hand, we intentionally change the control-flow of the source application, causing it to execute entirely new functionality and generate non-standard message sequences consisting of different new messages. Furthermore, MutaGen applies mutations at an instruction-level granularity, emulating the kind of changes that may be performed by a fuzzing engine, but applying these mutations to individual instruction operands. On the other hand, we fuzz the contents of individual function arguments. While we could have implemented MutaGen's approach as a technique for generating different valid messages, fuzzing function arguments makes more intuitive sense in the context of network protocols (since message fields are often directly passed in with function arguments), and function argument fuzzing can be more easily integrated with more mature fuzzing techniques.

3.1.3 Fault detection

In addition to being able to generate invalid messages as well as valid messages in non-standard message sequences, an effective testing tool has to be able to detect when a bug has occurred. The most common method for identifying when a network protocol target application has crashed is by monitoring it over the network. IoTFuzzer [18] and a fuzzing tool implemented by Muench et al. [61] use such an approach to detect bugs. If a connection is dropped by a lower layer protocol like TCP or if the target stops responding to any messages, the tools detect that the target has crashed. This is the approach that InSource takes when it is used to test black-box targets.

Since not all bugs lead to a crash, some other black-box protocol-specific testing tools [47][84] manually implement checks for expected target behavior, and detect any instances when an assumption about how the program should have responded to different valid messages or non-standard message sequences doesn't hold true. To reduce protocol-specific knowledge required for implementing checks manually, some tools use differences in behavior between different applications in order to detect when one of the targets contains a bug. This approach has been used to effectively detect non-crashing bugs in TLS implementations of certificate parsing code [12][21] as well as hostname verification code [83]. We think that this is an interesting future direction for extending how InSource could be used to detect faults that do not lead to easily-detectable crashes.

3.2 Testing gray-box targets

While protocol-specific testing tools can effectively generate messages and non-standard message sequences to test black-box targets, some testing techniques can generate effective testcases by taking into account the knowledge about how a target application operates instead of employing knowledge about the format of a network protocol. Specifically, gray-box targets, which give a testing tool access to the program binary, allow a testing tool to observe how different mutations of a test input affect the target, as well as to take into account the target application's control-flow structure when constructing test inputs. Because testing approaches that work for gray-box targets are usually implemented in a general way, protocol-specific testing tools can sometimes also benefit from these approaches.

3.2.1 Testing techniques

While testing black-box targets, unless a testing tool observes a crash, it has no feedback mechanism by which it can determine whether or not one message that it sends to the target is better at exercising the program's code compared to another message that it sends to the target. In a gray-box setting, however, a testing tool can use this information to, for example, save one message that it has generated through fuzzing, and use it again as an initial *seed* for further mutations. As a reminder, a seed is an input that is usually provided by a tester, and is mutated by a fuzzer before being inserted into a network protocol message which gets sent to the target. By saving good mutations that exercise new functionality in the target, and by using those mutated inputs as future seeds for the fuzzer, a testing tool can *evolve* the initial seed through multiple mutations. By evolving the seed input, a fuzzer is able to gradually learn inputs that can be used to test more code in the target application compared to a black-box fuzzer. This has been proven to be an effective technique for improving the bug-finding performance of fuzzers that target gray-box programs, as is evident from the fact that all fuzzers used by OSS-Fuzz [79], a large-scale fuzzing program that has found over 9000 bugs in the first 2 years of its existence [76], use evolutionary fuzzing techniques.

To differentiate good mutations that the fuzzer should keep as future seeds from bad mutations that can be discarded, testing tools use coverage information about the target program. Specifically, by checking which parts of the target application execute (i.e., are "covered") when an input is sent to the target, a fuzzer is able to determine if the mutations that it performs reveal any new functionality in the program. Fuzzers can use different techniques to gather coverage information from gray-box targets: honggfuzz [44] makes use of Intel Processor Tracing (Intel PT) hardware support to gather coverage information, while AFL [102] emulates a program binary's execution using QEMU. Although coverage guidance can help gradually evolve an initial seed into testcases that can trigger new code in the target, coverage-guidance is not a panacea: evolutionary guidance cannot, alone, allow a fuzzer to generate significantly new input types, since even an evolutionary fuzzer performs only simple mutations when fuzzing seeds. Indeed, recent tools for testing file parsing applications [3][94] showed that combining domain-specific valid test-generations techniques with evolutionary fuzzing can boost the amount of tested functionality and thus the number of bugs discovered in the target application in comparison to evolutionary fuzzing alone. In context of testing network protocol applications, evolutionary fuzzing can be combined with protocol-specific valid message generation methods, by guiding mutations to individual message fields. Because InSource uses AFL as its underlying fuzzing engine, it is able to support evolutionary coverage-guided fuzzing of gray-box targets. However, it does not rely on evolutionary fuzzing for discovery of entirely new message types, and uses a source application's implementation of a protocol for this instead. Furthermore, to generate different variations of valid messages, InSource employs evolutionary fuzzing by fuzzing individual message fields. Fuzzing of message fields does not guarantee that InSource generate only valid messages, but rather helps it generate more valid messages than a standalone fuzzer would.

A technique that is quite different from fuzzing but that has also been shown to be effective when testing gray-box targets is symbolic execution. Symbolic execution can be used to generate exact inputs to target binary programs by analyzing their control-flow. Symbolic execution tools like angr [82] use this technique to generate test inputs by accumulating input-dependent condition values along the control-flow path of the target and deducing exact inputs that can be given to the target program to make it execute specific paths through the code. In theory, this approach seems superior to fuzzing in the context of testing gray-box targets: by being able to analyze the control-flow of the target, a symbolic execution tool can generate inputs to the program that cover all input-dependent parts of the code, as well as deduce exact input values that trigger bugs. In practice, however, symbolic execution tools suffer from considerable drawbacks. When compared to fuzzing, two of the main drawbacks are the slow speed of symbolic execution tools and the problem of path explosion [4]. Unlike fuzzers, which are able to quickly generate and try many inputs to the target, a symbolic execution tool has to emulate binary instructions while also keeping track of relevant conditions in the code. While this is already slow, a symbolic execution tool then invokes a constraint solver, which is computationally intensive. Furthermore, because symbolic execution tools try taking every path through the target when generating constraints, the number of paths that a symbolic execution tool computes grows exponentially with every new branch that the tool encounters, resulting in "path explosion." In practice, therefore, symbolic execution is ineffective when applied to large programs, and thus would also not work well on complex real-world protocol applications.

To alleviate the poor scalability of symbolic execution, many tools take an alternative approach. They first concretely execute a target program with a specific input while recording every condition that is satisfied along the way. Then, by systematically negating various conditions, they use a constraint solver to generate variations of the initial input that are guaranteed to take slightly different paths through the target. While not as thorough, this approach is effective, as has been demonstrated by the bug-finding ability of SAGE [37]. An important insight gained from this approach is that symbolic execution tools can be very effective at helping improve the effectiveness of fuzzers. While fuzzers can effectively generate many concrete inputs, they are sometimes unable to generate inputs that satisfy certain constraints in the target. For example, if the target programs has the check `if(num == 0x9a23be02)` (this kind of predicate is often called a check for a "magic value"), a fuzzer that is given a seed where the "num" field is set to `0x00000001`, will be unlikely to generate a mutated input that contains the exact byte value required by the check. On the other hand, a symbolic execution tool will be able to quickly solve this constraint, and allow the fuzzer to continue performing mutations on other parts of the input. The most notable work to combine symbolic execution and fuzzing is Driller [85], although other tools have also taken a similar approach [39][64]. By using symbolic execution only when a fuzzer is unable to satisfy a challenging constraint, and bootstrapping the symbolic engine with concrete inputs that are already able to propagate deep into the program, Driller is able to outperform either approach in isolation, and find even deeper bugs in the code. A recent tool, Savior [20], further minimizes the time spent executing heavy-weight symbolic execution by automatically prioritizing certain parts of the target binary as more important for testing, and using symbolic execution only for the purpose of directing fuzzed inputs closer to those parts of the code.

Still, despite this way of using symbolic execution, it is still a bottleneck in the testing process. In an effort to further speed up the computationally-heavy constraint solving tools used in symbolic execution, tools such as NEUZZ [80] have invented ways to approximate the very precise operation of constraint solvers by instead applying the faster gradient-descent algorithms often used in machine learning. By representing rigid constraints instead as smooth functions, a gradient descent algorithm can be used to generate an approximate solution to a set of constraints, but do so quickly, thus not causing a bottleneck in the input generation process. Another way in which tools have bypassed symbolic execution in order to satisfy difficult constraints, has been to track an input as it propagates through the program and to correlate fields in the input with specific predicates in the target's code. VUzzer [75] uses this information to prioritize fuzzing of different parts of the input, as well as to set input fields to the correct "magic values." Angora [19] is an extension of this approach, which tracks inputs at a finer granularity, and also tries to solve predicate conditions by applying gradient descent rather than symbolic execution, to avoid the high overhead of the latter approach. The goal of these fuzzers remains the same, however: to generate inputs that are not immediately rejected by a program and find bugs that are hidden deeper in the code.

All of these tools significantly outperform evolutionary coverage-guided fuzzers when testing gray-box targets because they thoroughly use the information about the target application that is encoded in its binary. Specifically, these tools are able to generate test inputs that are tuned to the conditional checks that a target application performs, allowing these tools to bypass the requirement of understanding the format of the inputs that they are generating. In fact, it is likely that modern gray-box testing tools could outperform protocol-specific testing tools for *simple* network protocols, as protocol-specific testing tools can only take into account the structure of a protocol messages and not the structure of the target application's code. Unfortunately, gray-box testing methods do not help us solve some of the fundamental challenges of testing *complex* and *stateful* network protocol applications. First, while gray-box fuzzers are excellent at generating inputs that test more different parts of the code, they, just like other fuzzers, perform mutations given an initial seed and thus don't re-adjust message fields such as sequence numbers or nonces, which change with every execution of a stateful network protocol target. Second, methods for computing inputs in order to influence which predicates in the target program are satisfied, don't work well for very complex messages fields like hashes and fields that are encrypted or compressed. Because of these two constraints, gray-box fuzzers are intrinsically unable to generate non-standard message sequences that consist of valid messages that pass complex checks in the target program code. As such, gray-box fuzzers would be the most effective when used in conjunction with protocol-specific testing tools that can generate such message sequences, since, in principle, gray-box fuzzers can be used in lieu of other fuzzing engines to fuzz individual protocol message fields. We leave the work of experimenting with alternative fuzzing engines as future work, since, although gray-box fuzzing techniques could be effective at fuzzing message fields, they are only applicable to gray-box targets, which represent only a subset of targets that can be encountered by our tool.

There is another class of gray-box fuzzers that more bluntly addresses the complex checks that can be present in a target program. While most gray-box fuzzing tools attempt to change inputs in order to satisfy difficult constraints in the target code, several fuzzers have taken the approach of dynamically modifying program binaries of the targets themselves in order to force control-flow paths to skip difficult constraints entirely. Taintscope [96] is a relatively early work that took this approach by using taint tracking to identify where hash checks are performed by correlating portions of the input to branches in the code. After locating these hash checks in the target, Taintscope skips them by dynamically instrumenting the target binary. Note that VUzzer [75], which was discussed earlier, used taint tracking to identify correct values to use for "magic number" checks, and would not modify the target binary. More recently, T-Fuzz [67] iterated on the technique used by Taintscope. The tool observes program coverage during fuzzing and identifies any branches that the fuzzer gets stuck on (i.e., is unable to circumvent by simply mutating the inputs). When applied to network protocol targets, both of these approaches would likely over-approximate what counts as a check for a hash field in a target application, resulting in important protocol-related processing code like encryption and decompression in being skipped as well. Furthermore, both approaches inadvertently introduce false positives into the fuzzing process, since they augment the target's behavior.

Still, there are two interesting avenues that stem from these dynamic approaches, which are specific to testing stateful network protocol applications. First, by dynamically instrumenting the target program it may be possible to overcome different *state* checks that are present in network protocol code (e.g., in Figure 1.1) that prevent testing tools from thoroughly testing these parts of the network protocol application. We postulate, however, that the amount of false positives introduced into the testing process by skipping state checks would be substantial since, unlike the *hash* checks that are skipped by T-Fuzz and Taintscope, which are only used for testing the integrity of the received message, state checks are indicative of whether a target contains enough internal information to be able to process an input. We make this guess based on our experience of changing control-flow of a source application while using InSource: even simple changes that cause the source application to ignore a state variable in a predicate often led to a crash. The second interesting avenue for applying the philosophy used by Taintscope and T-Fuzz is to overwrite certain field values while they are being processed by the target application (e.g., after decryption occurs) instead of while they are being generated by the source program. However, this approach could only be applied to gray-box targets, and would likely introduce false positive results into the testing process, so we did not investigate it during development of our tool. InSource currently does not modify the target application, and thus does not generate false positives.

3.2.2 Library API fuzzing

Sometimes, target network protocol applications are available as libraries. Indeed, some of the targets that we test in our evaluation [59][73][101] are built using existing network protocol libraries. Libraries can be tested not only via inputs send as messages through the network layer, but also by directly testing the APIs provided by the network protocol library. Fuzzers like libFuzzer [54] are used to test individual library functions as long as a tester provides a manually-constructed wrapper program that interfaces with the fuzzer. The fuzzer IMF [40] automates the process of constructing wrapper programs for fuzzing library function by observing how real applications use the tested library calls. While such tools can be used to individually test the interface between library functions of a network protocol library and the program using these functions, the bugs that these tools expose will likely be complementary to those found by testing these library functions from the point of view of the network. Since we are looking for bugs that can be exploited by a malicious actor which accesses the target application via a network interface, we do not seek to find a way to fuzz individual library functions. Furthermore, we seek to fuzz entire target applications, and not individual features of the underlying libraries.

3.2.3 Fault detection

Observing crashes is easier in gray-box targets because such targets can usually either be run in an observable process on the host OS, or emulated. Memory corruptions, for example often lead to segmentation faults and other signals that directly notify the testing tool about a bug. Furthermore, dynamic binary instrumentation can be used to change the behavior of the target program in order to observe how it manages memory, in order to detect low-level software vulnerabilities caused by memory mismanagement that don't necessarily lead to a crash. For example, Valgrind [63] has built-in support for detection of some memory errors. In rare instances, a program binary can also be converted to a compiler representation [58] (i.e., "lifted") and recompiled with additional memory checks [77]. This is not always possible, however, meaning that any additional fault detection techniques used for testing a gray-box target have to be picked on a target-to-target basis by the tester. Therefore, when testing a gray-box target using InSource, we also consider the target at hand in order to select fault-detection techniques. In all cases, however, fault-detection methods increase bug discovery rates at the expense of some slow-down to testing speed.

3.3 Testing white-box targets

While many targets are inaccessible to a tester (black-box) or are only accessible in binary form (gray-box), in some instances the source code of the target application is available to the tester, allowing them to leverage the knowledge of the entire application, as it was written by a developer. We call these targets white-box targets. With the growing popularity of open-source software, the availability of source-code of a target application is becoming more likely, allowing testing tools to leverage various static analysis and compile-time instrumentation techniques.

3.3.1 Testing techniques

One interesting way in which a target's source code can be leveraged during fuzzing, is to allow fuzzers to prioritize testing certain parts of the target application. Specifically, while the fuzzers discussed previously perform changes to a seed input in order to generate inputs that trigger as much new code on the target as possible, "directed fuzzers" are used to automatically generate inputs that can exercise parts of the target's code that are specified by the tester. For example, AFLGo [11] uses compile-time instrumentation in order to track how close the currently-covered code in the target is to reaching a part of the code specified by the tester. AFLGo then prioritizes mutating those inputs which lead the application to execute closer to the specified code. Hawkeye [17] implements an improved version of this approach, by considering both short and long paths to target code, and using different metrics for code "closeness" when directing mutations to specific parts of the target program. While, in a stateful protocol target, these methods could be used to direct testing towards certain parts of the code that set state variables, these fuzzers would not be able to automatically deduce which blocks they would need to target in order to toggle different states in the application. Furthermore, just like fuzzers that target gray-box programs, directed white-box fuzzers would not be able to construct messages that pass hash checks, that satisfy variable message fields (e.g., nonces), and that get decrypted by the target correctly, limiting their effectiveness at being applied to generating non-standard message sequences in complex stateful protocols. Because InSource leverage a source application to generate messages, it can generate message sequences that include messages with complex message fields, and thus can be used to toggle new states in the target. Furthermore, InSource can direct execution in the *source* more easily than how directed fuzzers direct execution in a *target* because InSource takes full control over a source application's behavior.

Some existing gray-box fuzzing methods can benefit from increased control attained when having access to the target program's source code. For example, coverage-guided evolutionary fuzzers often support using compile-instrumentation as a way of recording coverage during fuzzing [44][54][102]. At compile time, fuzzers can either apply instrumentation that measures block-level coverage or edge-level coverage. Block-level coverage is used by libFuzzer [54], which merely counts the number of times basic blocks are executed. AFL [102] uses a fast approximation of edge-level coverage and CollAFL [34] uses a more refined edge-level coverage technique. Since we use AFL as the fuzzing engine in InSource, we can rely on this method of coverage-guided fuzzing when testing white-box target applications. Furthermore, since AFL inserts coverage in the *target* application and InSource inserts instrumentation into the *source* program, the instrumentation methods used by each tool do not interfere.

In addition to fuzzing, some other testing techniques that are applied to finding bugs in gray-box targets also have their white-box counterparts. Some of the earlier methods for symbolic execution operated on compiler representations of programs rather than fully-compiled program binaries. Notably KLEE [15], which is an early tool that has proven to be useful enough to still be used and supported to this day, can symbolically execute a program that has been compiled to the compiler representation used by LLVM. While symbolic execution for testing white-box targets still suffers from the problem of path explosion, a popular technique that has proven to be useful in practice has been to apply *under-constrained* symbolic execution to complex targets. The under-constrained variation of KLEE, UC-KLEE [74], was applied to testing a complex real-world network protocol application [101], and was successfully able to find bugs in this program. This was possible because UC-KLEE symbolically executes only individual sections of the target's code and not the entire application, effectively "skipping" complex checks in its analysis, such as checks of internal state variables or message hash fields. This means, however, that the tool generates many false positives, most of which would not be exploitable by real attackers targeting the application via network inputs. InSource does not generate false positives, although, at the expense of not being able to find as many bugs in a target application.

Another interesting application of symbolic execution may be to use it as a replacement for the instrumentation inserted by InSource into a source program. Specifically, symbolic execution could be used to make the source program generate different message sequences, by deducing which inputs a source program may be given in order for it to generate such sequences. InSource, however, does not take this approach because it attempts to generate not only message sequences that are intended by the program, but also non-standard message sequences that a source program cannot generate through normal execution (i.e., for which no corresponding program input exists).

3.3.2 Static Analysis

Another common automated approach to bug discovery in white-box targets is static analysis. Unlike all of the tools discussed previously, static analysis tools analyze source code of programs without ever executing or emulating the code, and instead search for patterns that are often correlated to bugs. Unlike symbolic execution, this approach scales exceedingly well to large programs, and can be used to find bugs in parts of the program that are difficult to test by constructing concrete inputs. Indeed, static analysis tools can easily be applied to testing any network protocol application for which the source code is available. The main drawbacks of static analysis, however, make it very unpopular with developers — many false positives are generated for every real bug found. The creator of Flawfinder [98], an early and popular static analysis tool, notes how developers often mislabel real bugs found by the tool, effectively negating the benefit that the tool provides.

Nevertheless, some techniques that static analysis tools use to analyze control-flow of a *target* have influenced how InSource analyzes control-flow in a *source* program, when trying to determine best ways to influence its behavior. For example, the approach used by the static analysis tool LRSan [97] for constructing a program call-graph of a target, has influenced the method that we use for generating the call-graph for the source application. Since LRSan generates the callgraph of the entire Linux kernel, it needs a fast, approximate technique to resolve function names at indirect call sites. Instead of performing careful pointer analysis as is done, for example, by Hawkeye [17], LRSan instead matches every indirect call site in a program to candidate called functions using function signatures. Since some source programs that we encountered are large and make heavy use of indirect calls, we perform the same coarse-grained analysis that is performed by LRSan in order to match indirect function calls to candidate called functions. We then equip InSource with the ability to force any function to be called at an indirect call site; thus, even though InSource over-approximates the set of candidate functions at each indirect call site, it can then force the function call to execute whichever candidate that it needs.

3.3.3 Fault detection

Perhaps the most significant benefit of being able to test a white-box target application is the ability to apply compile-time instrumentation that can be used for fault detection and thus significantly increase the number of bugs found in the target. For example, memory overflows can be easily exposed by instrumenting the program with AddressSanitizer [77], and any use of undefined behavior can be exposed with UndefinedBehaviorSanitizer [90]. While not all of the bugs that these tools will expose will be security vulnerabilities, all of the bugs that they expose will be real bugs.

As with fault detection techniques used for testing gray-box targets, not every white-box fault detection technique is useful for every target, so different kinds of compiler instrumentation can be applied depending on the kinds of application being targeted. For example, ThreadSanitizer [78] can be used to detect data races in multi-threaded targets, at the expense of introducing some false positives into the testing process. Nevertheless, compile-time fault detection techniques combine well with almost any testing tool and are extremely effective. Indeed, most testing tools recommend that their users at least enable AddressSanitizer during target compilation whenever a white-box program is being tested. We make the same recommendation for when InSource is used to test white-box targets.

Chapter 4

Design

We strive to achieve the ability to effectively test target applications that implement complex stateful network protocols, and to reveal low-level security bugs in these targets. To do so, we transform an existing source application, which already implements the protocol that is used by the target, into a protocol-specific testing tool. To measure our ability to exercise code in the target we compare our approach to both a blind fuzzer (which does not use protocol-specific knowledge) and to a manually-constructed protocol-specific testing tool. For the purpose of demonstrating our approach we run our tool with a client application as the instrumented source application, which is then used to test a target server; however, the approach can also be applied to test peers sending messages to each other over the internet, or to test any intermediate nodes in a network, as long as a source application that implements the same protocol as the target, is available.

4.1 End-to-end workflow

Before delving into the design details, we will briefly describe how InSource can be used for bug discovery in a target application. To use InSource, a tester (the user of our tool) needs to perform the following steps:

- Find a source application that implements the same network protocol as the target, and is able to initiate a connection and then communicate with the target application using this protocol. For example, a client can act as a source program for testing a target server.
- Compile the source application using the compiler provided by InSource to insert instrumentation that allows InSource to control the source program's behavior. Optionally, a tester can also manually specify files in the source application that are related to generating messages.
- Allow InSource to find out how the source application can be used to test a target by trying to modify the source application's behavior in different ways, so that the source application generates non-standard sequences of messages. The changes to the source program's behavior are protocol-independent, but the source program itself generates messages in a protocol-specific way. We call this step "state exploration," since by generating non-standard sequences of messages using a source application, InSource is able to toggle different states in the target.

- When InSource finds a change to a source application's behavior that generates a message sequence that elicits a previously-unseen state in the target, InSource generates different variations of every message in this message sequence. To do so, InSource first performs "fuzzing of whole messages," by overwriting entire messages after they are generated by the source. This approach helps generate different invalid variations of valid messages that are sent from the source to the target. If a tester specified files in the source application that are related to generating messages, then InSource also mutates message fields while they are in the memory of the source application. The source application then uses these new fields during message construction, and sends the changed messages to the target. We call this "fuzzing message fields," and use it to generate more different valid variations of messages to test the target.
- The source application is re-run multiple times by InSource, allowing it to generate many different test messages that get sent to the target. This continues indefinitely, or until the tester stops InSource.

InSource uses changes to the behavior of the source application to generate non-standard message sequences. There are, however, several alternatives ways to approach the goal of generating messages in non-standard order. We briefly discuss the pitfalls of the next-best alternative approach that we considered.

4.1.1 Motivation for our current approach

The alternative that we considered in our implementation of InSource was to simply intercept messages sent by the source application and to change the message order before the messages reached the target. This approach, too, can help generate non-standard message order, but it only works in simple scenarios. Consider, for example, some standard message orders that are shown in Figure 4.1, which are sent between a source application (colored gray) and a target application (colored black). In example a), messages `A`, `B`, and `C` have no inter-message dependencies, and are the same every time the source and target are executed. It is easy to, for example, replay message `A`, or to try preventing message `B` from reaching the target. Furthermore, in this case, it would be possible to reuse messages between each execution of a source and a target, allowing us to send message `C` before message `A`, by taking this message from a previous execution.

Consider, however, example b) in Figure 4.1. In this case, messages `B[y]` and `C[y]` both depend on state provided by the target in the message `Y`. Suppose that message `Y`'s content changes with every execution. Then, messages `B[y]` and `C[y]` are also different every time the source executes. While we could save `B[y]` and use it to replace `C[y]`, we would not be able use `C[y]` instead of `B[y]`, since we would not know what `C[y]` is until the source application generates it for us. We also cannot replace `B[y]` with `C[y]` from a previous execution of the source, since the message has to be constructed using the latest information provided by the target in the message `Y`. Furthermore, in both examples a) and b) in Figure 4.1, we would not be able to insert any new messages into the network that are not present in the standard message sequence sent by the source application already.

To circumvent the limitations of this approach of reordering messages in the network, we instead directly modify the source application to make it generate different messages in different orders. This allows us to produce a larger variety of non-standard message sequences that allow us to find more interesting states in the target application.

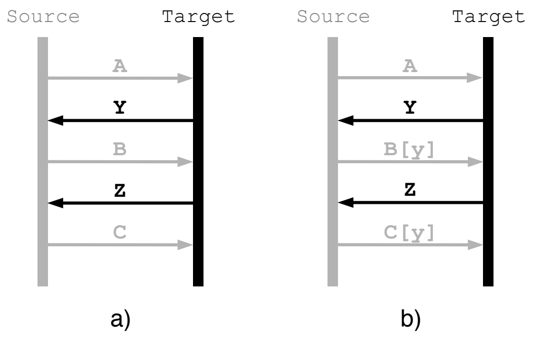


Figure 4.1: Standard message orders that can occur between a source and a target application. Generating non-standard message orders through network reordering is possible in example a) but not in example b), due to the dependencies of messages `B[y]` and `C[y]` on the message `Y`.

4.2 Controlling source application behavior

In order to be able to control the behavior of a source application for the purpose of generating different message sequences, InSource inserts compile-time instrumentation at every branch and at every function call of the program. When not enabled, this additional logic does not influence the behavior of the application, and allows it to execute normally. However, when one of the inserted instrumentations is toggled by InSource, it overrides the real behavior of the application, allowing InSource to dictate exactly what the program does at that point during execution.

The first kind of instrumentation that is inserted by InSource allows it to override which next basic blocks are executed at branches in the source application. We call this method of forcefully changing control-flow in the source application "forcing a branch," since, when directed by InSource, the source program ignores the predicate of the branch and instead "forces" the branch to the desired value. Consider, for example, Figure 4.2, in which our instrumentation changes execution in a branch from executing block `D` to executing `B` instead. Part a) of the figure shows the *default control-flow* of the application (highlighted in gray). The default control-flow is the sequence of basic blocks that are executed by the source application while it is executing normally, branching at `if(predicate)` according to the value of `predicate`. This is how the application behaves before it has been instrumented by InSource. Part b) of the figure shows the behavior of the application after it has been instrumented; however, in part b) InSource still does not interfere with the control-flow of the program, since `do_force ()` returns "false" by default, allowing the program to execute normally despite the additional logic. In part c), the branch is forced by InSource. Here, `do_force()` returns "true," indicating that the branch must now follow the control-flow specified by InSource. `force_to()` returns the value that InSource specifies, directing the next block that is executed. In this case, InSource dictates that the basic block `B` should be executed instead of `D`, allowing the control-flow to proceed along a non-default path. It is important to note, however, that the branch can also be forced to execute the block `D`. This is useful if, for example, the value of `predicate` is not yet known, so InSource simply ensures that this branch is run, even though this branch would have been taken anyway during normal execution. After basic block `B` finishes executing, the block `C` executes next, highlighting that forcing the branch causes a ripple effect on the execution of the source application. In diagrams later in this chapter, we will not show the additional inserted instrumentation, and will only show the basic blocks that are part of the original source program itself.

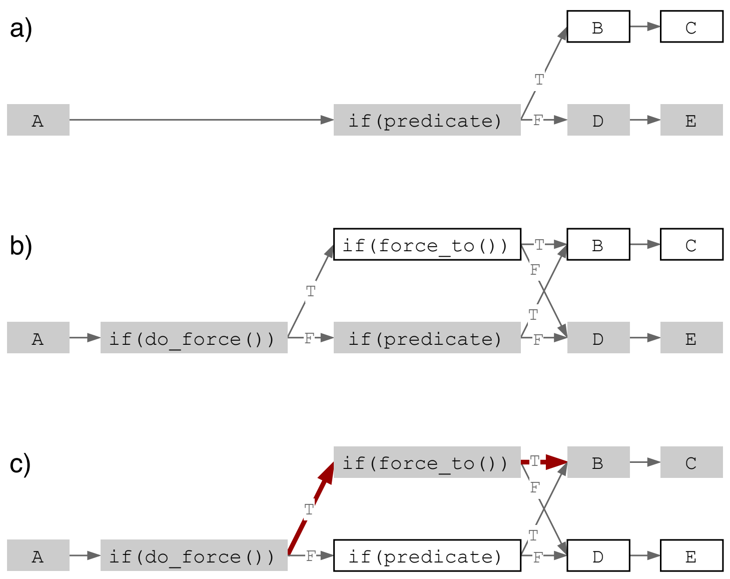


Figure 4.2: An example of a source application's control-flow being modified using instrumentation inserted by InSource. a) The default control-flow executes when no instrumentation has been inserted. b) The default control-flow is still executes when the instrumentation is inserted but is not toggled by InSource. c) An alternative path is taken, as dictated by InSource.

In addition to branch forcing instrumentation, InSource also inserts instrumentation at function calls. One part of the instrumentation inserted at function calls allows InSource to overwrite function call arguments, influencing the contents of the message that gets constructed by the source application. Section 4.4.2 discusses how this instrumentation gets used by InSource to overwrite individual message fields. Additionally, at *indirect* call sites InSource also inserts logic that allows it to change the function that gets called at the call site. We call this "forcing a call." This is another way by which InSource changes control-flow of the source application, and the usefulness of this approach will be motivated in Section 4.3.2. We don't present the details of function call instrumentations in this section because their implementations follow a similar pattern to Figure 4.2. First, the instrumentation checks if it should execute the default behavior of the program or the behavior specified by InSource. Then, it executes either the default code or the specified behavior.

4.3 State Exploration

By trying to modify the source application's behavior in different ways, InSource finds out how the source application can be used to test a target using non-standard message sequences. We call this step "state exploration," since by generating non-standard sequences of messages using a source application, InSource is able to toggle different states in the target.

4.3.1 Changing source application control-flow

To force different message sequences to be generated, InSource modifies the default control-flow of the source application so that the application executes along a different control-flow path. Consider, for example, the source application in Figure 4.3. In the top half of the figure, the unmodified source

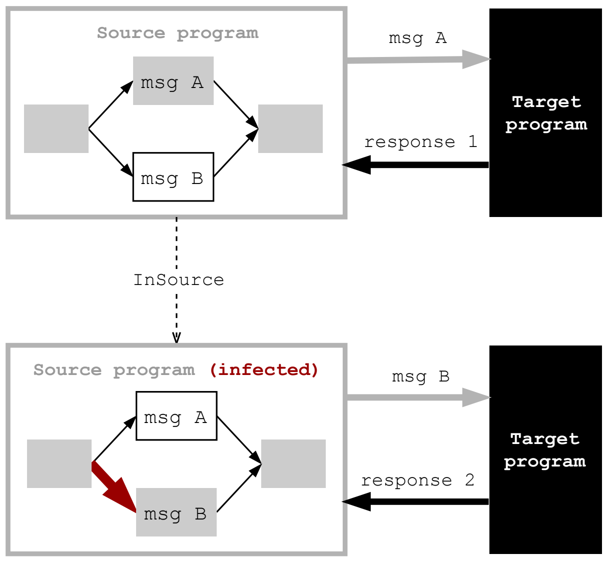


Figure 4.3: The overview of the operation of InSource. In a source program which has a default control-flow (top), which passes through code that generates "msg A," InSource can modify the default control-flow to send "msg B" instead of "msg A" (bottom). The different messages changes the internal state of the target, as is observed by its response.

application, during execution of its default control-flow, executes 3 basic blocks, one of which generates and sends `msg A` to the target program. By using an "infected" version of the program that has been instrumented by InSource, which gives InSource full control over every branch that is taken, InSource is able to generate and send `msg B` instead.

To make this approach more general, we allow chaining multiple branch forces together. Consider, for example, the real-world default control-flow observed in a source application, shown in Figure 4.4. To make this source application send the "plain" message instead of the "anon" message that gets generated as part of the default control-flow (highlighted in gray), InSource needs to force three branches: first it must force the branches that checks if `support == ANON` and if `support == SCRAMSHA1` to false, and then it must force the branch that checks if `support == PLAIN` to true, resulting in the block that makes and sends the "plain" message being executed. In this example, the "anon" message is only one in a sequence of messages that get sent by the source application; thus, by performing these branch forces, a different message gets placed instead of the original message, resulting in a non-standard message sequence being generated by InSource. Here, InSource can also generate the non-standard message sequence in which no message is sent in this step of the message sequence, by forcing all branches shown in the figure to false.

4.3.2 Forcing indirect function calls

When using branch forcing alone, however, we are not able to fully leverage the source application, since some of the messages that get sent by the source are only generated and sent when complex conditions are satisfied. Consider, for example, the program shown in Figure 4.5. Here, the default control-flow first assigns the address of the function `send_A` to the variable `s` and then calls `make_message`. However, because the condition `condition_2` evaluates to "true," the source program does not execute the function

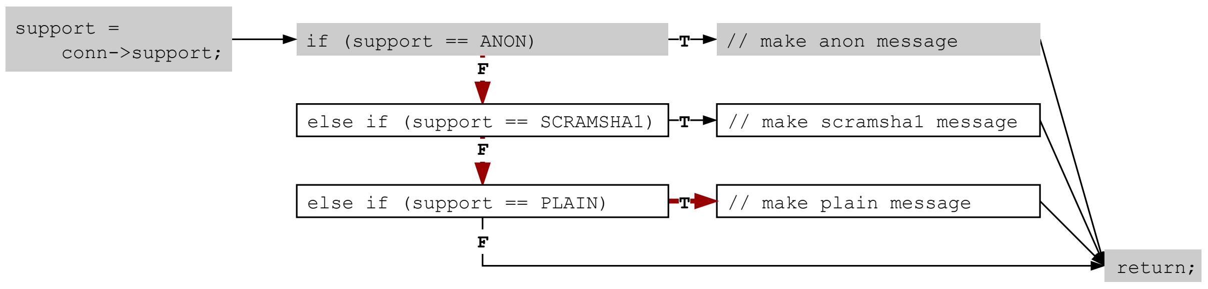


Figure 4.4: The default control-flow makes this source application execute the "anon" message. By forcing multiple branches using InSource we can cause a different message to be generated. The code shown is a simplified version of the code in the function `_auth` in the libstrophe library [59].

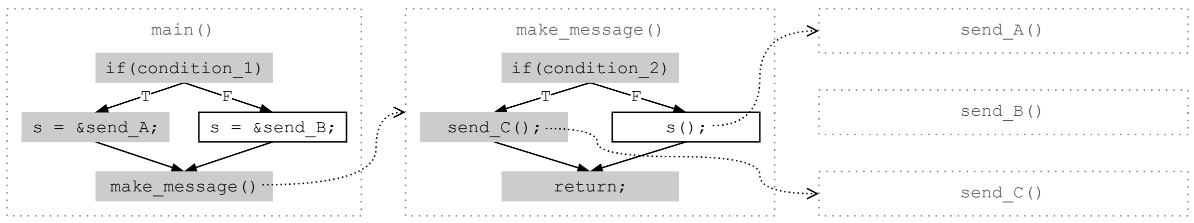
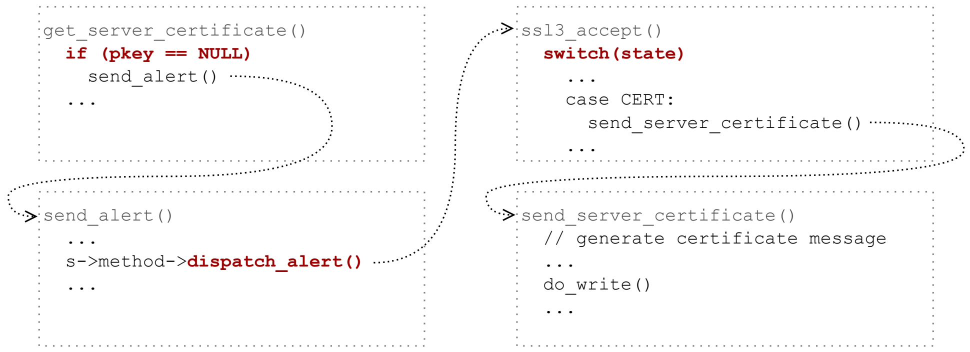


Figure 4.5: An example program in which simply forcing a chain of branches is not enough to execute a part of the source program which generates a different message.

pointed to by the variable `s`, and executes the function `send_C` instead, which sends the message `C` to the target. To generate message `A`, InSource can simply force the branch checking `condition_2` to "false" causing `s()` to execute. If, however, we want InSource to execute function `send_B`, InSource should be able to compute where the indirect variable `s` is assigned the address of the function `send_B`, and then ensure that this part of the code executes. Then it would need to also ensure that the part of the code where `s` gets called, executes as well. Instead of performing complex static analysis of the source application that would be required to compute exactly where the indirectly called variable is assigned (real applications are far more complicated than this example), we instead enable InSource to call any function at the indirect call site `s()`, allowing InSource to reach the function `send_B` by first forcing the branch `if(condition_2)` and then *forcing* the call in `s()` to execute function `send_B`. We can force an indirect call to any function, whose function signature (return and argument types) matches the signature at the call site.

We find that the method of *forcing* calls to be very effective in practice, especially for the purpose of using the client application to generate very unusual message sequences. Consider Figure 4.6, in which we change the control-flow of a source application implementing the client-side logic of the TLS protocol to send a target TLS server the "Server Certificate" message. The "Server Certificate" message is never sent by a TLS client under normal circumstances, yet InSource is able to leverage the server-side code embedded in the source application to make it generate this message. By first forcing the branch `if (pkey == NULL)` to "true," then forcing the indirect call `dispatch_alert()` to execute the function `ssl3_accept` and then forcing the branch `switch(state)` to the `CERT` condition, InSource is able to execute the function `send_server_certificate`, which generates and sends a "Server Certificate" message. `ssl3_accept` is intrinsically a server-side TLS function, and is not intended to be used by the source program, when it is communicating with a server. Still, because the source has already received



```
get_server_certificate()
  if (pkey == NULL)
    send_alert()
  ...

send_alert()
  ...
  s->method->dispatch_alert()
  ...

ssl3_accept()
  switch(state)
    ...
    case CERT:
      send_server_certificate()
    ...

send_server_certificate()
  // generate certificate message
  ...
  do_write()
  ...
```

Figure 4.6: Calls needed in order for a "Server Certificate" message to be sent as a message from a source TLS client application. This is possible in this case because the source application implements both client-side and server-side TLS functionality. The code in this figure is simplified code from the OpenSSL library [101].

a "Server Certificate" message with `get_server_certificate`, and because the source program internally uses the same data structure for storing the server certificate data in both its client-side and its server-side code, InSource is able to make the source application send back to the server the server's own certificate.

By making modifications to the control-flow of the source application, we are bound to encounter situations in which we try to generate a message for which the source application does not have all the necessary components. In the example in Figure 4.6, the source application was able to generate the "Server Certificate" message because it had received the server certificate in the prior message. If, however, InSource tried to change the control-flow of the program earlier (before the server certificate was received), it would try de-referencing memory where the certificate was supposedly stored, causing a memory error. After the program crashed, no more messages would get sent from the source program to the target. However, it would be beneficial for InSource to keep the source program running, so that the source at least sends a partially-correct message to the target, increasing the chances of placing the target into a new state or exercising some unusual parts of error handling code in the target. We thus increase the resilience of the source program by making it ignore memory errors during execution, and simply to continue execution from the next available instruction. Section 5.1.2 describes the exact details of our approach. We find that this is a surprisingly effective way of helping the client generate useful message sequences even when the client does not contain enough information to construct some messages completely. Appendix C gives more exact figures about how many states on various target program are found because of this approach.

4.3.3 Guiding control-flow changes

The control-flow changes that we described in the previous sections are used by InSource to generate non-standard message sequences. However, not all control-flow changes in the source application help generate non-standard message sequences, since not all of the source application code is related to sending messages to the target. Thus, we define a method by which InSource can select *targeted blocks* in the source application, which InSource then tries to reach by making control-flow changes to the default control-flow of the application. Targeted blocks are any basic blocks in the source program that are targeted by InSource by making changes to default control-flow. InSource tries to pick targeted blocks

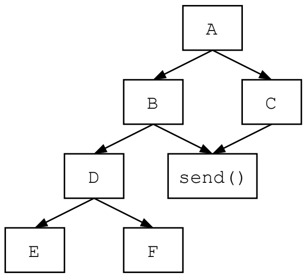


Figure 4.7: A simple control-flow graph in which the basic blocks `A`, `B`, and `C` are chosen as the *targeted blocks*.

in the source program that are related to message generation, since such blocks are more likely to change the message sequence that is generated by the source. For example, in Figure 4.6 InSource selects the basic block that contains the call to `do_write` as a targeted block, and reaches it using both branch forcing and indirect call forcing. This block is one of many targeted blocks that InSource selects and tries to reach in the source application. By rerunning the source program with a different targeted block each time, InSource makes the source generate multiple non-standard message sequences, which are in turn useful for triggering new states in the target application. For example, by selecting the targeted block `msg B` in Figure 4.3 and forcing the branch leading up to it, InSource causes the source application to generate the message "`msg B`" and send it to the target, toggling a different state in the target, as indicated by the change in its response.

Since targeted blocks guide which changes are made to the source program's control-flow by InSource, we select different targeted blocks in order to allow InSource to change control-flow in different ways. To select targeted blocks we perform a static analysis of the inter-procedural control-flow graph of the source application. In this analysis we identify any basic blocks that are predecessors to a function used to send a message to the network, and use these as our set of targeted blocks. For example, in the simple control-glow graph shown in Figure 4.7, the basic blocks `A`, `B`, and `C` are chosen as the targeted blocks. To compute edges at indirect call sites when creating an inter-procedural control-flow graph, we consider any functions with a matching function signature, and which have their address-of taken at least once, as candidate called functions at that call site. This means that the list of targeted blocks that InSource computes is a conservative estimate of the blocks that may be predecessors to network write calls. The function which is to be used as the indicator of a network write is specified by the tester.

We also allow a tester to manually specify files in the source program that should be targeted by InSource (i.e., all basic blocks in the selected files are targeted), and also allow a tester to select all basic blocks in the program as targeted blocks. The latter approach is feasible if the source program is small, but we find that the benefit from targeting all basic blocks is not usually significant and only slows down the state exploration process (Appendix A shows that in real programs the number of paths found and that have to be tested can be large when using all blocks as targeted blocks, and Appendix B shows that despite the larger number of paths, the number of states and the coverage gain on the server is not substantial when using all blocks as targeted blocks).

To thoroughly explore how a targeted block can be used by the source program to generate non-standard message sequences, we use InSource to try reaching it at different stages of execution of the source. Specifically, by detecting when the source application sends each protocol message to the network, we are able to tell which parts of the default control-flow are responsible for generating specific network messages. We can then use the default control-flow associated with each network message as the starting point for reaching a targeted block. For example, if two messages are generated and sent by the source application, we try changing control-flow at three different points in the execution. First, we try reaching the targeted block before the first network message is sent. Second, we try reaching the targeted block before the second message is sent, but after the first message is sent. Third, we try reaching the targeted block after the second message is sent, but before the source program closes the network connection.

Algorithm 1: Overview of how targeted blocks are used to generate message sequences.

```
Input:
    CG,                              /* Call graph, including indirect call candidates */
    CFGs,               /* List of control-flow graphs, one for each function in CG */
    default_blocks,          /* List of blocks executed in the default control-flow */
    num_msgs,     /* Number of network messages sent in the default control-flow */
    targeted_blocks                                     /* List of targeted blocks */
Output:
    all_forces,         /* List of sets of branches and calls that need to be forced */
    responses                             /* List of responses for each set of forces */
Function Explore(CG, CFGs, default_blocks, num_msgs, targeted_blocks):
    all_forces = empty_list
    responses = empty_list
    for msg_num in num_msgs + 1 do
        for targeted_block in targeted_blocks do
            all_paths = BFS(CG, targeted_block, default_blocks)    /* Breadth-first search */
            chosen_paths = choose_paths(all_paths)
            for path in chosen_paths do
                forces_set = empty_set                        /* Set of forces in this path */
                for cur_func, next_func in path do
                    forces_set += shortest_control_flow(CFGs[cur_func], next_func)
                end
                cur_responses = exec_source_target(forces_set)    /* Run source with forces */
                all_forces.append(forces_set)
                responses.append(cur_responses)
            end
        end
    end
    return all_forces, responses
end
```

Algorithm 1 shows how targeted blocks are used to generate message sequences in the source, during the step of exploring different states in the target. The primary input to the function is the inter-procedural callgraph. Here, it is represented separately as a *CG*, which is graph of all function calls, and a list *CFGs*, which is a list of control-flow graphs of basic blocks within each function. The *default_blocks* are those that are executed during the default handshake, and *num_msgs* is the number of network messages sent during the default handshake. The *targeted_blocks* are selected either manually or automatically as described earlier. For each targeted block in each network message, InSource first computes all the paths that can be used to reach the *targeted_block* from the default control-flow. To do so, InSource

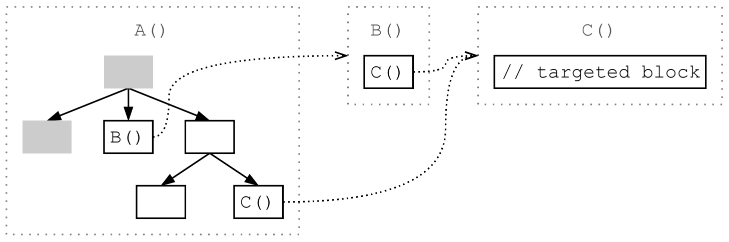


Figure 4.8: An example of a targeted block that can be reached differently using different path heuristics. The gray blocks are part of the default control-flow of the source application.

executes a bread-first search through the *CG*, starting from the function that contains the *targeted_block* and ending on any function that contains at least one of the *default_blocks*. InSource further narrows down the number of paths by calling *choose_paths*. Section 4.3.4 further describes how specific paths are chosen in *choose_paths*.

To be able to execute the source program and ensure that it reaches a targeted block during execution, InSource computes a set of *forces*. This is the sequence of control-flow changes, including branch forces and indirect call forces, which can be used to reach a targeted block through the current *path*. All branches starting from the first *default_block* in *path* up to the *targeted_block* are forced, since we do not know, before executing the program, whether their predicates will evaluate to the desired values. InSource computes the set of forces needed to go from one function to the next in *path*, by taking the *shortest_control_flow* within each function's control-flow graph from the start of the function to any block that calls the next function. If a function call is indirect, the function call is forced to execute the next function computed in *path*. As with branch forcing, indirect calls are always forced, to ensure that *path* is followed.

After computing the forces for each function in *path*, InSource combines the forces for each function into *forces_set*. Then, in *exec_source_target*, the source program is executed with all the forces in *forces_set* enabled, causing its control-flow to change, potentially making the source program generate a different message sequence for the target. The target's *responses* are later used to deduce whether the generated message sequence toggles a new state in the target, as will be described in Section 4.4. Each *forces_set* in *all_forces* is used to start a "fuzzing instance" to fuzz the target in each state, as will described in Section 4.4.1.

4.3.4 Choosing paths to targeted blocks

While picking only targeted blocks that are responsible for message generation helps InSource focus on trying to reach parts of the program that are useful for creating non-standard message sequences, InSource still needs to compute a sequence of control-flow changes that can be used to reach a given targeted block. If the targeted block is located within the same function as blocks which are covered by default control-flow (as in Figure 4.4), then InSource simply selects branch forces that follow the shortest sequence of branch forces from the default control-flow path to the targeted block. However, targeted blocks are also frequently located in other parts of the program, so chains of branch forces and indirect call forces have to be constructed to reach them. For example, to reach the targeted block in Figure 4.6, InSource needs to use a function call path that traverses 3 function calls.

We define several *path heuristics* for finding the function call path between the default control-flow

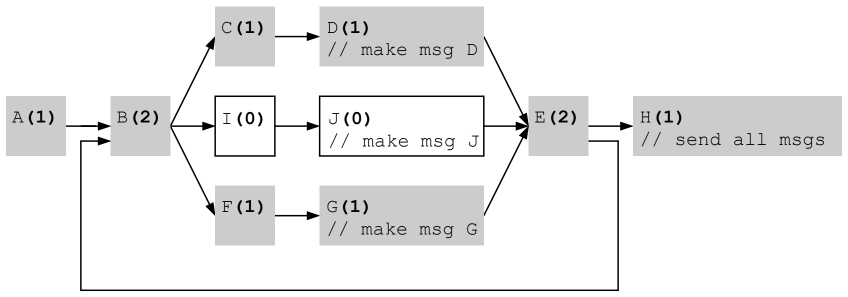


Figure 4.9: An example control-flow graph showing the necessity of trying to change execution on different counts of the first block in the default control-flow. Parentheses show the number of times each block is executed in the default control-flow. Blocks `D`, `J`, and `G` can all be used to generate messages, and block `H` sends all messages that were generated in bulk to the target.

and the targeted block: InSource can pick "all paths," it can pick the "shortest path," or it can pick the "simplest path." To demonstrate the differences, we present Figure 4.8. Here, the gray blocks are the ones that are part of the default control-flow, and the block in function `C` is the targeted block. All call paths to the block include the paths `A-B-C` and `A-C`. The shortest path to the block is `A-C`. It is the shortest path because it involves the fewest function calls. On the other hand, the simplest path to the targeted block is `A-B-C`. It is the simplest because this path involves the fewest number of branch forces to reach the targeted block: only 1 branch force is made in function `A`, compared to 2 branch forces made in `A` when taking the shortest path. We define these heuristics to give more flexibility to the tester. If the source program is small, a tester can specify all paths to be tried. If the source program is large, a tester can specify either the shortest paths or simplest paths to be used. We find that the shortest paths is the most likely to result in a new message sequence being generated by the source. On the other hand, when the shortest path heuristic does not work well for a program, we are sometimes able to generate new non-standard message sequences better by trying multiple simplest paths instead. Appendix B demonstrates some of the effects of using different path heuristics during state exploration.

In addition to allowing different call path heuristics to be specified, InSource can be configured to also try branch forcing during different iterations of the default control-flow. To demonstrate why this may be necessary, we show the control-flow graph in Figure 4.9. In the figure, the default control-flow is highlighted in gray. In addition, the number of times each block in the default control-flow is executed is shown in parentheses. Since there is a loop in the control-flow, block `B` is executed twice. The code in block `D` generates `msg D` and the code in block `G` generates `msg G`. Then, block `H` sends both messages to the target. In order to generate `msg J`, InSource picks `J` as a targeted block. However, in order to be able to generate multiple non-standard message sequences that include `msg J`, InSource must be able to try forcing the branch in block `B` during both count 1 of its execution and count 2. To account for this case, we allow InSource to be configured to try forcing the first branch in the path to a targeted block on multiple counts of its execution. We call this configuration "all counts" and find this configuration to be very useful in generating diverse non-standard message sequences, and leave it enabled whenever we use InSource. Data presented in Appendix B highlights its effectiveness in terms of increasing the number of states found in the target.

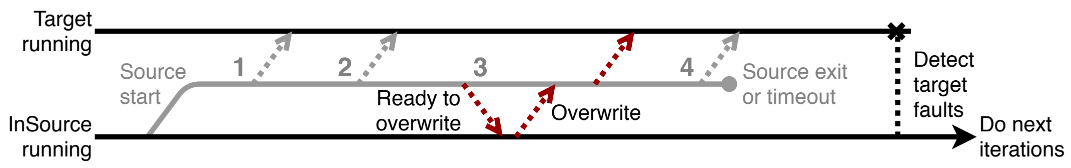


Figure 4.10: The execution flow of InSource that is used to fuzz individual network protocol messages that get sent to the target.

4.4 Testing the target in different states

In order to be able to effectively test the target application for low-level software vulnerabilities, InSource has to not only generate non-standard message sequences, but also generate different valid and invalid messages to send to the target. Because InSource generates many non-standard message sequences, however, trying to fuzz each message in each message sequence would be inefficient, since different non-standard message orders frequently trigger the same state in the target application. If we were to make InSource fuzz all messages in each non-standard message sequence that it found, we would waste a lot of testing effort doing redundant work.

Thus, to narrow down the amount of messages that are fuzzed by InSource and prioritize testing of only those message sequences that are able to trigger *new* states in the target, we analyze the responses received from the target after sending it each message sequence generated by the source, and only fuzz those messages that trigger new target states. We use the message response received from the target as an indicator of the target's state: if we see a different response to a sequence of messages sent from the source, we mark the target as being in a different state. We label responses as the same or different based on parts of the response message that stay constant across re-executions. For example, a message header or message body may be constant in the response every time a target is executed, while a nonce or sequence number would change. Since a target usually sends multiple responses throughout an exchange of messages with a source, the target changes state multiple times during execution. This means that a single new message sequence generated by the source can yield multiple new states in the target. If we receive the same response but in different positions in the exchange (i.e., the response message is the third message received from the target, rather than the second message received), we correspond these responses to different states in the target.

We use this approach to determine the target's state because it can be used with all targets, including black-box, gray-box, and white-box targets. Intuitively, the number of found states roughly correlates with the amount of new code uncovered on the server (Appendix B shows the extent to which this is true in one of the targets that we tested). Overall, the end-to-end evaluation of InSource in Section 6.4.1 demonstrates that this technique for identifying states in state exploration is sufficiently effective at discovering and executing new code in the target application, as is shown by the coverage increase gained during testing.

4.4.1 Fuzzing each state

Once the states have been identified, in order to only test the target while it is in a particular state, InSource re-executes the source application on every test iteration, so that it regenerates the message sequence that puts the target into the desired state. It uses the same set of control-flow changes that are needed to make the source execute along a specific path to a targeted block (in other words, it uses one of the *forces_set*s from *all_forces* in Algorithm 1). To see an example of how this works in practice, consider Figure 4.10, which shows a single test iteration used to fuzz message 3. Here, messages 1, 2, 3, and 4 are part of some non-standard message sequence found during the previous state exploration stage. When messages 1 and 2 are sent, the target is put into the found state.

To test the target while it is in this state, the source is allowed to regenerate messages 1 and 2 on every execution, and message 3 is fuzzed. While building message 3, the source notifies InSource that it has reached the part of the code where InSource wants to overwrite the message (Section 5.2.1 describes where this instrumentation gets inserted). InSource then sends the new value to the source, and the instrumentation that has been inserted into the source, substitutes the old value to the value provided by InSource. The source execution continues, the message is constructed, and the source sends the fuzzed message to the target. The source then continues executing until it either exits naturally or times out. It is worth keeping the source running for as long as possible, since bugs in the code can be revealed while the target is using the fuzzed value for constructing later response messages (e.g., as was the case in the Heartbleed vulnerability).

During fuzzing, we start a separate "fuzzing instance," each fuzzing either a message field or a whole message. For example, in 4.10 we show a fuzzing instance that fuzzes message 3. If we only choose to fuzz whole messages, the number of fuzzing instances is the same as the number of states discovered during state exploration. If we choose to fuzz multiple message fields, we start multiple fuzzing instances per state, each fuzzing a different part of the message. For example, in Figure 4.10 we can start multiple instances for fuzzing different fields in message 3. Fuzzing instances can be run in parallel if more than one copy of the target is available (e.g., if the target is white-box or gray-box and thus can be started in multiple instances), or in a round-robin fashion, if the target is a black-box and cannot be easily duplicated. To detect if a bug in the server has occurred we use whichever techniques are applicable to the target at hand (known fault detection techniques are discussed in Chapter 2). We do not define any novel fault detection techniques in InSource.

4.4.2 Generating test messages

In order to more thoroughly test the code of the target application that is associated with every state found during state exploration, InSource performs fuzzing of both whole messages and individual message fields. As described in Section 2.1, fuzzing entire messages is the approach that is taken by blind fuzzers, and, while it is useful for generating invalid messages, is unlikely to generate valid messages that are accepted and processed by the target. However, for completeness, and in order to also exercise the code related to handling invalid messages in the target, we still perform blind fuzzing of entire messages that are found during state exploration. In Section 6.4.1 we find that even by only using blind fuzzing together with state exploration we are already able to exercise many parts of the target application code.

In addition, by fuzzing individual message fields InSource can generate valid messages that exercise the target code associated with each state more thoroughly. To implement message field fuzzing we identify function call arguments in the source application that contain pointers to memory buffers, and overwrite the buffers and length fields passed in the arguments to each function call. This is an approach similar to that taken by [18], but the way in which we identify function arguments to fuzz, differs. We identify buffer arguments by looking for function calls that contain both a pointer and a length argument. To narrow down the number of function calls that we identify, we allow the tester to specify in which source files of the source program InSource should pick function calls to fuzz. InSource then executes the source program and checks whether each function is actually called with a reasonable value (i.e., if a pointer is `NULL` or the length is negative during normal execution, this function call is not used for fuzzing since it is not linked to a meaningful buffer value). This approach produces both valid and invalid messages, but, for the sake of evaluation of this approach, in Section 6.4.2 we only measure how many valid messages this approach is able to produce, that whole-message fuzzing is not able to generate.

While we do not yet fully automate this aspect of InSource, we note that the knowledge required by the tester to pick files for field fuzzing is not related to knowing the details of the network protocol, but rather related to being able to quickly parse through the folders of the source application and look for file names or folder names that hint at the fact that they may be somehow related to the protocol at hand (e.g., they contain the words **`send`** or **`handshake`**, or they contain the protocol acronym). This function call selection method usually results in the selection of many function calls that do contribute to the message that is created by the source (i.e., are used in the message as message fields), but also usually causes many functions to be selected which do not contribute data to the generated message. Still, as shown through the experiment in Section 6.4.2, this method does help us select some function calls that do contribute to message creation and thus to generating *different* valid messages that exercise target code. However, we do hope to address the precision of function call selection and the automation of this feature in our future work.

Chapter 5

Implementation

In the previous chapter we discussed how changing control-flow of the source program helps us reach new states on the target that are then fuzzed in individual fuzzing instances. In this chapter, we discuss how we implement the InSource compile-time instrumentation that allows us to make changes to the control-flow as well as the instrumentation that allows us to overwrite whole messages and individual message fields. We also describe how we integrate existing fuzzing techniques into InSource to perform mutations on messages and message fields.

5.1 Modifying the source program

5.1.1 Instrumentation

To implement compile-time instrumentation used by InSource, we rely on LLVM [50] version 4.0.1. Although we prioritize support for instrumenting C applications, our use of LLVM allows us to later extend our instrumentation to support LLVM instructions and control-flow features used by other languages that can be compiled with LLVM-based compilers. As mentioned in Chapter 4, we allow two different kinds of changes to control-flow of a source program: branch forcing and indirect call forcing. Both instrumentations are implemented such that they augment program control-flow only when instructed by InSource, but otherwise don't have any effect on program behavior.

To illustrate how instrumentation is applied, we show an example source basic block in Figure 5.1. This block has an indirect call, which calls the function pointed to by the variable `%7`, and passes in the arguments `%8` (a pointer, although the type isn't shown for brevity) and `%9` (a length argument). After the call, the basic block contains a branch. If the value at the variable `%3` is true, the branch executes the left basic block next; if the value is false, the right basic block is executed next.

To allow branch forcing, we insert several additional basic blocks for each block that contains a

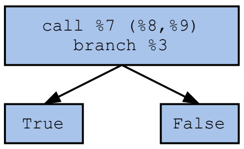


Figure 5.1: An example basic block that contains an indirect function call and a branch.

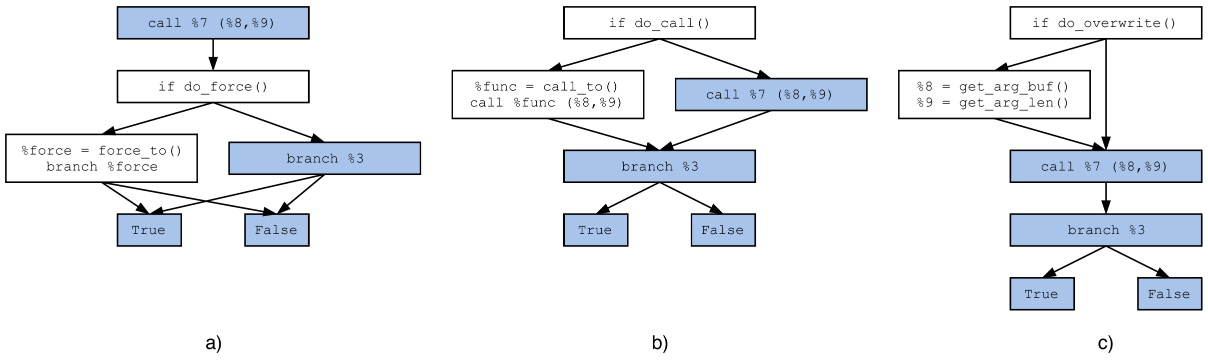


Figure 5.2: Different instrumentations that can be applied to basic blocks in the source application. a) Instrumentation that allows forcing branches to the desired value. b) Instrumentation that enables calling arbitrary functions at indirect calls. c) Instrumentation that allows overwriting function arguments to fuzz message fields.

branch, as shown in Figure 5.2.a. The first block that we insert performs a call to an additional function that we compile into the source, `do_force`, which checks whether this block was marked to be forced by InSource, or if it isn't marked and should be executed according to the default code. Because the call to `do_force` executes on every single branch, it has to be fast. Therefore, the only action performed by `do_force` is a check in a lookup array. The contents of the lookup array are pre-populated during initialization of the program based on the values provided by InSource. Only if the array value is set, does the `do_force` function succeed, and the `force_to` function gets called. This function simply checks which branch InSource has specified to force to. Based on the value provided by InSource, the `branch %force` instruction then executes the desired branch.

The mechanism used for forcing indirect calls is very similar to the mechanism for branch forcing, except that instead of a branch destination, InSource specifies the name of the function that should be called. The inserted blocks also follow a similar structure, as shown in Figure 5.2.b. First, the `do_call` function is called to check if InSource specified a function to overwrite. If it did, `do_call` returns "true" and `call_to` is called. This function fetches the function pointer based on the name provided by InSource. This pointer is found by parsing the source program binary during program initialization. Currently, we make an assumption about the format of the binary file and its location based on the host OS that we use. The line `call %func` executes the function pointer returned by `call_to` with the same function arguments that are used in the original function call `call %7`. While not shown in the figure for clarity, both instrumentations are combined together in this basic block.

The biggest effect on performance overhead stems from the functions `do_force` and `do_call` since they are always called on every branch and every indirect call in the source program. Appendix D shows the exact overhead induced on the source program by these functions.

5.1.2 Ignoring memory errors

Because we force the source application to execute along paths that it is not meant to execute, the source program sometimes encounters memory errors when trying to de-reference memory that isn't yet initialized. Specifically, the host OS that we use notifies the source program of memory errors using the `SIGSEGV` signal. We seek to ignore errors that happen on the source program so as to enable it to continue executing. To ignore them, we overwrite the default `SIGSEGV` signal handler and skip past the faulting instruction. We assume that the underlying architecture is `x86-64` and use the Capstone disassembler [72] to decode individual instruction lengths in order to skip past only one instruction. If we observe too many memory errors during the source program's execution (i.e., 10 or more) we exit the source program since the presence of this many memory errors is usually indicative that the control-flow path that is being forced is too non-standard to be useful for message generation.

5.2 Performing fuzzing

5.2.1 Instrumentation

In addition to instrumentations used for changing control-flow, we add instrumentation to allow overwriting individual function call arguments to allow field fuzzing of individual message fields while the message is being generated by the source program. The mechanism for overwriting function arguments is similar to the mechanism in the other instrumentations. Figure 5.2.c shows the blocks that are inserted by this instrumentation pass. First, the `do_overwrite` call checks whether InSource has specified the arguments to be overwritten. If this call returns "true," the new arguments are returned from calls to `get_arg_buf` and `get_arg_len`. These new values are then used in the original call in `call %7`. To get the new substitute arguments, the instrumentation notifies InSource and waits for a response which includes the new argument values.

In order to perform fuzzing of whole messages, we wrap functions used for sending network messages in the source program, and add the ability to overwrite the data that gets sent by the source over the network. These functions are defined by standard libraries like `libc`, and, as such, are used across source applications (e.g., functions `write`, `send`, and their variations). Since such functions can also be used to write to non-network connections or to files, we also perform simple checks in each function to ensure that the data is being written to the network, and don't perform fuzzing of the data if that is not the case. To get the message from InSource, the instrumentation notifies InSource and waits for a response, just as is done with function argument fuzzing.

5.2.2 Making mutations

In order to perform mutations of both individual message fields and whole messages, we integrate the AFL [102] fuzzing engine into InSource. We use AFL because it supports both blind fuzzing of black-box targets and coverage-guided fuzzing when the targets are either gray-box or white-box. Furthermore, it employs state-of-the-art input mutation techniques that are built on years of experience of its author. AFL is used in practice for fuzzing the Chrome browser at Google [2], and is frequently used as the base fuzzing engine for developing new bug-finding tools [53][67][85][94][95][103]. Since, by default, AFL only supports testing of the command-line or file inputs to a program, we only use it for performing mutations, not for coordinating the fuzzing process. InSource gives AFL an initial seed (which may be either a whole message or a message field), and AFL performs mutations of this seed. To use the mutated seed, InSource re-executes the source application and inserts the mutated seed into the source application either at a function call argument or at a network send function, using the instrumentation described in Section 5.2.1. If the target is gray-box or white-box, coverage information about the target is used by AFL to guide the mutations that it performs on the seed.

Chapter 6

Evaluation

To evaluate the ability of InSource to generate non-standard protocol message sequences as well as to construct different valid messages that can be used to find bugs in target applications, we perform several measurements that demonstrate the extent to which InSource is able to exercise a target program's code. These measurements demonstrate that InSource is able to thoroughly test complex and heterogeneous target applications. While we have not yet run InSource for sufficiently many iterations to find any security vulnerabilities in the target applications that we tested, based on the preliminary comparison of our technique to a manually-constructed protocol-specific testing tool and the fact that we already found several non-security bugs, we anticipate that, given enough time to run, InSource will be able to find security bugs in the target applications as well.

6.1 Chosen targets

For our evaluation we choose to test server implementations of several complex stateful network protocols: TLS, XMPP, and SIP. We specifically focused on testing heterogeneous *stateful* protocols, to show that InSource can be easily extended to testing different targets. As our target TLS server, we use the OpenSSL `s_server` application [101], since it is frequently targeted by TLS testing tools and thus provides a good baseline for comparison. As an XMPP target server we pick the sample server implementation provided with the QXMPP library [73]. For SIP, we use the server application provided by the PJSIP library [69]. We pick the sample server applications bundled with each protocol library since they allow us to specifically focus on testing features related to each network protocol, but any other applications that use these libraries can be tested as well.

To test the chosen server applications we choose corresponding clients that are used by InSource to generate different valid messages and non-standard message sequences. As our TLS source client we pick the `s_client` tool in OpenSSL. For generating XMPP messages we use the client application available with the libstrophe library [59]. Although QXMPP provides a client implementation of the XMPP protocol, the library is written in C++, instrumentation of which we do not yet support. For SIP message generation we choose the baresip [42] client application. Unlike the other clients that we use, baresip is not built on top of a library, but is instead built, from the ground up, as a command-line tool. InSource uses each default handshake as a starting point for state exploration. The default handshake for each protocol is shown in Figure 6.1.

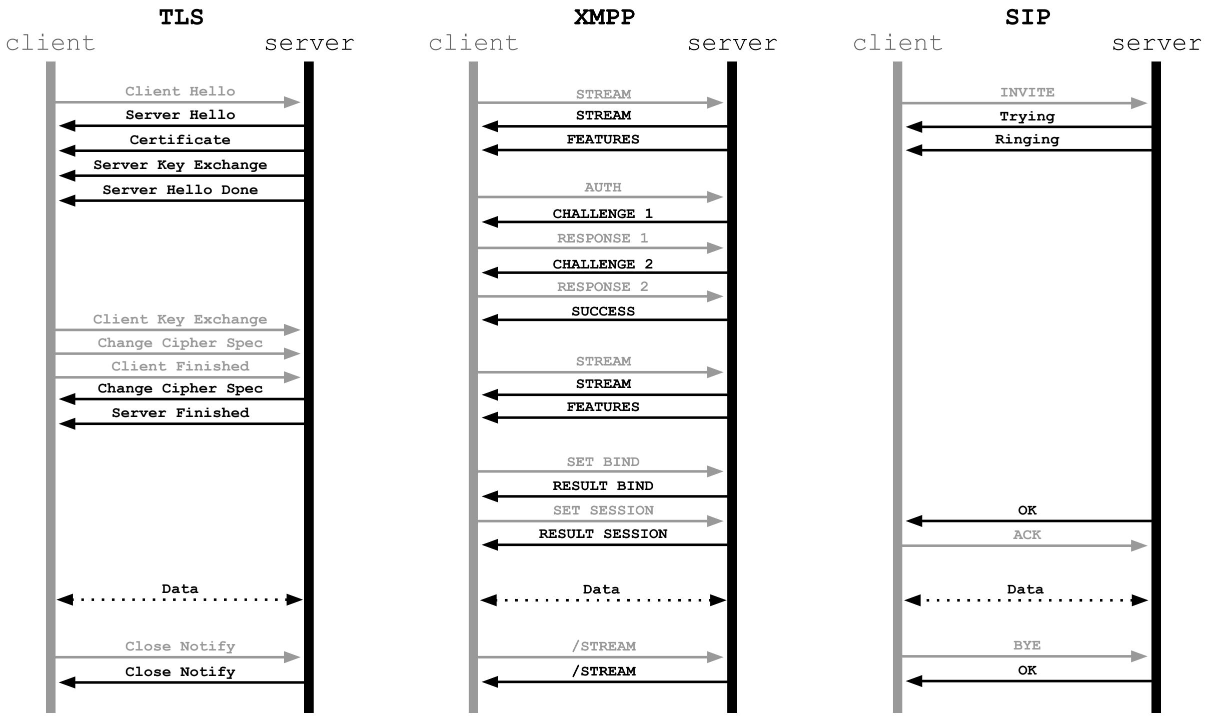


Figure 6.1: Stateful protocol handshakes as seen from the perspective of the client. Extra space roughly separates different stages of each handshake. Left: TLS, center: XMPP, right: SIP.

The protocols implemented in the code of the chosen target applications are quite different from one another, and testing each of them requires understanding and generating entirely different messages, with different dependencies, in different order, and with unique message fields. To highlight some of the challenges in testing servers that implement these protocols we give a brief overview of some of the complexities that make testing each protocol implementation difficult.

TLS

The complexity of the TLS handshake stems from two facts: first, the protocol is used to perform a complex negotiation of a common state that is established between the client and the server in order to send encrypted messages; second, the TLS handshake has evolved across different TLS versions, supports many different encryption methods, and can be extended with features that either give more flexibility or more security guarantees. Some fields in TLS handshake messages have inter-message dependencies that make testing difficult (e.g., encryption used in the "Client Key Exchange" message), while other fields make sanity checks in the target application difficult to satisfy (e.g., integrity hash used in record layer messages). Furthermore, message order is very important in the TLS handshake since it can influence a target's behavior, and is thus meticulously defined in the protocol's specifications [29].

XMPP

In an XMPP handshake there are many messages exchanged between the client and a server, but only a subset of them contain complex message fields. Notably, the "RESPONSE 1" message contains dependence on data sent in previous messages, and embeds this data both in plain-text form and as part of a hash. Generating different valid variations of this message is thus particularly difficult. More importantly, the abundance of exchanged messages means that the server is constantly re-adjusting its internal state, as it expects different messages at every stage of the handshake. Finally, the message structure in each message is formatted using XML, meaning that the messages are very verbose, so it is difficult to generate entirely new messages, even ones that do not have complex fields, with blind fuzzing alone.

SIP

The standard handshake performed between the SIP client and server[1] contains fewer messages than handshakes of other protocols; however, each message is comparatively long, with average length of messages sent by a client during the default handshake being almost 4 times as long as the messages sent in TLS and XMPP handshakes. The message fields used in SIP messages are less complex than in other protocols: neither hashing nor encryption is used as part of messages sent in the default handshake. However, SIP supports many modes of operation, so state computations in a SIP server tends to be more complicated. Furthermore, just like TLS and XMPP, SIP messages contain values that change on every execution of the client and server, making generation of valid messages difficult. Finally, the SIP protocol defines many different message types, generation of which is difficult through blind fuzzing alone.

6.2 Evaluation methods

We briefly highlight the evaluation metric used in most of our experiments in this chapter (Section 6.4.2 uses a slightly different evaluation technique, which will be discussed then).

6.2.1 Evaluation metric

In both the evaluation of InSource in comparison to a manually-created protocol-specific tool and in the evaluation of InSource's ability to test more code with the help of state exploration, we measure the effectiveness of each approach using coverage. However, this isn't an obvious choice for a metric: fairly comparing the effectiveness of bug-finding tools is known to be difficult, and experimental setups vary significantly, as explored by Klees et al. [48]. Particularly, bug-finding tools that perform fuzzing use non-deterministic mutations and are built to run indefinitely, so a fuzzing run can never be considered to be "complete." Coverage measurements help determine how much of the program can be reached by inputs generated by a testing tool, and are thus often used to help approximate how effective a testing tool can be at finding bugs; however, coverage does not exactly correlate to bug-finding ability: the seed generation engine of a testing tool can be good at generating seeds that reach new parts of the program, but the fuzzing engine may be bad at mutating these seeds in a way that can trigger bugs in the target. Still, coverage is commonly used for evaluation of fuzzers [48].

In addition to coverage, the LAVA-M [31] and the CGC datasets [16] are also frequently used for evaluation of fuzzing techniques. These datasets are collections of binary programs with intentionally-inserted vulnerabilities that allow creators of fuzzing tools to match the vulnerabilities that their tools

[1]We collectively treat the proxy server and target client as one entity, since both are implemented in a single network node in the PJSIP server program that we target.

find, to ground truth data. These datasets consist only of gray-box targets and are thus popular among developers of fuzzers that test such applications. Because these datasets are tailored to optimizing gray-box fuzzing techniques, however, the datasets consist only of small command-line utilities, and are thus not good candidates for network protocol testing. Furthermore, synthetic bug insertion techniques are best at helping evaluate specific fuzzer optimizations like sanitizers, input mutation heuristics, or symbolic solving techniques. Input generation techniques used by protocol-specific testing tools, on the other hand, usually rely on existing fuzzing techniques to perform mutations that trigger bugs, and instead concern themselves with finding ways to employ the fuzzer such that more of the functionality of the target program can be targeted by its mutation engine. Thus, since we are making a holistic comparison between two protocol-specific testing tools and not between the underlying fuzzing engine used in each, we resort to using coverage metrics as our evaluation criteria, since they allow us to more easily identify what previously-untested parts of the target program become tested due to the message generation techniques that we use. Intuitively, protocol-specific testing tools trust the underlying fuzzer to provide good mutation methods, while helping the fuzzer reach new code by providing protocol-specific knowledge about valid message structure and by changing message order.

As our coverage metrics we show function coverage, which highlights the number of functions that get executed, as well as region and line coverage, which are complementary to each other, but highlight slightly different statistics. Regions are roughly analogous to basic blocks, in that a predicate can map to multiple regions (e.g. `if (a == 2 || b == 3)` maps to 2 regions), whereas lines directly map to lines of source code. For completeness we present both statistics.

To measure coverage gained from every experiment, we first gather all of the mutated variations of the seed input that our underlying fuzzer (AFL) identifies to be triggering new code. Thus, while we disable coverage-guidance in the fuzzer, we still allow it to gather coverage so that we can identify which mutations, during the experiment, triggered new code on the target. Since the coverage metric used by the fuzzer uses edge-level coverage, and since it does not map to exact locations in the code, we use `clang`'s "Source-based Code Coverage" compiler flags to instrument the target and rerun (after fuzzing is stopped) each mutated message that was identified by the fuzzer as triggering new code. If the mutation was made to a field rather than a whole message, or if the message was sent as part of a non-standard message sequence, we recreate the sequence of messages and the message in which the fuzzed field was located, using the source application. For gathering coverage of the manually-constructed testing tool, we gather coverage for all inputs that it generates, as the tool is running.

6.2.2 Base case

To compare the coverage attained in each experiment to a common baseline, we measure the coverage gained from running a single handshake using unmodified source and target applications for each protocol. Table 6.1 shows these statistics, alongside the number of states that are known to InSource by default. States, in this case, are defined as in Section 4.4: each target response correlates to a different state. The number of states known to InSource is smaller than the total number of protocol messages sent by a target (as seen in Figure 6.1), since InSource determines the target's state by looking at a network message (e.g., underlying TCP or UDP message), which can contain multiple protocol messages.

For completeness, Table 6.1 also shows the total size of the underlying libraries that are used by each of the servers; however, we do not use these statistics for comparison of each tool. First of all, the total size of each library represents both the server-side code and client-side code, meaning that

		OpenSSL	**QXMPP**	**PJSIP**
	# default server states	4	8	3
Server coverage after handshake	Functions	1145	168	628
	Regions	14443	471	5992
	Lines	28444	1365	13581
Total size of libraries	Functions	5082	2137	1418
	Regions	102024	8891	25535
	Lines	184245	19572	52225

Table 6.1: The first half of the table shows statistics about each target application after only executing the source and the target programs together once, without making any modifications to message order or to individual messages. The second half of the table shows the total size of each library that is used by each server.

coverage of all code in each library is unattainable by only fuzzing server-side programs that use each of the libraries. Furthermore, libraries implement many features that are not used for parsing network messages. Protocol-specific testing tools only look for bugs, which are accessible from the network side. Finally, many library features are not used by servers that use these libraries, either due to how the target servers are written or how the libraries are configured by the tested servers. Thus, the numbers in this part of the table represent an upper bound for attainable coverage, but are far higher than the actual maximum number of lines of code that an ideal testing tool can cover.

6.2.3 Blind fuzzer

In addition to comparing InSource to a manually-created protocol-specific testing tool, we also compare both tools to a blind fuzzer. Despite its name, blind fuzzing is the next best option available to a manually-created protocol-specific testing tool, when considering methods that work for both black-box, gray-box, and white-box targets. We use AFL as the blind fuzzer that we compare to, since it is a state-of-the-art fuzzer that has been used to find numerous bugs in open-source software [102], and is currently used for continuously fuzzing large real-world software at Google [2]. InSource also uses AFL as the underlying fuzzing engine, but, in the case of blind fuzzing, AFL only fuzzes the messages that are present in the default handshake of each protocol (i.e., those shown in Figure 6.1). Since AFL cannot be used to test stateful protocols out-of-the-box (Network-AFL [10] and Preeny [81] can help only to fuzz messages in stateless protocols), we modify AFL to enable it to fuzz different network messages in the default handshake.

To test how InSource performs in comparison to a blind fuzzer, we can run the blind fuzzer in several different ways. To show why this is the case, part a) of Figure 6.2 shows how InSource partitions test iterations across instances that fuzz different states in the target. In this example, the first two states are known from a default handshake, while the other states are known from state exploration. Each fuzzing instance in Figure 6.2.a is fuzzing a whole message (messages `A`, `B`, `C`, etc.), and each message is sent to a target in a particular state. Thus, to fuzz the target for N total iterations, in this case, InSource dedicates $N/6$ iterations per instance.

A blind fuzzer, however, only has 2 states to fuzz in this case, since only 2 of the states are known by default, and are fuzzed via messages `A` and `B`. One way to run fuzzing of these states is to run each one for as long as possible (part b of Figure 6.2). Here, each state is fuzzed for $N/2$ iterations

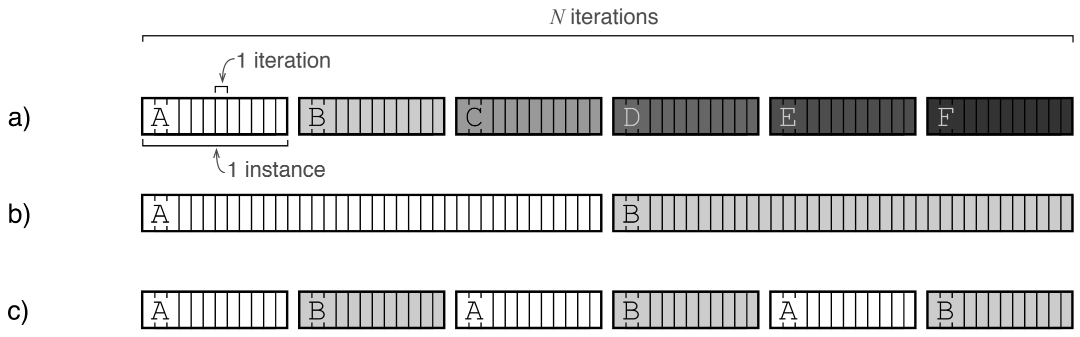


Figure 6.2: How iterations are partitioned across different states. a) InSource dedicates the same number of iterations for each fuzzing instance. b) A blind fuzzer can be partitioned to complete a maximum number of iterations per instance ("continuous" mode). c) To compare to InSource, the blind fuzzer is partitioned to complete only as many iterations per state as InSource does, and to restart fuzzing again from the initial seed the next time the instance runs ("split" mode).

continuously. When comparing this to InSource, however, this inflates the total coverage gain *within* the individual states `A` and `B` (this coverage would be attained by InSource if it fuzzed whole messages `A` and `B` for just as many iterations), and does not illustrate coverage that is gained *across* multiple states. In other words, fuzzing the whole message `A` for more iterations, helps a fuzzer gain more coverage due to it generating more invalid variations of message `A`, but does not illustrate a fuzzer's ability to find new states to fuzz. Thus, we also run the blind fuzzer as shown in part c) of Figure 6.2. Here, each of the default states is fuzzed in multiple *split* fuzzing instances, where each instance (e.g., the repeated instances for fuzzing message `A`) is restarted after it performs $N/6$ iterations. This allows us to more precisely compare the effect of fuzzing more states, since we discount the effect of fuzzing each state for more consecutive iterations.

6.3 Comparison to a manually-created protocol-specific testing tool

To evaluate how well InSource is able to test complex network protocols, we compare the coverage gained from testing an OpenSSL server using InSource, to the coverage gained from testing the same target server using a manually constructed TLS testing tool. While there are multiple TLS frameworks that enable the creation of testing tools for the TLS protocol [9][26][46][84], most of them only implement tests for well-known attacks on TLS applications, but do not provide support for continuously generating different valid and invalid messages and for automatic generation of non-standard message sequences. Only one of them, TLS-Attacker [84], implements a full-featured testing framework that supports these testing techniques and thus allows for the discovery of low-level security vulnerabilities in TLS implementations. In the evaluation performed by its authors, TLS-Attacker was able to find low-level security bugs in several TLS applications. Like InSource, TLS-Attacker is capable of testing black-box targets since it relies entirely on the ability to generate effective testcases that are based on the protocol format, rather than testcases that are tailored to the target's code.

To make our comparison as fair as possible, we run TLS-Attacker with the default settings, and enable all fuzzing stages performed by TLS-Attacker, including the tool's own versions of message re-

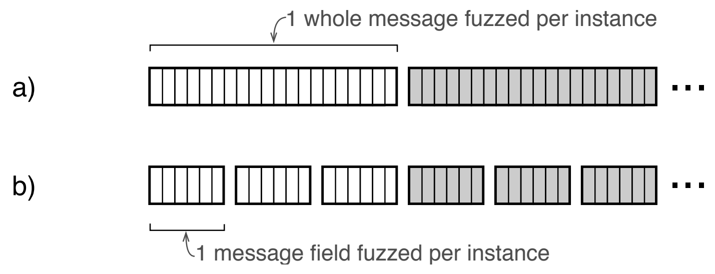

1 whole message fuzzed per instance
a)
b)
1 message field fuzzed per instance

Figure 6.3: How iterations are partitioned across different InSource instances. a) Only the whole message is fuzzed per instance. b) More instances are started by InSource, each fuzzing a separate field in the message. (Among these instances, 1 instances is still dedicated to fuzzing the whole message, but performs fewer iterations, matching the number of iterations used for fuzzing each other field.)

ordering and field fuzzing. When running InSource, we use the same total number of test iterations as TLS-Attacker does by default. As such, since TLS-Attacker runs all fuzzing steps in a single fuzzing instance, while InSource runs multiple fuzzing instances, we set InSource to run each fuzzing instance for a smaller number of iterations per instance, to make our total number of iterations match that of TLS-Attacker. Furthermore, we include the iterations that we use to perform state exploration into the count of the total number of iterations that we perform. Thus, while fuzzing only whole messages (column "Fuzz messages" in Table 6.2), we first do state exploration (which takes ∼55k iterations) and discover 110 potential states on the server. Then, we start 110 fuzzing instances to test each state by fuzzing the whole message sent to the server in that state, and run each fuzzing instance for ∼4k iterations, to make the total number of iterations made by InSource be the same as for TLS-Attacker. Part a) in Figure 6.3 visually demonstrates how instances are assigned a certain number of iterations to fuzz each message found during state exploration. If we also choose to fuzz multiple individual message fields (column "Fuzz fields & messages" in Table 6.2), we start an order of magnitude more fuzzing instances (1114, since multiple fuzzing instances are started per message), and thus run fuzzing for fewer iterations iterations per instance (∼400). Part b) in Figure 6.3 shows how there are more fuzzing instances per message, each running an equal number of iterations per message field. Furthermore, although the underlying fuzzing engine that we use (AFL) supports coverage-guided mutation methods, we disable this during comparison to TLS-Attacker, to compare how the tools operate in a black-box setting, since TLS-Fuzzer does not apply non-black-box fuzzing techniques.

We run a blind fuzzer using a variable number of instances, as described in section 6.2.3. The column "Continuous instances" in Table 6.2 shows the coverage attained when having 4 fuzzing instances, 1 for each network packet sent to the TLS server by default. Each instance makes 129k fuzzing iterations based on an initial seed. This corresponds to Figure 6.2.b. The column "Split instances," on the other hand, shows the coverage gained from blind fuzzing, when a known default state is fuzzed multiple times. Each time an instance runs again, it starts from the first initial seed, rather than continuing where the previous instance for the same message left off. This corresponds to 6.2.c. While the first measure shows the overall attainable coverage of the blind fuzzer, the latter measurement helps evaluate how much code is covered, when the blind fuzzer performs as much continuous fuzzing per message as InSource does.

The coverage that TLS-Attacker gains after 516,000 test iterations is shown in the first column of Table 6.2. When we fuzz whole messages using InSource (column "Fuzz messages"), we see a higher total coverage gain compared both to TLS-Attacker and to the "Blind fuzzing" experiments. This suggests that we are triggering a larger amount of new code in the target when the modified client generates new messages, and that fuzzing these messages helps us further stress-test the parsing logic in the OpenSSL server. This means that, although our approach does not require protocol-specific knowledge, by leveraging an existing TLS client we are able to generate a substantial amount of new message flows that reveal new states on the server application. Furthermore, by fuzzing each whole message, the total amount of coverage increases beyond the coverage attained by TLS-Attacker in the same number of test iterations.

While the total number of covered lines of code of either tool does not increase far beyond the baseline (i.e., coverage during the default handshake), it is important to note that all tools that are compared are specifically testing code associated with processing network-side inputs. In this regard, the effects of each testing tool are significant, since even several lines of new covered code that are associated with an esoteric feature of a protocol (e.g., TLS Heartbeat messages) may contain a potentially dangerous bug. Furthermore, since some code is reused by different parts of a target program, the gains in code coverage decrease as more and more parts of the target program are tested. For example, executing two different versions of TLS in the target will not double the amount of code that is covered, since some code is reused between TLS versions. Still, the way in which the same code is used when processing messages according to different TLS versions may contain subtle bugs and is worth testing.

		TLS-Attacker	**InSource**		**Blind fuzzing**	
		All stages	Fuzz messages	Fuzz fields & messages	Continuous instances	Split instances
	Total # iterations	516k	516k	516k	516k	516k
	# fuzzing instances	-	110	1114	4	110
Server coverage after testing	Functions	1234	1266	1241	1222	1172
	Regions	17092	17724	16943	16340	15169
	Lines	34733	36246	34285	32263	29755
Cov. union with TLS-Attacker	Functions	-	1273	1273	1246	1234
	Regions	-	18042	18008	17426	17126
	Lines	-	36895	36826	35701	34812

Table 6.2: Coverage comparison between TLS-Attacker, InSource, and blind fuzzing when testing an OpenSSL target server.

Still, it is interesting that the union[2] of the coverage gained by both TLS-Attacker and our approach is still greater than each approach in isolation. The functionality that is covered by TLS-Attacker but not by our tool include several encryption functions and timeout-related code. Since we use the same configuration for fuzzing the target application in both cases, it is likely that by fuzzing the list of cipher suites sent in the "Client Hello" message, TLS-Attacker is able to cause the server to negotiate a different encryption scheme from the one used by default, causing the server to use different encryption and decryption functions for processing some messages. We believe that, given enough test iterations, our fuzzing approach would be able to trigger these functions on the server, since we fuzz the "Client Hello" message and its individual fields as well. On the other hand, the timeout code triggered by TLS-Attacker is caused by session expiry, which occurs since TLS-Attacker has a larger timeout after each test iteration, to ensure that the server has fully processed each test message. This leaves enough

[2]The union is the total coverage attained on the target when both tools are used to test it.

time for the server to sometimes expire the session. To improve fuzzing speed, we use a smaller timeout and thus never allow the server's timeout function to be triggered. Most importantly, we don't observe TLS-Attacker triggering any code linked to particular states in the OpenSSL server that isn't already triggered by InSource, meaning that our approach for testing a target in specific states is comparable to manually-constructed protocol-specific testing tools.

Interestingly, blind fuzzing of individual messages (both "Blind fuzzing" columns) can be used to cover some server code that isn't tested by TLS-Attacker (as seen from the union of the total coverage gained by TLS-Attacker and blind message fuzzing being larger than the coverage gained by TLS-Attacker alone). This is because, although TLS-Attacker constructs different valid and invalid messages by fuzzing individual message fields, by not fuzzing the entire message it misses certain edge cases in the code related to handling of messages that are completely invalid. Still, TLS-Attacker is superior to the blind fuzzing method, as is expected from the fact that it can also generate valid messages and non-standard message sequences.

Finally, we compare TLS-Attacker to running InSource with not only whole-message fuzzing but also with fuzzing of individual message fields (column "Fuzz fields & messages"). In comparison to only doing whole-message fuzzing, this approach also adds fuzzing instances that fuzz individual message fields while the message is still being constructed by the source application. Since there are multiple message fields in each tested message, the total number of instances used for fuzzing is larger: 1114 as opposed to 110. The fuzzing instances started in this step constitute a superset of the fuzzing instances started when only whole messages are fuzzed, since whole-message fuzzing instance are started in this experiment as well. Given this fact, it may seem surprising then that the coverage for this experiment is lower than the coverage gained from fuzzing whole messages; however, it is important to note that the number of test iterations made per whole-message fuzzing instance in this experiment is lower ($\sim$400 as opposed to $\sim$4000). This is because in the mode where whole-message fuzzing and field fuzzing are combined, whole-message fuzzing is effectively deprioritized, and instead InSource uses more iterations for generating different valid messages instead. Valid messages, however, are less prone to covering new code, but are instead useful for exercising the code that is already covered. We therefore perform a more thorough evaluation of the effects of field fuzzing in Section 6.4.2.

Overall, when measuring coverage attained by InSource in comparison to a manual testing tool, we reach comparable coverage of the target application. Most importantly, our tool requires no knowledge of the tested protocol and thus minimal effort is needed to start fuzzing an application that uses a TLS protocol. This experiment demonstrates that we can bypass the need to learn the complexities of a stateful network protocol like TLS and instead rely solely on the protocol knowledge already embedded in a source application in order to test a target program.

6.4 Comparison across different network protocol applications

To evaluate how well InSource can be applied to heterogeneous protocols, we evaluate InSource on several different stateful network protocols by comparing its effectiveness to blind fuzzing. The method by which we start blind fuzzing instances is the same as is described in Section 6.2.3: we start the same number of blind fuzzing instances as we do fuzzing instances used by InSource, but each blind fuzzing instance only performs whole-message mutations of some default message, without using any protocol-specific approaches to find new states. We use the "split" mode of blind fuzzing (Figure 6.2.c), in order to measure the amount of code covered in the target with the same maximum number of continuous iterations per state. Since the techniques used by InSource are applicable to testing both black-box, gray-box, and white-box targets, when performing the experiments we disable test-generation techniques in the underlying fuzzing engine (i.e. AFL's coverage guidance) since these testing methods are only applicable to testing gray-box and white-box targets. This also helps us evaluate our methods in isolation from these techniques.

To guide the length of the experiments that we use to perform the comparison we take Klees' advice [48] to perform fuzzing runs of at least 24 hours. To speed up our experiments, we start multiple fuzzing instances in parallel; however, each instance performs a variable amount of executions in the 24 hours due to the variability of execution speeds of different fuzzing instances. As such, we further normalize the experiment to depend not on running time of every instance, but rather the number of executions performed by the slowest fuzzing instance in 24 hours. This means that the experiments that we run still take 24 hours, but are not skewed by the execution speeds of fuzzing instances that execute more slowly. This is also necessary to ensure fair comparison, since different fuzzing experiments are executed on different virtual machines, meaning that processing power may vary both across experiments and throughout time, so a time-based experiment would make it difficult to compare different approaches.

6.4.1 Fuzzing states found in state exploration

For the sake of testing the effects of state exploration techniques on the effectiveness of fuzzing a target, in this experiment we only fuzz entire messages when doing both blind message fuzzing and when performing fuzzing after state exploration. We do not enable field fuzzing since it is not linked to state exploration, and is evaluated separately in Section 6.4.2. The number of fuzzing instances used for testing in this section is directly linked to the number of server states that we discover through state exploration: for testing the OpenSSL server we start 110 fuzzing instances[3], for testing the QXMPP server we start 102 fuzzing instances, and for testing the PJSIP server we start 91 fuzzing instances. Thus, each instance fuzzes a single message sent to the target in a particular state, each of which is induced on the target by a non-standard message sequence sent by a source application.

For discovering states in state exploration we use different approaches for selecting targeted blocks and paths to targeted blocks. In the OpenSSL TLS client, we observe that the code structure cleanly separates functionality specific to coordinating messages from the functionality of lower-level utilities, so we manually select the `ssl` folder in the source program and use all the basic blocks located in the files in this folder as targeted blocks. We then apply "shortest paths + all counts" as our path heuristic. In the libstrophe XMPP client, we select all blocks as targeted blocks because the application is relatively small, and use "shortest paths + all counts" as our path heuristic. In the baresip SIP client, we automatically identify targeted blocks by identifying all predecessor blocks to the `sendto` network function, and use the "3 simplest paths" path heuristic, since, after a quick test, it identifies more states than the "shortest paths + all counts" path heuristic, for this source application.

Table 6.3 shows the amount of coverage that is gained from running a fuzzing campaign targeting each of the discovered target server states. The first set of coverage statistics shows the coverage gained from state exploration. Only InSource performs state exploration; a blind fuzzer only fuzzes messages that are present by default. The second set of coverage values shows the coverage attained from then fuzzing the messages that are sent to each found server state. As a reminder, Table 6.1 shows the

[3]This is the same number of instances as in Section 6.3.

server coverage that is already attained from executing the target and source applications together once normally, while keeping the control-flow of the source application untouched. Table 6.3 shows the total coverage, including the default handshake, and not just the coverage gain.

		OpenSSL		**QXMPP**		**PJSIP**	
	Cov. type	Explore + fuzz	Blind fuzzing	Explore + fuzz	Blind fuzzing	Explore + fuzz	Blind fuzzing
Server coverage after exploration	Functions	1238	-	196	-	665	-
	Regions	16617	-	595	-	6694	-
	Lines	33686	-	1682	-	15095	-
Server coverage after testing	Functions	1281	1195	196	186	707	688
	Regions	18151	15703	607	570	7420	6994
	Lines	37373	30948	1713	1603	16674	15716

Table 6.3: The total coverage on the target application after performing testing using different techniques. The first three rows show the total coverage of the target by modifying the source application using InSource in order to generate non-standard message sequences. The last three rows show total coverage of the target from fuzzing whole messages sent to the target, when it is in different states.

The statistics shown in Table 6.3 demonstrate that we are able to effectively test every target server by leveraging existing client applications implementing each protocol. For the OpenSSL server, only employing blind message fuzzing helps generate testcases that increase line coverage of the target server by 8.8% (28444 lines in Table 6.1 to 30948 lines in Table 6.3). On the other hand, by first performing an exploration of server states (28444 to 33686) and then fuzzing the newly found states (33686 to 37373), we increase the amount of code covered on the server by 31.4%. We see that InSource is effective when testing OpenSSL, which is in line with the results of the experiment presented in Section 6.3[4].

For the QXMPP server, blind message fuzzing can increase the amount of lines covered by 17.4% (1365 lines to 1603 lines). On the other hand, first applying state exploration and then fuzzing, yields a 25.5% increase in coverage (1365 to 1713). Although this is a noticeable increase, the results from testing the QXMPP server look less impressive compared to the results from testing OpenSSL with InSource, which prompted us to manually inspect the code that was covered on the QXMPP server. All of the code responsible for processing the client handshake (with the exception of several lines of error-handling code) was thoroughly executed in the QXMPP server during the testing process. However, some non-handshake-related features were left un-tested. XMPP specifies numerous enhancements [99] that can optionally be added to both client and server applications that use the XMPP protocol. While the QXMPP server implements some of them, the libstrophe client does not. This means that InSource is not able to generate messages related to these optional features, since they are not implemented in the source application. The results of testing QXMPP highlight the effectiveness of using a source application to thoroughly test common XMPP features implemented in an XMPP server, but also highlight a pitfall of this approach — the dependence of InSource on the protocol features implemented in the source application.

For the PJSIP server, blind fuzzing of default handshake messages for 24 hours yields a coverage increase of 15.7% (13581 lines to 15716 lines). On the other hand, first running state exploration and then fuzzing each individual state results in a 22.8% increase in server coverage (13581 to 16674). The

[4]This experiment ran for longer than the one in Section 6.3. Here we run ∼55k test iterations per OpenSSL fuzzing instance, as opposed to ∼4k iterations per instance.

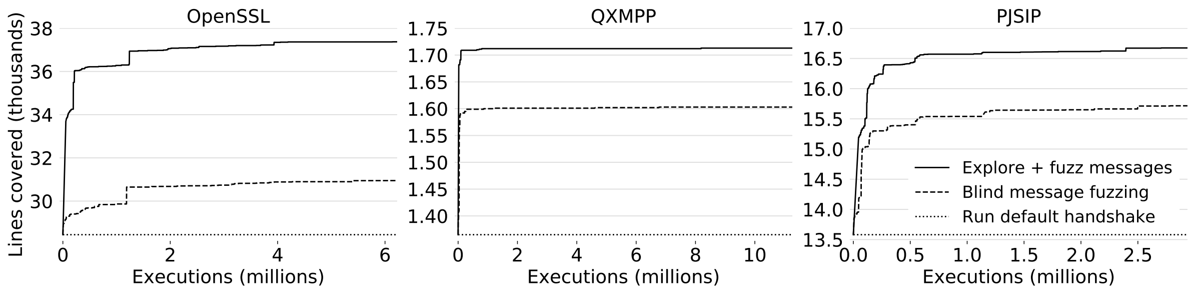


Figure 6.4: Line coverage attained over the number of executions for each target application. "Run default handshake" shows the number of lines gained from running the client and server without any mutations. This is used as a baseline for evaluating how much additional coverage testing tools can attain. "Explore + fuzz messages" shows the total amount of coverage attained while fuzzing messages after state exploration is performed. "Blind message fuzzing" is the coverage gained by only fuzzing default messages. Since the fuzzing instances run in parallel in our evaluation, the plot combines the coverage gains made by each fuzzing instance as iterations progress.

increase in coverage gain when testing PJSIP with InSource is the least pronounced among the targets that we tested, despite the fact that SIP contains numerous features that can be tested, which the source client also *does* implement. Observing the messages generated by the source client[5] that we used as the source application revealed that this client program was less accommodating to unusual control-flow changes compared to the clients that we used during testing of the other target protocol applications. We further discuss the difficulties that we faced with this source program in Section 7.1.

To further illustrate the effect of applying state exploration to testing, in Figure 6.4 we show the plot of coverage attained in the target over the number of test cases executed. From the figure, it is clear from the initial growth in coverage that happens during the state exploration phase that the amount of code that gets tested in the target application grows significantly during this step. The coverage shown is only that which is gained from the states that are selected for fuzzing using the method described in Section 4.4, and not the coverage from all the tried source control-flow paths (since we don't fuzz the paths that we do not select). The coverage gain from state exploration is followed by a more gradual growth when fuzzing is performed. The gradual growth is common among fuzzing tools, since many mutations have to be performed to attain new coverage through mutations of individual messages, and since fuzzing seeks to exercise each individual path to not only discover new code but also to discover bugs. Since testing through fuzzing can be performed indefinitely, this growth is likely to continue, but at a reduced rate. In the figure, the growth in line coverage gained from fuzzing QXMPP plateaus more dramatically than for the other targets. This is due to the nature of the implementation, which only implements high-level features, and delegates other functionality to other libraries. Thus, when new functionality is triggered and tested, the number of lines associated with that functionality is smaller, when compared to tools that implement many of their own features (such as OpenSSL and PJSIP).

Neither blind fuzzing nor our testing technique are likely to discover all code in the target program since many parts of the program are unrelated to the setting in which the target is being tested. For example, the OpenSSL library contains command-line utilities and library functions that are either not exposed to the network or not used in the application that we tested. However, in the context of testing the stateful message exchange between network nodes, a protocol-specific testing tool like InSource still

[5]We used baresip [42] as the source client for testing the PJSIP server.

is more effective at discovering and testing new states in the target program, as seen from Figure 6.4.

As mentioned earlier, coverage information by itself only helps demonstrate that we can increase the amount of functionality on the server that gets tested by generating non-standard message sequences. In addition, we also implement a way for InSource to generate different variations of valid messages via fuzzing of individual message fields, as discussed in Section 4.4.2. This aspect of testing, however, is used to exercise code related to processing of valid messages rather than discover new states, so it is not as easy to measure with coverage metrics.

6.4.2 Generating different valid messages

One of the testing techniques used by InSource is the ability to generate different variations of valid messages through fuzzing of individual message fields. Rather than discovering new states, however, this technique aims to exercise code related to the processing of valid messages in the target. Specifically, fuzzing of valid message fields helps generate messages that pass complex checks like hashes and encryption, which can cause a target application to reject messages that don't follow a valid message structure. To measure how well this technique is able to exercise these parts of the code in the target program, we measure the amount of test inputs that are generated by InSource, which are able to reach and test parts of the target program that are related to valid message handling, and that don't get tested by a blind fuzzer.

To show why this metric is necessary and why coverage measurements are insufficient for evaluating this aspect of testing network protocol applications, we present a simple example of target in Figure 6.5. This figure shows the same code and relates to the same message structure and vulnerability that was shown in Figure 2.3 in Chapter 2. As a reminder, when the "bytes" field is not `NULL`-terminated in the first 5 bytes and when the `process` function is called, a buffer overflow bug is triggered in the target. In Figure 6.5, if the "hash" field does not match the value computed by the target in "test," the vulnerable code does not run. The numbers in each edge in part a) for the figure show how frequently each edge executes when a blind fuzzer is used to fuzz the message: only the first time when the fuzzer executes, does the hash check pass. This is because, on the first iteration, a fuzzer doesn't overwrite the message that gets sent to the target, it simply tests how the program executes under normal conditions. Once the fuzzer starts mutating the message by using the initial message as the seed, it stops being able to generate messages that pass the hash check in the target code, and only produces invalid messages. Thus, after performing 99 more mutations to the seed, all of the mutated inputs to the target fail the hash check, since the nonce used by the target in subsequent re-executions is not the same as the nonce that was used when the fuzzer's seed was generated. In fact, if the fuzzer did not make any mutations to the message and simply re-executed it 99 times, the hash check would still fail every time, due to the data dependence between the nonce used in the "hash" field of the message and the nonce stored in the target server.

Consider instead part b) in the figure, which shows the exact same code in the target. This time, however, the target is executed using a protocol-specific fuzzer that can generate valid messages. Specifically, a fuzzer that generates messages in which only the "bytes" field is mutated, but in which the "hash" field is re-computed during every execution, is able to more thoroughly exercise both paths in the code. In this example, a fuzzer performs 50 iterations of mutations to the "bytes" field and 50 iterations of mutations to the "hash" field, thus testing the target more thoroughly. This kind of fuzzer is able to discover the vulnerability found in the `process` function, since it is able to generate valid

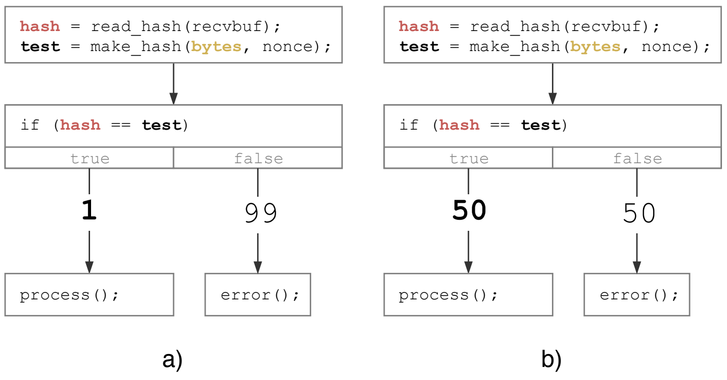

hash = read_hash(recvbuf);
test = make_hash(bytes, nonce);
if (hash == test)
true
false
1
99
process();
error();
a)
hash = read_hash(recvbuf);
test = make_hash(bytes, nonce);
if (hash == test)
true
false
50
50
process();
error();
b)

Figure 6.5: This figure shows the same code as that shown in Figure 2.3, but highlights the edge coverage difference between only sending invalid messages (a) and sending both valid and invalid messages to test a target (b).

messages that do not get rejected by the target program's hash check.

This example highlights that coverage is insufficient to measure all aspects of a fuzzer's effectiveness and that a slightly different approach has to be used for measuring the effectiveness of valid message generation. Both the line, region, function, and edge coverage for a blind fuzzer (part a) and a protocol-specific fuzzer (part b) would be the same in this case, but only the protocol-specific fuzzer would be able to discover the bug in function `process`. Instead, we use the metric of the number of test-cases generated, which are able to execute the program along the *correct path*. We define the correct path in terms of the behavior of the target application when tested by a blind fuzzer. Specifically, the correct path in the program gets executed on the very first iteration of the blind fuzzer, during which the fuzzer does not mutate the message, but does not get executed on subsequent iterations of a blind fuzzer. For example, in part a) of the figure, the blind fuzzer executes the left path once, but does not execute it on any subsequent iterations, so we consider the path through function `process`, to be the correct path.

To refine this measurement further, we only count the messages generated by the fuzzer which cause a measurable change in the behavior of the program: if, for example, the "bytes" field is changed from an original value of "`Hi\0`" to a mutated value "`Hello\0`," we will measure an increase in the number of times that the loop in the function `process` gets executed. On the other hand, the mutation to "`Ii\0`" would not be recorded, because the number of times each edge in the target executes would stay the same. To gather this coverage information, we rely on AFL's internal edge-level coverage metric, and measure how often AFL detects that a mutation that it performs to the seed value (i.e., to the "bytes" field) allows execution along the correct path *and* executes edges in the target a different number of times.

Thus, this metric helps us estimate how many messages are generated by InSource that are *different* valid messages. One caveat of this metric is that it ties measuring message validity to the implementation of the target application, rather than the actual validity of the message according to specifications. We still consider this measurement to be useful, since, even if our tool generates invalid messages that get processed along the correct path, such messages will still be useful for exercising the correct path in the target in search of low-level bugs. Another important thing to note about this metric is that it does not capture all different valid messages that our tool generates, but only those that are also not generated by a blind fuzzer. Thus we get a lower bound on the number of different valid messages that get generated, and are able to observe the benefit of using InSource when exercising the correct path, as opposed to using a blind fuzzer.

In order to evaluate how well InSource performs valid message generation through field fuzzing in isolation from testing how well InSource generates non-standard message sequences, we only run field fuzzing on the messages performed during the standard handshake used in each protocol. For example, if there are 4 network messages that are sent by a source application to a target, we only fuzz the fields in these 4 messages. We first perform blind fuzzing of each of these standard handshake messages to identify the correct path. Then, we allow InSource to identify message fields that can be fuzzed during the execution of the source application, and use InSource to fuzz each field in each message. Table 6.4 shows how many message fields we identify for fuzzing in each message of each *source* application. Then, we fuzz each individual field using InSource, and measure how many messages InSource is able to generate that execute the *target* program along parts of the correct path that are not executed by tests generated with a blind fuzzer.

	OpenSSL		**QXMPP**		**PJSIP**	
#	Fields fuzzed	Messages generated	Fields fuzzed	Messages generated	Fields fuzzed	Messages generated
1	6	725	2	0	51	218
2	24	657	2	0	29	108
3	8	368	63	288	9	116
4	4	0	2	0		
5			2	0		
6			2	0		
7			2	0		
8			2	0		

Table 6.4: Number of valid messages generated by InSource, which exercise parts of the *correct path* in a target program that are not exercised by a blind fuzzer. "#" refers to the number of the message sent over the network (i.e., a TCP or UDP message). "Fields fuzzed" refers to the number of fields fuzzed while a source application is generating a particular network message (one field is fuzzed at a time).

Table 6.4 shows the count of the number of inputs that get generated by InSource, which are able to execute parts of the correct path, that also are not executed by a blind fuzzer. From the table it is evident that fuzzing fields in the client source applications leads to a significant number of messages being generated by InSource that propagate along the correct path in the target application. Specifically, in TLS messages 2 and 3, some of the messages that get sent to the server are encrypted (e.g., "Client Key Exchange" and "Application Data"), leading to a larger number of lines being executed due to successful decryption of these messages, as well as the fact that each message's hash is computed correctly. Surprisingly, fuzzing fields that make up message 4, which is an encrypted alert message, doesn't reveal more executions of the correct path, even though four different fields are being fuzzed on the source application for this message. Closer inspection reveals that, although the function arguments that are fuzzed on the source do contribute to generating different valid messages, the code for handling alert messages on the target simply is not executed for any valid alerts that the source sends. This is because, in this state of the target application, when message 4 is received, any message that is an alert is ignored, so the alert message processing code cannot be exercised while the server is in this state. On the other hand, we find that the alert *is* processed when an alert is received in other server states that we find through state exploration in Section 6.4.1, further motivating the usefulness of combining the two approaches. Message 1 sees a significant number of messages that execute the correct path despite the fact that the "Client Hello" message does not contain encryption or hashes. This is because blind fuzzing of the entire message quickly distorts many parts of its complex structure, preventing the server from executing along the correct path. On the other hand, fuzzing fields that compose the message helps keep the message mostly valid, while only mutating small parts of it at a time.

A similar pattern is observed in the first PJSIP message. Fuzzing individual fields helps keep the majority of the message intact, preventing many error conditions in the target server. On the 2nd and 3rd messages, messages sent to the target use sequence numbers and `Call-ID` fields that are execution-specific and are checked by the server before it begins processing the remaining fields. InSource is therefore able to generate messages that pass these checks and are not rejected immediately by the target program. On the other hand, a blind fuzzer does not exercise the code used for processing these messages well.

For the QXMPP server we observe that fuzzing individual fields enables InSource to overcome the hash check found in the code for processing message 3. By fuzzing individual fields, the source is able to construct messages with a valid hash, that are not rejected by the target. For the remainder of the messages we do not observe any benefit from using valid message generation capabilities provided by InSource, because our tool does not identify any more message fields in the source program that can be fuzzed. This is due to the limitations of how we currently identify message fields in the source application. The two fields in each message that are identified by InSource are simply buffers in the source application which hold the entire network message before it gets sent to the target, so fuzzing these "fields" is equivalent to fuzzing the entire message using blind fuzzing techniques. Thus, as expected, fuzzing these "fields" does not allow InSource to exercise any parts of the correct path any better than is done by blind fuzzing.

6.4.3 Early found bugs

While we have not yet run InSource for a sufficiently long time to exhaustively find all of the bugs that it could potentially discover in the tested target programs, we found several early bugs that are an indication that InSource is able to trigger and detect bugs in target programs. One of the bugs that we found was in an earlier version of the OpenSSL server[6], in which the target crashes when AddressSanitizer [77] detects an incorrect use of `memcpy`, in which the source and destination buffers overlap in memory. We are able to test the target program with AddressSanitizer enabled because it is a white-box target. While the bug is not triggered in a state that is found by InSource, it is instead triggered when fuzzing the first ("Client Hello") message in a non-standard message sequence. Since the Client Hello message is different from that which is sent in the default handshake, InSource is able to find the bug faster than a blind fuzzer can. In other words, the different "Client Hello" message generated by the source program acts as a good seed for the underlying fuzzing engine.

Another bug that we found was an assertion failure in the newest version of the PJSIP server[7]. In this case, the bug occurs both during fuzzing of a message in a non-standard message sequence and when fuzzing a message in the default handshake.

[6]OpenSSL version 1.0.1g

[7]PJSIP version 2.8

Chapter 7

Limitations

The field of security vulnerability detection in software is ever-expanding and tools are constantly being improved. As such, we present observations about InSource that present opportunities for us to further strengthen the techniques that we implement.

7.1 Dependence on source application

The usefulness of InSource hinges on having access to source code of an application that implements the same network protocol as the target that is being tested. This is in contrast to manually-constructed protocol-specific testing tools that, once they are built, don't require any additional *source* program in order to operate. While we have found that open-source applications that implement network protocols are abundant, there are cases when target applications implement protocols that are not widely used. In such cases, only binaries of these applications may be available. Using such gray-box programs as source programs is a possibility that we are considering for future expansion of our tool, but with which we have not yet experimented in the current version. We anticipate that the limitations in the flexibility of tools used for dynamic binary instrumentation will limit the number of current techniques that we can apply to transform binary-only source program.

Even when the source code of a source program is available, however, the way in which a program is implemented can limit the effectiveness of InSource. For example, the XMPP source program that we used for testing an XMPP target did not support as many XMPP features as the target, meaning that the source application could not generate all non-standard message sequences or produce all valid messages. However, we share this limitation with manually-constructed protocol-specific testing tools, since these tools are also dependent on the features that they implement, and cannot generate protocol messages that they do not support.

Furthermore, even if a source program does support all the desired features, the control-flow changes that InSource tries during state exploration, may, for some source programs, not be as effective as for other source programs. During our evaluation of InSource we found that the "baresip" source SIP client application was not as responsive to control-flow changes, meaning that when we augmented the program's behavior, the program would eagerly exit rather than generating new non-standard message sequences. To see why this occurred, consider the example control-flow graph in Figure 7.1. The gray blocks shown are part of the default control-flow, and the targeted block selected by InSource is

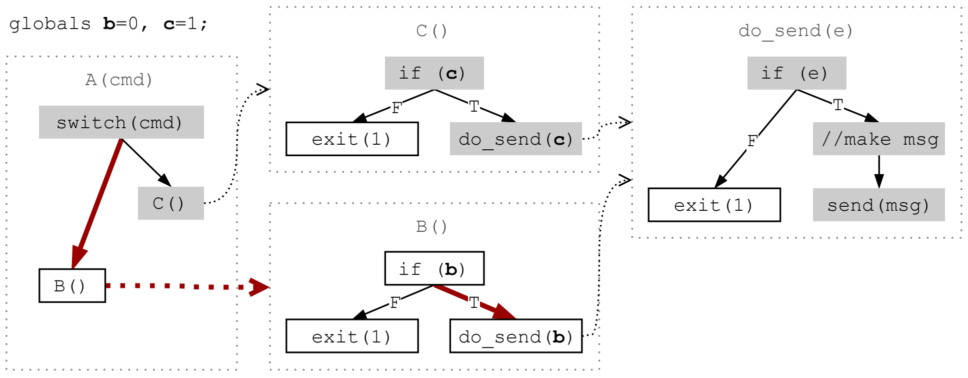


Figure 7.1: An example where control-flow changes do not enable a different message to be sent.

`do_send(b)` in function `B()`. To reach this block InSource computes the path `switch(cmd)-> B()-> if(b)-> do_send(b)` and executes the source program. These changes to control-flow allow the source program to pass the first sanity check `if(b)` even if the global variable `b` is not set, but the source still fails the check `if(e)` when the variable `e` that is passed in as an argument to `do_send` is set to `0` (as is the case here). This kind of scenario occurs less frequently in the TLS and XMPP source clients that we used, since, in these programs, downstream functions like `do_send` simply assume that they are given correct argument values. To make the check `if(e)` succeed, InSource would have to compute a path traversing all the way down to `send(msg)`; however, we do not yet support computation of paths that pass through the default control-flow blocks in the source program (gray blocks in the figure), which constrains us in scenarios such as this one. Furthermore, we do not take the approach of setting data in the program (e.g., variable `b`) to guide control-flow, since it is our goal to have the ability to generate non-standard message sequences, which do not entirely follow the internal logic of the program.

Finally, we are limited by the programming language used to implement the source application, since we only implement our instrumentation in context of instructions that are used in LLVM when compiling C programs. For example, we do not handle exception handling control-flow present in C++. While `longjmp` and `setjump` instructions are also present in C programs, we don't include them in control-flow path computations because they are rarely used and thus don't have a large influence on which paths we can compute in the source program. Finally, we disable multi-threading when compiling a source program in order to be able to use count execution numbers when forcing branches (such as in the example in Figure 4.9). While we plan to extend the support for more languages that use LLVM to implement their compilers, we do not currently have plans to support any programming languages that do not use LLVM.

7.2 Fuzzing limitations

InSource is able to fuzz individual message fields in protocol messages by overwriting them before the messages are constructed in the source application; however, the amount of valid message fields that are selected is constrained by how well the tester using InSource is able to select the files that are related to message generation. While we found this to not be a difficult task, we hope to automate this step in the future. Furthermore, by making the selection at a per-file granularity, a tester inadvertently selects some functions that do not relate to any message fields. In these cases, the fuzzing instances that are used to fuzz these values do not contribute any useful work, and simply take away computational power from those instances that do fuzz valid fields. Implementing additional automated checks will allow us to alleviate this problem in future versions of InSource.

When we perform fuzzing of various targets, we are constrained by existing generic fault-detection capabilities available for various kinds of targets, and are thus not able to find all kinds of bugs in all possible targets. While support for generic low-level security bug detection techniques in white-box targets is excellent (see Section 3.3.3 for a discussion of available techniques), generic fault detection techniques used for black-box targets don't always catch all bugs. As discussed in Section 3.1.3, manually-constructed protocol-specific testing tools that implement protocol-specific fault-detection methods can catch more errors in the target's implementation. For example, TLS-Attacker [84] is able to not only find low-level security vulnerabilities, but also to detect subtle oracles that can be used to attack TLS encryption stacks. We hope to, in the future, automate the creation of some of the fault-detection techniques that are employed by such testing tools in order to be able to detect more bugs in black-box targets.

Finally, one of the challenges that is unique to testing network protocol applications is the speed at which different test iterations can be performed. Since we re-execute the source application on each test iteration, we are constrained by the execution speed of the source program. In cases when the source application is able to complete execution and exit normally, we can quickly restart it and let it execute again. However, in the context of fuzzing, a source application frequently times out, bounding the execution speed of the fuzzer by the timeout set for the source application. Client timeouts happen frequently when fuzzing, because a fuzzed message can trigger the target to wait. While the target is waiting for the source, the source is also waiting for the target, causing deadlock in execution, and necessitating a timeout to be used. While we plan to develop techniques to address this for gray-box and white-box targets, we do not know of a way to develop a general solution to this problem that would work for black-box targets as well. In addition, we plan to improve runtime by reducing the overhead induced by our instrumentation (exact amounts of overhead are shown in Appendix C), but note that the timeout induced by the client is a more severe bottleneck.

7.3 Ethical considerations

While testing a network protocol implementation it is beneficial to run the target application in its natural environment; for example, when testing a target application that performs a DNS look-up on an address provided by the source application, allowing the look-up to return real values from a real DNS server can help the target code pass the DNS lookup step and proceed to other processing logic. As a side-effect of sending many inputs to the target, however, the DNS server may end up becoming overloaded during testing. Indeed, we observed high loads on a local DNS server while fuzzing the SIP target used in our evaluation, resulting in denial of service for other users of the network. While the effects in this scenario were isolated, we highlight the need to consider the potential consequences of testing network targets that may depend on external network nodes. As we move towards testing even more complex targets, we strive to strike a balance between correctness of execution of the target applications and the sufficient isolation of the target during testing.

Chapter 8

Conclusion

The complexities of stateful network protocols make them difficult to test for low-level security vulnerabilities. As such, testers today rely on manually-constructed protocol-specific testing tools to test such target applications. However, the creation of effective manually-constructed protocol-specific testing tools for stateful protocol applications is difficult because of the complexities ingrained in these protocols. Furthermore, the growing number of heterogeneous network protocols being used today, makes the development of multiple protocol-specific testing tools a daunting task.

To address the challenges of manually developing protocol-specific testing tools, we developed a technique for converting existing applications that already implement network protocols, into effective protocol-specific testing tools. We implement this technique in a tool called InSource, which can test any network protocol application, for which there exists a corresponding open-source program implementing the same protocol. We find that InSource is able to perform comparably to a manually-constructed testing tools on a real, complex network protocol, despite not being implemented using manual techniques for defining network protocol features. Furthermore, we show that InSource works well across different network protocols like TLS, XMPP, and SIP, suggesting that our approach can be effectively applied to other network protocols as well.

Overall, we make the conclusion that the use of a source application to test a target is an efficient way to automate parts of the process of creating protocol-specific testing tools, and hope to use it for testing more target applications. While, during the timespans used of our evaluation, we did not find low-level software vulnerabilities in the programs that we targeted, we demonstrated the efficacy of InSource along other metrics used for comparing testing tools. As such, we believe that we will eventually find security vulnerabilities by running longer testing campaigns using InSource, and by running InSource on a larger variety of target applications.

Appendix A

Number of call paths in source programs

Table A.1 shows how many paths are found in the source application by InSource using each path heuristic used for reaching targeted blocks. The main takeaway from this result is that when larger programs are used as the source application, such as OpenSSL, the conservative heuristics like "All paths" result in a large number of paths that need to be tested by InSource. Instead, by using the heuristics "Shortest paths" or "3 simplest paths" a tester can allow InSource to perform the state exploration stage more quickly. Another takeaway from this table is that adding consideration of "all counts" used (i.e., the iterations on which a default control-flow branch is forced), does not significantly impact the amount of paths that InSource needs to try.

	OpenSSL client		**libstrophe**		**baresip**	
	Pred. to `write`	Blocks in all files	Pred. to `xmpp_send`	Blocks in all files	Pred. to `sendto`	Blocks in all files
Target blocks	65k	413k	4k	30k	24k	152k
Shortest paths	51k	180k	3k	11k	16k	58k
All paths	910k	3,215k	4k	17k	156k	566k
Shortest + all counts	68k	235k	3k	19k	19k	72k
All + all counts	1,344k	4,700k	5k	27k	316k	1,140k
3 simplest paths	136k	479k	4k	15k	42k	153k

Table A.1: Number of control-flow paths found in each source application using different heuristics for path selection. "Pred. to `<function>`" shows the number of paths that are computed to reach the targeted blocks that are selected which are predecessors to the specified function. "Blocks in all files" shows the same information but when considering all blocks in the source program as targeted blocks.

Appendix B

Number of states found in target

Here we compare the effectiveness of different path heuristics using the libstrophe [59] source application, since this program is small enough to allow comparing all the possible approaches. Table B.1 shows the effectiveness of each heuristic by identifying the number of states found on the QXMPP target server, as well as the number of lines covered by executing the target together with a client that has its control-flow modified by InSource. One takeaway from this table is that by picking targeted blocks which are predecessors to a network send function (in this case `xmpp_send`), we can reduce the number of test iterations needed to discover almost as many states in the target as through considering all blocks in the source program as targeted blocks. The second takeaway from the statistics shown is that by considering "all counts" of a control-flow path in the source application, we can increase the coverage attained by testing the target application with a modified client, meaning that this is a useful heuristic to consider. Note also that, as seen in the table, discovering more states in the target allows us to cover more code in our tests.

Heuristic	# source paths	# target states	# target lines cov.
No exploration	1	8	1365

	# source paths		# target states		# target lines cov.	
Heuristic	Pred. to `xmpp_send`	Blocks in all files	Pred. to `xmpp_send`	Blocks in all files	Pred. to `xmpp_send`	Blocks in all files
Shortest paths	2.7k	10.8k	74	87	1646	1674
All paths	4.2k	16.6k	85	96	1646	1674
Shortest + all counts	3.1k	19.3k	78	102	1682	1682
All + all counts	5.2k	27.0k	88	111	1682	1682
3 simplest paths	3.9k	15.0k	82	98	1646	1674

Table B.1: Coverage gained and number of states found by applying different path heuristics when discovering states in the QXMPP server by using a libstrophe client. The top row shows the number of states that are know in the server just from running the default control-flow in the source.

In practice, we resort to using either the "Shortest path + all counts" or the "3 simplest paths" path selection heuristics, since they provide a good balance between trying fewer control-flow paths on a source application and attaining better coverage on the target. We also use all blocks when possible, but resort to using only blocks that are predecessors to network calls when the source program is large.

Appendix C

States found by ignoring memory errors

By ignoring memory errors in the source application we allow it to keep running even when not enough information is available for it to generate new messages. Specifically, we ignore segmentation faults as a blunt way to prevent the source from crashing prematurely. We find that using this simple technique to increase source program resilience has allowed us to force the source application to execute messages that it would not otherwise have executed. In fact, the messages that it executes help InSource discover new stats in the target.

Table C.1 shows that some states in the target are only found by keeping the source application running despite memory errors. The last row indicates the total number of states found on the target (these are also the same states that are fuzzed during evaluation of InSource). It is interesting to realize that ignoring segmentation faults in the source enables us to discover more states in the target program; while only 9% of states found in the QXMPP target can be attributed to this technique, 25% of states found in the PJSIP target are discovered by messages that are generated in a source application *after* a memory error occurs and is intentionally ignored.

States type	**OpenSSL**	**QXMPP**	**PJSIP**
Found with memory errors in source	12	9	23
Found without memory errors in source	98	93	68
Total number of target states	110	102	91

Table C.1: The number of states found in different targets by ignoring segmentation faults in the corresponding source applications.

Appendix D

Instrumentation overhead

To measure overhead, we execute the client together with the server, and measure the amount of time that passes before each network message is sent by the client. We discount the time spent by the client waiting for a network response from the server. The total amount of time needed to execute each client and the time per message are shown in Figure D.1. Each bar is separated into sections, where each section shows the amount of time taken to construct each network message sent by the client (e.g. the OpenSSL client sends 4 TCP messages, and the last section shows how long it takes for the client to exit after sending the last message). The instrumentation types shown are: "none" is no instrumentation, "branch" is used for forcing branches, "call" is used to force indirect calls, "buffer" is responsible for overwriting function call arguments (i.e., individual message fields). To measure runtime, each client is executed 5000 times and the average runtime per message is taken.

The total overhead induced by all of our instrumentation combined is relatively low: 63% for OpenSSL, 5% for libstrophe, and 62% for baresip. To put this in perspective, the overhead of AFL's coverage instrumentation has been measured to be around 36% [62]. It is interesting to note that the overhead of libstrophe is noticeably lower than for other clients. This is likely due to the fact that libstrophe delegates a lot of computation to external libraries (e.g., libxml), whereas other source programs implement most of their own features.

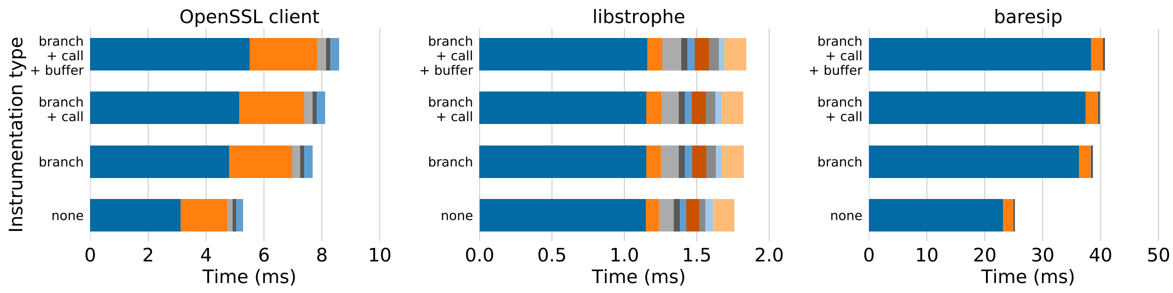


Figure D.1: The amount of time each client is executed with different levels of instrumentation.

Bibliography

[1] Dave Aitel. The advantages of block-based protocol analysis for security testing. *Immunity Inc., February*, 105:106, 2002.

[2] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. Announcing OSS-Fuzz: Continuous fuzzing for open source software. *Google Security Blog*, 2016.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono Delia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[5] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: Toward a stateful network protocol fuzzer. In *International Conference on Information Security*, pages 343–358. Springer, 2006.

[6] Marshall A Beddoe. Network protocol analysis using bioinformatics algorithms, 2004.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[8] Black box testing for software and hardware. `https://www.beyondsecurity.com/blog/black-box-testing`, 2018.

[9] Benjamin Beurdouche, Antoine Delignat-Lavaud, Nadim Kobeissi, Alfredo Pironti, and Karthikeyan Bhargavan. FLEXTLS: A tool for testing TLS implementations. In *9th USENIX Workshop on Offensive Technologies, WOOT'15*, 2014.

[10] Doug Birdwell. Network-AFL - AFL for network fuzzing. `https://github.com/jdbirdwell/afl`, 2015.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344. ACM, 2017.

[12] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 114–129. IEEE, 2014.

[13] Derek Bruening and Saman Amarasinghe. *Efficient, transparent, and comprehensive runtime code manipulation.* PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering , 2004.

[14] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 317–329. ACM, 2007.

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[16] DARPA cyber grand challenge binaries. `https://github.com/CyberGrandChallenge/`, 2015.

[17] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2095–2108. ACM, 2018.

[18] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing. *Proc. 2018 NDSS, San Diego, CA*, 2018.

[19] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725. IEEE, 2018.

[20] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. SAVIOR: Towards bug-driven hybrid testing. *arXiv preprint arXiv:1906.07327*, 2019.

[21] Yuting Chen and Zhendong Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 793–804. ACM, 2015.

[22] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Security Symposium*, pages 1–14, 2007.

[23] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.

[24] CVE-2014-0160. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`, 2014.

[25] CVE-2019-3568. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3568`, 2019.

[26] Joeri De Ruiter and Erik Poll. Protocol state fuzzing of TLS implementations. In *USENIX Security Symposium*, pages 193–206, 2015.

[27] Defensics fuzz testing. `https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html`, 2015.

[28] Synopsys: Fuzzing test suites. `https://www.synopsys.com/software-integrity/security-testing/fuzz-testing/defensics.html`, 2019.

[29] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, August 2008.

[30] Vaibhav Hemant Dixit, Adam Doupé, Yan Shoshitaishvili, Ziming Zhao, and Gail-Joon Ahn. AIM-SDN: Attacking information mismanagement in SDN-datastores. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 664–676. ACM, 2018.

[31] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. LAVA: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[32] Pete Evans. Heartbleed bug: RCMP asked Revenue Canada to delay news of SIN thefts. *CBC*, 2014.

[33] Paul Fiterău-Broştean, Toon Lenaerts, Erik Poll, Joeri de Ruiter, Frits Vaandrager, and Patrick Verleg. Model learning and model checking of SSH implementations. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 142–151. ACM, 2017.

[34] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path sensitive fuzzing. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[35] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. PULSAR: stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*, pages 330–347. Springer, 2015.

[36] The state of the Octoverse. `https://octoverse.github.com/projects#languages`, 2018.

[37] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[38] The hacker-powered security report 2018. `https://www.hackerone.com/resources/hacker-powered-security-report`, 2018.

[39] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *USENIX Security Symposium*, pages 49–64, 2013.

[40] HyungSeok Han and Sang Kil Cha. IMF: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358. ACM, 2017.

[41] Kenneth V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[42] Alfred Heggestad. Baresip - open source SIP user agent. `http://www.creytiv.com/baresip.html`, 2014.

[43] Aki Helin. A crash course to radamsa. `https://gitlab.com/akihe/radamsa`, 2019.

[44] Honggfuzz - security oriented fuzzer with powerful analysis options. `http://honggfuzz.com/`.

[45] Ulf Kargén and Nahid Shahmehri. Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 782–792. ACM, 2015.

[46] Hubert Kario. tlsfuzzer - SSL and TLS protocol test suite and fuzzer. `https://github.com/tomato42/tlsfuzzer`, 2015.

[47] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the LTE control plane. In *Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane*. IEEE, 2018.

[48] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2018.

[49] Frederic Lardinois. Google open sources ClusterFuzz. *TechCrunch*, 2019.

[50] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[51] Dave Lee. WhatsApp discovers 'targeted' surveillance attack. *BBC*, 2019.

[52] Seungsoo Lee, Changhoon Yoon, Chanhee Lee, Seungwon Shin, Vinod Yegneswaran, and Phillip A Porras. DELTA: A security assessment framework for software-defined networks. In *NDSS*, 2017.

[53] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 809–814. ACM, 2018.

[54] libFuzzer - a library for coverage-guided fuzz testing. `http://llvm.org/docs/LibFuzzer.html`.

[55] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *NDSS*, volume 8, pages 1–15, 2008.

[56] Jonathan Looney. Linux and FreeBSD kernel: Multiple TCP-based remote denial of service vulnerabilities. `https://github.com/Netflix/security-bulletins/blob/master/advisories/third-party/2019-001.md`, 2019.

[57] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[58] mcsema - Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. `https://github.com/trailofbits/mcsema`, 2014.

[59] Jack Moffitt. Libstrophe - a simple, lightweight C library for writing XMPP clients. `https://github.com/strophe/libstrophe`, 2013.

[60] mozPeach - fuzzing framework which uses a DSL for building fuzzers. https://github.com/MozillaSecurity/peach, 2015.

[61] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[62] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. *CoRR*, abs/1812.11875, 2018.

[63] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[64] Brian S Pak. Hybrid fuzz testing: Discovering software bugs via fuzzing and symbolic execution. *School of Computer Science Carnegie Mellon University*, 2012.

[65] Peach - a smart fuzzer that is capable of performing both generation and mutation based fuzzing. http://community.peachfuzzer.com/WhatIsPeach.html, 2004.

[66] Peach Tech - discover unknown vulnerabilities. https://www.peach.tech, 2015.

[67] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710. IEEE, 2018.

[68] Joshua Pereyda. boofuzz - a fork and successor of the Sulley fuzzing framework. https://github.com/jtpereyda/boofuzz, 2015.

[69] PJSIP - open source SIP, Media, and NAT traversal library. https://www.pjsip.org/, 2016.

[70] PROTOS - security testing of protocol implementations. https://www.ee.oulu.fi/roles/ouspg/Protos, 1999.

[71] 3GPP agrees on plan to accelerate 5G NR — the global 5G standard — for 2019 deployments. https://www.qualcomm.com/news/onq/2017/03/09/3gpp-agrees-plan-accelerate-5g-nr-global-5g-standard-2019-deployments, 2017.

[72] Nguyen Quynh. Capstone: The ultimate disassembler. http://www.capstone-engine.org/, 2013.

[73] QXMPP - cross-platform C++ XMPP client and server library. https://github.com/qxmpp-project/qxmpp, 2014.

[74] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, 2015.

[75] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[76] Matt Ruhstaller and Oliver Chang. A new chapter for OSS-Fuzz. *Google Security Blog*, 2018.

[77] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

[78] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71. ACM, 2009.

[79] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. *USENIX Security Symposium*, 2017.

[80] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *IEEE Security & Privacy*. IEEE, 2018.

[81] Yan Shoshitaishvili. Preeny - some helpful preload libraries for pwning stuff. `https://github.com/zardus/preeny`, 2015.

[82] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[83] Suphannee Sivakorn, George Argyros, Kexin Pei, Angelos D Keromytis, and Suman Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *Proceedings of the 38th IEEE Symposium on Security & Privacy*, 2017.

[84] Juraj Somorovsky. Systematic fuzzing and testing of TLS libraries. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1492–1504. ACM, 2016.

[85] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[86] Chris McMahon Stone, Tom Chothia, and Joeri de Ruiter. Extending automated protocol state learning for the 802.11 4-way handshake. In *European Symposium on Research in Computer Security*, pages 325–345. Springer, 2018.

[87] Sulley - a pure-python fully automated and unattended fuzzing framework. `https://github.com/OpenRCE/sulley`, 2012.

[88] State of fuzzing 2017: Where the zero days are, 2017.

[89] The connected future: Internet of things forecast. `https://www.ericsson.com/en/mobility-report/internet-of-things-forecast`, 2019.

[90] UndefinedBehaviorSanitizer - a fast undefined behavior detector. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`, 2019.

[91] Guillaume Valadon and Arnaud Ebalard. Scapy - the python-based interactive packet manipulation program & library. `https://github.com/secdev/scapy`, 2015.

[92] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. Security testing of GSM implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 179–195. Springer, 2014.

[93] Vulnerabilities by type. `https://www.cvedetails.com/vulnerabilities-by-types.php`, 2019.

[94] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 579–594. IEEE, 2017.

[95] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 724–735. IEEE Press, 2019.

[96] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 497–512. IEEE, 2010.

[97] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in OS kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913. ACM, 2018.

[98] David Wheeler. Flawfinder. `https://dwheeler.com/flawfinder/`, 2001.

[99] XMPP extensions. `https://xmpp.org/extensions/`, 2019.

[100] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, pages 283–294. ACM, 2011.

[101] Eric Young and Tim Hudson. OpenSSL - TLS/SSL and crypto library. `https://www.openssl.org/`, 1999.

[102] Michał Zalewski. American Fuzzy Lop (AFL). `http://lcamtuf.coredump.cx/afl/`, 2013.

[103] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: High-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.