

Vrije Universiteit Amsterdam



MASTER THESIS

Safe Patch Fingerprinting

Author:

**Nathan
Schagen**

Supervisors:

**Koen Koning (Msc)
Dr. Cristiano Giuffrida
Prof. Dr. Herbert Bos**

*A thesis submitted in fulfilment of the requirements
for the degree of Parallel and Distributed Computer Systems*

in the

Faculty of Sciences
Department of Computer Science

August 2017

Declaration of Authorship

I, Nathan SCHAGEN, declare that this thesis titled, 'Safe Patch Fingerprinting' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Master's degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

VRIJE UNIVERSITEIT AMSTERDAM

Abstract

Faculty of Sciences

Department of Computer Science

Parallel and Distributed Computer Systems

Safe Patch Fingerprinting

by Nathan SCHAGEN

In this thesis, we explore a novel method called *safe patch fingerprinting*. It allows for potentially automated discovery of *inputs* that safely discriminate vulnerable from patched servers for the latest vulnerabilities. This enables rapid updates to scanning tools, allowing administrators to scan and secure their networks more quickly. To ensure such scans are safe, it is required to reject inputs having malicious side-effects.

We focused on testing both the safety and this discriminative property for given inputs. We have implemented a framework that uses delta execution, to successfully recognize these so called *discriminators* for both the infamous heartbleed vulnerability and a number of vulnerable test servers we built ourselves.

We made further attempts to automate the process by generating candidate inputs which were then tested using the aforementioned framework. This allowed us to automatically find discriminators for most of our vulnerable test servers. Although many challenges are ahead, we provide many insights that can lead to the development of more effective patch fingerprinting tools.

Acknowledgements

This project initially sparked my interest because potentially had a high impact and allowed me to learn many different things about computer systems and security. However, doing this project with a demanding part-time job on the side proved very challenging. Also, it turned out that many problems were more difficult than initially assumed. Finally, I have high expectations of myself, which drives me to achieve great things, but can also cause my productivity to grind to a halt when hope of reaching my goals diminishes. During the last 2 years and 2 months, I had multiple episodes where I was about to give up on this project. Now that I've made it, it seems appropriate to thank some people for their help and support.

First of all, I would like to thank my parents, Cees Schagen and Anja Kroon, for their unwavering support, especially at times when I was losing hope of ever finishing the project. I would also like to thank Chris Dekker for his support and understanding and bringing up my thesis at parties. He could relate to my struggles, but also made me laugh about it at times. Christian Ouwehand is a longtime friend of mine who is currently working on his thesis in Herbert's group. I would like to thank him for the conversations we had about the challenges we both faced in our own journeys. As opposed to a work environment, writing a thesis is a highly solitary job and having people to discuss the subtleties of your work with helps you to move forward.

I would also like to thank Dirk Jonker, my employer, for giving me space and time to finish this project. Working on my thesis in the office helps against the feeling of isolation I experienced after working at home for a few days. Many of my colleagues know what thesis writing is like and I thank them for their support.

Last but not least, I would like to thank Koen Koning, Cristiano Giuffrida and Herbert Bos for their patience and guidance while I was ploughing through all the challenges that I encountered. I often had unreasonable expectations of myself, because I was aiming for near-perfect solutions and finishing the project as soon as possible. Not being able to live up to such high standards had negative effects on my confidence about the project. Fortunately, my supervisors were understanding and made me feel better about my achievements. Besides that, being part of the world of systems security research whilst writing my thesis has taught me a lot.

Contents

Declaration of Authorship	i
Abstract	ii
Acknowledgements	iii
List of Figures	vii
List of Tables	ix
1 Introduction	1
2 Design	5
2.1 Discriminators	5
2.1.1 Infection	7
2.1.2 Safety	7
2.1.3 Propagation	9
2.1.4 Examples	10
2.2 Discriminator testing	11
2.2.1 Delta execution	12
2.2.2 Confidentiality	13
2.3 Discriminator discovery	14
2.3.1 Symbolic Execution	15
2.3.2 Concolic Execution	15
2.3.3 Fuzzing	16
2.3.4 Hybrid approaches	17
3 Implementation	19
3.1 Deltafy Compiler Pass	19
3.2 Libdelta runtime library	21
3.2.1 Splitting	21
3.2.2 Merging	22
3.2.3 Instrumenting I/O functions	24
3.2.4 Whitelisting	24
3.2.5 Deltastub	25
3.3 Sanitizer integration	25

3.3.1	Compilation	26
3.3.2	Initialization	26
3.3.3	Thread-local-storage	26
3.3.4	Controlling the heap	26
3.3.5	Interposing functions	28
3.4	Fuzzing	29
4	Vulnerability Class Analysis	31
4.1	Null-pointer dereferencing	31
4.2	Integer overflows	32
4.3	Use-after-free	33
4.4	Buffer overflows	34
4.5	Buffer over-reads	35
4.6	Conclusions	35
5	Results	36
5.1	Heartbleed	36
5.1.1	Propagation in heartbleed	38
5.1.2	Safety in heartbleed	39
5.1.3	Infected state in heartbleed	40
5.1.4	Fuzzing	40
5.2	Experiments	41
5.2.1	Echo server 1	41
5.2.2	Echo server 2	42
5.2.3	Replace server	42
5.2.4	Quote server	44
6	Related work	46
6.1	Fingerprinting in network reconnaissance	46
6.2	Web vulnerability scanners	47
6.3	Mutation testing	48
7	Discussion	52
7.1	Discriminator testing	53
7.1.1	Limitations	53
7.1.2	Accuracy of splitting and merging	53
7.1.3	Inspecting Diverged State	55
7.1.4	Retaining Diverged State	56
7.1.5	Confidentiality	57
7.2	Discriminator Discovery	58
7.3	Alternative applications	59
7.4	Future work	60
8	Conclusion	62

Bibliography

List of Figures

1.1	This figure shows a patched and an unpatched server processing the same request. The request is a discriminator because it causes both versions to respond differently as indicated by the colors	2
2.1	Overview of the patch fingerprinting framework. The boxes indicate the two subsystems and the arrows show their interactions and their inputs and outputs	6
2.2	Echo server 1 contains a buffer overread vulnerability, allowing attackers to leak memory contents.	10
2.3	Echo server 2 contains a buffer overflow vulnerability. It can be safely detected using a small overflow that only overwrites padding bytes	11
3.1	The deltafy pass merges the patched and unpatched version. It replaces modified functions with proxies, which forward the call to either the original or patched version of the function	20
3.2	Shows all LLVM compiler passes used to generate instrumented binaries that work with libdelta	21
3.3	Shows how an instrumented program splits and merges when a patched function F is called	22
3.4	The VUzzer uses a session script to fuzz the instrumented server	30
4.1	Shows a venn-diagram of the RIPS model. The blue-ish parts are all safe (S) but only with respect to the vulnerability being analyzed. This picture may be different per vulnerability as some sets may be empty or may not overlap with others.	32
4.2	The original and patched code of CVE-2017-7308	33
5.1	The patch for heartbleed (CVE-2014-0160)	38
5.2	Replace server replaces characters in a string and sends the result back. It contains a buffer overflow.	43
5.3	The quote server escapes quotes (and backslashes) and returns the escaped string. A buffer is overflowed when a string contains too many characters that need to be escaped.	44
6.1	Shows a mutation on the $<$ operator. The state (variable c) is only infected when $a == b$	50

-
- 7.1 Left side shows *write-before-merge* resulting in a lot of manual work to analyze/whitelist all diverged state. Right side shows *merge-before-write*. When the merge is successful, the patched process exists and all infected state is lost even though whitelisted infections may still exist. Any propagation that may happen due to the subsequent `write()` call will not be detected by our delta framework 54
- 7.2 This is what could happen if we retain diverged state. The first merge checks the integrity of the process but retains the diverged state. As a result, the execution splits again when this diverged state is accessed and used by a `write()` call 56

List of Tables

5.1	Patched test programs for which we know that discriminators exist	41
5.2	Shows number of discriminators, safe (S) and propagating (P) inputs found for each toy program.	45

Chapter 1

Introduction

Reconnaissance is the first step (ethical) hackers take to see whether a system can be penetrated. Their end goal is to prove a system is vulnerable without doing any damage. By selecting a number of target hosts and gathering information about them, hackers can explore increasingly specific attack vectors. They use techniques such as port-scanning, DNS lookups, banner grabbing, OS fingerprinting and vulnerability scanning. All of these, except vulnerability scanning, do not directly indicate that the system is vulnerable but rather allow an attacker to focus his search.

Banners are strings describing a service and the host, which can be easily fetched after connecting to the said host. They are intended for use by system administrators, but since hackers can access them, they are now often hidden.

OS fingerprinting allows attackers to learn the host operating system by analyzing the response to particular TCP/IP packets. The technique exploits ambiguities in the protocol specifications which were interpreted or at least implemented differently across networking stacks. OS fingerprinting is still effective, but does not give detailed information such as the kernel version. Therefore it is only useful in the very early stage of reconnaissance.

The concept of OS fingerprinting can also be applied to server applications. [1] shows that it is possible to fingerprint many Apache 1.3.x versions based on server responses. Fingerprinting application versions is interesting to attackers as some versions are known to be vulnerable. Yet, security patches are often backported, so we cannot use a version number to get a conclusive answer. Also, servers can be configured in a wide variety of ways, complicating creation of fingerprints for specific versions.

Vulnerability scanning identifies hosts in a network that susceptible to a particular attack. The process is highly automated, while also providing actionable output that

increases the security of scanned hosts. However, vulnerability scanners often report false positives, thus manual validation is necessary. Various scanners exist, such as Nessus[2], Secubat[3] and Acunetix[4] which can scan for vulnerabilities remotely.

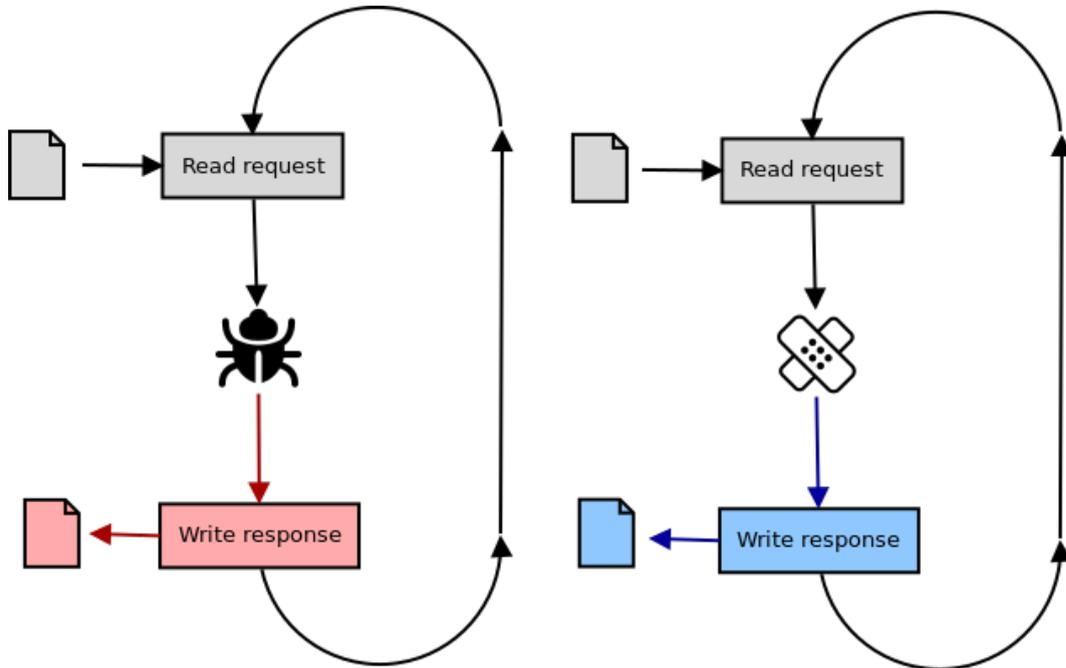


FIGURE 1.1: This figure shows a patched and an unpatched server processing the same request. The request is a discriminator because it causes both versions to respond differently as indicated by the colors

Most of the research on vulnerability scanners is focused on scanning web applications. These are much easier to scan than compiled server applications because the vulnerability classes such as *cross-site-scripting (XSS)* and *SQL injection* are well understood and can be detected remotely without exploitation. This is discussed in section 6.2. Vulnerability scanning of compiled server applications is more complicated because the vulnerabilities are more application and architecture specific. Also, since many of these vulnerabilities are memory errors, scanning can very well result in corruption or crashes. Therefore, it takes manual effort from security experts to develop plugins that can scan hosts in a safe manner. Development of such plugins requires discovery of a particular request to send to a server that reveals the presence of the vulnerability in a non-disruptive way.

In this thesis, we investigate a novel method we call *safe patch fingerprinting*, which aims to fingerprint behavioral changes of a server as a result of applying a security patch. More specifically, we want to discover specific inputs which provoke a response that allows us to remotely discriminate between patched and unpatched versions. We shall call such inputs *discriminators*. The concept is visualized in figure 1.1. With an *input* to a server, we refer to the stream of bytes delivered after establishing a new

connection. Such a stream may be different every time, due to interaction between the client and the server, timestamps, crypto etc.

Automating the fingerprinting process will aid the rapid development of vulnerability scanner plugins. When new security vulnerabilities are discovered, it is essential to distribute a plugin as soon as possible. Yet, we must ensure that such plugins do not harm production servers during a scan. This can be quite challenging since many different versions of a vulnerable server may be in production and manually testing a vulnerability scanner plugin against all versions may be prohibitively time consuming.

Because this method is patch-based, its detection capabilities may generalize across many versions of the vulnerable server. Open-source projects may be forked and modified many times, yet all variations of the software could allow detection using the same discriminator.

Note that exploitation of the vulnerability trivially allows detection of vulnerable servers, but may also do damage or leak sensitive information. We aim to find non-malicious discriminators. What 'malicious' exactly means depends on the vulnerability that is being analyzed. Ideally, we want to be stealthy by sending the least number of packets possible and also have the ability to prove we did not do any harm, based on the packet trace.

Not all vulnerabilities or patches may be fingerprintable. When the vulnerable code is buried deep inside the application, it may not affect any data that is send back to the attacker.

Our main research question is whether safe patch fingerprinting is practical and whether it can be automated. We mainly focus on *discriminator testing*, which involves testing whether a given input has the discussed discriminative property. We use a technique called *delta execution*, which allows analyze how both the unpatched and patch server respond under exactly the same circumstances. Although the technique works in our experiments, we were limited by the lack of known discriminators.

Furthermore, we made a first attempt to automate discovery of new discriminators. We leveraged our discriminator tested and used a fuzzer to generate candidate inputs. This seems to work well for very small programs, but more work is needed to make the technique scale to larger pieces of software. Our contributions are:

- A useful formalization of the safe patch fingerprinting problem, which we call the RIPS model. It was inspired by existing work on mutation testing, discussed in section 6.3.

- A delta-execution framework that confirmed a safe discriminator for the *heart-bleed* vulnerability(CVE-2014-0160). and helped us to detect discriminators for a number of simple test servers.
- An adaptation of an evolutionary coverage-guided fuzzer that helped us find discriminators automatically for simple test servers.
- Recommendations on how to design more powerful discriminator testing and discovery instrumentation based on our findings.

In chapter 2, we will discuss the design of both our discriminator testing and discovery tools and in chapter 3 we will discuss their implementation. We give a qualitative analysis on the applicability of safe patch fingerprinting on various types of vulnerabilities in chapter 4. In chapter 5, we will discuss the results of both discriminator testing and discovery for heartbleed and a selection of test programs, followed by the related work and discussion in chapters 6 and 7 respectively. We conclude this thesis with chapter 8.

Chapter 2

Design

Fingerprinting a patch is about finding a discriminator. Doing this automatically is challenging, because of the sheer number of constraints that must be satisfied. For that reason, we first focused on *discriminator testing*, which is a method to decide whether an input safely discriminates the patch and unpatched version. Next, we attempted *discriminator discovery*, where we aim to find discriminators automatically. Our discovery approach uses our earlier developed discriminator testing system and generates candidates to be tested. The whole system is shown in figure 2.1.

Given a vulnerability and its patch, we compile both the original and the patched version of the server. A fuzzer is then used to generate a set of inputs that may be discriminators. We bootstrap the fuzzing process using a number of *seed inputs*, which we need to create manually using the exploit script. Once candidate inputs are generated, we can test whether any of them is a discriminator. We do this by passing them into an instrumented server, which uses delta execution to analyze how both versions of the server behave. When the input didn't cause problems, but provoked a different response, we have found a discriminator.

In the next section, we will explain in depth what a discriminator is and give some examples. We then describe how we test whether an input is a discriminator, using delta execution. Finally, we discuss ways to discover potential discriminators.

2.1 Discriminators

A patch typically changes one or multiple sets of lines, which can be in distinct functions. These are referred to as *hunks* by diffing an patching tools. After compilation, this results in a one or more groups of modified instructions, which we refer to as *patch-sites*.

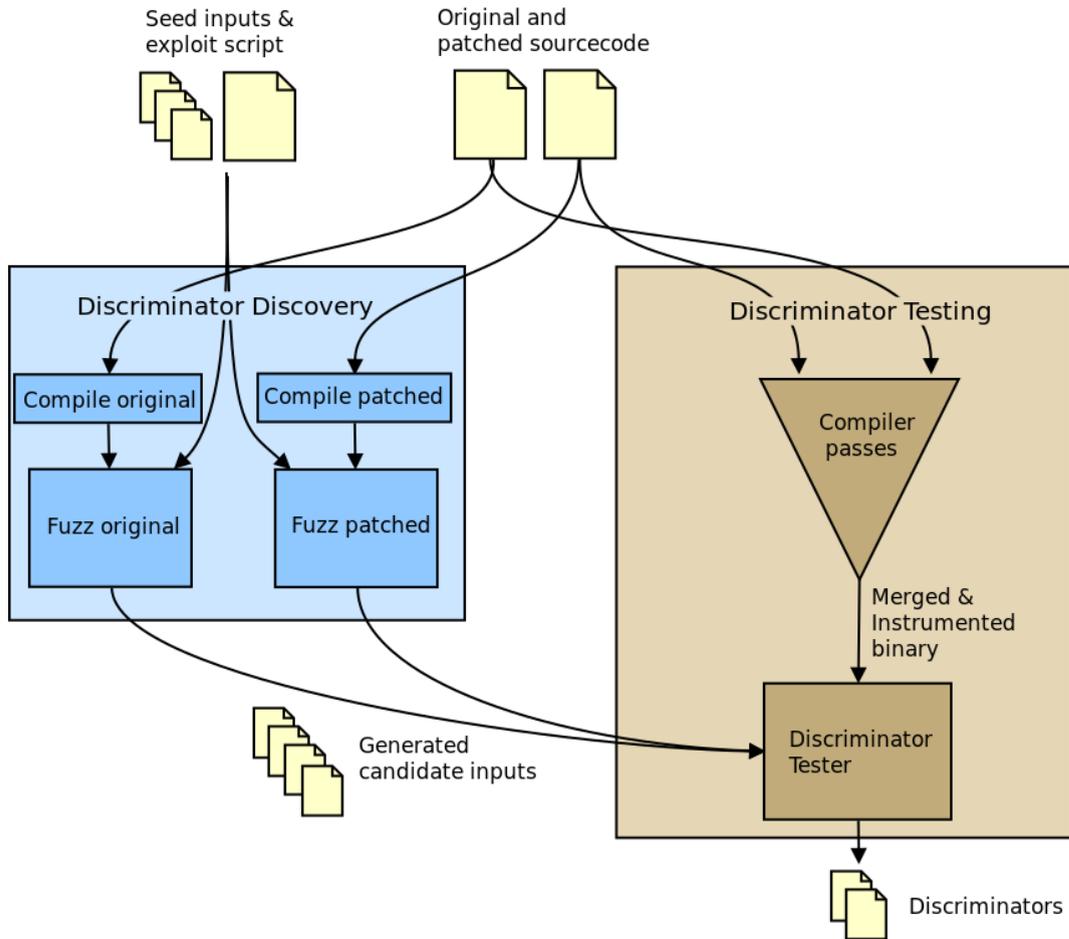


FIGURE 2.1: Overview of the patch fingerprinting framework. The boxes indicate the two subsystems and the arrows show their interactions and their inputs and outputs

Discriminators must have the following properties:

- The program must **reach** a *patch-site*. In other words, the input must cause added or modified instructions to be executed. In case the patch has removed instructions, they should be executed when sending the same input to the original unpatched server.
- The input must **infect** the program state resulting in different states for the patched and the unpatched server after executing one or more patched instructions.
- The *infected program state* must **propagate** to the output, such that the difference between the patched and unpatched server can be observed. Inputs that have this property are said to be *propagating*.
- It must be **safe**, meaning that input does not exploit the vulnerability in a harmful way. What this means depends on the vulnerability. Therefore, the safety requirements must be established by an analyst.

To summarize, we need to test whether an input exercises the vulnerable code, resulting in a state divergence causing an observable effect, discriminating the two versions, without exploiting the vulnerability. This input is said to *reach* the patch, *infect* the program state and *propagate* the infected state to the output in a *safe* manner. Since *propagation* implies *infection* and *reachability*, we can say that discriminators are safe propagating inputs. The *reachability*, *infection*, *propagation* and *safety* properties together constitute to the *RIPS model*, which is derived from the *RIP model*, used in mutation testing [5][6][7].

Besides testing whether an input is a discriminator, we will also attempt to find discriminators ourselves. We start our search using an exploit for the vulnerability which is assumed to be known. The inputs generated by an exploit script will always have the *reachability* and *infection* property, because they exercise the vulnerable code that is patched. However, these inputs will not be *safe* as the exploit has a malicious effect. Moreover, it is not known which input(s) will *propagate* and allow us to discern the patched from the unpatched version. Automatically finding discriminators, however, is not the main focus of this thesis.

2.1.1 Infection

Infection can only happen when the input *reaches* a patched instruction. We speak of infection when execution of such instruction causes either the memory state or the control flow to diverge between the patched and unpatched servers. Control flow divergence will subsequently cause divergence in memory state. We use the terms *diverged state* and *infected state* interchangeably to describe this phenomenon.

The infected state will then propagate either affecting memory writes or conditional jumps. Infections are removed when functions return (infected local variables), heap memory is freed or when memory is overwritten by constant values or non-infected state.

2.1.2 Safety

For a discriminator to be usable in real-world scenarios, it must be safe to use on both the patched and unpatched version of the server. Only well-tested safe inputs can be used for vulnerability scanning. What it means to be safe depends on the nature of the vulnerability and the server that is attacked and therefore needs to be decided by an analyst.

We define two broad types of safety to be used for two types of vulnerabilities:

Integrity The program state or control flow was not corrupted by the input. By corruption, we refer to state changes that should not be permitted and may cause the program to malfunction or act on behalf of an attacker. This type of safety is used to analyze all vulnerabilities based on memory errors and insufficient input sanitization.

Confidentiality No internal information is leaked to the attacker. Used for all information disclosure vulnerabilities.

This distinction is important as both require different methods of ensuring safety. Note that *Availability* is missing in the list, which would provide safety against *denial-of-service (DoS)* attacks. We consider it to fall under integrity as an attacker typically needs to corrupts program state with the goal of making the program hang, crash or otherwise become unavailable.

Besides the type of vulnerability, safety also depends on characteristics of the server. For example: Crashing a process is normally considered unsafe. However, some servers such as nginx[8] and apache[9] will automatically fork off new worker processes once this happens, undoing the damage that was done. One can argue that process crashes are safe in this scenario. Alternatively, a buffer-overflow vulnerability might only be detectable remotely if we attempt an overflow, forcing us to overflow at least a single byte. Depending on the memory layout, this could be harmless or very damaging.

In case of overflows, multiple levels of safety may exist because safety depends on the size of the overflow and the data that is corrupted by it. Because modern compilers add padding between datastructures, small overflows may not even corrupt the neighboring datastructure. When the overflow is larger, it could overwrite variables. The effect of this depends on how the variables are used and could be completely safe. However, automatically proving that such overflows are safe is a difficult task. Tools that do bounds checking such as [10] and [11] are often conservative and do not allow any overflow at all.

We consider safety only in the context of the vulnerability being analyzed. *Safety* only guarantees that the vulnerability of interest is not (fully) exploited. We assume that no exploitation or corruption is possible in the patched version, so we are only concerned with the impact of an input on unpatched servers. Even when the vulnerable code was successfully executed, we must ensure that there are no side-effects that could cause problems later. We do not consider the case that an input triggers other unknown bugs or vulnerabilities.

2.1.3 Propagation

We want our inputs to *propagate* infected state, allowing us to discriminate the patched from the unpatched version. Besides local I/O and network traffic, other channels may exist through which the running version can be determined, such as timing side-channels. In this work, we only consider *network-based* propagation which allows discrimination between versions over the network.

As opposed to the *reachability* and *infection* properties, it is not known whether propagating inputs exist for a vulnerability. Given a patch, inputs that both have the reachability and infection properties always exist. We know this because security patches *must* change the behaviour of a program in certain scenarios, meaning that it must modify executable statements that cause changes in state or control flow. However, after infecting the state, it may very well happen that these changes are never exposed. In our work, we focus on determining whether a given input propagates, rather than inventing a method to establish whether a propagating input exists, for an arbitrary vulnerability.

In this work, we only consider propagation of infected state towards the *write()* system call. Obviously, it is easy to consider other system calls as well such as *sendmsg()*. Patches do not normally introduce new *write()*'s. Instead, the infected state should cause an existing *write()* call to write different data in both versions or should cause the *write()* call to be executed for only one of the two versions

This leads to two forms of propagation:

1. *Propagation through difference*. Both versions of the server respond with data, but some bytes are different or one response is larger than the other.
2. *Propagation through absence*. One of the versions responds while the other closes the connection.

In the first case, infected state is written back to the client, while in the second case infected state determines whether a response is sent at all. In *propagation through absence*, it is often the case that the patch detects a malicious request and drops the connection after it fails the check. It is important to note that other system calls could also have effects that are observable by an attacker. In other cases, no writing of data is needed at all (e.g when using a timing side-channel).

Propagation through difference is ideal, because we reliably receive the information we need. In case of *propagation through absence*, we should also verify that the server is

<pre>1 typedef struct { 2 short type; 3 short size; 4 char buf[32]; 5 } echo_t; 6 7 void handle_request(int clientfd) { 8 echo_t e; 9 10 11 read(clientfd, &e, sizeof(echo_t)); 12 write(clientfd, &e, e.size + 4); 13 } 14 15 16</pre>	<pre>typedef struct { short type; short size; char buf[32]; } echo_t; void handle_request(int clientfd) { echo_t e; ssize_t real_size; read(clientfd, &e, sizeof(echo_t)); if (e.size > 0 && e.size <= 32) { write(clientfd, &e, e.size + 2*sizeof(short)); } }</pre>
Original echo server	Patched echo server

FIGURE 2.2: Echo server 1 contains a buffer overread vulnerability, allowing attackers to leak memory contents.

still up and that the packet was not dropped. This can be done by repeating the input and also send another request for which a response is expected, regardless of version.

2.1.4 Examples

Figure 2.2 shows a trivial example of a vulnerable and patched server program. After accepting a new connection, a client provides a simple struct, of which *size* bytes are echo'ed back. Obviously, the value of *size* can be larger than 32 bytes, resulting in extra memory being written back to the client. This memory may contain memory addresses and other information useful to an attacker. If this example was embedded in a larger server application, it could lead to disclosure of sensitive information such as cryptographic material.

The server was later patched by ensuring that the *size* does not exceed 32 bytes. Moreover, using a negative or zero *size* is now also prevented.

We can remotely test whether the server is vulnerable by providing a large *size* value, because patched servers will make themselves known by closing the connection without responding. This test would essentially exploit the vulnerability by leaking information, which is not *safe*. Instead, we could also provide zero or a very small negative value, which would not cause information to be leaked. However, it would still lead to a *write()* in the vulnerable server only. This is an example of *propagation by absence*.

<pre> 1 typedef struct { 2 int size; 3 char buf[99]; 4 } echo_t; 5 6 void handle_request(int clientfd) { 7 echo_t *e = malloc(sizeof(echo_t)); 8 read(clientfd, &e->size, 9 sizeof(int)); 10 read(clientfd, e->buf, e->size); 11 write(clientfd, e->buf, e->size); 12 free(e); 13 } 14 15 16 </pre>	<pre> typedef struct { int size; char buf[99]; } echo_t; void handle_request(int clientfd) { echo_t *e = malloc(sizeof(echo_t)); read(clientfd, &e->size, sizeof(int)); if (e->size > 0 && e->size <= 99) { read(clientfd, echo->buf, e->size); write(clientfd, echo->buf, e->size); } free(echo); } </pre>
--	---

Original echo server 2

Patched echo server 2

FIGURE 2.3: Echo server 2 contains a buffer overflow vulnerability. It can be safely detected using a small overflow that only overwrites padding bytes

In figure 2.3, we demonstrate a second echo server, which has a simple buffer overflow vulnerability. By providing a *size* that is larger than 99, we can overflow the buffer on the heap.

This time, we cannot use a size of zero to discern the two versions. Passing zero to *read()* or *write()* will have no effect and nothing will be written to the network. Thus, we are forced to overflow the buffer and see how the server responds. Fortunately, compilers align structures in memory to at least 4-byte boundaries. This means that *echo_t* has a size of 104 bytes. The very last byte is padding and we can safely overwrite it. Thus, a size of 100 followed by 100 bytes of data constitute to an input that propagates but is also safe.

2.2 Discriminator testing

Finding out whether a given input is a discriminator could be done by running a patched and an unpatched server and see if a particular input leads to different observable outcomes. However, due to non-determinism, timing and other factors, it can be difficult to compare how the servers respond and isolate observable effects that are caused by diverged state. This is especially true for bigger more complex servers. Also, unsafe inputs could still corrupt the state of a server, which would lead to problems later on, for example, by corrupting a variable that is used much later. In order to ensure that no harm is done, we would need to compare the program state of the two executions to see if nothing bad has happened. When both versions converge to the same program state, we know that the effect of the input was temporary. All infected state is gone and integrity is preserved.

However, comparing the state of two servers is quite difficult because two separate executions result in many spurious differences. First of all, the binaries are subtly different because one of them is patched, resulting in different return addresses and code pointers. Also, differences and interactions with the environment can cause significant changes in the memory layout and memory contents of the application as well as its libraries. Finally, any non-determinism such as thread or process interleaving, timing and crypto functions quickly introduce differences between otherwise equivalent executions. These spurious differences make it hard to find the actual infected state.

2.2.1 Delta execution

To mitigate these problems, we have implemented *delta execution* [12], which allows us to only run a *patch-site* of the program (*delta code*) in parallel. By default, the common code is running as a single execution (*merged execution*). When we reach delta code, we split off into two different executions. We are now running a *split execution*. These executions will continue to run and state differences introduced by delta code can be easily extracted. At specific moments, we attempt to *merge*, which means that we compare the state of the two executions. When there are no differences, we drop one of the executions and let the other continue as a *merged execution*.

Delta execution has a number of advantages. First of all, the application will run as a single merged execution most of the time. Therefore, all state differences found after splitting must result from execution of a patch-site or code executed directly after it. Because the delta-executed server is running as a single binary in a single environment, spurious differences will be eliminated, making it easier to manually inspect and reason about the remaining differences.

Secondly, we merge when all state differences are gone, which is a strong indication that the input is safe. Our assumption is that the patched execution can not be corrupted. When the non-patched execution converges with this clean patched execution, we can claim that the input is safe. When the merge attempt fails, we shall continue and retry merging a number of times, hoping that state differences will dissolve. Because some popular protocols are stateless such as HTTP and DNS, we hypothesize that servers will clean up their state after a request is handled. Even when a server keeps connection state, we hope to find inputs that have a minimal and benign impact on such state.

For example, a server may track the number of bytes written to the network, update pointers into I/O buffers, update timeout information or simply fail to reset any data that will be overwritten anyway when a new request arrives. For these cases, we support

whitelisting, which allows us to skip comparing particular data during a merge. This allows for a successful merge despite a few benign modifications to program state.

Finally, delta execution helps us to detect propagating inputs. To make delta execution work at all, we need to instrument I/O calls that are done during split state, to ensure that both executions perceive the same version of the environment and to hide the fact that two executions are running. We leverage this mechanism to detect differences between the executions that are observable by an attacker. More specifically, we intercept *write()* system calls to detect both *propagation-through-difference* and *propagation-through-absence*, discussed earlier in section 2.1.3.

One notable difference between our delta execution framework and the original one is that we do handle *delta data* differently. *Delta data* is what we call *diverged state* or *infected state*.

In the original paper[12], a merge can happen as soon as both executions are running the same (non-patched) code again. A merge results in one execution copying its delta data to the other and exiting. The other execution will now continue being the *merged execution*. All pages containing delta-data will be `mprotect`'ed, taking away all permissions, allowing the framework to intercept accesses to delta-data. When delta-data is being accessed, the execution will split such that each execution can operate on its own version of the delta-data.

In our implementation, we do not merge as long as delta-data exists. We keep trying to merge a number of times, after which we decide that the executions have diverged permanently. We did this because our goals are substantially different: we do not care about performance and resource consumption as is the case in the original paper. We assume a small amount of delta-data because the security patches are typically quite small[12]. Furthermore, we assume that when a request is handled, data-structures will be re-initialized, causing the executions to converge.

2.2.2 Confidentiality

Although checking for state differences will give us strong safety guarantees, we also need some protection against information leakage. However, it is difficult to distinguish sensitive from non-sensitive data without exhaustive annotation of the program and non-trivial instrumentation. Therefore, we assume that information leaks violate memory boundaries or read from uninitialized memory. This should work well when the size argument to a copy operation is attacker-controlled, because everything between the source pointer and the sensitive data must be read.

The *address sanitizer (ASan)*[10] can be used to catch out-of-bounds memory accesses to the stack, heap and globals, while also detecting a number of other memory errors. It instruments all loads and stores and uses shadow memory and redzones to check if a memory access is safe. `Malloc` and `free` are replaced in order to provide a special heap implementation which adds red-zones around allocations and detects use-after-free errors.

The *memory sanitizer (MSan)*[13] is used to detect *uninitialized memory reads (UMR)*. It is implemented using bit-precise shadow memory which tracks whether a bit is properly defined. The memory sanitizer propagates shadow bits when data is copied or when a safe operation is performed. As soon as uninitialized memory is used in a conditional branch, system call or pointer dereference, an error is reported.

We have designed our delta framework for use with the Memory Sanitizer or Address Sanitizer. The sanitizers themselves were not designed to be used simultaneously, so we will need to compile and run our server for both separately.

2.3 Discriminator discovery

Discriminator discovery is concerned with automating the process of finding discriminators based on some existing knowledge or inputs. Since we will develop a discriminator testing tool, we will need candidate inputs which we can test. In this section, we will discuss some techniques that could help discover discriminators and we motivate our choice for a fuzzer based approach.

We assume that an exploit for the patched vulnerability is available. The reason is that we need to know how to execute the patch-site, which may be in difficult-to-reach portion of the server. The task of automatically finding an input that exercises the vulnerability is a difficult one, that has been studied in depth [14][15][16] and is outside the scope of our research.

As said before, we cannot use the exploit directly because it has malicious effects. Instead, we use it just to be able to generate new *reaching* and *infecting* inputs. Therefore, the inputs should only be slightly different from the exploit. It is up to the analyst to identify portions of the input that need to be manipulated automatically, as opposed to the parts that are *boilerplate*.

2.3.1 Symbolic Execution

Symbolic execution would be a promising technique to help us generate candidate inputs. The idea behind symbolic execution is to let an interpreter explore many executions at the same time by using symbolic expressions rather than concrete values. Instead of using concrete input values, we express each value as a symbolic expression which is a function of the program inputs and constants. These symbolic expressions grow more complex as more program statements are executed.

At branches, the interpreter *forks off* two interpreters, each exploring a different branch. Each of these adds the true condition (or the false condition respectively) to its *path constraints*. The *path constraints* of an interpreter describe which constraints must be satisfied to follow the exact same code path. Each time an interpreter follows a new branch, it checks whether its path constraint is satisfiable and only continues when this is the case.

As more branches are executed, the number of interpreters increases exponentially, leading to the scalability problem known as the *path explosion problem*. However, because security patches are small, we can focus our efforts on a small piece of the code. Thus, depending on how we apply symbolic execution, we may not run into such scalability problems.

If the symbolic execution engine is made aware of both versions of the server, it could be used to generate an *infection constraint*, by comparing the *diverged symbolic expressions* caused by the patched statements. We could also see how such *infected expressions* propagate towards *write()* calls or control whether these are executed.

Even though this may be possible, it requires us to modify the symbolic execution engine to be aware of both the patched and unpatched version of the server and be able to mark expressions as *infected*. For example, when executing the first patched statement, the interpreter could fork and continue executing the two different statements, generating *infected symbolic expressions*. The executable must be represented in such a way that the symbolic execution engine knows which instructions are part of a patch-site so it can act accordingly when it executes them. It is not yet clear how this type of symbolic execution would work, but will probably require substantial engineering effort.

2.3.2 Concolic Execution

Concolic execution [17] is a popular derivative of symbolic execution. In concolic execution the binary is executed normally, but instrumentation accumulates all the conditional

branches that are encountered and builds a symbolic *path constraint*. A path constraint is a set of constraints on the inputs that, when satisfied, causes the program to take the same execution path. This path constraint can be used to generate new inputs that will exercise the same path. Also, it can be modified which allows us to generate inputs for new execution paths, given that the modified constraints together are still satisfiable.

Concolic execution does not suffer from the path explosion problem, because only one path is executed at a time. However, not all paths are explored and it is the responsibility of the test driver (component that repeatedly runs instrumented program) to generate new satisfiable path constraints that can be used to execute new paths.

Concolic execution would allow us to generate *reachability* constraints. Using these constraints, we can easily generate inputs reaching the patch. If we could make concolic execution aware of both versions of a server, we could let it continue and possibly let it generate *infection* or *propagation constraints* as well.

However, concolic execution is an advanced technique and we did not have the time to run experiments with it.

2.3.3 Fuzzing

Fuzzing is the act of generating inputs hoping that they will satisfy particular constraints and thus lead to some desired behaviour. Fuzzers are often used to automatically discover unknown bugs. They do this by either randomly generating inputs (possibly using a specification) or mutating known inputs for a program. Also, fuzzers may instrument the program to observe the effects of particular inputs and generate or mutate inputs based on these observations.

Because we are dealing with many complex constraints, simple random fuzzing will not be adequate. Our search is highly focused on a particular vulnerability, so it makes sense to mutate inputs generated by a known exploit because these will exercise the vulnerable code.

Evolutionary fuzzers generate a population of inputs and measure the fitness of each input through run-time instrumentation. Based on the results, a new population is generated. This feedback loop makes evolutionary fuzzers very efficient.

In our experiments, we employ VUzzer^[15], which is an evolutionary fuzzer which aims to maximize basic-block coverage. Each time it discovers a basic block, the used input will be written to a file. We run the VUzzer on the original and patched server separately

resulting in two sets of input files, each of which will be fed to our discriminator testing framework, in the hope that discriminators are found.

VUzzer uses PIN[18] which is a dynamic program analysis framework, to record which basic blocks are executed and selects good inputs for the next generation. It also uses *libdft*[19] to see how individual bits in the single input buffer affect different *cmp* instructions. It will mutate these bits in an attempt to execute both sides of a conditional branch (the if-block and the else block).

There are a few reasons why we chose the VUzzer. First of all, we can easily seed the fuzzer using a few inputs generated by an exploit script. This is a rudimentary way of teaching the fuzzer how to *reach* a patch-site and *infect* program state. Since security patches almost always modify or introduce conditional branches, it is likely that the fuzzer finds additional infecting inputs.

Secondly, we can make the fuzzer focus on basic blocks inside the patched function. We do this by providing the start and end address of the patched function, which we manually extract using a tool such as *objdump*. The VUzzer will then focus mutations on bits in the input string that correspond to *cmp* instructions who lie in this address range. As such, we can focus on *interesting* basic blocks, rather than simply maximizing coverage.

Finally, because the main focus of this work is implementing and testing the discriminator testing framework, we preferred a simple method that generates candidate inputs. Fuzzing is definitely one of the simplest methods to test first.

We realize that the VUzzer has no way of generating inputs that are more likely to have the desired propagation and safety properties. As such, methods based on symbolic, concolic execution or constraint solving may be more efficient at satisfying all properties. However, implementing such a system would be too much work for this thesis.

2.3.4 Hybrid approaches

An important observation is that symbolic execution and fuzzing are opposites. Fuzzing is very scalable but often has problems penetrating deep parts of the program logic. Symbolic execution is very precise and offers more guarantees at the cost of scalability. When trying to solve a problem like finding discriminators, it may pay off to combine these approaches and have the best of both worlds.

Driller[14] is a good example of such a hybrid tool. Its goal is to find vulnerabilities in deeper regions of the program. It uses fuzzing to explore the shallow code paths. When

no new paths are found for some time, it resorts to concolic execution to *drill* through more difficult constraints, reaching deeper parts of the program, where it can go back to using the fuzzer with a few additional constraints. A similar hybrid approach may be useful to find discriminators since the constraint-solving effort is clearly focused on a single patched function.

Chapter 3

Implementation

In this chapter we will describe how we have implemented our discriminator testing framework. We have implemented delta execution using a number of LLVM compiler passes shown in figure 3.2 and a runtime library called libdelta. We will also explain how we integrated the *address sanitizer (ASan)* and the *memory sanitizer (MSan)* into this framework. Finally, we will explain how we used a fuzzer to generate candidates for discrimination testing.

3.1 Deltafy Compiler Pass

We have designed an LLVM compiler pass to support delta execution. It takes two almost identical LLVM bitcode files, one being the original version and the other being the patched version. It merges them into a single output file, without duplicating unchanged code or globals. Code that has changed between input files is kept and *split hooks* are inserted right before these changes, allowing the delta library to split the execution into two.

The deltaxy pass operates at function granularity. All functions with identical names in both input files are compared. We keep both versions of all modified functions and we rename these. Then, we introduce a new *proxy function*, which will call the split hook. Based on the return value of the split hook, either the original or the patched version of the function will be called. All calls to modified functions will be replaced by calls to their respective proxies to make sure that the split hook is executed before running delta code.

The transformation is described in figure 3.1. We start with two programs, where the function `foo` was modified by the patch. The deltaxy pass will copy all the unmodified

<pre> 1 int foo(int a) { 2 return a; 3 } 4 5 int bar(int a) { 6 return foo(a) + foo(a); 7 } 8 </pre> <p style="text-align: center;">Without patch</p>	<pre> 1 int foo(int a) { 2 if (split_hook()) { 3 return foo_original(a); 4 } else { 5 return foo_patched(a); 6 } 7 } 8 9 int foo_original(int a) { 10 return a; 11 } 12 13 int foo_patched(int a) { 14 return a + 1; 15 } 16 17 int bar(int a) { 18 return foo(a) + foo(a); 19 } </pre> <p style="text-align: center;">After deltafy pass</p>
<pre> 1 int foo(int a) { 2 return a + 1; 3 } 4 5 int bar(int a) { 6 return foo(a) + foo(a); 7 } </pre> <p style="text-align: center;">With patch</p>	

FIGURE 3.1: The deltafy pass merges the patched and unpatched version. It replaces modified functions with proxies, which forward the call to either the original or patched version of the function

functions to the output program. In our case this is just `bar`. It will then copy and rename both versions of `foo` and will introduce a new `foo` function, which we call the *proxy function*. The task of the proxy function is to call `split_hook`, to decide which version of `foo` will be executed. `split_hook` is implemented in *libdelta* which is discussed later.

Deltafying at function granularity is much simpler than doing it at basic block granularity since we don't need to manipulate the control-flow-graph and references to local variables. The downside of the approach is that a small patch affecting a big function, causes all code in that function to be treated as delta code. The fact that this function is executed does not imply that an input satisfies the *reachability* property, as the patch-site could have been skipped altogether. This is discussed in more depth in 7.1.2.

The deltafy pass does not support global data changes such as changing structs or introducing new global variables. This does not matter since security patches typically do not affect globals or structs [12].

Figure 3.2 shows the deltafy pass together with all other compiler passes that are used to instrument binaries for use with *libdelta*. The other passes will be explained in later sections.

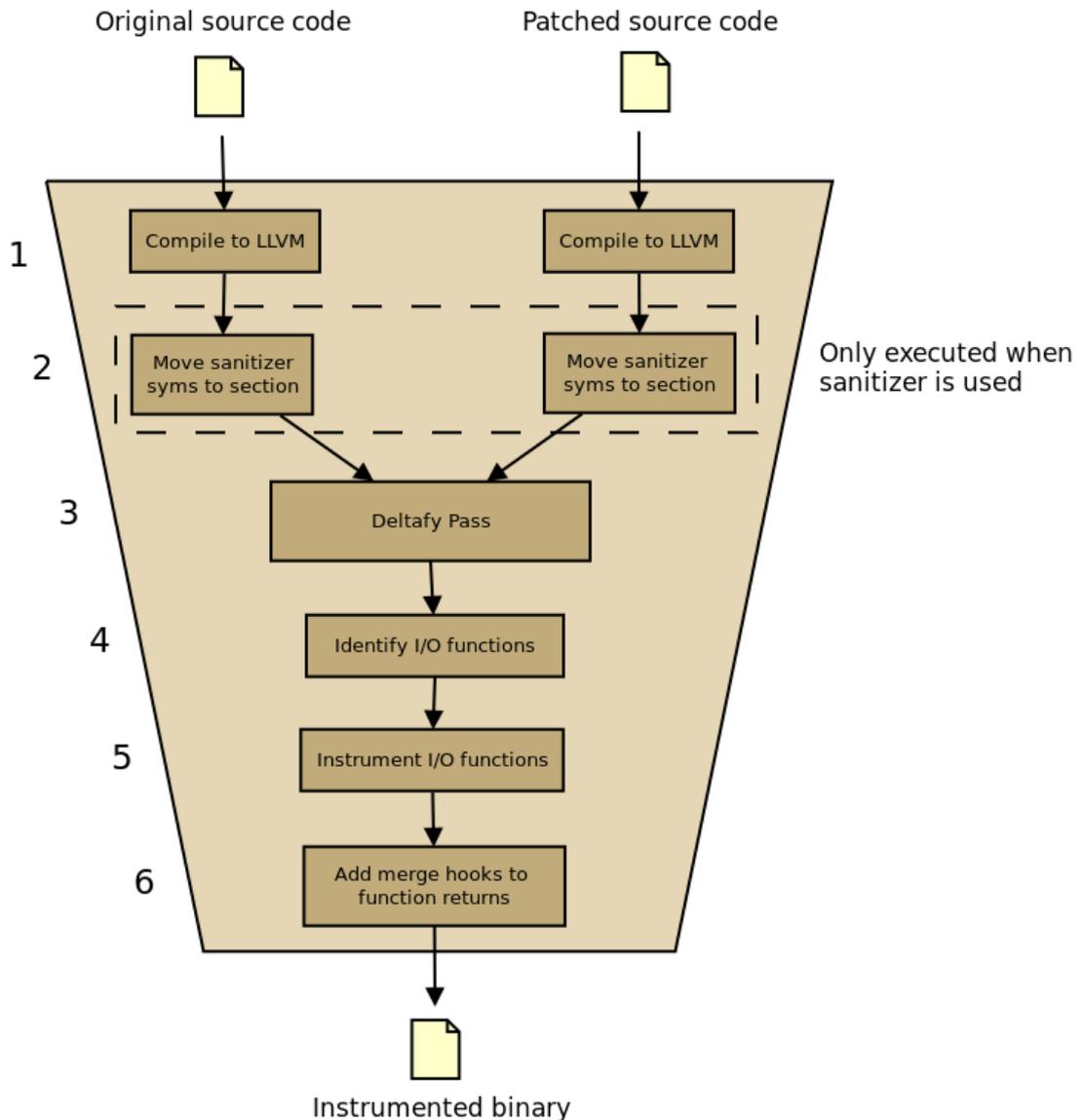


FIGURE 3.2: Shows all LLVM compiler passes used to generate instrumented binaries that work with libdelta

3.2 Libdelta runtime library

Libdelta is the beating heart of the delta execution framework. It takes care of splitting and merging, relying on hooks installed by various compiler passes into the analyzed program.

3.2.1 Splitting

How splitting is implemented is shown in figure 3.3. The *executions* we talked about so far are implemented as processes. In merged execution, a single process is running. Suppose our program contains a function F that has been patched. When we call it (step

1), the proxy function `F` will invoke the `split hook` implemented by `libdelta`. `libdelta` will call `fork()` to create a child process. Both processes will return from the `split hook` with different values causing the proxy function `F` to run both the original and the patched version, each in its own process. This child process will run the patched code, while the parent runs the original code.

When the execution splits, `libdelta` will take note of all writeable memory maps (step 4) that need to be compared when merging. Some writeable maps are not taken into account, such as the ones used by `libdelta` itself.

`libdelta` uses soft-dirty bits[20] to efficiently find out which memory the process has written to and thus needs to be compared during a merge attempt. After establishing which memory maps are relevant, `libdelta` will reset the soft-dirty bits to start recording dirty pages (step 5). Whitelist calibration (step 6) will be explained later.

Note that split-execution is *re-entrant*. If we take the code in figure 3.1, a first call to `foo` will result in a fork. Now when the original execution calls `foo` again, it will not fork, but continue to execute `foo_original`.

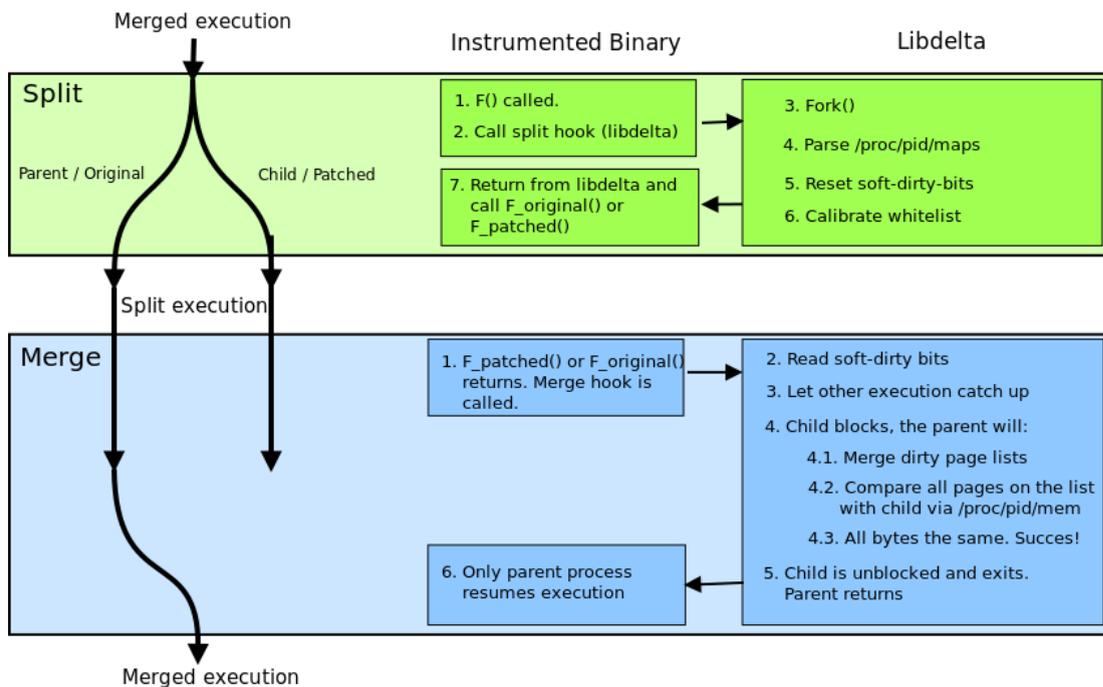


FIGURE 3.3: Shows how an instrumented program splits and merges when a patched function `F` is called

3.2.2 Merging

Once in split execution, merge attempts should be done to detect whether both processes have already converged. We inject *merge hooks* (step 6 of figure 3.2) at every function

return as is done in [12]. These do nothing during merged execution.

When a merge is attempted, the merge hook is called on a function return (step 1 of the merge in figure 3.3). We employ soft-dirty bits (since kernel version 3.11) to see which pages have been modified ever since we split (step 2). All dirty pages that are not within the previously saved memory maps will not be taken into account.

Libdelta will then wait until the other process also calls its merge hook (step 3) and we will block the child process. The parent process will then compare it's the state with the blocked child process.

When both are ready, the parent process proceeds comparing all mapped memory. For each of the dirty pages found (in either process), we do a byte-by-byte comparison (step 4.2). The parent process will access the memory of the child process through the `/proc/pid/mem` interface. Registers do not need to be compared because both processes are blocked inside libdelta and thus the registers do not contain application data.

When all compared memory matches, we go back to merged execution (step 5). This means that the child process that executed the patched code will exit. When diverged state was found between the processes, the respective pages will be dumped to files and a summary of all diverged state will be written to a file.

Libdelta will perform a limited number of merge attempts and will abort the program when all attempts have failed. We restart counting our attempts after a successful merge.

To prevent unnecessary merge attempts, we use the depth of the call stack to decide whether an attempt is warranted. Merges will only be attempted when the call stack is of equal or shorter depth than at the previous attempt (or using the stack depth at the split when there is no previous attempt). That's because it is very likely that attempts performed at deeper levels will fail because of some diverged state higher up the stack.

The stack requires some special attention, because it both contains *application stack frames* (at higher addresses) and *instrumentation stack frames* (at lower addresses). The merge hook records the address that separates these. When comparing pages byte-by-byte, we whitelist all stack bytes below this address to ensure that the libdelta stack is not compared.

Finally, the return value of a function may also have diverged, so it requires special handling. The merge hook copies the return value and sends it to libdelta where it is compared along with the dirty pages.

3.2.3 Instrumenting I/O functions

When in split execution, we cannot let both processes do I/O themselves, because we want to hide the fact that two processes are running. We want external effects to happen once, while duplicating the effect a call has on process memory. For example, when both processes call *write()*, only one should be actually performed. However, both processes should see the same return value. Similarly, when both processes *read()*, only one call is executed and the contents of the buffer will be copied over to the other process, keeping them synchronized.

We automatically instrument a large number of I/O calls, all of which happen to be system calls. We have a compiler pass that marks these calls (step 4 of figure 3.2. Another pass is used to instrument all these function calls (step 5). It places *pre-call* and *post-call* hooks around them, which allows libdelta to intercept function arguments, skip the call and provide a return value or another result e.g by writing to buffer(s). When in merged execution, libdelta will not affect I/O calls in any way.

When in split execution, libdelta will process I/O calls in pairs. When one process calls the pre-call hook, libdelta will block it and wait for the other process to also arrive at a pre-call hook. When both processes appear to call the same function, the arguments to both I/O calls are copied and compared. For some I/O calls, referenced buffers will also be copied and compared.

Libdelta will then report on any differences between the calls. Differences in *write()* buffers are of special interest to us, since these show that infected state is propagating to the attacker. It can happen that both processes do calls to different I/O functions, when they have diverged. libdelta makes no effort to pair up I/O calls, to synchronize the executions, so manual analysis is required when this happens.

When one process wants to attempt a merge and is waiting for the other process to catch up, all I/O calls in the other process will run as normal. These I/O calls are executed because the other process is executing a different code path and may allow us to remotely discriminate between the two versions.

3.2.4 Whitelisting

It is unlikely that the memory contents of both processes end up being the same, when a merge is attempted. Applications often modify global or heap datastructures, which may never return to the same state. Libraries may have their own state as well. Also failure to clean up garbage data may result in memory differences between the processes. In order to ignore certain memory areas, libdelta keeps track of a whitelist.

First of all, particular symbols can be added to the whitelisted section. This causes them to be placed into one contiguous piece of memory, which is ignored during a merge. Secondly, address ranges can be provided using a file. This is of limited use because the exact addresses to whitelist may change between runs of the program. To work around this, Libdelta also exposes an API that allows us to build and maintain a whitelist at run-time.

Finally, as shown in step 6 of *splitting* in figure 3.3, we calibrate the whitelist after calling `fork`. This is needed because the runtime environment keeps retains process specific information such as PID's in memory. These must be added to the whitelist automatically.

3.2.5 Deltastub

Libdelta is implemented as a shared library. It relies on a shared library called *libtaskctl*, which offers various services such as a custom heap in shared memory, allowing all delta processes to communicate. It also allows us to interpose `fork()`, `exit()`, `exec()` and others, allowing us to track different processes, any of which may transition between split and merged execution independently. Interposing `fork()` is done by loading both *libtaskctl* and *libdelta* using `LD_PRELOAD`.

Because it is a shared library, loaded with `LD_PRELOAD`, we cannot directly call into our library from the binary. Hence, we need to introduce the *deltastub* static library, which contains trampolines that call into *libdelta*.

3.3 Sanitizer integration

Integration of the memory sanitizer and address sanitizer is very important to test that inputs are safe. However, integrating these into the delta framework proved to be a challenge. ASan and MSan are very invasive as they both instrument the code, replace the heap and interpose many libc calls. Similarly, the delta-framework constrains what the sanitizers can do during split state, because operating on non-whitelisted memory results in spurious differences that are difficult to track down. In this section, we discuss various problems we encountered and how we worked around them.

3.3.1 Compilation

We are not aware of a way to use the sanitizer as a separate LLVM pass. Therefore, we must first compile both versions of the code with the sanitizer before we run `deltafy`.

As a result, `deltafy` will see symbols introduced by the sanitizer's compiler pass, which are generated based on the code structure. `Deltafy` may therefore not be able to match all symbols to equally named counterparts and aborts. To solve this, we moved these symbols into the whitelist section using an extra compiler pass (step 2 in figure 3.2).

3.3.2 Initialization

The sanitizers call `mmap` regularly during their initialization. These areas need to be whitelisted as well. However, a sanitizer may initialize before `libdelta` can (`ASan` uses `.preinit_array` which executes before any other constructor does) and the whitelist is not created yet. To work around this, we introduced a temporary whitelist in `deltastub`, which is later copied into the whitelist in `libdelta`. `Deltastub` is available before `libdelta` is, because it is statically linked.

3.3.3 Thread-local-storage

The memory sanitizer also uses *thread-local storage (TLS)* to pass shadow values for function parameters and the return value. However, these variables turned out to diverge. Also, failure of a system call in one execution could also cause `errno` to diverge, which is a TLS variable. Unfortunately, it is not possible to selectively whitelist individual TLS variables. This is because our whitelisting mechanism assumes that each variable can be moved into the whitelist section. This is not true for TLS variables, which reside in special thread-local data sections managed by the thread library. We decided to whitelist the entire virtual memory area that contains TLS data. We believe this is relatively safe, since the amount of data in thread local storage is very limited, given that we do not support threaded software.

3.3.4 Controlling the heap

Both `libdelta` and the sanitizers impose some requirements on how the heap should work. This is the reason why both `ASan` and `MSan` interpose `malloc` and `free`. As a result, `libdelta` can not intercept heap allocations, making it difficult to integrate them properly. These are the requirements imposed on the heap:

1. **Detect heap overflows:** In order to detect unsafe inputs, we need a heap-implementation that help us detect out-of-bounds reads and writes using guards or red-zones. Detecting these also requires compatible instrumentation of load and store operations.
2. **Control heap base-address:** Both ASan and MSan use shadow memory where mapping addresses to shadow addresses must be very fast. Controlling the base-address of the heap allows one to make mapping addresses extremely fast using offsets or bitwise operators.
3. **Whitelisted heap for instrumentation:** Instrumentation also needs memory to run. ASan and MSan default to use their own heap, thus mixing instrumentation and application allocations. This becomes problematic when state divergence causes a sanitizer to perform allocations, leading to differences in heap layout. We must ensure that allocations done by the instrumentation are done on a secondary heap that is completely whitelisted.
4. **Synchronize allocations between executions:** When both executions perform different allocations, their heap layouts will be desynchronized, causing many spurious differences. [12] fixed this by replicating each `malloc` call to both executions. This wastes memory, but ensures that the heap layout stays synchronized. Unfortunately, this solution requires us to interpose `malloc`, which is not possible. This is discussed in section 3.3.5.
5. **Control `mmap`'s:** Heaps will resort to `mmap` for larger allocations. It is important that we whitelist such `mmap`'ed memory when it is used by instrumentation.
6. **Keep allocation metadata:** Finally, when differences are reported, it is very time-consuming to see where they come from. GDB can be used to identify changes to global and stack data, but the heap poses a challenge. Being able to store additional metadata such as source line of the allocation or the datatype can help to create new whitelisting policies and reduces the manual work that is currently needed to interpret the reported diverged state.

Because the heap is controlled by a sanitizer, we could only partially meet the last three requirements using cumbersome tricks. Therefore, the delta framework is less robust than we like it to be and sometimes difficult to work with. In order to meet the 3rd requirement, we had to subvert all allocations done by MSan into libdelta's heap, which is located in shared memory that is whitelisted.

Intercepting the `mmap` calls required us to modify the sanitizers. Both MSan and ASan perform lots of `mmap` calls to allocate or protect memory (using `PROT_NONE`). Also, some

calls to `mmap` use `MAP_NORESERVE`, which allows the sanitizer to claim enormous chunks of virtual memory not backed by physical memory. We need to whitelist these regions because libdelta will hang and eventually crash as it tries to compare terabytes of memory not backed by physical RAM.

In order to make it easier to see where reported differences originate, we used the *origins tracking* feature of the memory sanitizer. Origins tracking keeps track which allocation and copy operations are involved before uninitialized memory is being used in a system call, branch or pointer dereference. These are reported as a series of stack traces and make it easy to track down the uninitialized bytes. We modified MSan such that libdelta can extract these origin traces for memory addresses that have diverged. Unfortunately, MSan only keeps these traces when uninitialized data is involved, so this is not a complete solution to the problem.

3.3.5 Interposing functions

Instrumentation tends to interpose functions to see what the application is doing and possibly provide an alternative implementation. Functions can be interposed at compile-time by replacing or instrumenting them, but also at run-time, by providing a symbol with the same name that is linked first.

ASan and MSan both provide a static library that interposes a large number of functions by providing a symbol with the same name. As such, equally-named functions defined in `LD_PRELOAD`'ed shared objects or `libc` will not be called, because functions defined in the binary are considered first.

A nasty side-effect is that sanitizers will also interpose calls performed by libdelta. Libdelta requires calls such as `memcmp` to compare arbitrary pages, which causes sanitizers to report uninitialized or out-of-bounds reads. We solved this by grabbing `libc` function pointers and calling those directly, thus bypassing the sanitizer. Unfortunately, this only works for code we compile ourselves. When libdelta calls into `libc`, `libc` may very well call an interposed function, resulting in an error. We faced *interesting challenges* as `dLError`, which was required for function pointer loading, could call `malloc` or `free`, denying libdelta access to non-interposed functions.

Since libdelta is an `LD_PRELOAD` library, it can only interpose application calls that are not already interposed by a sanitizer. Fortunately, it can still intercept `fork`, `exit` and `exec` in order to track application processes, which can individually split and merge.

3.4 Fuzzing

We are using VUzzer[15] to discover potential discriminators. However, a few modifications were required to make it work for our use-case.

First of all, We made it target basic blocks of the patched function, by providing it's start/end address. The fuzzer will focus it's mutations on bits in the input that are used in conditional branches within this address range.

Secondly, the VUzzer was designed to provide input files that would be sent to the program over `STDIN`. The program is expected to read all data at once, allowing the VUzzer to monitor, using taint tracking, how bytes at different offsets in the read-buffer propagate to different `cmp` instructions. Since servers do not receive data via `STDIN` and often require interaction, this design did not fit our use-case.

Recall that we defined an input as a concatenation of all data sent to the server, which will cause the patched code to be executed. However, this input may consist of separate messages that must be delivered in succession. Many of these messages may be *boilerplate*, which means that they have little or no influence over how patched code is executed. Also, there may be interaction where the contents of the messages depend on preceding messages. It is therefore unpractical to *fuzz* the entire input string.

Hence, we decided to add a *session script* to the design which transforms *fuzzed inputs* from the server into a series of messages that exercise the patched code on the server side. This session script opens a network connection to the server via the loopback interface and performs any required setup such as handshakes, authentication or other interactions. The session script can be derived from the exploit script, by replacing the hardcoded payload by a payload provided by the fuzzer.

When the setup is done, the session script will inform the server's runtime instrumentation that the payload is coming and that taint tracking must be enabled. It will then send the fuzzed input to the server and will wait for a response. The instrumented server will call a single `read` and taint (with buffer offsets) will start to propagate. It will send a response, or hang, in which case the session script times out. When the session script exits, the VUzzer will kill the server and restart it for the next input to test.

The design is shown in figure 3.4. The VUzzer uses the seed inputs to create the first generation of inputs. For each input, the server is started (step 1) and the session script is launched (step 2), providing it with a fuzzed input to test. The setup script will send and receive messages as is required to prepare the server for the payload (step 3) and after enabling taint tracking, will deliver the payload (step 4). The setup script terminates after receiving a response or after a timeout occurred. After all, we must

ensure that the server has enough time to run the patched code and provide feedback (step 5) in the form of executed basic blocks and executed *cmp* instructions and offsets used by them.

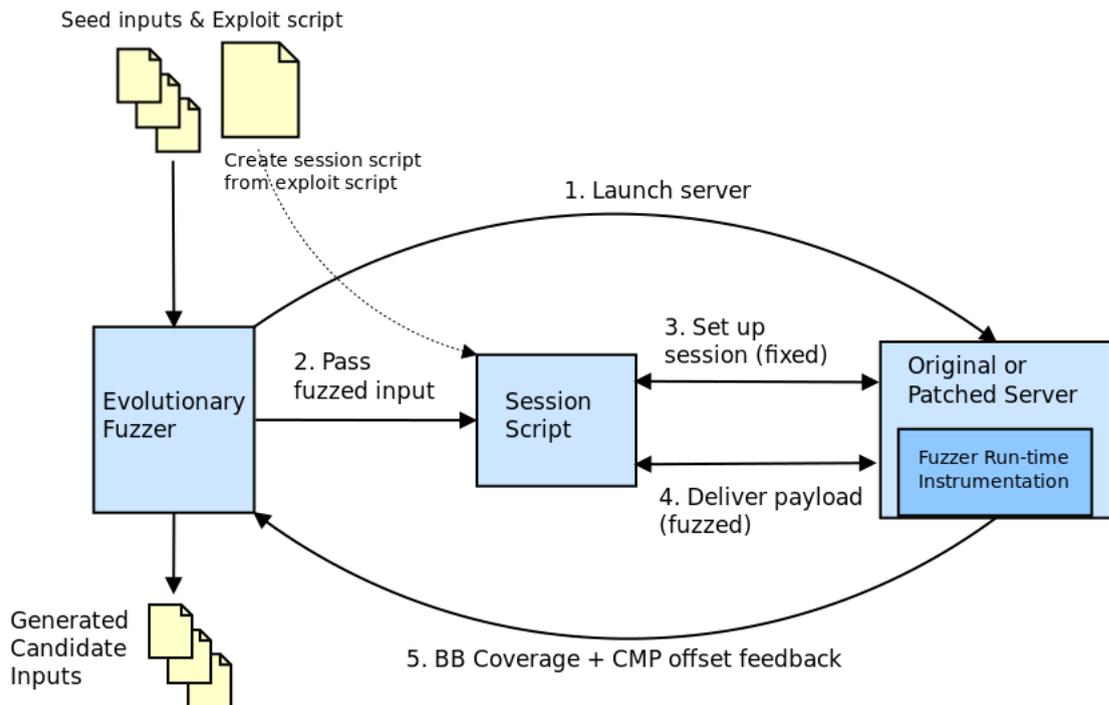


FIGURE 3.4: The VUzzer uses a session script to fuzz the instrumented server

One major problem with this design is performance. The server must be restarted to ensure inputs are tested in isolation. This is because fuzzed inputs can bring the server into an erroneous state, resulting in failures for succeeding inputs. Also, simply not providing a large enough payload can keep a server hanging. Fuzzers can typically test hundreds of inputs per second, making them much more effective.

Unfortunately, we did not end up fuzzing real-world exploits, so we could not benefit from the flexibility offered by the session script. The vulnerable toy programs we experimented with could all be tested by sending a single buffer.

Chapter 4

Vulnerability Class Analysis

In this section, we will give a qualitative analysis of the effectiveness of patch fingerprinting for different classes of vulnerabilities. We do this by relating vulnerability- and patch characteristics to the properties in the *RIPS* model to see if and how they might be satisfied.

Given the properties defined by the *RIPS model*, we can subdivide the input space of a server in context of a patched vulnerability. Let R , I , P and S be the sets of inputs that have the *reach*, *infect*, *propagate* and *safety* properties respectively.

We know that $P \subset I \subset R$, because an input can only *infect* state after *reaching* a patch-site and can only *propagate* infected state after it has *infected* the state.

In this analysis, we define safety solely within the scope of the vulnerability being analyzed. As a result, all inputs that do not exercise the vulnerable code are considered safe, regardless of any other bugs that may exist. Also, we assume the patch completely remedies the problem, such that no malicious effects are possible in the patched version. Therefore, all inputs $s \notin I$ are safe because no harm can be done when no state is infected.

Figure 4.1 shows the sets as defined here in a venn diagram. We will now analyse some general vulnerability classes to see what is needed to find a discriminator for them.

4.1 Null-pointer dereferencing

The simplest class of vulnerabilities are *null-pointer dereferences*. Programmers can easily miss the fact that in some condition, a pointer can be null, resulting in at least a

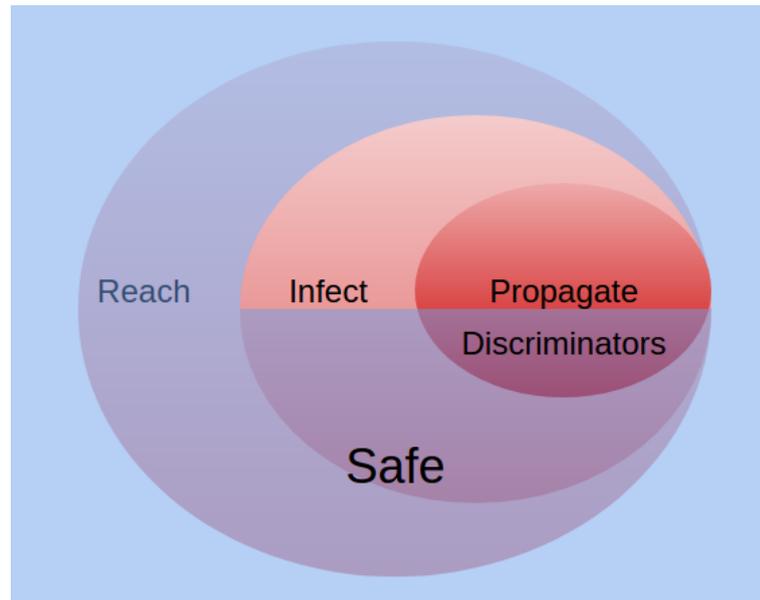


FIGURE 4.1: Shows a venn-diagram of the RIPS model. The blue-ish parts are all safe (S) but only with respect to the vulnerability being analyzed. This picture may be different per vulnerability as some sets may be empty or may not overlap with others.

denial-of-service (DoS) vulnerability. A patch will typically add a check which prevents execution of code that dereferences the pointer. When this is the case, all infected inputs are malicious ($I \cap S = \emptyset$), because all infecting inputs will cause the pointer to be dereferenced. Therefore, safe discriminators will not exist, making it impossible to fingerprint these patches. The same goes for *reachable assertions* which can also be used for DoS attacks.

4.2 Integer overflows

Integer overflows or underflows are a result of arithmetic on extreme values which were not anticipated by the programmer. Depending on how the operands or the result is used, it could open security vulnerabilities. This type of vulnerability puts constraints on the attacker controlled values, forcing them to be very different from what the program expects. It is very likely that such extreme values can cause the program to crash before an attacker can actually do something interesting. For example, extreme values could lead to extremely large memory allocations, segmentation faults or failing library or system calls. In other words, values causing the integer overflow are often unsafe and need to be carefully crafted to pass all safety constraints.

Although not found in a server, one interesting example is CVE-2017-7308 which was recently found in the linux packet socket interface. An integer underflow followed by a

```

4207 if (po->tp_version >= TPACKET_V3 &&
4208     (int)(req->tp_block_size -
4209         BLK_PLUS_PRIV(req_u->req3.tp_sizeof_priv)) <= 0)
4210     goto out;

```

Vulnerable code of CVE-2017-7308

```

4207 if (po->tp_version >= TPACKET_V3 &&
4208     req->tp_block_size <=
4209     BLK_PLUS_PRIV((u64)req_u->req3.tp_sizeof_priv))
4210     goto out;

```

Patched code

FIGURE 4.2: The original and patched code of CVE-2017-7308

typecast allows an attacker to bypass a check, resulting in an exploitable heap-out-of-bounds write. The vulnerable code is shown in figure 4.2.

We control `req->tp_block_size` (int) and `req_u->req3.tp_sizeof_priv` (unsigned int). `BLK_PLUS_PRIV` simply adds 48 to it's input. By setting the high bit of `req_u->req3.tp_sizeof_priv`, we can underflow the result of the subtraction, such that the it becomes a high positive number (e.g `0x7ffffffd0`) causing us to pass the check. This bug leads to a kernel heap-out-of-bounds write where the size and offset are under the attackers control.

The values of `req->tp_block_size` and `req_u->req3.tp_sizeof_priv` propagate to other datastructures and are used in various places. Additional checks are performed on them, further constraining the possible values they can have.

The patch of CVE-2017-7308 rewrites the vulnerable expression to eliminate the subtraction that caused the underflow. It also casts `req_u->req3.tp_sizeof_priv` to 64 bits to ensure that the `BLK_PLUS_PRIV` macro can safely add 40 without overflowing.

Integer overflows can often fixed by properly validating input values before they are used, or by fixing sanitization code itself as is the case for CVE-2017-7308. When an attacker can learn that this check failed for her input, because of error-handling by the server, all infecting inputs will also be propagating inputs ($I = P$). However, automatically finding safe inputs within this set remains difficult.

4.3 Use-after-free

Use-after-free vulnerabilities allow an attacker to control memory reads or writes by reallocating the memory pointed to by a dangling pointer. When this pointer is later dereferenced, it will read data provided by the attacker, allowing him to manipulate the

execution, possibly resulting in execution of arbitrary code. Exploitation of a use-after-free vulnerability requires three things:

- A pointer to an object that has been deallocated, also known as a *dangling pointer*.
- An *attacker-controlled object* that reuses the memory pointed to by the dangling pointer.
- A *use-after-free instruction*, which causes the dangling pointer to be dereferenced, thus using attacker controlled object.

Dereferencing the pointer can have various consequences. An attacker can cause the program to read un-sanitized data, write to a chosen memory location or execute arbitrary code when case a function pointer is called.

Patching a use-after-free vulnerability is done by either preventing dangling pointers to appear or by refactoring the program to ensure that the *use* always happens before the *free*. Either way, these patches typically do not change how the program behaves and rather prevent specific exploitable scenarios. This results in few (if any) infected values. Also, since use-after-free bugs are often patched in the *teardown* part of the code, it does not make sense to let any values propagate to the attacker, even for locally running software. We believe that, in general, use-after-free vulnerabilities will be impossible to fingerprint.

4.4 Buffer overflows

Buffer overflows are caused by the absence of a bounds check on the destination before copying data, resulting in adjacent bytes being overwritten. Patches can either add a check that causes malicious inputs to be rejected. Sometimes, the input can be truncated such that it fits into the destination.

Finding completely safe inputs is often not possible, because the patch may only infect inputs that cause at least a one-byte overflow. Whether such one-byte overflows are safe depends on the memory layout and probably requires some manual inspection. In other words, we require the program to work correctly even though a small overflow has occurred. Fortunately, compilers always align datastructures as well as elements inside them, so there might be room for a small overflow. Also non-critical or unused variables may be overwritten, but manual inspection is required.

As for propagation, the input check will usually cause the vulnerable code to abort, and this event may very well propagate to the attacker.

4.5 Buffer over-reads

Buffer over-reads bugs are a powerful class of information disclosure vulnerabilities. Buffer over-reads arise when no bounds-checking is performed on the source before copying data, leading to disclosure of additional data to the attacker. An over-read can be easily prevented by checking the length parameter before copying the data.

Buffer over-read vulnerabilities have attractive properties making them relatively easy to fingerprint. First of all, we know that they expose information to an attacker, so the propagation property is given. Also, leaking a single byte a single time is harmless, but allows an attacker to learn that the host is vulnerable. Therefore, these vulnerabilities are the easiest to fingerprint.

4.6 Conclusions

Vulnerabilities in C source code manifest themselves in a wide variety of ways. Despite our attempts, it remains difficult to draw conclusions on how patch fingerprinting will work on various classes of vulnerabilities. However, there are a few general observations:

Patches that perform some kind of validation or sanitization are often good candidates for fingerprinting. Such patches meet two conditions: They enforce a constraint on the input that was initially missed by the developer and jump to error handling code when an illegal input was provided. The larger the set of handled illegal inputs, the more likely it is that a harmless discriminator exists among them. Buffer overflows where a length field is attacker-controlled, force a developer to handle the error-case. It is this error handling that enables propagation and allows us to fingerprint the patch.

On the other hand, for many low-level errors, patching involves preventing execution of vulnerable code without the need for *error handling*. For such vulnerabilities it is possible to craft *seamless* patches, which do not change application behaviour in any way. Null-pointer dereferences and use-after-free vulnerabilities can often be patched in such fashion, making fingerprinting difficult if not impossible.

Chapter 5

Results

We have done a number of experiments with our system. Heartbleed was an important driver behind this research so we will discuss our findings in depth.

Unfortunately, we could not test our system on other real-world vulnerabilities because we had trouble finding server vulnerabilities in open source software for which both a patch and a working exploit is available.

Even if we had these, they would still not provide a ground-truth for our experiments because we do not know the discriminators for these patches. Therefore, we cannot effectively measure the performance of our approach. So, instead of analyzing real-world vulnerabilities, we resort to a few hand crafted vulnerable programs to see if our method can find discriminators for these.

5.1 Heartbleed

Heartbleed (CVE-2014-0160) is a critical information disclosure vulnerability, which enables hackers to leak memory of an OpenSSL server, which may lead to disclosure of cryptographic keys. The vulnerability was found in OpenSSL's implementation of the heartbeat extension (RFC6520) which is part of the TLS/DTLS protocol.

Heartbeats are necessary to keep existing TLS/DTLS sessions alive. Also, because DTLS is implemented on top of an unreliable transport protocol (UDP), discovering the *Maximum Transmission Unit (MTU)* becomes the responsibility of OpenSSL. By allowing heartbeats to have a variable size payload, we can discover the *Path MTU* or *PMTU* for short.

Apart from 16 bytes of random padding, the payload of a heartbeat is echo'ed back. Unfortunately, the 2-byte length field of the heartbeat request is not validated and can be used to retrieve a large heartbeat response containing 64kb worth of arbitrary memory contents, possibly containing sensitive keying material. The patch for heartbleed is shown in figure 5.1.

Heartbleed serves as an example to motivate this research, because it is possible to detect vulnerable OpenSSL servers in a safe way. As such, it provides us with a ground-truth to test our method. It works as follows: The first if-statement will cause `tls1_process_heartbeat` to return when the required 16 bytes of padding are not present, which results in closing the connection. The vulnerable version would send a heartbeat response in that scenario. Thus, sending heartbeats without this padding provides us with a safe avenue to scan for vulnerable OpenSSL servers.

```

diff --git a/ssl/t1_lib.c b/ssl/t1_lib.c
index a2e2475..bcb99b8 100644
--- a/ssl/t1_lib.c
+++ b/ssl/t1_lib.c
@@ -3969,16 +3969,20 @@ tls1_process_heartbeat(SSL *s)
     unsigned int payload;
     unsigned int padding = 16; /* Use minimum padding */

-     /* Read type and payload length first */
-     hbtype = *p++;
-     n2s(p, payload);
-     pl = p;
-
     if (s->msg_callback)
         s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

+     /* Read type and payload length first */
+     if (1 + 2 + 16 > s->s3->rrec.length)
+         return 0; /* silently discard */
+     hbtype = *p++;
+     n2s(p, payload);
+     if (1 + 2 + payload + 16 > s->s3->rrec.length)
+         return 0; /* silently discard per RFC 6520 sec. 4 */
+     pl = p;
+
     if (hbtype == TLS1_HB_REQUEST)
     {
         unsigned char *buffer, *bp;

```

FIGURE 5.1: The patch for heartbleed (CVE-2014-0160)

5.1.1 Propagation in heartbleed

Finding a discriminator proved to be more difficult than initially thought. The reason lies in the fact that heartbeats can be sent at any moment, even before the TLS handshake is completed. Heartbeats sent during the handshake are unencrypted and are handled differently from the encrypted ones sent after the handshake is completed.

For unencrypted heartbeats, the heartbeat response is buffered in `write_buffer()` in `crypto/bio/buff.c`, which inhibits propagation. The only way to get a response is to ensure that the heartbeat response has a payload of at least 4073 bytes. Furthermore, since the server did not expect a heartbeat at this point, it aborts the handshake. There is no way we can flush the buffer to get the heartbeat response.

Exploit scripts can simply send these unencrypted heartbeats because they do not care that the TLS handshake is aborted, as long as they can fetch the heartbeat response that contains the desired cryptographic material. Creating the exploit using encrypted heartbeats is substantially more difficult because it requires you to duplicate parts of OpenSSL in order to complete the handshake and craft a malicious encrypted heartbeat.

ZGrab, which is part of ZMap[21] contains a heartbleed scanner that uses encrypted heartbeats, which we modified to carry out our experiments. Encrypted heartbeats

were not buffered like the unencrypted ones, allowing us to detect vulnerable OpenSSL servers without leaking a single byte of information.

Although this is a single example, heartbleed proves to us that *vulnerability exploitation* can be substantially different from *vulnerability detection*.

5.1.2 Safety in heartbleed

In case of heartbleed, safety is defined in terms of information leakage. Leaking 64kb of memory would be considered full exploitation and leaking no information is completely safe. Modifications to program state are not of interest because they do not serve a purpose in the exploit.

In order to catch information leakage, the delta framework supports the *address sanitizer (ASan)* which reports *out-of-bounds reads* and the *memory sanitizer (MSan)* which reports *uninitialized reads*. Because heartbleed `memcpy`'s the memory contents into the response buffer, the use of a large size argument may result in one of these sanitizers to report a violation. The following experiments were done with a heartbeat request having no payload.

The *address sanitizer (ASan)* allowed us to leak up to 17725 bytes, before reporting a heap-out-of-bounds read. This is probably because the heartbeat request is stored in a heap-allocation that can accommodate the largest possible TLS record, which encapsulates 16 kilobytes of encrypted application data.

The *memory sanitizer (MSan)* did much better. It reported an uninitialized read when trying to leak 114 bytes or more. This number may differ between executions since heap allocations may be re-used while already initialized.

It turned out that OpenSSL contains quite a few uninitialized reads that are unrelated to heartbleed exploitation. We had to blacklist quite a few functions to allow the memory sanitizer to initialize and reach the heartbleed patch. Using existing tools to catch unsafe inputs, requires the instrumented code to be of high quality. Fixing or blacklisting all uninitialized reads may require significant effort.

We can conclude that the accuracy of MSan and ASan highly depends on the memory layout of the application. Since OpenSSL is highly optimized, it works with fewer and larger memory allocations, making it hard for us to accurately detect information leakage. As such, automated discovery of discriminators may result in many false positives.

5.1.3 Infected state in heartbleed

Our delta framework allows us to determine the infected state after running the vulnerable function. This infected state must be inspected to rule out the case that heartbleed detection has undesirable side-effects on the OpenSSL. When both executions merge, we know that OpenSSL’s integrity was preserved.

Unfortunately, sending a discriminator to our instrumented OpenSSL server results in numerous infections that do not dissolve. The main problem is that only the vulnerable execution will call `ssl3_write_bytes` to send a response. This call will encrypt the given buffer and will wrap the result into a TLS record. These operations modify a lot of state, such as I/O buffers and fields in the `SSL3` struct that keeps track of our session. Also, the `lh_new` function creates a hashmap that keeps various counters that track calls to the hashmap. New allocations modify the heap and the random-number-generator also updates its internal state.

Libdelta supports whitelisting, which allows us to tell the framework that some infections can be considered safe and are ignored when attempting to merge. This can be done by providing hardcoded memory ranges upfront. Because heap allocations may differ every execution, we need to add calls to OpenSSL’s sourcecode, which will instruct libdelta to whitelist some heap memory. However, it is very time consuming to track down all the infections and investigate whether they can be safely whitelisted. Since for heartbleed, safety is mainly enforced using the sanitizers described earlier, it might be better to disable the *merge check*. This is unfortunately not supported by our delta execution framework.

5.1.4 Fuzzing

In order to generate a set of candidate inputs to test, we selected VUzzer. Recall that VUzzer focuses its mutations on parts of the input that are used in `CMP` instructions, allowing it to discover new basic blocks more easily. Taint tracking is used to relate input bytes (offsets) to particular `CMP` instructions. The read buffer of a single `read` call will act as a *taint-source*. The offset of each byte in this output buffer will propagate along with the taint, allowing the fuzzer to learn how each byte is used.

Unfortunately, only encrypted heartbeats support propagation towards an attacker. This means that we have to use ZGrab to generate encrypted heartbeat requests, which will be `read` and used as taint-source. This results in taint-tracking on ciphertext instead of plain-text, which cripples the VUzzer, as it is now fuzzing the decryption logic instead. An in-memory fuzzer that starts with the plain-text buffer could be used to bypass the

TABLE 5.1: Patched test programs for which we know that discriminators exist

	Vulnerability type	Safety Policy	Known discriminator
echo server 1	Buffer over-read	Merge + ASan + MSan	Size = 0
echo server 2	Heap-based buffer overflow	Merge	Size = 100 (overflow into padding)
replace server	Stack-based buffer overflow	Merge	Size >100 (overflow into padding or from/to chars)
quote server	Stack-based Buffer overflow	Merge + ASan + MSan	99 character inputs without '/' or quotes

encryption logic. We did not explore this further as it again requires extra manual work, specific to heartbleed.

5.2 Experiments

We have crafted a four simple toy programs for which discriminators exist. These programs are created to test whether our system is able to find the discriminators automatically, but also to determine if handcrafted discriminators are indeed confirmed by our delta framework.

Table 5.1 lists the programs, the type of vulnerability they have and the *safety policy*, which describes how we test that an input is safe. By default, an input is considered safe when the delta framework successfully merges the two executions. In other cases, the analyst may opt to include the memory- or address-sanitizer as well.

The results are shown in table 5.2. It shows the number of candidate inputs generated by the fuzzer and the number of discriminators found. Also, it shows the number of propagating (P) and safe (S) inputs. The latter number is established using the safety policy from table 5.1. The framework can not reliably detect whether an input is *reaching* or *infecting*, so all non-propagating unsafe inputs are listed under *other*.

5.2.1 Echo server 1

Echo server 1 is shown in 2.2. It contains a buffer over-read vulnerability that may allow disclosure of sensitive information. This code mimics heartbleed, which we could not fully analyze as we explained earlier.

Sending a request having `size=0` is a discriminator, because the patch causes such requests to be ignored, resulting in *propagation through absence*. Other less-safe discriminators exist, where `size` is slightly larger than 32. These will leak a few bytes, but the impact of this is probably quite limited.

Our system was able to find the `size=0` discriminator automatically, as seen in table 5.2. We started the process using three seed inputs, which all had a length larger than 32. The VUzzer managed to set the `size` to zero, because `size` is compared with the zero constant, which was extracted from the program. The discriminator testing framework intercepted the `write()` that only takes place in the original version. Also, it detected that program state converged.

The 9 safe inputs are inputs that have a length field below 32. High length values (> 1000) will likely access unmapped memory, which is detected and reported by the framework (11 inputs). The remaining 10 inputs have shorter lengths, but 9 of them were marked as unsafe by the memory sanitizer, leaving just a single discriminator. We used both the memory sanitizer and address sanitizer, because merges will never detect information leakage.

5.2.2 Echo server 2

Echo server 2 shown in listing 2.3 contains a classic heap-based buffer overflow. Overflowing the buffer is required to discriminate. Fortunately, compilers align datastructures at 4 byte boundaries, allowing us to safely overflow at least one byte, as the buffer size is 99 bytes. Depending on the compiler and architecture, more padding bytes may be available.

None of the 138 generated candidate inputs was a discriminator. Again, finding a discriminator revolves around finding the correct `size` value, which is 100 in this case. The VUzzer is not aware of memory layout, buffer sizes and data types in the input buffer. Since it is not aware that the first four bytes are an integer, it can not gradually increment them to arrive at the desired value. When looking at the set of candidate inputs, most of them had random `size` values, which result in extremely large reads that reach into unmapped memory.

5.2.3 Replace server

The replace server will take a buffer containing a string and will replace all occurrences of a character by another one, as specified by `from` and `to` in the `hdr` struct. However, the string length is bounded to 100 bytes and the second `read` call on line 14 can be

<pre> 1 struct header { 2 char from; 3 char to; 4 unsigned short len; 5 }; 6 7 void handle_request(int clientfd) { 8 struct header hdr; 9 char string[100]; 10 int i; 11 12 read(clientfd, &hdr, 13 sizeof(struct header)); 14 read(clientfd, (char*)string, 15 hdr.len); 16 17 for (i=0; i<hdr.len; i++) { 18 if (string[i] == hdr.from) 19 string[i] = hdr.to; 20 } 21 22 write(clientfd, (char*)string, 23 (size_t)hdr.len); 24 } 25 26 27 28 </pre>	<pre> struct header { char from; char to; unsigned short len; }; void handle_request(int clientfd) { struct header hdr; char string[100]; int i; read(clientfd, &hdr, sizeof(struct header)); if (hdr.len > 100) { hdr.len = 100; } read(clientfd, string, hdr.len); for (i=0; i<hdr.len; i++) { if (string[i] == hdr.from) string[i] = hdr.to; } write(clientfd, (char*)string, (size_t)hdr.len); } </pre>
---	---

FIGURE 5.2: Replace server replaces characters in a string and sends the result back. It contains a buffer overflow.

overflowed by specifying a larger length. Because the overflow is stack-based, it can lead to control-flow hijacking when `handle_request` returns.

However, we can do a small overflow of `string`, which only causes `from` and `to` to be overwritten, modifying how the application behaves. Therefore, if we can send a buffer with the appropriate length where the last two bytes are the same, we can disable the character replacement logic. In the patched version, the string will be truncated and characters will be replaced.

We were surprised to find that 79 out of 265 candidates were discriminators. Again, the fuzzer needs to find a length that results in a small overflow. We think the large number of discriminators is due to the fact that our seed inputs contain readable ASCII bytes. The entire lower-case alphabet spans from decimal byte 97 until 122. These characters may end up in the length field as a result of cross-over and mutations done by the fuzzer. These values will result in small overflows. We have repeated the experiment again but now with a fully randomized inputs where only the header was fixed. The results were the same.

There is another reason that explains the large number of discriminators. The compiler passes used by the delta framework add local variables and thus create extra stack space which can be overflowed. Many of such overflows may not result in crashes, thus resulting

<pre> 1 void handle_request(int clientfd) { 2 char out[100], in[100], *i, *o; 3 memset(in, 0, 100); 4 memset(out, 0, 100); 5 6 read(clientfd, in, 99); 7 8 for (i = in, o = out; 9 *i != 0; 10 i++, o++) { 11 if (*i == '"' *i == '\\') { 12 *o = '\\'; 13 ++o; 14 } 15 *o = *i; 16 } 17 18 write(clientfd, (char*)out, 19 strlen(out) + 1); 20 } </pre>	<pre> void handle_request(int clientfd) { char out[100], in[100], *i, *o; memset(in, 0, 100); memset(out, 0, 100); read(clientfd, in, 99); for (i = in, o = out; *i != 0 && o < &out[98]; i++, o++) { if (*i == '"' *i == '\\') { *o = '\\'; ++o; } *o = *i; } write(clientfd, (char*)out, strlen(out) + 1); } </pre>
--	--

FIGURE 5.3: The quote server escapes quotes (and backslashes) and returns the escaped string. A buffer is overflowed when a string contains too many characters that need to be escaped.

in more discriminators. However, using the same inputs on an un-instrumented version of the server may indeed result in a crash. Therefore, the use of compile passes may cause our framework to give false positives. There is only 4 bytes of space between the end of `string` and the start of `hdr`, but in the instrumented version, this can be an order of magnitude more.

5.2.4 Quote server

The quote server reads up to 99 characters from the network connection and escapes all quotes using backslashes. The escaped string is sent back to the client. Input strings which have lots of quotes and backslashes in the first 99 characters may cause an overflow on the `out` buffer.

The vulnerability was patched by ensuring that the output pointer (`o`) never moves past the 98th character. The patch is slightly too conservative and stops after the 98th character to ensure that there is room in the buffer in case the 98th character needs to be escaped. The patch would be more *seamless* if it allowed the 99th character to be processed as well, given that it doesn't need escaping. However, this character is now dropped, which makes all 99+ character strings infect the state. More generally, all strings for which $N + E \geq 99$ will infect the state, where N is the number of characters and E is the number of characters that need to be escaped. All these infections will propagate to the attacker.

TABLE 5.2: Shows number of discriminators, safe (S) and propagating (P) inputs found for each toy program.

	Candidates	Discriminators	P	S	Other
echo server 1	30	1	9	9	11
echo server 2	138	0	8	1	129
replace server	265	79	0	90	96
quote server	278	1	210	67	0

In this example, overflowing and non-overflowing discriminators exist. When $N+E = 99$, the infection relies on the patch being too conservative about the last character and no overflow will occur. When $N + E > 99$, an overflow will happen in the unpatched version. This may be problematic, depending on the stack layout. A discriminator for which $N + E = 99$ is obviously more desirable.

It turns out that our framework finds one non-overflowing discriminator. This input is longer than 99 characters and has no characters that need to be escaped. Apart from that, there are 53 overflowing discriminators, but these are marked as unsafe by the address sanitizer and are therefore classified as *propagating (P)* in table 5.2. It turns out that a couple of these cause very small overflows and are actually usable on the un-instrumented un-patched version of the server. 67 inputs did not propagate. The main reason was that they contain a zero-byte in the first 99 characters, causing the output string to be less than 99 characters.

As shown in 5.2, Discriminator discovery resulted in discriminators for 3 out of 4 toy programs. However, the toy programs are designed to be simple and it will be more difficult to find discriminators for real-world vulnerabilities.

Chapter 6

Related work

6.1 Fingerprinting in network reconnaissance

Patch fingerprinting is a reconnaissance technique that directly addresses the question whether a server is vulnerable to an attack. Existing reconnaissance techniques typically only narrow down the search of vulnerabilities.

One of the oldest techniques to learn more about services running on a host is banner grabbing. Banners are easy to extract and may provide valuable information to both system administrators and malicious attackers. Hence, banners are now often hidden to keep adversaries in the dark, making it harder to launch attacks. ZMap [21] is an efficient internet-wide scanner that supports banner grabbing on a large scale.

OS fingerprinting [22] [23] leverages ambiguity in network protocol specifications to distinguish different network stack implementations in order to identify the operating system. Since the modern TCP/IP stack is complicated and always exposed to the outside world, these different implementations expose a large accessible *surface* that can be fingerprinted, akin to the surface on our fingertips. This technique is popular and difficult to defend against. Yet, it does not give detailed information such as the kernel version numbers. Also, countermeasures exist such as [24].

Since TCP/IP implementations change slowly over time, it is essential to keep fingerprints up-to-date. Automating OS fingerprinting was attempted [25][23][26], but results were mixed. It is possible to discern between OS families such as Linux, Microsoft windows and Solaris. More fine-grained fingerprints remain a challenge.

Even though there seems to be resemblance, OS fingerprinting and patch fingerprinting are very different. Security patches, as opposed to TCP/IP implementations only expose

a very tiny *surface* to fingerprint. However, OS fingerprinting has to deal with a lot of noise generated by non-determinism, hardware differences, application behaviour and system configuration. Also, OS fingerprinting aims to classify many OS families and kernel versions. Hence, OS fingerprinting techniques often send hundreds of probing packets and use database of fingerprints to generate a list of matching operating systems, based on the probe responses. It is often up to the user to draw conclusions from a list of matching fingerprints.

Fingerprinting of application versions was attempted in [1]. We are currently not aware of generic remote fingerprinting methods for application versions.

6.2 Web vulnerability scanners

Web applications are often targeted by attackers and thus methods to automatically find vulnerabilities in web applications are heavily researched. Automatically finding *SQL-injection* and *Cross-site-scripting (XSS)* vulnerabilities is possible with black-box scanners such as Secubat[3] and Acunetix[4]. Black-box scanners are popular because they are easy to use and do not depend on the server-side technology that was used to build the scanned application. They do depend on the use of HTML, Javascript and SQL which are used in pretty much any web application.

These scanners typically employ a *crawler module*, an *attacker module* and an *analysis module*. The crawler explores linked pages within a domain and finds interesting input points such as GET parameters and forms. The attacker module will send requests carrying particular inputs known to trigger vulnerabilities when such inputs are not properly sanitized. The analysis module will inspect the responses to such requests and will assign a score indicating the likelihood that a vulnerability was discovered.

For many web vulnerabilities, particular inputs exist that allow detection of vulnerabilities. For example, passing `<script>alert('hello');</script>` to a GET parameter and finding it in the response, proves that a cross-site-scripting vulnerability exists without doing any harm. Similar prefabricated inputs exist to detect SQL-injection vulnerabilities.

Such prefabricated inputs exist because, unlike compiled server applications, web applications are built using high level components and programming languages. These components expose a solid well-defined interface and in case of un-sanitized inputs, a broad attack surface. The problem of finding safe inputs is trivial, because we don't need to worry about memory errors at this level of abstraction. Because of the rich

content sent back to the client, it is often easy to remotely discriminate the vulnerable from the secure.

6.3 Mutation testing

The field of mutation testing shows remarkable similarities with patch fingerprinting. Therefore, we will explore the topic in this section and express the problems in terms of patch fingerprinting.

In software testing, one tries to prove the correctness of a program for its entire input domain, by testing whether a finite set of test inputs all result in correct output. In practice, it is difficult to know whether a set of test inputs sufficiently exercises a program to find all its programming errors.

Howden et al[27] introduced the notion of test set *reliability*. Given a program P that accepts inputs in domain D , test set $T \subset D$ is said to be *reliable* when passing all tests $t \in T$, implies that P is correct for its entire domain D . In other words:

Definition 1. If P is a program to implement function F on domain D , then a test set $T \subset D$ is *reliable* for P and F if: $\forall t \in T, P(t) = F(t) \Rightarrow \forall t \in D, P(t) = F(t)$

Although Howden[27] showed that reliable test sets exist, he also showed that there is no efficient method to find them. Researchers found that rather than checking for correctness, it is more practical to prove that a test set is able to identify a finite number of bugs. Subsequently, the notion of test set *relative adequacy* was introduced, also known as *mutation adequacy*. Adequacy is defined relative to a set Φ of incorrect variations of program P . A test set T is said to be *relatively adequate* to Φ when all programs in Φ fail to pass the tests.

Definition 2. If P is a program to implement function F on domain D and Φ is a finite set of programs, then a test set $T \subset D$ is *adequate* for P relative to Φ if: \forall programs $Q \in \Phi$, if $Q(D) \neq F(D) \Rightarrow \exists t \in T, Q(t) \neq F(t) \Rightarrow \forall t \in D, P(t) = F(t)$

Relative adequacy is of course an approximation and its effectiveness strongly depends on the size of Φ as well as the faults introduced in members of Φ .

In order to construct Φ , researchers generate many variations of a program, which are called *mutants*, each having a single source-level modification, which likely causes the

mutant to behave incorrectly. A *mutation adequate* test set will *kill all mutants*, meaning that it will identify all of them as incorrect, because their output does not match with the output of the correct program.

We assume that the *program-under-test* is correct which may seem strange. However, note that when the program-under-test has a fault, a mutant exists that can only be killed by a test that would also fail as a result of this fault. Thus, by introducing a large number of faults and writing tests that identify them, we can discover real faults caused by human error.

Some mutations will not cause the program to behave differently, even though a statement of the program has changed. These are called *equivalent mutants* and can not be killed. As such, they do not help us to establish the adequacy of a test set and should not be used. A complete solution to this problem is not possible[6] so most mutation systems require human intervention to remove equivalent mutants.

Besides mutant equivalence, another problem in mutation testing is automated mutant killing. By generating tests for all our mutants, we can automatically enhance a test set and possibly find existing bugs. While generating mutants is not very difficult, automatically killing them is much more difficult and requires advanced tools [28][29][7] or manual work.

For a test to kill a mutant, it has to satisfy three conditions:

- The test must *reach* the mutated statement
- Execution of the mutated statement should *infect* the program state, such that it is different from the correct program. In the example shown in figure-6.1, an input of $A = 6$ and $B = 6$, would have this result, while $A = 6$ and $B = 5$ (or vice versa) would not.
- The infected program state must *propagate* to the output of the program, allowing the test to check it.

This is also known as the *RIP model*[5]. The infection and propagation conditions are also commonly referred to as the *necessity condition* and the *sufficiency condition*.

When only the first two conditions are met, the test input is said to *weakly kill* a mutant. This means that the execution of the mutated statement causes a temporary *infection* of program state, which ultimately converged to match the state of the original program. *Weak mutation testing* is an optimization because we no longer need to execute all mutants to completion. When we observe a state divergence after executing the mutated

<pre> 1 if (a < b) { 2 c = 1; 3 } else { 4 c = 0; 5 } </pre>	<pre> if (a <= b) { c = 1; } else { c = 0; } </pre>
Original	Mutated

FIGURE 6.1: Shows a mutation on the $<$ operator. The state (variable c) is only infected when $a == b$

statement, we assume that this divergence will be reflected in the output of the program and the mutant is *weakly killed*. Weak mutation testing is more practical than *strong mutation testing*, since it is difficult to strongly kill all mutants. Many infecting inputs do not propagate and finding ones that do is difficult. This all results in many live mutants, which falsely indicate that the test suite does not cover all code.

It could happen that a mutated statement causes a temporary divergence of state, which converges before any output is returned. Budd et al[6] called this coincidental correctness, because a temporary erroneous state is later erased causing the output of the program to be correct. It could be that the mutated statement is not participating in creating the output that is checked by the test. Also, protection mechanisms in programs could erase certain state if it was found to be incorrect.

Mutation testing has a strong resemblance with our work. A security patch is often a small, local modification and a patched program is in that respect similar to a mutant. Our aim is to *strongly kill* this mutant, as to ensure an observable response discriminating the patched from the unpatched version.

A difference is that we are trying to apply mutant killing to servers, which in principle have an infinite execution time as opposed to stand-alone programs. This is the motivation for our safety property. In other words, we want to kill a mutant without disrupting the server.

Although security patches are simple, they are not as simple as mutations and we know less about how they affect the program. Mutations typically add or replace a single operator or operand. More importantly, the mutation testing framework *knows* what kind of constraint needs to be fulfilled to infect the state of each generated mutant. In figure 6.1, $<$ is replaced by $<=$, so the state will be infected if and only if $A = B$. Demillo et al[7] defines such constraints as *necessity constraints*. For security patches, we do not know these necessity constraints. We do know that an exploit of the vulnerability will *exercise* the patch in some way. Automated extraction of such infection constraints may be a key step towards evolving patch fingerprinting.

Also notice how equivalent mutants are very much like *seamless* security patches. When a patch modifies the program in such a way that this is not noticeable from the outside, we speak of a *seamless* patch. Similarly, no input exists that exposes an equivalent mutant through propagation. The only way to detect vulnerabilities that are seamlessly patched is full exploitation. In chapter 4, we discussed some types of vulnerabilities that are often seamlessly patched. Seamless patches are interesting from a defense point of view since it prohibits automatic scanning. Truly malicious attackers on the other hand, may simply try to run the exploit, so seamless patches do not offer any real protection.

Chapter 7

Discussion

The main research questions we had, was whether our delta execution framework is effective at recognizing discriminators for real-world vulnerabilities. We quickly discovered that we did not have a realistic test set of vulnerable programs for which discriminators were known. The main vulnerability that led to our question was heartbleed, for which a discriminator is known. Unfortunately, we were unable to find other cases.

Even though thousands of vulnerabilities are documented, the vast majority affects servers running on non-linux operating systems or closed-source servers. There were quite a few vulnerabilities in open source software, which had a patch but lacked a working exploit. The limitations imposed by our framework, discussed in 7.1.1 made our search even more difficult and we eventually decided to create some toy programs to test against.

Although it was not the main focus of our work, we also attempted to discover discriminators. One major reason for doing so, was the lack of known discriminators in real-world software to test.

In the remainder of this chapter, We will first discuss discriminator testing in 7.1. We will go over the limitations of the current delta framework and discuss a few insights that could lead to an improved version of such a framework. We will then discuss how discriminator discovery 7.2 could be improved and how we could perhaps merge discriminator testing and discovery to further improve the patch fingerprinting technique.

7.1 Discriminator testing

7.1.1 Limitations

First of all, the current systems we have created only work on server programs written in C that do not have threads. Making threads work in our delta framework takes a lot of time and is outside the scope of this research.

Also, diverged state in thread local storage is not detected and `mmap`'ing memory during split execution is not supported.

7.1.2 Accuracy of splitting and merging

Our delta execution framework is quite coarse-grained, when it comes to splitting and merging. The `deltafy` pass, discussed in section 3.1, compares both programs at function-granularity and split as soon as a patched function is called. We attempt to merge when this patched function returns. This leads to two execution paths which should not be ran in split execution:

- **split-to-patch-site path:** These are the first instructions of the patched function, up to the first patch-site.
- **patch-site-to-merge path:** These are the instructions executed after the last patch-site, until the function returns which is where we will attempt to merge.

Depending on how the program is organized, these paths may contain a lot of instructions. This is problematic for a number of reasons:

First of all, we need to take care of any I/O that happens during split execution. I/O calls and access to hardware performance counters during split execution will introduce state differences unrelated to the presence or absence of the patch. These spurious differences between the processes show up as infections and increase the amount of manual work. We can solve this by intercepting all these I/O operations to ensure both processes perceive the same environment, but this is a lot of work. It may be more effective to reduce the time spent in split execution, by splitting immediately before the patch-site and merging immediately afterwards.

Besides pollution caused by I/O, a long *patch-site-to-merge* path has more problems. It gives infections, a patch-site, more time to spread to other memory areas, resulting in more analysis work. This is shown in figure 7.1 on the left. It is obviously much easier to

distinguish safe and unsafe inputs by inspecting a few infected bytes immediately after the patch than to consider many infections that all result from an initial small infection.

Heartbleed is an excellent example of why a long *patch-site-to-merge path* is problematic. When processing the discriminator, the patched version returns from the vulnerable `tls1_process_heartbeat` function while the original version continues. The original version calls `ssl3_write_bytes` which touches a lot of state. Because we try to merge when both executions return from `tls1_process_heartbeat`, we see a lot of diverged bytes, the vast majority of which was caused by the `ssl3_write_bytes` call. We had to inspect all this diverged state to ensure that the discriminator is safe.

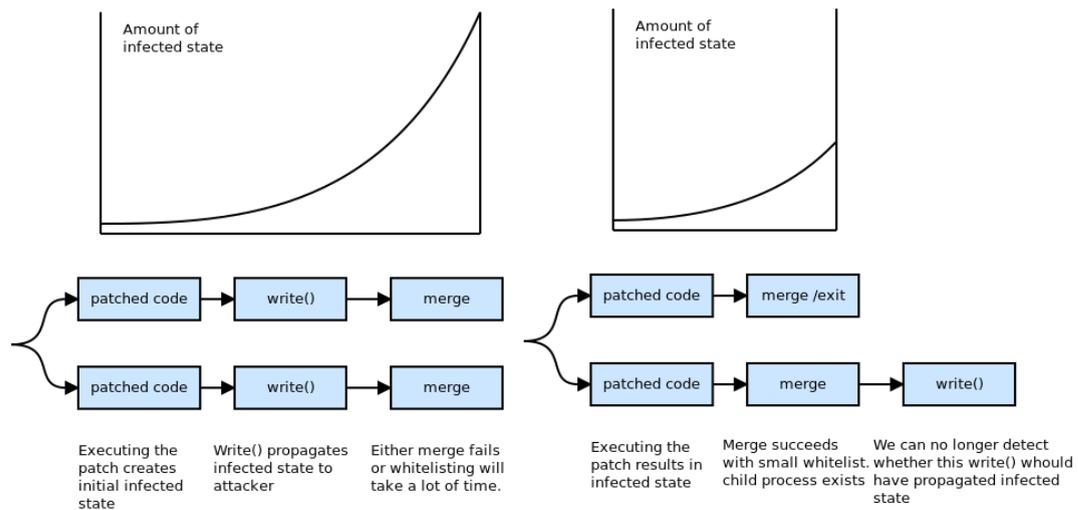


FIGURE 7.1: Left side shows *write-before-merge* resulting in a lot of manual work to analyze/whitelist all diverged state. Right side shows *merge-before-write*. When the merge is successful, the patched process exists and all infected state is lost even though whitelisted infections may still exist. Any propagation that may happen due to the subsequent `write()` call will not be detected by our delta framework

Finally, our delta framework cannot reliably detect whether an input *reaches* the patch or *infects* state. Knowing which properties are satisfied is an important measure for how good an input is and could be used by future methods to incrementally solve all RIPS constraints. One might say that splitting may indicate reachability (patched function is called) and failure to merge may indicate infection. However, this is a very coarse approximation. Splitting obviously does not guarantee patch-site execution. Merge failures could also be false positives caused by pollution resulting from unhandled I/O operations discussed earlier. Finally, successful merges can still hide the fact that there was a temporary infection. Establishing reachability and infection for an input will be more accurate when the mentioned paths are reduced in length or eliminated completely.

So how can we increase split/merge accuracy and shorten the undesirable code paths?

It could be done by improving our current compiler instrumentation by analyzing differences at basic block granularity. One could mark a set of basic blocks as *patch BB's* and then instrument all incoming edges with split hooks and outgoing edges with merge hooks. Such improvements are difficult to implement, which is the reason we did not try it.

Another interesting approach is to use dynamic binary instrumentation such as PIN[18]. A big advantage is that one can dynamically decide after which instruction to split or merge. Since dynamic binary instrumentation is only concerned with a single execution, it may be much easier to implement accurate delta execution for patch fingerprinting.

Tucek et al[12] actually use PIN to implement delta execution. Besides supporting function-granularity splits and merges, they allow an analyst to place macros that signify the start and end of a patch. They do not elaborate on how they merge two versions of the code into a single binary, such that the execution splits when this section of the code is executed.

7.1.3 Inspecting Diverged State

When we attempt to merge, the delta framework will always report on all diverged state that was found. It will do so by dumping all pairs of memory pages that diverged and also an overview of the offsets and sizes of the diverged bytes.

Unfortunately it is very difficult to interpret such reports because we do not know which variables and datastructures are affected, let alone by which instruction. This makes it nearly impossible for an analyst to determine whether some differences should be whitelisted.

To partially mitigate this, we can leverage *origins tracking information* from the memory sanitizer to find out when the memory is allocated from where it is copied. This tells us where the data was first allocated, allowing us to find out which datastructure was infected.

However, this origins tracking information is only guaranteed to be available for memory areas that contain uninitialized bits. And even if is available, inspection with GDB is still required to find out which part of a datastructure is infected. Apart from tracing infections to source-level, it requires some understanding of the applications internals to be able to judge whether it can be safely whitelisted.

7.1.4 Retaining Diverged State

Our implementation of delta execution is simpler than the original implementation in [12]. When merging, the original implementation copies all diverged pages to the parent process and `mprotect()`'s them, thus retaining two versions of all the so called *delta data*. As soon as the application reads or writes to these pages, the execution splits again and each execution operates on it's own version of the data. In the implementation, merging has nothing to do with asserting that both processes have converged, but rather saving resources by exiting one of the processes.

Initially, we did not think that retaining *delta data* like this would be helpful. After all, we simply want to assert that all memory contents, except what is whitelisted, has fully converged. Paradoxically, we also want infected state to propagate via an I/O call. As a result of `exit`'ing one process and not retaining it's diverged page, we essentially erase all diverged state. Therefore, we also lose legal diverged state in whitelisted memory. This diverged state could, in theory, still propagate via an I/O call, but because we do not retain it, we can not detect such propagation. This results in false negatives as shown on the right side of figure 7.1. Hence, we must retain both versions of the diverged state to also detect propagation in a *merge-before-write* situation.

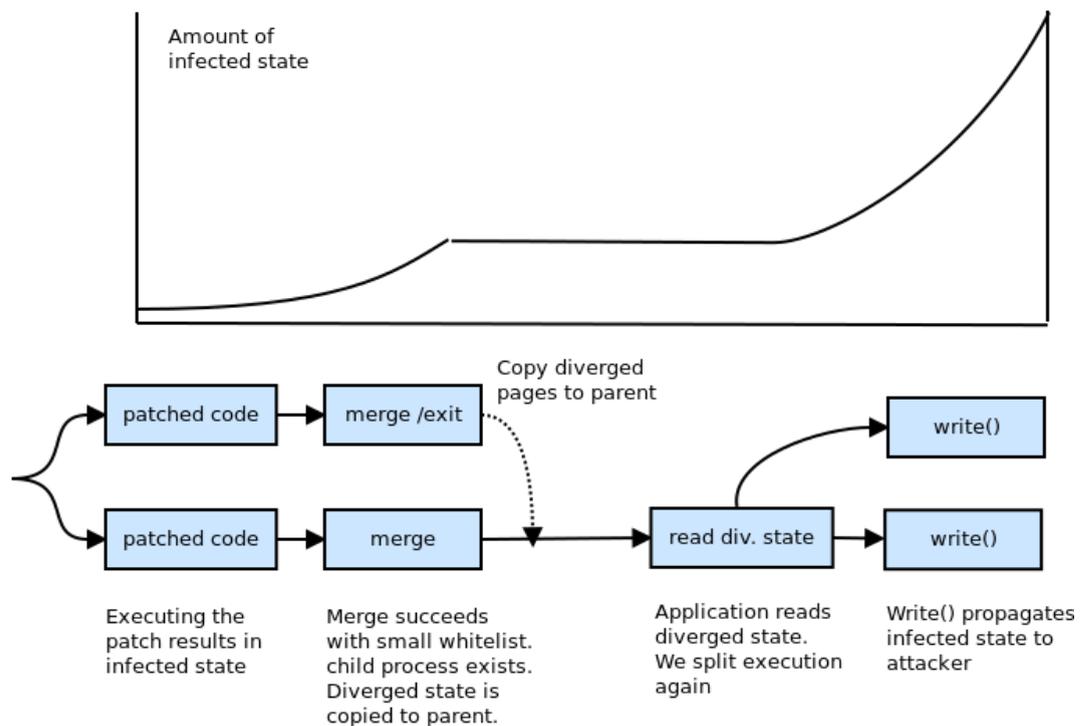


FIGURE 7.2: This is what could happen if we retain diverged state. The first merge checks the integrity of the process but retains the diverged state. As a result, the execution splits again when this diverged state is accessed and used by a `write()` call

In section 7.1.2, we discussed the problems of prolonged split execution and the merits of late splitting and early merging. If we also would like to retain diverged state, we arrive at the original implementation of delta execution. When we merge, we check the integrity of the unpatched process, like we currently do and abort when the input is deemed unsafe. However, if the integrity check is successful, we retain all diverged pages to see if the infection spreads or dissolves. Figure 7.2 shows how this works. We can now detect propagation long after we checked the integrity of the process.

This approach to delta execution has some consequences for how we check the integrity of the unpatched process. When retaining diverged state, splits and subsequent merges are not only caused by execution of modified code but also by accessing diverged state. The integrity check must only be done after executing the modified code, because that is the only place where corruption of the unpatched process can happen. The integrity check asserts that all diverged state (if any) is whitelisted and that it can freely propagate as is. Therefore, no further integrity checks are needed as long as the patched code is not executed again.

7.1.5 Confidentiality

Because merges only check integrity, we have used the address sanitizer and memory sanitizer to find out about information disclosure. We only considered buffer over-read vulnerabilities where controlling a *size* argument to a call such as `memcpy` leads to reading a contiguous chunk of memory. We hope that this large read violates buffer bounds or touches uninitialized memory, which would be detected by the address- or memory sanitizer.

We found that the success of this method highly depends on the memory layout of an application. For example, in heartbleed, we could leak 17 kilobytes of memory before any boundary was crossed, allowing many leaks to go undetected. Fortunately, the memory sanitizer performed much better, allowing only leaks of up to 114 bytes of memory.

In some cases, we found that the address sanitizer is too strict. We have shown that we can use small legal overflows to craft discriminators (See section 2.1.4). It is important to check that an overflow is *safe*, meaning that it only touches padding bytes. Unfortunately, the address sanitizer does not tolerate any single byte overflow. As such, we cannot easily verify that the overflow is indeed small and harmless.

It might be possible to modify the address sanitizer to support *orange zones*. An orange zone is a portion of the red-zone, which we may overflow. The size of the orange zone

could be parameter set by the analyst and must be smaller than the red-zone. This would allow the analyst to find the minimal overflow needed to craft a discriminator.

What can also be problematic is that the sanitizers do not focus specifically on the patch-site, but on the complete server. Therefore, the sanitizers may report a vast number of unrelated problems. This was the case when we used the memory sanitizer with OpenSSL. We had to blacklist 6 functions to suppress existing errors.

Finally, we had a lot of problems integrating the sanitizers into the delta execution framework. The sanitizers produced spurious state differences, cause merge failures and also intercepted library calls done by the delta framework, resulting in a variety of problems.

We believe that addressing confidentiality in patch fingerprinting requires more specialized instrumentation that integrates better with delta execution. If it is possible to use off-the-shelf tools such as the sanitizers, they should be ran separately as integration may not bring significant advantages compared to the required efforts.

7.2 Discriminator Discovery

Discriminator discovery can currently not consume feedback from discriminator testing. Since the fuzzer is currently not aware of the kind of constraints we want to solve, we mostly rely on brute-force. We did not integrate discriminator discovery with discriminator testing because the used tools required different processor architectures. Even if they would use the same architecture, the different pieces of instrumentation would require radical changes to make them compatible.

We believe huge improvements can be made by making the fuzzer aware of some constraints. Using concolic execution, one could extract the path constraints generated by exploitation of the vulnerability. Using this path constraint for input generation, one could guarantee that the reachability property is satisfied for all inputs. It might be possible to introduce infection constraints as well, further narrowing down the search.

Despite the path explosion problem, symbolic execution can still be an interesting technique, as the amount of patched code to analyze is often really small. The symbolic execution engine could be modified to *fork* the symbolic execution as soon as the first patched instruction is executed. Analyzing the path constraints could result in an *infection constraint*, which describe the set of infecting inputs. After learning these constraints, more lightweight methods can be used to find safe and propagating inputs that satisfy these constraints.

There are some scalability challenges with discriminator discovery using a fuzzer: First, observe that fuzzing for discriminators is very different from using a fuzzer to excavate bugs or vulnerabilities. When using a fuzzer to find discriminators, we fuzz a known broken piece of code, resulting in many crashes and memory corruption. When fuzzing for bugs, most inputs will result in a normal execution and finding a single crashing input is the end-goal.

Also, most existing fuzzers target locally executing programs rather than servers. They start the binary for each input and wait for it to exit. When fuzzing a server, one could start the server and let the fuzzer launch a client script that sends inputs and collects their responses. When trying to find bugs, a server crash would indicate that the last input triggered a bug.

Unfortunately, when searching for discriminators in servers, crashes and corruption are the norm and the server has to be restarted for every input. Since starting a server is done infrequently in real-world usage, as opposed to starting local programs, developers do not optimize for it, resulting in poor fuzzer performance due to slow restarts.

Furthermore, some number of interactions may be required before a server will be in a state where vulnerable code will be triggered. To customize these interactions per vulnerability, we introduced the session script into our design, which takes a fuzzed payload and delivers it after the required setup interactions. For example, Heartbleed required an entire SSL handshake. We modified the code offered by ZGrab[21] to perform the handshake and to provide the desired heartbeat request. Since the session script is launched as a new process for each input, it also incurs significant overhead.

These issues could be solved if the fuzzer could run the session setup once, freeze the server and fork off one process for each fuzzed input. Since we are using a fuzzer to solve highly complex constraints, optimizing for brute-force usage may substantially improve the likelihood of finding a discriminator. AFL[30], a popular state-of-the-art fuzzer, uses this fork-server approach as well [31]. As discriminator discovery was not a core part of this thesis, we did not attempt to implement such optimizations. However, it is important to keep these observations in mind for future fuzzer-based implementations.

7.3 Alternative applications

An alternative application of patch fingerprinting is to run the analysis before releasing a patch to the public. This allows software maintainers to remove discriminators, thus defending against malicious vulnerability scans. In other words, the aim to make their patch *seamless*, eliminating all network-facing behavioral differences.

As discussed in section 4.6, some types of memory errors are almost always patched seamlessly. Input validation errors on the other hand, are forced to handle the error which can be detected remotely.

Also, safe patch fingerprinting could be integrated into the software development process, making developers aware of how their code changes could be fingerprinted, exposing information to the public. This is difficult though, because of our current assumption that fingerprintable patches are small. Discriminators could also hint on subtle possibly unintended implementation changes that are not sufficiently tested by the existing test suites.

7.4 Future work

So far, we have discussed various aspects of discriminator testing and discovery. In this section, We will try to consolidate these insights in order to formulate interesting future research questions.

This research focused on patch fingerprinting for open-source software. However, a substantial number of vulnerabilities exist software for which source-code is not available. Our research was hampered by the lack of vulnerabilities for which source-code, a patch and a working exploit was available. Performing patch fingerprinting on compiled binaries weakens these requirements and may lead to more results and use-cases. Obviously, it will be much harder to interpret diverged state making it harder to create a good whitelist.

Studying the effects of patches, as we do in this thesis, requires us to merge the patched and unpatched version into a binary which we can analyze. This merging, which we call *deltafying*, determines the granularity of splitting and merging which was discussed earlier. We have done this using a compiler pass and Tucek et al[12] did the same. Yet, it may be possible to deltafy compiled binaries, enabling patch fingerprinting without source-code. Although this is complicated, it will probably enable accurate placement of split hooks, addressing the problems outlined in section 7.1.2.

After we have a deltafied program, we can add instrumentation to either test or discover discriminators. Both of these problems can be approached using a wide variety of techniques and most probably hybridizations of techniques. Also, merging discriminator discovery with discriminator testing into a single analysis process would make it more effective, since discovery can then rely on the notion of diverged state and its propagation. Finally, the interpretation of safety demands that we modularize our instrumentation, to support various types of vulnerabilities.

We found that combining multiple pieces of compiler instrumentation can become very complex. We have seen that they affect the memory layout, causing unsafe inputs to be flagged as safe. Compiler passes have to be executed in an order, often leading to instrumentation of instrumentation. This interference happens at runtime too, as the runtime components of the instrumentation are not aware of each other. It is essential to express the problems of patch fingerprinting in terms of an instrumentation framework, which cleanly separates responsibilities.

We believe that such a framework should be built as dynamic binary instrumentation (DBI) such as PIN[18]. It is less intrusive and allows many modules to listen to runtime events at instruction granularity. Also, it does not require source-code, opening the door towards patch fingerprinting on binaries. It has been shown that complex analysis frameworks can be built with PIN, such as Triton[32]. Triton offers concolic execution, snapshotting and taint tracking, which could be instrumental in the creation of a patch fingerprinting framework.

The problem outlined in section 7.1.3 remains unsolved. We think that leveraging debug information, together with instrumentation of store operations during split execution, would be the most promising approach. We have spent considerable effort to let our deltafy compiler pass preserve debug information, but the results were disappointing. It remains to be seen whether debug information can be easily preserved when merging compiled binaries. As it turns out, accurately merging two versions of the same code, while retaining high-level debug information remains one of the core challenges that need to be addressed.

We currently focused only on `write` calls, while other I/O events could also facilitate propagation. Besides other I/O system calls, different side-channels may exist that could be used to learn whether the server is patched. When the attacker can run code on the server, different side-channels attacks may be feasible. Exploring these possibilities further remains future work.

Chapter 8

Conclusion

In this thesis, we took the initial steps towards safe patch fingerprinting. It is a novel approach that could vastly improve the effectiveness of vulnerability scanners by automatically providing fingerprints for the most recent vulnerabilities. It supports detection of vulnerable servers, without having to depend on the software version of such servers.

Delta execution turned out to be an effective technique to test whether an input is a discriminator. Due to the unavailability of suitable vulnerabilities and patches, we could not extensively test it. Yet, our framework confirmed the known discriminator for heartbleed and was used to find discriminators for a number of vulnerabilities in test servers that we created ourselves. These experiments provided many insights which can lead to more practical and accurate implementations.

The RIPS model, derived from the RIP model used in mutation testing, breaks the problem of finding discriminators down into better understood sub-problems, which can be approached using existing analysis techniques. Findings in the field of mutation testing will be instrumental in further developing safe patch fingerprinting.

Safety, which is being added to the existing RIP model, turned out to be difficult to implement and work with since its definition is vulnerability dependent. Checking integrity using merges is very conservative and whitelisting memory areas results in a lot of manual work. This problem is worsened by inaccurate placement of split and merge hooks. Existing tools that help us catch unsafe inputs, such as the address- and memory sanitizer work to some extent. However, they were inflexible and difficult to integrate with delta execution.

We did a rudimentary attempt to automatically discover discriminators. The results were good for small test servers, but much more work is needed to make automated discovery scale to realistic software. We believe that efficiently finding discriminators

requires a solution based on symbolic or concolic execution, allowing one to extract constraints and use them in a more focused search. Also, integration of discriminator testing and discovery could make safe patch fingerprinting much more practical.

To improve automatic discovery of discriminators and address the issues with safety, a more integrated approach is needed. We envision a framework based on dynamic binary instrumentation that harbors many different modules, each responsible for solving a sub-problem of patch fingerprinting. Dynamic binary instrumentation is suitable because it is non-invasive, allowing more accurate safety checks and better supports modularization than compiler based approaches, because analysis modules will be less coupled with the program under analysis.

Even though many improvements can be made upon our approach, there may be fundamental limits to this technique. We have analyzed a number of vulnerability classes typically found in compiled software. We argue that many of them can be seamlessly patched, leaving no discriminators that are required for a safe vulnerability scan. Also, we currently need the source-code, a patch and an exploit, which are often not available. It might be possible to do fingerprinting on compiled binaries directly, but it won't make the problem easier to solve.

Bibliography

- [1] Jason Damron. Identifiable fingerprints in network applications. *USENIX; login*, 28(6):16–20, 2003.
- [2] R Deraison. Nessus. URL <http://www.nessus.com/>.
- [3] Stefan Kals, Engin Kirda, Christopher Kruegel, and Nenad Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256. ACM, 2006.
- [4] Acunetix vulnerability scanner: Web application security. URL <https://www.acunetix.com/vulnerability-scanner/>.
- [5] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.
- [6] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.
- [7] RA DeMilli and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, 1991.
- [8] nginx. URL <https://nginx.org/en/>.
- [9] Roy T. Fielding and Gail Kaiser. The apache http server project. *IEEE Internet Computing*, 1(4):88–90, 1997.
- [10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.
- [11] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

- [12] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. *ACM SIGARCH Computer Architecture News*, 37(1):193–204, 2009.
- [13] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 46–55. IEEE, 2015.
- [14] 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.
- [15] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.
- [16] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.
- [17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.
- [18] 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.
- [19] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *Acm Sigplan Notices*, volume 47, pages 121–132. ACM, 2012.
- [20] The linux kernel archives: Soft dirty pte’s. URL <https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt>.
- [21] Zakir Durumeric, Eric Wustrow, and J Alex Halderman. Zmap: Fast internet-wide scanning and its security applications. In *USENIX Security Symposium*, volume 8, pages 47–53, 2013.
- [22] Fyodor Yarochkin. Remote os detection via tcp/ip stack fingerprinting. *Phrack Magazine*, 17(3):1–10, 1998.

-
- [23] David W Richardson, Steven D Gribble, and Tadayoshi Kohno. The limits of automatic os fingerprint generation. In *Proceedings of the 3rd ACM workshop on Artificial intelligence and security*, pages 24–34. ACM, 2010.
- [24] Kathy Wang. Frustrating os fingerprinting with morph. In *Talk at the fifth HOPE conference*, 2004.
- [25] Juan Caballero, Shobha Venkataraman, Pongsin Poosankam, Min G Kang, Dawn Song, and Avrim Blum. Fig: Automatic fingerprint generation. *Department of Electrical and Computing Engineering*, page 27, 2007.
- [26] François Gagnon, Babak Esfandiari, and Leopoldo Bertossi. A hybrid approach to operating system discovery using answer set programming. In *Integrated Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on*, pages 391–400. IEEE, 2007.
- [27] William E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, (3):208–215, 1976.
- [28] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.
- [29] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.
- [30] Michal Zalewski. American fuzzy lop, 2015.
- [31] Peter Gutmann. Fuzzing code with afl.
- [32] Florent Soudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*, pages 31–54. SSTIC, 2015.