Precise and Accurate Patch Presence Test for Binaries

Hang Zhang and Zhiyun Qian, ***University of California, Riverside***

https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

Open access to the Proceedings of the 27th USENIX Security Symposium is sponsored by USENIX.

Precise and Accurate Patch Presence Test for Binaries

Hang Zhang
University of California, Riverside
hang@cs.ucr.edu

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Abstract

Patching is the main resort to battle software vulnerabilities. It is critical to ensure that patches are propagated to all affected software timely, which, unfortunately, is often not the case. Thus the capability to accurately test the security patch presence in software distributions is crucial, for both defenders and attackers.

Inspired by human analysts' behaviors to inspect only small and localized code areas, we present FIBER, an automated system that leverages this observation in its core design. FIBER works by first parsing and analyzing the open-source security patches carefully and then generating fine-grained binary signatures that faithfully reflect the most representative syntax and semantic changes introduced by the patch, which are used to search against target binaries. Compared to previous work, FIBER leverages the source-level insight strategically by primarily focusing on small changes of patches and minimal contexts, instead of the whole function or file. We have systematically evaluated FIBER using 107 real-world security patches and 8 Android kernel images from 3 different mainstream vendors, the results show that FIBER can achieve an average accuracy of 94% with no false positives.

1 Introduction

The number of newly found security vulnerabilities has been increasing rapidly in recent years [3], posing severe threats to various software and end users. The main approach used to combat vulnerabilities is patching; however, it is challenging to ensure that a security patch gets propagated to a large number of affected software distributions, in a timely manner, especially for large projects that have multiple concurrent development branches (*i.e.,* upstream versus downstream). This is due to the heavy code reuse in modern software engineering practice [16, 23, 20]. Thus, the capability to test whether a certain security patch is applied to a software distribution is crucial, for both defenders and attackers.

To better facilitate the discussion of the paper, we differentiate the goal and scope of *patch presence test* from those of the more general *bug search*. Patch presence test, as its name suggests, checks whether a specific patch has been applied to an unknown target, assuming the knowledge of the affected function(s) and the patch itself, *e.g.,* "whether the heartbleed vulnerability of an openssl library has been patched in the `tls1_process_heartbeat()` function". Bug search, on the other hand, does not make assumptions on which of the target functions are affected and simply look for all functions or code snippets that are similar to the vulnerable one, *e.g.,* "which of the functions in a software distribution looks like a vulnerable version of `tls1_process_heartbeat()`." Our study focuses on the more specific problem of *patch presence test*, which aims to offer a precise and accurate answer. With this in mind, both lines of work have been studied in the following contexts:

Source to source. This type of work operates purely on source code level. Source code is required for both the reference and target. In recent studies, it is also typically assumed that patches about specific bugs are available.

Binary to binary. These work do not need any source code. Both the reference and target are in binary, thus all comparisons are based on binary-level features only. It does not assume the availability of patch information (about which binary instructions are related to a patch).

In this paper, we consider a new category of **"source to binary"**, which is a middle ground between the above two, based on the following observations. First, open source has become a trend in computer world nowadays with an exploding number of software open sourced with full history of commits and patches (*e.g.,* hosted on github) [4]. In fact, most of the binary-only bug search studies include software such as Linux and

openssl. Second, many open-source code or components are widely reused in closed-source software, *e.g.,* libraries and Linux-based kernels in IoT firmware [13, 26]. This is a critical change that allows us to leverage the source-level insight that can inform the binary patch presence test.

Unfortunately, the closely related work on binary-only bug search misses an important link in order to be twisted to perform accurate patch presence test. Due to its extremely large scope, they are forced to use similarity-based fuzzy matching (inherently inaccurate) to speed up the search process, instead of the more expensive yet more accurate approaches. As a result, most of the existing solutions usually take the whole functions for comparison [26, 27, 13, 31]. However, since security patches are mostly small and subtle changes [30], similarity-based approaches cannot effectively distinguish patched and un-patched versions.

In this paper, we propose FIBER, a complementary system that completes the missing link and takes the similarity-based bug search to the next level where we can perform precise and accurate patch presence test. Fundamentally, FIBER addresses the following technical problem: "how do we generate binary signatures that well represent the source-level patch"? We address this problem in two steps: First, inspired by typical human analyst's behaviors, we will pick and choose the most suitable parts of a patch as candidates for binary signature generation. Second, we generate the binary signatures that preserve as much source-level information as possible, including the patch and the corresponding function as a whole.

We summarize our contributions as follows:

(1) We formulate the problem of patch presence test under "source to binary", bridging the gap from the general bug search to precise and accurate patch presence test. We then describe FIBER — an automatic, precise, and accurate system overcoming challenges such as information loss in the binaries. FIBER is open sourced[1].

(2) We design FIBER inspired by human behaviors, which picks and chooses the most suitable parts of a patch to generate binary signatures representative of the source-level patch. Besides, the test results can also be easily reasoned about by humans.

(3) We systematically evaluate FIBER with 107 real word vulnerabilities and security patches on a diverse set of Android kernel images [2] with different timestamps, versions and vendors, the results show that FIBER can achieve high accuracy in security patch presence test. We discover real-world cases where critical security patches fail to propagate to the downstreams.

[1] https://fiberx.github.io/

[2] Although Android follows open-source license, many Android device vendors still do not publish their source code or only do that periodically (with significant delays) for certain major releases.

2 Related Work

In this section, we discuss the related work primarily under bug search and how they are currently applied to the patch presence test problem. We divide them as source-level and binary-level.

Source-level bug search. Many studies focused on finding code clones both inside a single software distribution and across distributions [18, 22, 17, 16, 20]. The general goal is to find code snippets similar to a given buggy one — a more general goal that can be twisted to also conduct patch presence test. Since bug search typically does not limit the search scope to only a single function, it needs to face potentially millions of lines of code in large software [16]. Due to the scalability concern, bug search solutions are typically framed as some form of similarity matching using features extracted from the source code, including plain string [8], tokens [18, 22, 16, 20], and parse trees [17]. Unfortunately, this makes it challenging to ascertain whether the identified similar code snippets have been patched; this is because the patched and un-patched versions can be similar (especially for security patches that are often small) [16].

Binary-level bug search. Similar to the source-level work, binary-level approaches follow a similar principle of finding similar code snippets. To overcome the challenge of lack of source-level information, *e.g.,* variable type and name, these solutions need to look for alternative features such as structure of the code [19, 13, 31]. Since the "binary to binary" bug search does not assume the availability of symbol tables, they are forced to check out every single function in the target even if it only intends to conduct an accurate patch presence test on a specific function. For example, given a vulnerable function, Genius [13] and Gemini [31] are essentially looking for the same affected function(s) in the complete collection of functions in a target binary. Due to the scalability concern again, these features and solutions are engineered for speed instead of accuracy. BinDiff [2] and BinSlayer [9] check the control flow graph similarity based on isomorphism. As more advanced solutions, Genius [13] and Gemini [31] extract feature representations from the control flow graphs and encodes them into graph embeddings (high dimensional numerical vectors), which can speed up the matching process significantly. Unfortunately, under the huge search space, more accurate semantics-based solutions are not believed to be scalable [13, 31]. For instance, Pewny *et al.* [26] computes I/O pairs of basic blocks to

match similar basic blocks in a target function. BinHunt [14] and iBinHunt [24] use symbolic execution and theorem provers to formally verify basic block level semantic equivalence.

FIBER is in a unique position that leverages the source-level information to answer a more specific question — whether the specific affected function is patched in the target binary. To our knowledge, Pewny *et al.*'s work [26] is the only one that claims source-level patch information can be leveraged to generate more fine-grained signatures for bug search (although no implementation and evaluation). However, its goal is still focused on bug search instead of patch presence test, which means that it still attempts to search for similar (un-)patched code snippets (in binary) in the entire target, making it too fuzzy to answer the problem of patch presence test.

Finally, binary-level bug search has been extended to be cross-architecture [27, 26, 13, 31]. FIBER naturally supports different architectures with the assumption that source code is available, allowing us to generate different signatures for different compiled binaries.

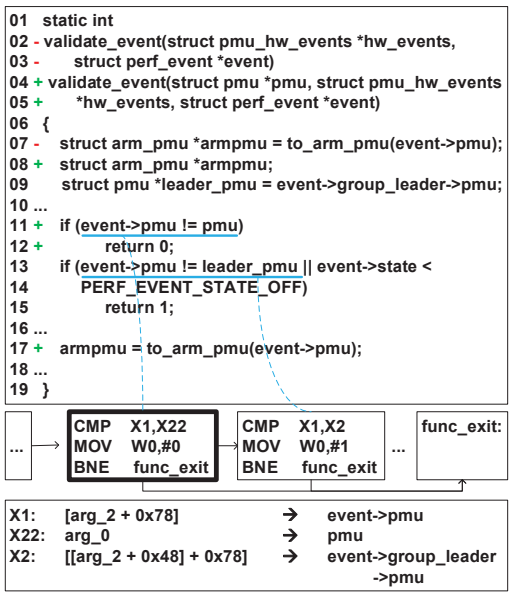

Figure 1: Patch of CVE-2015-8955

3 Overview

In this section, we first walk through a motivating example to summarize FIBER's general intuition, then position FIBER in a larger picture.

A motivating example. We pick the security patch for `CVE-2015-8955`, a Linux kernel vulnerability, to intuitively demonstrate a typical workflow of patch presence test which FIBER closely emulates. The patch is shown in Fig 1.[3] To test whether this patch exists in the target binary, naturally we will follow the steps below:

Step 1: Pick a change site (*i.e.,* sequence of changed statements). At first glance, we can see that the patch introduces multiple change sites. However, not all of them are ideal for the patch presence test purpose. Line 1-5 adds a new parameter "`pmu`" for original function, which will be used by the added "if" statement at line 11. Another change is to move the assignment of "`armpmu`" from line 7 to line 17. The "`to_arm_pmu()`" used by the assignment is a small utility macro, which will result in few instructions without changing the control flow graph (CFG), making it difficult to be located at binary level. However, the added "if" statement at line 11 will introduce a structural change to the CFG, besides, it also has a unique semantic as it involves the newly added function parameter. Therefore, it is natural to consider line 11 a more suitable indicator of patch presence.

Step 2: Rough matching. Now we have decided to search in the target binary function for the existence of line 11 in Fig 1, typically we will start from matching the CFG structure since it is easy and fast. This step can be similarly carried out in the source code level also. Specifically, one condition in the "if" statement will generally lead to a basic block with two successors, Thus for line 11, we will first try to locate those basic blocks with out-degrees of 2. Besides, one successor of the basic block should be the function epilogue since at line 12 the function will return if passing the checks at line 11. In Fig 1 we also show a part of the CFG generated from a patched Android kernel image, we can see that both the bolded basic block and the basic block right of it satisfy this requirement.

Step 3: Precise matching. Out of the two candidate basic blocks in the target binary, we now should need some semantic information to further distinguish them. Ideally, if we have the source level information such as variable names, a human can typically make a decision already (assuming the target function does not change variable names). With limited information at the binary level, we need to map the binary instructions to source-level statements somehow. This is usually a time-consuming process for human analysts, since they typically need to understand which register or memory location corresponds to which source-level variable.

[3] For simplicity, we include only one of the two changed functions in the patch and removed comments and context lines. The full patch can be found in [6].

Following the same example in Fig 1[4], an analyst needs to inspect the registers used in the "cmp" instruction of candidate blocks. Specifically, by tracking the register's origin (listed at the bottom of Fig 1), we can finally tell the differences of the two "cmp" instructions and correctly decide that the bolded basic block is the one that maps back to line 11.

System architecture. Fig 2 illustrates the system architecture, which is abstracted from human analysts' procedure. It has four primary inputs: (1) the source-level patch information; (2) the complete source code of a reference; (3) the affected function(s) in the compiled reference binary; (4) the affected functions in the target binary. It is obvious that (4) is readily available if the symbol table is included in the target binary (*e.g.,* true in most Linux-based kernel images). However, in the more general case we do not make this assumption, neither do the state-of-the-art binary-only bug search work [13, 31, 26]. Fortunately, these similarity-based approaches solve this very problem by identifying functions in the target binary that look similar to a reference one, thus the symbol table of the target binary can actually be inferred — in addition to research studies [13, 31], BinDiff [2] also has a built-in functionality serving this purpose. We leave the integration of such functionality into FIBER as future work, since all kernel images as test subjects in our evaluation have embedded symbol tables.

This shows that the similarity-based bug search and the more precise patch presence test are in fact not competing solutions; rather, they complement each other. The former is fast/scalable but less accurate; the latter is slower but more accurate. In a way, bug search acts as a coarse-grained filter and outputs a ranked list of candidate functions which can be used as input (4) of FIBER for further processing. Since the search space of FIBER is now constrained to only a few candidate functions (one if with symbol table), it opens up the more expensive analysis.

With the inputs in mind, we now describe the three major components in FIBER:

(1) Change site analyzer. A single patch may introduce more than one change site in different functions and one change site can also span over multiple lines in source code. Change site analyzer intends to pick out those most representative, unique and easy-to-match source changes by carefully analyzing each change site and the corresponding reference function(s), mimicking what a real analyst would do. Besides, during this process, we can also obtain useful source-level insight regarding the change sites (*e.g.,* the types of statements and the variables involved), which can guide the later signature generation and matching process.

[4] We use AArch64 assembly instructions in this example, if not explicitly stated, the same assembly instructions will also be used in all other examples across the paper.

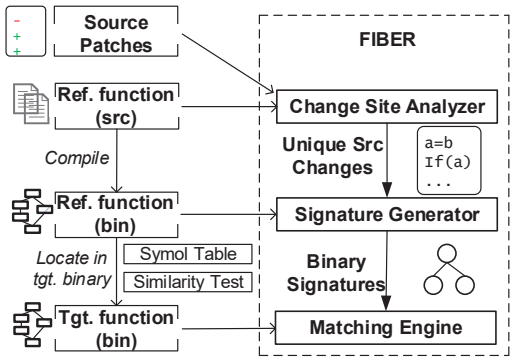

Figure 2: Workflow of FIBER

(2) Signature generator. This component is responsible for translating source-level change sites into binary-level signatures. Essentially this step requires an analysis to ensure that we can map binary instructions to source-level statements, which is challenging because of the information loss during the compilation process. The key building block we leverage is binary symbolic execution for this purpose.

(3) Matching engine. The matching engine's task is to search a given signature in the target binary. To do that, we first need to locate the affected function(s) in the target binary with the help of the symbol table. Then the search is done by first matching the syntax represented by the topology of a localized CFG related to the patch (a much quicker process), and then the semantic formulas (slower because of the symbolic execution). This process is similar to the one described in the motivating example.

It is worth noting that as long as a signature is generated for a particular security patch, it can then be saved and reused for multiple target binaries, thus we only need to run the analyzer and generator once for each patch.

Scope. (1) FIBER naturally supports analyzing binaries of different architecture and compiled with different compiler options. This is because of the availability of source code, which allows us to compile the source code into any supported architecture with any compiler options. More details will be discussed in §5 and §6.

(2) FIBER is inherently not tied to any source language although currently it works on C code. We do require debug information to be generated (for our reference binary) by compilers that can map the binary instructions back to source level statements as will be discussed in §4.3. All modern C compilers can do this for example.

Potential users and usage scenarios. We envision third-party auditors/developers will be FIBER's primary users, such as independent security researchers, security companies, software integration companies that rely on code/binaries supplied from others. Even for first-party developers, checking security patches at the binary level offers an extra layer of safety. As will be shown in §6.4, some vendors indeed forgot to patch critical vulnerabilities even though they have source access (*i.e.,* human errors), while systems like FIBER could have caught it.

4 System Design

In this section, we describe FIBER's design in depth by walking through the requirement of signatures and the design of each component.

4.1 Signature

The signature is what represents a patch. In general, we have two criterion for an "ideal" signature:

(1) Unique. The signature should not be found in places other than the patch itself. Otherwise, it is not unique to the patch. Specifically, it should not exist in both the patched and un-patched versions. This means that the signature should not be overly simple, which may cause it to appear in places unrelated to the patch.

(2) Stable. The signature should be robust to benign evolution of the code base, *e.g.,* the target function may look different than as the reference due to version differences. This means that the signature should not be overly complex (related to too many source lines), which is more likely to encounter benign changes in the target, creating false matches of the signature.

As we can see, the above two seemingly conflicting requirements ask for a delicate balance in signature generation, which we will elaborate in this section. Fundamentally, we need to pick a unique source change from a patch for which we believe a corresponding binary signature can be generated that well represents it. What works in our favor is that the reference and target function should share significant variable-level semantics. Assuming both versions are patched, things like "how a variable is derived and dereferenced" and "how a condition is derived" should be the very much the same. The binary signature simply need to carry this necessary information to recover the semantics present in the source.

Informally, we define a binary signature to be a group of instructions, that not only structurally correspond to the source-level signature, but also are annotated with sufficient information (*e.g.,* variable-level semantics) so that they can be unambiguously mapped to the original source-level change site. We will elaborate the translation process in §4.3.

4.2 Change Site Analyzer

The input of the change site analyzer is a source patch and the reference code base. It serves two purposes. (1) Since a patch may introduce multiple change sites within or across different functions, the analyzer aims to pick a suitable signature according to the criterion mentioned in §4.1. (2) Another goal is to gain insights of the patch change sites, from which the binary signature generator will benefit. We divide this process into two phases and detail them as below.

4.2.1 Unique Source Change Discovery

A patch can either add or delete some lines, thus we can either changes based on either the absence of patch (*i.e.,* existence of deleted lines) or presence of patch (*i.e.,* existence of added lines). For the purpose of discussion, we assume that our signature generation is based on the presence of patch and focused on the added lines; the opposite can be done similarly. The general strategy is to start from a single statement and gradually expand if necessary. For each added statement in the patch, the following steps will be performed:

(1) Uniqueness test. Basically, a statement has to exist in only the added lines of the patch and nowhere else (*e.g.,* un-patched code bases)". For this, we can apply a simple token-based sequence matching using a lexer [16]. We wish to point out that this uniqueness test captures not only token-based information but also semantic-related information. For instance, the example source signature in Fig 1 at line 11 encodes the fact that the first function parameter is compared against a field of the last parameter, and this semantic relationship is unique (which we need to preserve in binary signatures).

(2) (*optional*) Context addition. If no single statement is unique, we consider all its adjacent statements as potential context choices. The "adjacent" is bi-directional and on the control flow level (*e.g.,* the "if" statement has two successors and both of which can be considered the context), thus there can be multiple context statements. We gradually expand the context statements, *e.g.,* if one context statement is not enough, we try two.

(3) Fine-grained change detection. By convention, patches are distributed in the form of source line changes. Even when a line is partially modified, the corresponding patch will still show one deleted and one added line. We detect such fine-grained changes within a single statement / source line, by comparing it with its neighbouring deleted/added lines. This is to ensure that

we do not include unnecessary part of the statement which will bloat the signature. For example, if only one argument of a function call statement is changed, we can ignore all other arguments in the matching process to reduce potential noise, improving the "stability" of the signature.

(4) Type insight. The types of variables involved in source statements are also important since it will guide the later binary signature generation and matching. Theoretically, we can label the type of every variable in the reference binary (registers or memory locations in the binary) and make sure the types inferred in the target match (more details in §4.3.1). However, sometimes type match is not good enough to uniquely match a signature. A special case is a const string which is stored statically at a hardcoded memory address. If the only change in a patch is related to the content of the string, then both binary signature generation and matching should dereference the `char*` pointer and obtain the actual string content; otherwise, the signature will simply contain a const memory pointer whose value can vary across different binaries. Even if the pointer type matches as `char*` in the target, it is still inconclusive if it is a patched or un-patched version (we give some real examples in §6 as case studies).

After the above procedure, we now have some unique and small (thus more stable) source changes.

4.2.2 Source Change Selection

Previous step may generate multiple candidate unique source changes for a single patch. In practice, the presence of one of them may already indicate the patch presence. In addition, some source changes are more suitable for binary signature generation than others. In FIBER, we will first rank all candidate changes and pick the top N for further translation. The ranking is based on three factors (from least important to most):

(1) Distance to function entrance. Short distance between statements in the source-level signature and the function entrance will accelerate the signature generation process because of its design which we will detail in §4.3.

(2) Function size. If the source code signature is located in a smaller function, the matching engine will benefit since the search space will be reduced and it is less likely to encounter "noise". In addition, the matching speed will be faster. Note that this is more important than (1) because the signature generation process is only a one-time effort while matching may be repeated for different target binaries.

(3) Change type. The kinds of statements involved in a change matters. As shown previously in §3, if the change involves some structural/control-flow changes (*e.g.,* "if" statement), we can quickly narrow down the search range to structurally-similar candidates in the target binary, affecting the matching speed. More importantly, it can also affect the stability of the binary signature. Unlike statements such as a function call, which may get inlined depending on the compiler options, structural changes in general are much more robust.

We categorize the source changes into several general types: (1) function invocations (new function call or argument change to an existing call), (2) condition related (new conditional statement or condition change in an existing statement), (3) assignments (which may involve arithmetic operations). Actual source changes can have multiple types, *e.g.,* a function invocation can have an argument derived from an assignment or follow a conditional statement. Generally, we rank "new function call" (if FIBER determines that it is not inlined in the reference binary[5]) the highest because one can simply decide the patch presence by the presence of the function invocation, which is straightforward with the symbol table. We also rank "condition" related signatures (*e.g.,* "if" statement) high because it introduces both structural changes and semantic changes. On the other hand, a simple assignment statement, including assignment derived from arithmetic operations (*e.g.,* `a=b+c;`), will not affect the structure in general, so it is less preferred. Besides, pure control flow transfer (*e.g.,* addition of a "goto") is not preferred as well since we may need to include extra context statements that are unrelated to the change site, which is less stable.Note that there are certain source-level changes are simply not visible at the binary level (*e.g.,* source code comments) or difficult to locate (variable declaration).

4.3 Signature Generator

We first need to compile the reference source into the reference binary, from which the binary signatures will be generated according to the selected unique source change. As discussed in §4.2, we will still assume that the signature is based on the patched version. Also, during the compilation process, we will retain all the compiler-generated debug information for future use.

4.3.1 Binary Signature Generation

Identify and organize instructions related to the source change. Given the reference binary, the first thing is to locate the corresponding binary instructions related to the source change. This can be done with the

[5] It looks the presence of the corresponding binary instruction that calls to the exact function.

help of debug information since it provides a mapping from source code lines to binary instructions. We will then construct a local CFG that includes all the nodes containing the identified instructions, which is straightforward if these nodes are connected to each other, otherwise, we need to add some padding nodes to make a connected local CFG, which by nature is a steiner tree problem [15]. For this purpose we use the approximation steiner tree algorithm implemented in the NetworkX package [5]. The topology of such a local CFG reflects the structure of the original source change. Compared to full-function CFG, this local CFG structure is more robust to different compiler options and architectures since it excludes unrelated code. That being said, compilation configurations may still affect the signature. Therefore, ideally we should use the same compilation configuration of the reference kernel as the target. As will be described in §6.1, we follow a procedure to actively probe the compilation configuration of the target kernel.

Identify root instructions. Theoretically all these instructions identified in the local CFG above will be part of the binary signature. However, this is not a good idea in practice as only a subset of instructions actually summarizes the key behavior (data flow semantic); we refer to such instructions as "root instructions". The more instructions we include in a binary signature, the more specific and less "stable" it becomes. For instance, a compiler may insert additional "intermediate" instructions to free up some registers (by saving their values to memory). If we unnecessarily include all these instructions, we may not get a match in the target. Take the two source-level statements in Fig 3 as examples, the first statement is an assignment where 3 binary instructions are generated to perform the operation. However, capturing the last instruction alone is already sufficient, because we know through data flow analysis that `X1` is equal to `X0+0x4` and can therefore discard the first and second instruction. Similarly, instruction 03 and 04 corresponding to the second statement already sufficiently capture its semantic, because the outputs of instruction 00, 01 and 02 will later be consumed by other instructions.

Simply put, we define "root instructions" to be the last instructions in the data flow chains (where no other instructions will propagate any data further), along with some complementary instructions that complete the source-level semantic. For instance, by this definition, the `cmp` instruction will be the root instruction. However, we need to complement it with the next conditional jump instruction to complete its conditional statement semantic. For function call instructions, the root instructions will include the push (assuming x86) of arguments (as they each become the last instruction in a

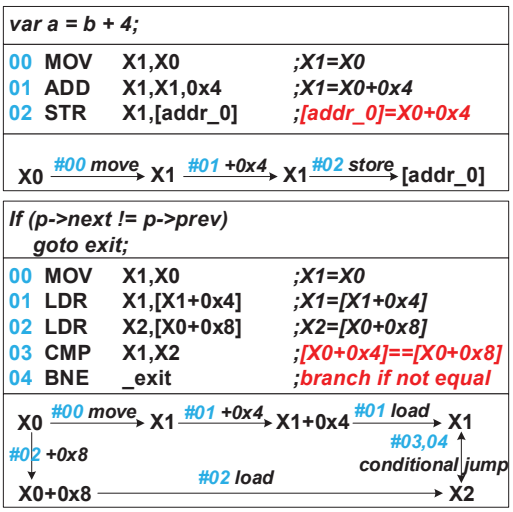

Figure 3: Data flow analysis of example basic blocks

Signature Type	Root Instructions (x86 example)
Function call	call,push
Conditional statement	cmp, conditional jmp
assignment (incl. arithmetic ops)	mov,add, sub,mul,bit ops...
Unconditional control transfer	jmp,ret

Table 1: Types of root instructions

data flow chain to prepare a specific argument), and the call instruction (to complete the function invocation semantic).

Note that compilers may still generate slightly different root instructions for the same statements (due to compiler optimizations, *etc.*). To facilitate signature matching, we deem root instructions equivalent as long as their types are the same (normalization of root instructions). We illustrate this in Table 1 where we show the different types of instructions that may be generated from the same source change. For instance, a compiler may choose to use bit operations instead of multiplications for an assignment statement.

Annotate root instructions. Now we need to make sure that the root instructions are sufficiently labeled (which is our binary signature) such that they can be uniquely mapped to source changes.

Following the observation mentioned earlier in §4.1 that the target and reference function should share variable-level semantics (as they are simply different versions of the same function), we formulate the goal as *mapping the operands (registers or memory locations) of the root instructions back to source-level variables*. This is sufficient because if the target function indeed

arg:	***function argument***
var:	***local variable***
ret:	***callee return value***
imm:	***immediate value***
[ ]:	***dereference***
op:	***binary operators***
expr:	***arg* | *var* | *ret* | *imm***
	| [*expr*] | *expr op expr*
	| if(*expr*) then *expr* else *expr*

Figure 4: Notation for formula (expression) annotating root instruction operands

applied the patch, the variables related to the patch should be the same ones as what we saw in the reference function. Now, our only job here is to ensure that the binary signature retains all such semantic information. To this end, we compute a full-function semantic formula for each operand (up to the point of root instructions). As shown in Fig 1, these formulas are in the form of ASTs – essentially formulated as expressions following the notation in Fig 4.

Note that from a function's perspective, any operand in an instruction can really be derived from only four sources:

(1) a function parameter (external input), *e.g.,* `ebp+0x4` if it is x86, `X0` or `X1` if it is aarch64;

(2) a local variable (defined within the function), *e.g.,* `ebp-0x8` in x86 or `sp+0x4` in aarch64 (which use registers to pass arguments);

(3) return values from function calls (external source), *e.g.,* a register holding the return value of a function call;

(4) an immediate number (constant), *e.g.,* instruction/data address (including global variables), offset, other constants;

These sources all have meaningful semantics at the source level. The question is how do we leverage them in the binary signature. Do we require the binary signature to state something precise "the fourth parameter of the function is used in a comparison statement", or something more fuzzy "a local variable is dereferenced at an offset, whose result is passed to a function call"? These choices all have implications on the unique and stable requirement of the signature. We discuss how we handle these four basic cases:

(1) Function parameter. From the calling convention, we can at least infer where memory location corresponds to which parameter. Despite the fact that function prototpye may change in the target, our current policy assumes otherwise (as the change happens rather infrequently). As an extension, we could use the type of the parameter (as mentioned in §4.2), or even its usage profile to ensure the uniqueness of the parameter. Note that this would also require analysis of the target function to derive similar information (which will require more expensive binary-level type inference techniques [21, 10]).

(2) Local variable. This is similar to the function parameter case, except that local variables are much more prone to change, *e.g.,* new variables may be introduced. In theory, we could similarly use type information and the way the local variable is used to ensure the uniqueness the variable in the signature. For now, we do not conduct any additional analysis and simply treats all local variables as the same class without further differentiation. Interestingly, we will show in §6 that this strategy already can generate signatures that are unique enough.

(3) Return values from function calls. This is a relatively straightforward case, we simply tag the return value to be originated from a specific function call.

(4) Immediate number. It is generally not safe to use the exact values of the immediate numbers, especially if it has to do with addresses. For instance, a `goto` instruction's target address may not be fixed in binaries. A field of a `struct` may be located at different offsets, *e.g.,* the target binary has a slightly different definition. We need to conduct additional binary-level analysis to infer if a target address is pointing to the right basic block (*e.g.,* by checking the similarity of the target basic block), or the offset is pointing to a specific field (*e.g.,* by type inference [21, 10]). Our current design allows for such extensions but at the moment simply treats immediate numbers as a class without differentiating their values, unless the values are related to source-level constants and unrelated to addresses, *e.g.,* `a = 0;`.

In our experience, we find that even without having a precise knowledge of these basic elements in the signature, the semantic formula that describe them is typically already unique enough to annotate the operands; ultimately allow us to uniquely map the root instructions to source-level statements. We show a concrete example in Fig 5 with both reference and target in comparison. As we can see, the patch line is in red: `a=n*m+2;`, a fairly straightforward assignment statement, which is used as a unique source change. In the binary form, we would identify the store instruction as root instruction, and annotate both operands accordingly. In this case, we know that `X3=X0*X1+0x2` which represents `arg_0*arg_1+0x2` and it is being stored into a local variable at `sp+0x8`. Similarly, the target source has the same patch statement (and should be considered patched) even though it has also inserted some additional code with a new local variable. When we attempt to match the binary signature, there are three points worth noting:

First, the local variable `a` is now located at a different offset from `sp`, *i.e.,* `sp+0x10`. We therefore cannot

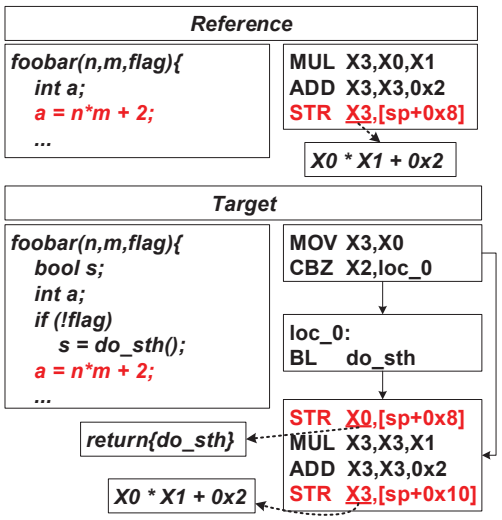

Figure 5: Illustration of the binary signature matching

blindly use a fixed offset to represent the same local variable across reference and target. Instead, we could apply the additional strategies mentioned above: (1) Inferring the type of local variables in the target binary and conclude that `sp+0x10` is the only integer variable and therefore must correspond to `sp+0x8`. (2) Profiling the behaviors of all local variables in the target binary and attempt to match the one most similar to `sp+0x8` in the reference. For example, we know `sp+0x8` in the binary (*i.e.,* `s`) takes the value from a function return, while `sp+0x10` (*i.e.,* `a`) did not (and `sp+0x10` is the more likely one). Interestingly, even if we do not perform the above analysis, the fact that there is a root instruction storing a unique formula `X0*X1+0x2` to a local variable (any) is already unique enough to be a signature that lead to a correct match in the target.

Second, to show that isolated basic block level analysis is not sufficient, we note the `mov` instruction in the first basic block of the target binary which saves X0 to X3 to free up X0 for the return value of `do_sth()`. It is imperative that we link X3 to X0 so that the final formula at the root instruction (*i.e.,* last instruction of the last basic block) will be the same as the one computed in the reference binary.

Third, there is an additional store instruction in the last basic block of the target binary, which saves X0 (return value of `do_sth()` to `sp+0x8` (*i.e.,* `s`). Note that this may look like a root instruction as well from data flow perspective. However, since it is attempting to store a return value instead of the formula in the original signature, it will not cause a false match.

4.3.2 Binary Signature Validation

Even though we have the best intention to preserve the uniqueness and stability of the selected source change, due to the information loss incurred in the translation, we still need to double check that the candidate binary-level signatures actually still satisfy the requirements.

(1) Unique. For each patch, we will prepare both the patched and un-patched binaries as references and then try to match the binary signature against them, with the matching engine (detailed in §4.4). For a binary signature based on the patched code, it will be regarded as unique only when it has no match in the reference un-patched binary. A unique binary signature may still have multiple matches (although rare) in the reference patched binary, in this case, we will record the match count as auxiliary information. When using it to test the target binary in real world, only when the match count is no less than previously recorded one, will we say that the patch exists in target binary.

(2) Stable. Our previous effort in §4.2 to keep a small footprint of the unique source change can also help to improve the binary signature stability here, since the sizes of source change and binary signatures are related. Besides, we can also prepare multiple versions of patched and un-patched function binaries (if more ground truth data are available) and test the generated binary signature against them. This can help to pick out those most stable binary signatures that exist in all patched binaries but none of un-patched binaries.

4.4 Signature Matching Engine

Matching engine is responsible for searching a given binary signature in the target binary (*i.e.,* the test subject). This section will detail the searching process. As briefly mentioned in §3, we first need to locate the target function in the target binary by its symbol table, then we will start to search the binary signature in it. We divide the search into two phases: rough matching and precise matching.

Rough matching. This is a quick pass that intends to match the binary signature by some easy-to-collect features. These features include:

(1) CFG topology. The binary signature itself is basically a subgraph of the function CFG. This step is useful unless the binary signature resides in only a single basic block (*e.g.,* the signature for an assignment statement).

(2) Exit of basic blocks. In general each basic block has one of two exit types: unconditional jump and conditional jump, the former can be further classified into call, return, and other normal control flow transfer for most ISAs. Thus, basic blocks can be quickly compared by their exit types.

(3) Root instruction types. As described in §4.3.1, we will analyze each basic block in the signature and decide its root instruction set. The instruction types can then be used to quickly compare two basic blocks. This requires generating the data flow graph for each basic block in target function binary, which is more expensive than previous steps but still manageable.

With above features, we can quickly narrow down the search space in the target function. If no matches can be found in this step, we can already conclude that the signature does not exist, otherwise, we still need to precisely compare every candidate match further.

Precise matching. In this phase, we leverage the annotation produced in §4.3.1 to perform a precise match on two groups of root instructions. We essentially just need to compare their associated annotation (*i.e.,* semantic formulas).

To fulfill the semantic comparison, we first need to generate semantic formulas for all the matched candidate root instructions, which can be done in the same way as detailed in §4.3.1. If all formulas of the signature root instructions can also be found in the candidate root instructions, the two will be regarded as equivalent (*i.e.,* they map to the same source-level signature/statements).

To compare two formulas (essentially two ASTs), there have been prior solutions that calculate a similarity score based on tree edit distance [12, 27]; however, FIBER intends to give a definitive answer about the match result, instead of a similarity score. Alternatively, theorem prover has been applied to prove the semantic equivalence of two formulas [14], which definitely provides the best accuracy but unfortunately can be very expensive in practice. In this paper, we choose a middle ground. Based on the observations that semantic formulas capture the dependency and therefore the order of instructions cannot be swapped, we know that the structure of formulas is unlikely to change (our evaluation confirms this), *e.g.,* `(a+b)*2` will not become `a*2+b*2`. In addition, with normalization of the basic elements of the formula, the matching process is also robust to non-structural changes. Basically, the matching process simply recursively match the operations and operands in the AST, with some necessary relaxations (*e.g.,* if the operator is commutative, the order of the operands will not matter). We also simplify the AST with a Z3 solver [11] before comparison.

5 Implementation

We implement the prototype of FIBER with 5,097 LOC in Python on top of Angr [29], as it has a robust symbolic execution engine to generate semantic formulas. To suit our needs, we also changed the internals of Angr (including 1348 LOC addition and 89 LOC deletion). Below are some implementation details.

Architectural dependencies. As mentioned, FIBER in principle supports any architecture as we can compile the source code into binaries for any architecture. Further, since we use Angr which lifts the binaries into an intermediate language VEX (which abstracts away instruction set architecture differences), most of our system works flawlessly without the need of tailoring for architectural specifics. This not only allows FIBER to be (for the most part) architectural independent, but also facilitates the implementation. For instance, when searching for root instructions, the data flow analysis is performed on top of VEX. However, some small engineering efforts are still needed for multi-architectural support, such as to deal with different calling conventions. At current stage FIBER supports aarch64.

Root instruction annotation. To generate semantic formulas for root instruction operands, it is necessary to analyze all the binary code from the function entrance to the root instruction. We choose symbolic execution as our analyze method since it can cover all possible execution paths and obtain the value expression of any register and memory location at an arbitrary point along the path.

Symbolic execution is well known for the path explosion problem, which makes it expensive and not as practical. We employ multiple optimizations to address the performance issue as detailed below.

(1) Path pruning. Before starting the symbolic execution we will first perform a depth first search (DFS) in the function CFG to find all paths from the function entrance to the root instructions. We will then put only the basic blocks contained in these paths in the execution whitelist, all other basic blocks will be dropped by the symbolic execution engine. Besides, we also limit the loop unrolling times to 2 to further reduce the number of paths.

(2) Under-constrained symbolic execution. As proposed previously [28], under-constrained symbolic execution can process an individual function without worrying about its calling contexts, effectively confining the path space within the single function. Although the input to the function (*e.g.,* parameters) is un-constrained at the beginning, it will not affect the extraction of the semantic formulas since they do not need such initial constraints. Un-constrained inputs may also lead the execution engine to include infeasible paths in real world execution, however, our goal for semantic formulas is to make them comparable between reference and target binaries, as long as we use the same procedure for both sides, the extracted formulas can still

be compared for the purpose of patch presence test. In the end, we use intra-function symbolic execution, *i.e.,* without following the callees (their return values will be made un-constrained as well), which in practice can already generate the formulas that make root instructions unique and stable.

(3) Symbolic execution in veritesting mode. Veritesting [7] is a technique that integrates static symbolic execution into dynamic symbolic execution to improve its efficiency. Dynamic symbolic execution is path-based, a same basic block belonging to multiple paths will be executed for multiple times, greatly increasing the overhead especially when there is a large number of paths. Static symbolic execution executes each basic block only once, but its formulas will be more complicated since it needs to carry different constraints of all paths that can reach current node. However, FIBER does not need to actually solve the formulas, instead, it only needs to compare these formulas extracted from reference and target binaries, thus, the formula complexity matters less for us. Note that this means an operand may sometimes have more than one formulas: consider when the true and false branch of a `if` statement merges. When we regard a binary signature as matched in the target, we require that the computed formulas in the target contain all of the formulas in the signature (could be a superset). If at least one formula is missing, we consider the corresponding source code in the target to have missed certain important code that contributes to the signature.

6 Evaluation

In this section, we systematically evaluate FIBER for its effectiveness and efficiency.

Dataset. We choose Android kernels as our evaluation dataset. This is because Android is not only popular but also fragmented with many development branches maintained by different vendors such as Samsung and Huawei [25]. Although Google has open-sourced its Android kernels and maintained a frequently-undated security bulletin [1], other Android vendors may not port the security patches to their own kernels timely. Besides, even though required by open source license, many vendors choose not to open source their kernels or make it extremely inconvenient (with substantial delays and only periodic releases). This makes Android kernels an ideal target. We collect two kinds of dataset specifically:

(1) Reference kernel source code and security patches. We choose the open-source "angler" Android kernel (v3.10) used by Google's Nexus 6P as our reference. We then crawl the Android security bulletin from June 2016 to May 2017 and collect all published vulnerabilities related security patches[6] for which we can locate the affected function(s) in the reference kernel image (*e.g.,* it may use a different driver than the one gets patched, or the affected function itself may be inlined). We also exclude one special patch that changes only a variable type in its declaration, requiring type inference at the binary level to handle, which we don't support currently as mentioned in §4.2.2. In total we collected 107 security patches that are applicable to our reference kernel.

(2) Target Android kernel images and source code. Besides the reference kernel, we also collect 8 Android kernel images from 3 different mainstream vendors with different timestamps and versions as listed in table 2. Note that vendors publish way more binary images (sometimes once every month) than the source code packages. We only evaluate the binary images for which we can find the corresponding source code, which serves only as ground truth of the patch presence test.

All our evaluations are performed on a server with Intel Xeon E5-2640 v2 CPU and 64 GB memory.

6.1 Experiment Procedure

To test patch presence in the target binary, we follow the steps below:

Reference binary preparation. As shown in Fig 2, we first need to compile the reference source code to binary, based on which we will generate the binary signatures. The availability of source code enables us to freely choose compilers, their options, and the target architecture. Naturally, we should choose the compilation configuration that is closest to the one used for target binary, which can maximize the accuracy. To probe the compilation configuration used for the target binary, we first compile multiple reference binaries with all combinations of common compilers (we use gcc and clang) and optimization levels (we use levels O1 - O3 and Os[7]), then use BinDiff [2] to test the similarity of each reference binary and the target binary, the most similar reference binary will finally be used for binary signature generation. Following this procedure (which is yet to be automated), we observed in our evaluation that kernel 6 and 7 as shown in table 2 use gcc with O2 optimization level, while all other 6 kernels use gcc with Os optimization level, which is confirmed by our inspection of the source code compilation configurations (*e.g.,* Makefile).

Offline signature generation and validation. For each security patch, we retain at most three binary

[6] Some security patches are not made publicly available on the Android Security Bulletin.

[7] Optimize for size.

Device	No.	Patch Cnt*	Build Date (mm/dd/yy)	Kernel Version	Accuracy				Online Matching Time (s)			
					TP	TN	FP	FN	Total	Avg	∼70%	Max.
Samsung S7	0	102	06/24/16	3.18.20	42	56	0	4(3.92%)	1690.43	16.57	8.47	306.47
	1	102	09/09/16	3.18.20	43	55	0	4(3.92%)	1888.06	18.51	8.24	438.76
	2	102	01/03/17	3.18.31	85	11	0	6(5.88%)	2421.44	23.74	5.49	1047.10
	3	102	05/18/17	3.18.31	92	4	0	6(5.88%)	1770.66	17.36	5.33	386.94
LG G5	4	103	05/27/16	3.18.20	32	65	0	6(5.88%)	2122.37	20.61	8.90	648.93
	5	103	10/26/17	3.18.31	95	0	0	8(7.77%)	1384.47	13.44	4.76	229.46
Huawei P9	6	31	02/22/16	3.10.90	10	20	0	1(3.23%)	390.35	12.59	8.47	89.35
	7	30	05/22/17	4.1.18	25	2	0	3(10.00%)	515.64	17.19	7.4	279.49

* Some patches we collected are not applicable for certain test subject kernels.

Table 2: Binary Patch Presence Test: Accuracy and Online Matching Performance

signatures, after testing their uniqueness by matching them against both patched and un-patched reference kernel images. If nothing is unique, we will add more contexts to existing non-unique signatures.

Online matching. Given a specific security patch, we will try to match all its binary signatures in the target kernels. Note that all Android kernel images are compiled with symbol tables. We therefore can easily locate the affected functions. As long as one signature can be matched with a match count no less than that in reference patched kernel, we will say the patch exists in the target. As a performance optimization, we will first match the "fastest-to-match" signature.

6.2 Accuracy

We list the patch presence test results for target Android kernel images in table 2. It is worth noting that our patch collection is oriented to "angler" kernel, which will run on the Qualcomm hardware platform, while kernel 6 and 7 intend to run on a different platform (*i.e.,* Kirin), thus many device driver related patches do not apply for kernel 6 and 7 (we cannot even locate the same affected functions).

Overall, our accuracy is excellent. There are no false positives (FP) across the board and very few false negatives (FN). In patch presence test, we assume that all patches are not applied by default. It has to be proven otherwise. In practice, FP may lead developers to wrongly believe that a patch has been applied while in reality not (a serious security risk). In contrast, FN only costs some extra time for analysts to realize that the code is actually patched (or perhaps unaffected due to other reasons) while we say it is not. Thus, we believe FN is more tolerable than FP. Since we have no FP, we manually inspect each FN case to analyze the root causes:

(1) Function inline. Function inline behaviors may vary across different compilers and binaries. A same function may be inlined in some binaries but not others, or inlined in different ways. Some of our signatures (*e.g.,* the signature for `CVE-2016-8463`) model inline function calls based on the reference kernel image, if the target kernel has a different inline behavior, our signatures will fail to match. To address this problem, we need to generate binary signatures based on a collection of different kernel images to anticipate such behaviors.

(2) Function prototype change. Although rare, sometimes the function prototype will change across different kernel images. Specifically, the number and order of the function parameters may vary. As discussed in §4.3.1, we will differentiate the parameter order, thus, if a same parameter has different orders in reference and target kernels, the match will fail. We have one such case (`CVE-2014-9893`) in the evaluation. To solve this problem, we can extend our current implementation with techniques such as parameter profiling (see §4.3.1).

(3) Code customization. As discussed in §4.2, extra contexts are necessary if original patch change site is not unique. However, the contexts may be different across various kernel images due to code customization, although the patch change site remains the same. If this happens, our signature (with contexts extracted from the reference kernel) will not match, although the target kernel image has been patched. We encountered such a case in Samsung kernels for `CVE-2015-8942`. Such customizations are generally hard to anticipate and it will likely still cause a FN even if the source code of the target is given. This is why we prefer not to add contexts. If we can use more fine-grained binary analysis such as parameter and local variable profiling, we may be able to avoid using contexts.

(4) Patch adaptation. A patch may need to be adapted for kernels maintained by different vendors since the vulnerable functions are not always exactly the same across different kernel branches. Adaptation can also happen when a patch is back-ported to an older kernel version. In our evaluation, we find that this happens in some target images for `CVE-2016-5696`. Strictly

Step	Total	Cnt. **	Avg.	∼70%
Analyze	21.52s	107	0.20s	-
Translation	1608.52s	293	5.49s	6.29s
Match Ref.0 *	2647.78s	293	9.04s	6.00s
Match Ref.1 *	3415.54s	293	11.66s	7.56s

* Match against reference kernels for uniqueness test.
* 0 for un-patched kernel, 1 for patched kernel.
** Analyze: Patch. Others: Binary Signature.

Table 3: Offline Phase Performance

speaking, FIBER intends to detect exactly the same patch as appeared in the reference kernel, however, to be conservative, we still regard such cases as false negatives.

(5) Other engineering issues. Some FN cases are caused by engineering issues. For example, certain binary instructions cannot be recognized and decoded by the frontend of angr (two cases in total), which will affect the subsequent CFG generation and symbolic execution.

6.3 Performance

In this section we evaluate FIBER's runtime performance for both offline signature generation and online matching. We list the time consumption of the offline phase in table 3 and that of online phase in table 2. From the tables, we can see that a small fraction of patches needs much longer time to be matched than average, this is usually because the change sites in these patches are positioned in very large and complex functions (*e.g.,* `CVE-2017-0521`), thus the matching engine may encounter root instructions deep inside the function. However, most patches can be analyzed, translated and matched in a reasonable time. In the end, we argue that a human will take likely minutes, if not longer, to verify a patch anyways. An automated and accurate solution like ours is still preferable, not to mention that we can parallelize the analysis of different patches.

6.4 Unported Patches

As shown in table 2, for all the test subjects except kernel #5, FIBER produces some TN cases, which suggests un-patched vulnerabilities. If related security patches had already been available before the test subject's release date, then it means that the test subject fails to apply the patch timely. Table 4 lists all the vulnerabilities whose patches fail to be propagated to one or multiple test subject kernel(s) timely in our evaluation. Note that for security concerns, we do not correlate these vulnerabilities with actual kernels in table 2.

CVE	Patch Date * (mm/yy)	Type**	Severity*
CVE-2014-9781	07/16	P	High
CVE-2016-2502	07/16	P	High
CVE-2016-3813	07/16	I	Moderate
CVE-2016-4578	08/16	I	Moderate
CVE-2016-2184	11/16	P	Critical
CVE-2016-7910	11/16	P	Critical
CVE-2016-8413	03/17	I	Moderate
CVE-2016-10200	03/17	P	Critical
CVE-2016-10229	04/17	E	Critical

* Obtained from Android security bulletin.
** **P:** Privilege Elevation **E:** Remote Code Execution
** **I:** Information Disclosure

Table 4: Potential Security Loopholes

From table 4, we can see that even some critical vulnerabilities were not patched in time, indicating a good potential that they can be leveraged to compromise the kernel entirely to execute arbitrary code. One such case is a patch delayed for more than half a year affecting a major vendor (who confirmed the case and requested to be anonymized). This illustrates the value of tools like FIBER.

Besides, we also identify 4 vulnerabilities in table 4 that eventually got patched in a later kernel release but not in the earliest kernel release after the patch release date, indicating a significant delay of the patch propagation process.

It is worth noting that FIBER intends to test whether the patch exists in the target kernel, however, the absence of a security patch does not necessarily mean that the target kernel is exploitable. So the further verification is still needed.

6.5 Case Study

In this section, we demonstrate some representative security patches used in our evaluation to show the strength of FIBER compared to other solutions.

Format String Change. There are 5 patches in our collection that intend to change only the format strings as function arguments. Take the patch for `CVE-2016-6752` in Fig 6 as an example, the specifier `p` is changed to `pK`. It will be impossible to detect it at binary level without dereferencing the string pointer, since all other features (*e.g.,.* topology, instruction type.) remain exactly the same. However, without patch insights, it is extremely difficult to decide which register or memory location should be regarded as a pointer and whether it should be dereferenced in the matching

```
CVE-2016-6752
-       pr_debug("UNLOAD_APP: qseecom_addr = 0x%p\n", data);
+       pr_debug("UNLOAD_APP: qseecom_addr = 0x%pK\n", data);

CVE-2016-3858
-       strlcpy(subsys->desc->fw_name, buf, count + 1);
+       strlcpy(subsys->desc->fw_name, buf,
+                       min(count + 1, sizeof(subsys->desc->fw_name)));

CVE-2014-9785
-       if (__copy_from_user(&load_img_req,
+       if (copy_from_user(&load_img_req,

CVE-2016-8417
-       if (hw_cmd_p->offset > max_size) {
+       if (hw_cmd_p->offset >= max_size) {

CVE-2015-8944
-       proc_create("iomem", 0, NULL, &proc_iomem_operations);
+       proc_create("iomem", S_IRUSR, NULL,
+                       &proc_iomem_operations);
```

Figure 6: Example Security Patches

process, rendering all binary-only solutions ineffective in this case. While FIBER can correctly decide that the only thing changed is the argument format string (see §4.2) and then test patch presence by matching the string content.

Small Change Site. It is very common that a security patch will only introduce small and subtle changes, such as the one for `CVE-2016-8417` shown in Fig 6, where the operator ">" is replaced with ">=". Such a change has no impact on the CFG topology and only one conditional jump instruction will be slightly different. Thus, it will be extremely difficult to differentiate the patched and un-patched functions without the fine-grained signature. FIBER handles this case correctly because the conditional jump is part of the root instruction and we will check the comparison operator associated with it.

Patch Backport. A downstream kernel may selectively apply patches (security or other bug fixes), which can cause functions to look different from upstream. Our reference kernel (v3.10) is actually a downstream compared to all test subjects except #6 as shown in table 2. The patch for `CVE-2016-3858` (shown in Fig 6) has a prior patch in the upstream (which deletes a "if-then-return" statement) for the same affected function, which was not applied to our reference kernel, making the two functions look different although both patched. FIBER is robust to such backporting cases because the generated binary signature is fine-grained and related to only a single patch.

Multiple Patched Function Versions. After a security patch is applied, the same function may be modified by future patches as well. Thus, similar to the backporting cases, two patched functions can still be different because they are on different versions. `CVE-2014-9785` is such an example. FIBER can still precisely locate the same change site as shown in Fig 6 even when faced with a much newer target function, which differs significantly with the reference function.

Constant Change. Patch for `CVE-2015-8944` in Fig 6 only changes a function argument from 0 to a pre-defined constant `S_IRUSR` (0x100 in reference kernel). Once again, such a small change makes the patched and un-patched functions highly similar. Even though a solution wants to strictly differentiate constant values, it is in general unsafe because the constants are prone to change across binaries. However, with the insights of the fine-grained change site, FIBER can correctly figure out that only the value of the 2nd function argument matters in the matching and it should be non-zero if patched, thus effectively handle such cases.

Similar Basic Blocks. FIBER generates fine-grained signatures containing only a limited set of basic blocks (see §4.3.1). It is likely that there will be other similar basic blocks as the signature if we only look at the basic block level semantics. One such example has been shown in Fig 1 and discussed in §3. Previous work based on basic block level semantics [27, 26] may fail to handle such cases, While FIBER tries to integrate function level semantics into the local CFG, resulting in fine-grained signatures that are both stable and unique.

7 Conclusion

In this paper, we formulate a new problem of patch presence test under "source to binary" scenario. We then design and implement FIBER, a fully automatic solution which can take the best advantage of source level information for accurate and precise patch presence test in binaries. FIBER has been systematically evaluated with real-world security patches and a diverse set of Android kernel images, the results show that it can achieve an excellent accuracy with acceptable performance, thus highly practical for security analysts.

Acknowledgement

We wish to thank Michael Bailey (our shepherd) and the anonymous reviewers for their valuable comments and suggestions. Many thanks to Prof. Heng Yin and Prof. Chengyu Song for their insightful discussions. This work was supported by the National Science Foundation under Grant No.1617573.

References

[1] Android Security Bulletin. https://source.android.com/security/bulletin/.

[2] BinDiff. https://www.zynamics.com/bindiff.html.

[3] CVE: Vulnerabilities By Year. https://www.cvedetails.com/browse-by-date.php.

[4] Github Annual Report. https://octoverse.github.com/.

[5] NetworkX Python Package. https://networkx.github.io/.

[6] Security Patch for CVE-2015-8955. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/commit/?id=8fff105e13041e49b82f92eef034f363a6b1c071.

[7] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. ICSE'14.

[8] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, October 1997.

[9] M. Bourquin, A. King, and E. Robbins. Binslayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*.

[10] W. Cui, M. Peinado, Z. Xu, and E. Chan. Tracking rootkit footprints with a practical memory analysis system. USENIX Security'12.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[12] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin. Extracting conditional formulas for cross-platform bug search. ASIACCS'17.

[13] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin. Scalable graph-based bug search for firmware images. CCS '16.

[14] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, 2008.

[15] F. K. Hwang, D. S. Richards, and P. Winter. *The Steiner tree problem*, volume 53. Elsevier, 1992.

[16] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. Oakland'12.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. ICSE'07.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[19] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[20] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. Oakland'17.

[21] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. NDSS'11.

[22] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[23] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. ACSAC'16.

[24] J. Ming, M. Pan, and D. Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology*.

[25] OpenSignal. Android Fragmentation Visualized. https://opensignal.com/reports/2015/08/android-fragmentation/.

[26] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. Oakland'15.

[27] J. Pewny, F. Schuster, C. Rossow, L. Bernhard, and T. Holz. Leveraging semantic signatures for bug search in binary programs. ACSAC'14.

[28] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. USENIX Security'15.

[29] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. Oakland'16.

[30] Y. Tian, J. Lawall, and D. Lo. Identifying Linux bug fixing patches. ICSE'12.

[31] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. CCS '17.