PDiff: Semantic-based Patch Presence Testing for Downstream Kernels

Zheyue Jiang*
Fudan University
zyjiang18@fudan.edu.cn

Yuan Zhang*
Fudan University
yuanxzhang@fudan.edu.cn

Jun Xu
Stevens Institute of Technology
jxu69@stevens.edu

Qi Wen
Fudan University
16212010024@fudan.edu.cn

Zhenghe Wang
Fudan University
17212010075@fudan.edu.cn

Xiaohan Zhang
Fudan University
xh_zhang@fudan.edu.cn

Xinyu Xing
Pennsylvania State University
xxing@ist.psu.edu

Min Yang
Fudan University
m_yang@fudan.edu.cn

Zhemin Yang
Fudan University
yangzhemin@fudan.edu.cn

ABSTRACT

Open-source kernels have been adopted by massive downstream vendors on billions of devices. However, these vendors often omit or delay the adoption of patches released in the mainstream version. Even worse, many vendors are not publicizing the patching progress or even disclosing misleading information. However, patching status is critical for groups (*e.g.*, governments and enterprise users) that are keen to security threats. Such a practice motivates the need for reliable patch presence testing for downstream kernels. Currently, the best means of patch presence testing is to examine the existence of a patch in the target kernel by using the code signature match. However, such an approach cannot address the key challenges in practice. Specifically, downstream vendors widely customize the mainstream code and use non-standard building configurations, which often change the code around the patching sites such that the code signatures are ineffective.

In this work, we propose PDiff, a system to perform highly reliable patch presence testing with downstream kernel images. Technically speaking, PDiff generates summaries carrying the semantics related to a target patch. Based on the semantic summaries, PDiff compares the target kernel with its mainstream version before and after the adoption of the patch, preferring the closer reference version to determine the patching status. Unlike previous research on patch presence testing, our approach examines similarity based on the semantics of patches and therefore, provides high tolerance to code-level variations. Our test with 398 kernel images corresponding to 51 patches shows that PDiff can achieve high accuracy with an extremely low rate of false negatives and zero false positives. This significantly outperforms the state-of-the-art tool. More importantly, PDiff demonstrates consistently high effectiveness when code customization and non-standard building configurations occur.

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; **Vulnerability management**.

KEYWORDS

Patch Presence Test, Patch Semantics, Linux Kernel Security

ACM Reference Format:
Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3372297.3417240

1 INTRODUCTION

Sitting in the core of the Operating System (OS), the kernel is the most important piece of software in various types of computing devices. However, the development of kernels has been a major challenge for tremendous device vendors, as these vendors either cannot afford the cost of handling the extreme complexities or lack the expertise in achieving the high requirement of efficiency, reliability, and security. Open-source kernel projects significantly alleviate this situation. In particular, the Linux kernel has been adopted by hundreds of downstream vendors on millions of devices [13, 44] since its first release in 1991. More remarkably, variants of the (Linux-based) Android kernel are running on over 2.3 billion of smartphones and IoT devices [57].

Despite the large group of downstream kernels boost the diversity of devices and extend all-sided functionalities, they do not provide the same level of security as the mainstream version. A major reason is that the downstream vendors often fail to timely adopt the released patches [2, 20, 21, 29, 38, 49]. The delay of patching can range from months to years, exposing a significant attack surface [23, 45]. To mitigate this type of threat, groups that have high demands of security, such as government

*co-first authors

agents, enterprise users, and security service providers, often take proactive actions. For instance, DARPA recently started the AMP program [10] to identify and remediate un-adopted patches, and various anti-virus vendors seek un-patched vulnerabilities and develop corresponding exploit protections [8, 39]. In those actions, the first and indispensable step is to understand the presence/absence of patches in the target kernels.

By intuition, a straightforward idea of understanding patch presence is to acquire related information from the vendors. However, as unveiled by our study presented in § 2.1, this is often infeasible or unreliable. On the one hand, many downstream vendors are not publicizing their patching progress. On the other hand, presumably due to high complexities in patch management, downstream vendors (even large ones like Google and Huawei) can unintentionally disclose misleading patch information. These practices indicate an urgent need of alternative approaches for patch presence testing with downstream kernels.

Technically speaking, there have been two approaches to perform patch presence testing with kernels – ❶ penetration testing by developing a proof-of-concept (PoC) program from one particular version of the kernel, running it against the target versions, and examining whether it triggers the corresponding vulnerability; ❷ deriving a signature from the mainstream version (with patch applied) and then searching that signature in the target kernels (e.g., FIBER [61]). In practice, both approaches, however, are insufficient for our problem domain. As we will shortly explain in § 2.3, downstream vendors prevalently customize the mainstream kernel and use non-standard building configurations. These factors greatly contribute to the variation in code layout, making the above two approaches ineffective. For the first approach, the PoC programs are developed to be working only for a specific kernel. When applied to another version of the kernel with code changes, the PoC programs often fail to create the contexts of triggering the vulnerabilities. For the second approach, kernel customization and non-standard building configurations can frequently change the code around the patching site. Therefore, the signature derived from the mainstream kernel cannot remain in the target kernels. Take the state-of-the-art patch presence testing system FIBER [61] for example. In cases where the target kernels are customized or built with non-standard configurations, FIBER demonstrates a significant decrease in accuracy (§ 5). In short, patch presence testing with downstream kernels remains an open problem under practical settings.

To address the problem above, we propose PDiff, a system to facilitate patch presence testing with downstream kernels. At the high level, PDiff generates summaries carrying the semantics of a corresponding patch. Then, it utilizes the summaries to perform patch presence testing. The intuition behind the design of PDiff is that the target kernel and its reference version should have similar semantics, regardless of the variations at the code level. Technique wise, PDiff proceeds patch presence testing by following three steps. Given the pre-patch code and post-patch code of the reference versions, it first slices the paths that are affected by a patch. Second, it generates semantic digests from those paths, using formulas constructed with symbolized values as representation. Finally, patch summaries are synthesized by combing the path digests from both the pre-patch and post-patch reference versions. In the course of patch presence testing, PDiff measures the distance between the patch summaries in the target kernel and those in the pre-patch and post-patch reference versions, preferring the closer reference version to determine the patching status.

Admittedly, this is not the first work that performs patch presence testing with kernels. To the best of our knowledge, PDiff however is the first work that considers code similarities at the semantic level for patch presence testing. As such, it has a high tolerance to noises introduced by code variance, capable of handling complex scenarios, in particular, the cases where non-standard building or code customization occurs. Moreover, PDiff has a minimal set of assumptions. PDiff only requires the pre-patch and post-patch versions of the mainstream kernel, and it can seamlessly work with arbitrary binary-only downstream versions. This makes PDiff significantly more practical than the existing techniques.

We have implemented a prototype of PDiff for Linux kernels on AArch64 and ARM32. To evaluate the utility of PDiff, we gather a group of 398 real-world kernel images corresponding to 51 released patches. In particular, these test-cases include thousands of (image, patch) pairs that are affected by customized code or non-standard building configurations. The results show that PDiff achieves highly accurate and reliable patch presence testing, with a false negative rate lower than 4.5% and no false positives. In addition, PDiff has nearly perfect tolerance to building configuration variation and code customization. These significantly outperform the state-of-the-art techniques.

In summary, we make the following contributions.

- **Deep Understandings of the Patch Presence Testing Problem.** We perform a comprehensive study on the patch presence testing problem by using a large-corpus of real-world kernel images. Throughout this study, we identify the essential challenges tied to reliable patch presence testing and analyze how these challenges affect the state-of-the-art techniques.
- **New Semantic-based Patch Presence Testing Approach.** We design and implement PDiff, a system that utilizes semantic-based similarity comparison to achieve highly accurate and reliable patch presence testing. PDiff is tolerant to variance at the code level, which overcomes the challenges that limit the existing techniques.
- **Comprehensive Evaluation.** We prototype PDiff for Linux AArch64 and ARM32 targets, and conduct an extensive evaluation of PDiff with a large corpus of test cases.

2 PROBLEM UNDERSTANDING

This section first presents a study to motivate patch presence testing for downstream kernels, then defines our problem scope and elaborates on the major challenges of this research.

2.1 Motivating Study

This research is motivated by the fact that downstream vendors prevalently delay the adoption of available patches and they are not reliably reporting their patching progress. To better unveil this less-understood fact, we perform an empirical study with 715 Linux-based kernel images from 9 popular vendors. Detailed distribution of the images is presented in Table 1. In this study, we examine the patching status pertaining to 152 vulnerabilities in the

Table 1: Patch delay and inaccurate patch reports by downstream vendors.

Vendor	# of Images	# of [Image, Vul] Pairs[1]	# of Omitted Patches	Max Patch Age (day)	# of Wrong Patch Reports[2]
Google	152	4,690	0/0%	0	2/0.04%
Samsung	120	3,414	133/3.89%	643	0/0%
Xiaomi	52	1,585	57/3.60%	1,018	0/0%
Vivo	22	652	94/14.42%	893	4/0.61%
Huawei	186	3,911	9/0.23%	373	3/0.08%
Meizu	102	2,563	349/13.62%	1,085	235/9.17%
Oppo	29	852	25/2.93%	935	15/1.76%
D-Link	25	422	97/22.99%	1,451	N/A
NETGEAR	27	496	48/9.68%	1,322	N/A
Total	715	18,585	812/4.37%	-	259/1.39%

[1] Each **[Image,Vul]** pair is a unique combination of an image and a vulnerability.
[2] A **wrong patch report** means a patch is reported as adopted but actually not.

mainstream Linux kernel and we verify the public patch reports from the vendors. Among the 152 vulnerabilities, 120 are reported to the CVE database with a CVSS score higher than 5 across the past 5 years and the remaining 32 are used by FIBER [61]. More details on how we perform the examination are explained in § 5.5.

As noted in Table 1, the vendors widely miss available patches. On average, 4.37% of the patches are not adopted when these vendors release their images. In particular, over 13% of the patches are omitted by Meizu, Vivo, and D-Link. Further, the missed patches can be as old as years. These results, well matching the previous observations [2, 20, 21, 29, 38, 49], strongly support that downstream vendors commonly delay or even ignore the patches.

Looking into the patch reports released by the vendors, we surprisingly observe tremendous missing or even misleading information. Overall, these vendors have three styles of reporting patching progress:

- Vendors such as D-Link and NETGEAR do not disclose their patch information in any means.
- Unlike D-Link and NETGEAR, many vendors release partial information that covers a specific group of patches. In particular, Huawei reports certain patches on its non-Android devices through an official website. Other than that, Samsung, Meizu, Xiaomi, and Huawei attach a patch tag in the format of *20YY-MM-01* while releasing most of their Android ROMs. A *20YY-MM-01* tag indicates all patches in the vendor's security bulletin for month "*MM*" have been applied. From such a tag, the status of patches out of the *20YY-MM-01* bulletin remains known.
- Finally, Oppo, Google, and Vivo are reporting comprehensive patching information for their Android devices[1]. Specifically, they prefer to attach their device ROMs with a tag like *20YY-MM-05*, indicating they have adopted patches covered in all their security bulletins until month "*MM*".

By checking the available patch reports for our 152 vulnerabilities, we find that Vivo, Huawei, Meizu, and Oppo are reporting patches that are actually not applied. In particular, nearly 10% of the patches reported by Meizu are not adopted. While Google has no wrong reports for our 152 vulnerabilities, it, however, attaches security tag *2016-12-05* to an image built on `2016-11-02`. The tag covers two patches that are publicized after *2016-11-02*. Through manual analysis, we verify that the two patches are indeed not applied.

[1]These vendors also use *20YY-MM-01* tags for a small number of their Android devices.

Table 2: Impacts of third-party customization on patch-related functions.

Vendor	# of Sources	# of Patch -related Func	# of Customized Func	Customize Rate
Google	126	741	554	74.76%
Samsung	42	1,256	980	78.03%
Xiaomi	51	1,069	784	73.34%
Vivo	0	-	-	-
Huawei	151	1,092	822	75.27%
Meizu	7	341	231	67.74%
Oppo	8	673	457	67.90%
D-Link	12	313	220	70.29%
NETGEAR	9	542	335	61.81%
Total.	406	6,027	4,383	72.72%

Overall and in general, today's downstream vendors are largely disregarding patching reports or releasing unreliable information.

Summary: Our study empirically confirms our motivation and strongly indicates the necessity of techniques for patch presence testing with downstream kernels.

2.2 Problem Scope

This work focuses on patch presence testing for downstream OS kernels that are derived from an open-source mainstream version. Specifically, we assume a vulnerability in the mainstream kernel is disclosed with a patch at the source level. Given a piece of downstream kernel that inherits the vulnerability, we aim at determining the kernel's patching status.

In this research, we consider a general and common setting where the downstream vendors are not intentionally malicious but they act as follows. First, the vendors may disregard or delay the release of their source code. This frequently happens in the reality, despite the vendors face the risk of license violations [32, 35, 51, 53]. Second, the vendors may not publicize the patching progress or may release misleading information. As we have illustrated above, such cases are surprisingly prevalent.

On account of the conditions above, we have the following assumptions in our research. First, we mainly consider testers that are users of the downstream kernels or providers of security service. Second, the testers can only access the binary code of the downstream kernels. Third, the testers are unable to acquire genuine patching information from the downstream vendors. Finally, the target kernels are free of obfuscation. This complies with our observations on 715 real-world downstream kernel images.

2.3 Challenges of Patch Presence Testing

As briefly introduced, the major challenge of patch presence testing derives from the code-level variance between the mainstream version and the downstream kernels. In practice, we observe two major sources introducing such variance — *third-party customization* and *non-standard building configurations*. To fully understand their prevalence and their effects on patch presence testing, we perform another study as follows.

2.3.1 Third-party Code Customization. Open source kernels are widely customized by third-party vendors for extended functionalities. For instance, a variety of smartphone vendors are porting

Table 3: Impacts of third-party customization on patches (516 samples in total).

Vendor	C1[1]	C2[2]	C3[3]	C4[4]	(C1+C2+C3) / (C1+C2+C3+C4)
Google	11	5	28	22	66.67%
Samsung	21	9	44	42	63.79%
Xiaomi	18	4	34	31	64.37%
Vivo	-	-	-	-	-
Huawei	24	8	45	34	69.37%
Meizu	6	2	13	11	65.63%
Oppo	10	3	12	18	58.14%
D-Link	3	1	10	9	60.87%
NETGEAR	6	3	13	12	66.67%
Total.	99	35	202	180	65.12%

[1] Customization directly modifies the patch (e.g. Figure 6).
[2] Customization modifies the patch context considered by Fiber [61] (e.g. Figure 1).
[3] Customization changes patch-related control flow (e.g. Figure 7).
[4] Patch irrelevant customization.

Android kernels to accommodate their own devices. To unveil the effects of third-party customization on patches, we exhaustively search source code[2] for the 715 images used in § 2.1 and successfully obtain 406 sources[3]. With these sources, we observe prevalent and various types of changes to patches.

Specifically, official patches to the 152 vulnerabilities affect 285 functions in the mainstream Linux kernel. As shown in Table 2, the 285 functions correspond to 6,027 unique counter-parts in our 406 sources. By examining the 6,027 functions, we find that 4,383 cases (over 72%) contain code different from their mainstream versions. In particular, Samsung has over 78% of its patch-related functions varying from the mainstream versions. From the 4,383 customized functions[4], we randomly pick 516 cases (around 12%) to understand how the code variations actually affect the patches. Overall, we observe three types of impacts as summarized in Table 3. Specifically, among the 516 cases, 19.19% contain direct changes to the patching code, 6.78% have changes to code nearby the patching sites (*a.k.a.* patching contexts considered by Fiber [61]), and 39.15% modify patch-dependent control flows. Examples for the three types are respectively demonstrated in Figure 6, Figure 1, and Figure 7.

The above results are clear evidence that third-party customization commonly leads to code changes that indeed affect the patches. More importantly, as we will detail in § 5.2, this type of code variance remains an open challenge to the state-of-the-art patch presence testing tools such as FIBER [61].

2.3.2 Diversities in Building Configurations. Modern OS kernels carry all-sided building configurations to accommodate the needs of functionalities. For instance, the aforementioned 406 Linux-based sources provide three major categories of configurations, including compilation options, self-designed macros, and optimization levels (by GNU GCC). Among these 406 sources, the three categories contain 170 specific configurations on average that may affect the patches to 152 vulnerabilities. Our study below reveals that the downstream vendors widely alter these building configurations, which truly introduce code-level variances that affect the patches.

From the 406 source code, we extract the configurations for 398 of them (configurations from the 8 Oppo source code cannot be obtained). As summarized in Table 4, the configurations can affect the patches in three ways. First, the configurations often enable/disable macros that are different from their mainstream versions[5]. In our data-sets, this type of macro variations affect the patches in 785 (image, patch) pairs. Second, these configurations are adding new macros. Overall, among the 398 images, 254 (image, patch) pairs are affected by new macros. Finally, downstream configurations may use non-standard optimizations. In our 406 images, 212 of them use `Os` instead of `O2`. Difference at the optimization level can in general incur code changes to patches.

To handle the possible variations in building configurations, past research [61] proposes to build the mainstream kernel using different configurations. The insight is to produce a group of reference versions such that one of them matches the target kernel. Not surprisingly, this approach is impractical because of the tremendous number of configurations and their combinations. Therefore, practical patch presence testing needs to tolerate code changes incurred by building configurations. In § 5.2, we will detail that existing techniques have limited utilities with doing so.

Table 4: Impacts of building configurations on patches.

Vendor	Varied-macro-impacted (image, patch)	New-macro-impacted (image, patch)	Os/O2
Google	98	15	126/0
Samsung	71	77	38/4
Xiaomi	144	26	40/11
Vivo	-	-	-
Huawei	416	86	8/143
Meizu	41	17	0/7
Oppo[1]	-	-	-
D-Link	11	20	0/12
NETGEAR	4	13	0/9
Total.	785	254	212/186

[1] Oppo provides no guidance of compiling its kernels and we cannot build those kernels with default configurations. Therefore, we cannot get its configurations.

Summary: Our study demonstrates that third-party code customization and non-standard building configurations are prevalent. The two issues can largely affect the patches, which remain significant challenges to patch presence testing.

3 APPROACH OVERVIEW

In this work, we propose PDiff for patch presence testing on binary-only downstream kernels. Going beyond achieving high accuracy, this approach also aims to carry resilience to code changes due to customization and building configurations. In the following, we first explain our insights and then overview our approach with a running example.

[2]We consider a source code matches an image if (1) the source and the image are used for the same device model; (2) the source and the image share the same kernel version; (3) the source and the image have exactly the same building configurations (if available); and (4) the source and the image have consistent public information, such as firmware version, PDA and CSC.
[3]This shows that downstream vendors only release source code for part of (around 55%) their images, indicating the necessity of binary-only patch presence testing.
[4]Beyond customization, code differences in those functions may also be because the downstream images use mainstream versions that are different from the ones used in our study. However, regarding impacts to patches, such code differences are similar to third-party customization. Hence, we deem those functions also as being customized.

[5]We consider macros that (1) change code in/around the patch or (2) affect patch-dependent control flow.

3.1 Insights

The insights behind the design of PDiff are three-fold. First of all, PDiff bases patching status on similarity comparison. Specifically, PDiff measures the distance from the target kernel to the pre-patch and post-patch reference versions, respectively. It considers that the target kernel shares the patching status with the closer reference version. By intuition, this strategy has high resilience to patch-irrelevant changes, since such code changes would equally affect both of the pre-patch and post-patch reference versions. The noise would be, therefore, balanced. Second, PDiff considers semantic-level properties of patch-affected regions for similarity comparison. Such properties can be largely preserved even when the code layout around the patching site changes. To avoid missing information, PDiff considers all-sided semantics and capture the semantics at a fine granularity. Last but not least, PDiff elaborately determines patch-affected regions. It maximizes the coverage of patch-related code while minimizing the other code to reduce noise.

```
diff --git a/mm/oom_kill.c b/mm/oom_kill.c
@@ -565,11 +564,13 @@ static bool __oom_reap_task_mm(struct
task_struct *tsk, struct mm_struct *mm)
-  tlb_gather_mmu(&tlb, mm, 0, -1);
   for (vma = mm->mmap ; vma; vma = vma->vm_next) {
     ...
     if (vma_is_anonymous(vma)||!(vma->vm_flags&VM_SHARED)){
+      tlb_gather_mmu(&tlb, mm, vma->vm_start, vma->vm_end);
       unmap_page_range(&tlb,
                            vma,vma->vm_start,vma->vm_end,NULL);
+      tlb_finish_mmu(&tlb, vma->vm_start, vma->vm_end);
     }
   }
-  tlb_finish_mmu(&tlb, 0, -1);
   ...
   up_read(&mm->mmap_sem);
```

(a) Patch for CVE-2017-18202.

```
Source snippet of __oom_reap_task_mm in test case
  for (vma = mm->mmap ; vma; vma = vma->vm_next) {
    ...
    if(vma_is_anonymous(vma)||!(vma->vm_flags&VM_SHARED)){
      tlb_gather_mmu(&tlb, mm, vma->vm_start, vma->vm_end);
      mmu_notifier_invalidate_range_start(mm,
                                   vma->vm_start, vma->vm_end);
      unmap_page_range(&tlb,
                           vma,vma->vm_start,vma->vm_end,NULL);
      mmu_notifier_invalidate_range_end(mm,
                                   vma->vm_start, vma->vm_end);
      tlb_finish_mmu(&tlb, vma->vm_start, vma->vm_end);
    }
  }
  ...
  up_read(&mm->mmap_sem);
```

(b) Code snippet of patch-related function in the patched target kernel. The code changes nearby the patching site are marked in orange color.

Figure 1: An running example of patch presence testing.

3.2 Running Example

Guided by the insights above, our approach proceeds with three steps. We brief these steps with a running example presented in Figure 1. In the example, the patch is shown in Figure 1(a), and the target kernel, which has been patched and contains code changes nearby the patching site, is presented in Figure 1(b).

Step 1: Identifying Patch-affected Regions and Collect Patch-affected Paths. The testing by PDiff starts with identifying the code regions for similarity comparison. To avoid missing information, PDiff is designed to include every piece of patch-related code. Specifically, PDiff first collects all the functions containing changes introduced by the patch and we call them patch-related functions. Considering that many functions are large and most of the code is patch irrelevant, PDiff further picks *anchor blocks* from each patch-related function to help reduce unrelated code by only keeping patch-affected paths. Simply speaking, an anchor block ensures that first, no path after the anchor block can reach patch-changed code, and second, any path visiting at least one patch-changed code block will reach the anchor block. Details about picking anchor blocks are covered in § 4.1. Upon the determination of the anchor block, PDiff then collects all patch-affected paths — paths that start at the function entry and end at the anchor block. These paths sufficiently cover all the code regions that are affected by the patch and will be later used for comparison.

Figure 2 illustrates the anchor block and patch-affected paths from our running example as presented in Figure 1. Specifically, `__oom_reap_task_mm` is a patch-related function and a, f, h, i are patch-changed nodes. As g post-dominates all the patch-changed nodes in both the pre-patch and post-patch reference version, PDiff picks it as an anchor block. Using this anchor block, PDiff collects 4 patch-affected paths from the pre-patch version and other 4 patch-affected paths from the post-patch version, respectively shown in Figure 2(a) and Figure 2(b). As noted in this example, the patch-affected paths cover all the patch-changed nodes but include only a few unrelated nodes.

Step 2: Generating Semantic-based Patch Summary. Given the group of patch-affected paths, PDiff then extracts the properties that can capture the semantics behind a patch. Inspired by previous research [16, 17, 19, 43, 59], we focus on properties including function calls, memory status, and path constraints. More details about our selection of semantics are presented in § 4.2. For the simplicity of presentation, we call the group of properties on a patch-affected path a *path digest*. The path digests of all patch-affected paths construct the patch summary.

In Figure 3, we present two path digests in our running example, corresponding to the path *[a,b,c,e,f,g]* in Figure 2(a) and the path *[b,c,e,g]* in Figure 2(b). The two path digests have different function calls, which well represent the patch semantics behind the removal of `tlb_gather_mmu` and `tlb_finish_mmu`.

Step 3: Patch Presence Testing Based on Patch Summary. Following a similar idea as Step 1, PDiff is able to locate the anchor block in the target kernel and then construct the path digests. With path digests for the reference version with/without patch and the target kernel, the idea of testing is then to measure the distance between those path digests. The distance algorithms is detailed in § 4.3. In this step, PDiff assigns the target kernel with the patching status of whichever reference version that has a smaller distance.

Referring back to the running example, the testing target shares three common paths with the post-patch version and the digests on the remaining path is quite similar. By contrast, the path digests of the pre-patch version and those of the testing target are significantly different. This enables PDiff to determine that the target kernel has been patched.

4 APPROACH DESIGN

Following the steps in the running example, PDiff has a workflow as presented in Figure 4. In this section, we elaborate on the key steps, including anchor block selection (§ 4.1), patch summary generation (§ 4.2), and patch presence testing (§ 4.3).

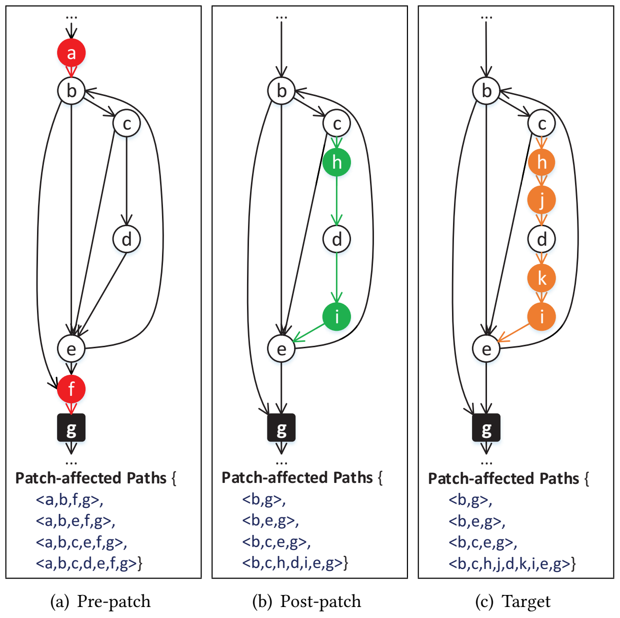
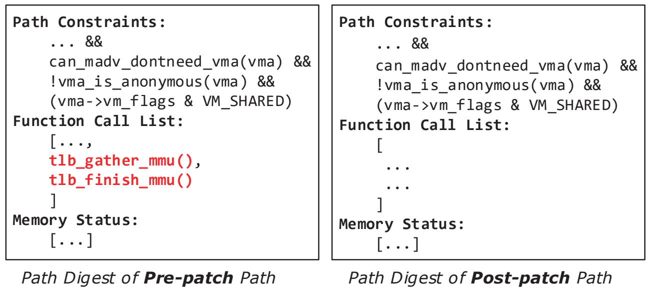


Figure 2: Anchor block and patch-affected paths for the example in Figure 1. Nodes in red, green, and orange are code deleted by the patch, code added by the patch, and code changes nearby the patching site, respectively.



```
Path Constraints:
    ... &&
    can_madv_dontneed_vma(vma) &&
    !vma_is_anonymous(vma) &&
    (vma->vm_flags & VM_SHARED)
Function Call List:
    [...,
    tlb_gather_mmu(),
    tlb_finish_mmu()
    ]
Memory Status:
    [...]
```

*Path Digest of **Pre-patch** Path*

```
Path Constraints:
    ... &&
    can_madv_dontneed_vma(vma) &&
    !vma_is_anonymous(vma) &&
    (vma->vm_flags & VM_SHARED)
Function Call List:
    [
      ...
      ...
    ]
Memory Status:
    [...]
```

*Path Digest of **Post-patch** Path*

Figure 3: Path digests for pre-patch path <a,b,c,e,f,g> and post-patch path <b,c,e,g>, using node g in Figure 2 as the anchor block.

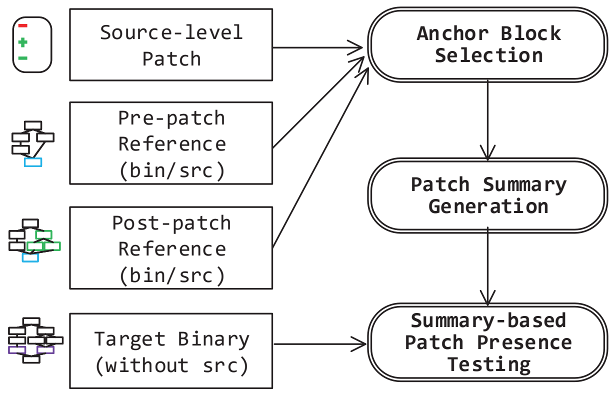

Source-level Patch
Pre-patch Reference (bin/src)
Post-patch Reference (bin/src)
Target Binary (without src)
Anchor Block Selection
Patch Summary Generation
Summary-based Patch Presence Testing

Figure 4: Overall workflow of PDiff.

4.1 Anchor Block Selection

As illustrated in our running example, PDiff summarizes patch semantics from patch-affected paths. For better accuracy, it is crucial to ensure that the patch-affected paths, on the one hand, entirely cover patch semantics, while on the other hand, include as little irrelevant code as possible. To achieve this goal, we first locate patch-related functions from the reference versions and then introduce the concept of anchor block to refine the patch-affected paths. We present the details in the following.

Preparing Reference Versions. In the first step, PDiff builds the reference versions by compiling the source code of the mainstream kernel with and without the patch. Correspondingly, it generates the pre-patch and post-patch reference images. In the compilation process, PDiff uses the default configurations (*e.g.*, -O2) unless changes are required to include patch-related modules. For the ease of further analysis, PDiff enables debugging information.

Identifying Patch-related Functions and Determining Patch-affected Blocks. Given the reference versions, we then determine the patch-related functions and patch-affected code blocks. Technically, we parse the patch file to extract the source locations of code changes and determine the functions that contain these code changes. Considering that the function names in the patch file might be masked by tags or macros (*e.g.*, SYSCALL_DEFINE(func)), we choose to parse the source code while determining the functions. Finally, using the debugging information, we map the code changes to basic blocks in the image. To be specific, we consider the deleted code in the pre-patch image and the added code in the post-patch image. To avoid missing any code, we include all basic blocks that have at least one instruction pertaining to any changed source code. In this process, the functions and basic blocks that we identify are deemed as patch-related functions and patch-affected blocks.

Selecting Anchor Blocks for Reference Versions. Following the above step, we then identify patch-affected paths. An intuitive idea is to enumerate paths from patch-related functions that visit patch-affected blocks. These paths, however, often have long ending parts that are patch irrelevant. To mitigate this issue, we introduce anchor blocks to truncate patch-affected paths such that irrelevant postfixes are excluded. Specifically, an anchor block has the following properties:

- *P-1:* Any path going through a patch-affected block will reach at least one anchor block. In this way, we guarantee all patch-affected blocks are covered.
- *P-2:* No path after an anchor block can reach patch-affected blocks. This ensures that no patch-affected blocks will be pruned.
- *P-3:* An anchor block in the pre-patch version should have a counter-part in the post-patch version and vice versa. Our patch presence testing requires this to align paths for comparison.

To satisfy the above properties, we devise Algorithm 1 to pick anchor blocks from each patch-related function. The algorithm takes the CFG of a patch-related function and the corresponding patch-affected blocks (from both reference versions) as inputs. It outputs an anchor block for each of the patch-affected blocks. Specifically, given a patch-affected block, Algorithm 1 first extracts its post-dominators (at basic block level) from the current function (line `10`). It then iterates the topologically sorted post-dominators (line 11) and picks the first one that also appears in the other version

Algorithm 1 Algorithm for selection of anchor blocks for one patch-related function

```
Input: $cfg_{pre}$: CFG of a pre-patch function; $cfg_{post}$: CFG of a post-patch
    function; $C_{pre}$: patch-affected blocks in pre-patch function; $C_{post}$:
    patch-affected blocks in post-patch function.
Output: Anchor blocks
function MAIN($C_{pre}, cfg_{pre}, C_{post}, cfg_{post}$)
    anchors ←LIST()
    SELECT_ANCHOR_BLOCKS($C_{pre}, cfg_{pre}, cfg_{post}$)
    SELECT_ANCHOR_BLOCKS($C_{post}, cfg_{post}, cfg_{pre}$)
    return anchors
end function

function SELECT_ANCHOR_BLOCKS($C_{bin}, cfg_{ref}, cfg_{assist}$)
    for $c \in C_{bin}$ do
        doms = POST_DOMINATORS($c$)
        for block in TOP_SORT(doms) do
            if block == c then
                Continue
            end if
            if ¬ CHECK_EXISTENCE(block, $cfg_{assist}$) then
                Continue
            end if
            anchors add block
            Break
        end for
    end for
end function
```

(line 12-17). In this process, the requirement of post domination ensures property *P-1*. With the topological sorting, we can pick the nearest post-dominator for maximal trimming of irrelevant code. Finally, we need an anchor block to also exist in the other reference version such that we can satisfy *P-3*. Note that to match an anchor block to a basic block in the other version, we require the two basic blocks to have identical assembly code and identical source code.

As noted in Algorithm 1, different patch-affected blocks may share the same anchor block and we do no de-duplication. This is intended as our summary-based comparison equally considers each patch-affected block. Anchor block de-duplication would eliminate the weights of certain patch-affected blocks. In addition, a path may go through two anchor blocks and there may exist patch-affected blocks in-between. This seemly breaks *P-2*. However, that path will eventually be captured as it ends at the second anchor block. Therefore, essentially, *P-2* is guaranteed.

Selecting Anchor Blocks in the Target Kernel. Our selection of anchor blocks starts with locating patch-related functions in the target kernel. Technically, we leverage symbols carried by the `KALLSYMS` section [55] to find a patch-related function. Once we determine a patch-related function in the target kernel, we search the counter-parts of reference anchor blocks (*i.e.*, anchor blocks that we have identified from the pre-patch/post-patch reference image) from its basic blocks. Our search follows two rules:

- *Termination type.* We require the reference anchor block and the counter-part have identical termination type (signed/unsigned conditional jump, unconditional jumps, function calls, and return). The rationale is that termination type is usually determined by semantics in the original code, which will not be changed by compilation or building.
- *Number of global memory access.* We also require the reference anchor block and the counter-part have the same number of global memory accesses. Similar to termination type, global memory access represents semantics of the original code, which should preserve across different binary versions.

Using the above approach, we may find multiple candidates for a single anchor block. In such cases, only one of them would be picked in the phase of patch summary comparison with the strategies discussed in § 4.3. We also want to note that if we find no appropriate anchor blocks (which is rare), we alternatively consider the function exit node as an anchor block. This strategy also applies to anchor block selection from the reference versions.

4.2 Patch Summary Generation

Given anchor blocks in a patch-related function, PDiff enumerates the paths that start from the function entry and end at anchor blocks. To avoid path explosion in the enumerating, we unroll each loop only once. As aforementioned, PDiff deems the extracted paths as *patch-affected* paths.

To support our semantic-based testing, we choose to preserve patch related semantics with the path digests. However, accurately extracting patch semantics would require human intelligence. To overcome this challenge, our idea is to over-approximate the semantics. We argue that patch semantics are generally represented by the control flow and data flow on patch-affected paths. As such, inspired by previous works [16, 17, 19, 43, 59], we consider fine-grained properties of control flow and data flow for path digests. The properties we consider include path constraints, memory status, and function calls. In the following, we explain the extraction and representation of our path digests.

Extracting Path Digests. To extract the path digests, we run symbolic execution along the patch-affected paths to collect memory accesses, function calls, and path constraints. Different from normal symbolic execution, we start from the entry of a patch-related function and skip function calls. As such, we often need to handle uninitialized contexts, including initial arguments to the patch-related functions, undetermined memory regions, and return values of function calls. We handle them as follows:

- *Function arguments.* Arguments to the patch-related functions are identified based on the calling convention and initialized as uniquely identified symbol values (*e.g.*, *arg0* is assigned to the first argument).
- *Undetermined memory regions.* Undetermined memory regions include uninitialized memory regions and memory regions with symbolic addresses. For uninitialized global memory regions (which access the `.data` and `.bss` segments), we give them unique symbolic values. Other than that, we assign 0 to local uninitialized memory regions. For a memory with symbolized address, we also create a new symbolic value and intercept the interpreter to simply read/write that symbolic value. Meanwhile, we maintain an address-value mapping between the symbolized address and the corresponding symbolic value. In this way, we can correctly reuse the symbolic value when the same symbolized address is de-referenced.

- *Return value from function calls.* Return value from a function call is assigned with a symbolic value in the format of *{funcname}_ret_{idx}*, where *funcname* is the name of the callee and *idx* indicates how many times this function has been called on the current path. If the name of the callee cannot be determined, we assign it a symbolic name.

During our symbolic execution, we exclude paths that carry unsolvable constraints. This helps remove infeasible paths that are previously collected.

Representation of Path Digest. In the course of symbolic execution, we record the aforementioned path digest elements. The key challenge is to use a representation that is semantic-catching but less code-dependent. In other words, the path digests extracted from two semantically-similar paths should be close to each other. In our design, we re-use the representation used by the symbolic executor. This representation converts binary-specific operations and data objects to uniform symbol-based formats. Further, it simplifies the operations into the most concise level, ensuring that identical semantics are similarly represented.

- *Path constraints* are formatted as Abstract Syntax Trees (ASTs). Each AST uses the comparison operator as the root and the expressions as the left/right sub-trees. To make the representation insensitive to the binary code, as aforementioned, we re-use the format from the symbolic engine (*e.g.*, `[arg1 + 0x10] ule [kmemdup_ret_0 + 0x29]`).
- *Memory status* is a set of memory accesses along a patch-affected path. In our design, we only consider global memory regions because they are usually decided by the semantics while the use of local variables are binary specific. Note that for conservativeness, we consider all memory regions with symbolized addresses as global memory regions. In our representation, we use a key-value pair for each global memory access. Considering that different binaries may have different addresses for the same global access, we use symbolized address in the format of `g_idx`, where `idx` represents the number of global access or memory with symbolized addresses we have been encountered. Regarding the value of a memory access, we simply use the concrete values or the IDs of the symbolized values.
- *Function call list* is an ordered list that records all invoked functions along the path. For each function call, we simply record its name and ignore its arguments. For functions without names, we assign a special name `func_unknow`.

All path digests are represented following the above rules and organized into the *Backus-Naur Form* (BNF) [58] (see Figure 9).

4.3 Summary-based Patch Presence Testing

After obtaining the patch summaries from the two reference versions and the target kernel, we measure their similarities to determine the patch status. Briefly speaking, PDiff determines the similarity of two patch summaries based on their path digests. Therefore, we first introduce the comparison of path digests and then explain how we use that to compare patch summaries.

Path Digest Similarity. Recall that a path digest is composed of three elements, including path constraint, memory status and calling list. In our design, we consider the average similarity of these elements as the similarity metric for path digests. We explain the details as follows.

- *Similarity of path constraints.* Constraints on a path are represented as a set of AST expressions. As such, we measure the closeness between two groups of constraints based on *set similarity*. Specifically, given two sets $S = \{s_1, s_2, ..., s_n\}$ and $S' = \{s'_1, s'_2, ..., s'_m\}$, we formally define their similarity as:

$$sim(S, S') = \sum_{i=1}^{n} \sum_{j=1}^{m} m_{ij} \times sim(s_i, s'_j) \tag{1}$$

  where m_{ij} defines the mapping relations between s_i and s'_j while $sim(s_i, s'_j)$ indicates their similarities. Following the literature [42, 46], we require $\{m_{11}, m_{12}, ..., m_{nm}\}$ to enforce a one-to-one mapping. That is, assuming $n < m$, n elements from S' map to unique elements in S and all other pairs are considered unmapped (vice versa if $n > m$). Accordingly, $m_{ij} = 1$ if s_i maps to s'_j and 0 otherwise. To determine $\{m_{11}, m_{12}, ..., m_{nm}\}$, PDiff re-uses the Kuhn-Munkres algorithm [34] to find the one that maximizes Equation 1.
  As noted, we need the similarity between individual elements (*i.e.*, AST expressions) for set similarity. To measure the similarity between two AST expressions, we re-use the tree edit distance [3]. More specifically, we leverage the bipartite graph matching algorithm [46] which is used by [16] to calculate tree edit distance. We notice certain AST expressions are inter-changeable but have different formats (*e.g.*, $a \leq b$ and $b \geq a$). To avoid wrongly considering such pairs to be different, we gather the group of AST expressions where the operators are inter-changeable and force them to follow consistent formats.
- *Similarity of memory status.* Memory status of a path digest is also represented as a set and we calculate its similarity based on the aforementioned set similarity. For individual elements (key-value pairs), their similarity is measured by the product of the key similarity and the value similarity.
- *Similarity of function call list.* We leverage the List Edit Distance (LED) [47] to calculate the similarity between two function call lists. Given two function call lists L_1 and L_2, we calculate their similarity using the following equation:

$$sim(L_1, L_2) = 1 - \frac{LED(L_1, L_2)}{\max(len(L_1), len(L_2))} \tag{2}$$

  where the similarity of two elements is 1 if they correspond to the same function and 0 otherwise.

Patch Summary Comparison. Using the similarity measurement of path digests, we perform patch-summary comparison following Equation 3. Specifically, given an anchor block a_n, we extract the group of paths ending at that anchor block respectively from the pre-patch and the post-patch reference versions, and correspondingly build their path digests. Similarly, we obtain a set of path digests from the target kernel. Based on set-similarity between the path digests from the pre-/post-patch reference version and the target kernel, we are able to calculate $sim(a_n, pre)$ and $sim(a_n, post)$, representing the similarity between the target kernel and the pre-/post-patch reference version on anchor block a_n. If a_n has multiple candidates in the target kernel (recall § 4.1), we will pick the one that maximizes the sum of $sim(a_n, pre)$ and $sim(a_n, post)$. We define a_n is closer to the pre-patch reference version if $sim(a_n, post) < sim(a_n, pre)$ and vice versa. Finally, we consider the target kernel shares the patching status as the reference version that is closer to more anchor blocks.

$$final\_result = \begin{cases} patched, & S > 0 \\ unpatched, & S \leq 0 \end{cases}, \tag{3}$$

$$\textbf{where} \quad S = \frac{\sum_{a_n \in anchor} r(a_n)}{|anchor|},$$

$$r(a_n) = \begin{cases} 1, & sim(a_n, post) > sim(a_n, pre) \\ 0, & sim(a_n, post) == sim(a_n, pre) \\ -1, & sim(a_n, post) < sim(a_n, pre) \end{cases}$$

5 EVALUATION

We have proto-typed PDiff on the top of Angr [52] and Clang Python [15]. Our prototype consists of about 5.5K lines of Python code. Currently, PDiff supports AArch64 and ARM32 targets and its source code will be made available at [41]. To better understand the utility of PDiff, we perform a group of evaluation centering around three questions:

- *Q1: Can PDiff do reliable patch presence testing in practice use?*
- *Q2: Can PDiff tolerate code customization?*
- *Q3: Can PDiff tolerate diversities in building configurations?*

5.1 Experiment Setup

To support the evaluation, we collect two image-sets as follows.

Testing Image Set. We re-use the wild images corresponding to the 406 source code as described in § 2.3 for testing. These images carry a high level of diversities: (1) they are distributed by 8 vendors (D-Link, Huawei, Meizu, NETGEAR, Samsung, Google, Oppo and Xiaomi); (2) they run on various types of devices (routers, mobile phones and tablets); and (3) they are migrated from different versions of Linux kernel (ranging from v3.4 to v4.9). Among these images, 345 are AArch64 targets and the remaining 61 are ARM32 targets. Further, we consider the 152 vulnerabilities used in § 2.1 as patching targets. To determine the ground truth of patches in a wild image, we take a two-step approach. First, we determine the patch status in the corresponding source code via a semi-automatic manner. Specifically, if the code added by a patch is present in the source code, we consider the patch is adopted. For other cases, we manually check the source code to determine the patching status. Second, considering that the patching status in the source code and the corresponding wild image may differ, we perform a further verification. Technically, we compile the source code to generate an image that preserves the patching status. Given a patch, if the patch-related functions in the wild image have identical op-code sequences as the self-compiled image, we consider the wild image has the same patching status as the source code. Otherwise, we exclude the wild image for that patch. Finally, we collected 16,836 (image, patch) pairs where we can identify patch-related functions. For 5,325 of them, we cannot confirm the ground truth with the above approach. As such, we ran our evaluation on the remaining 11,511 (image, patch) pairs which correspond to 51 vulnerabilities.

We also investigated the other pairs that we cannot locate patch-related functions. Specifically, we randomly checked 544 pairs of such pairs with manual analysis. We found that 86% of the cases (468) are missed because the corresponding modules are not included, and the remaining cases are because the patch-related functions are in-lined.

Reference Image Set. For each of the 51 patches as mentioned above, we build AArch64 and ARM32 reference versions by compiling the mainstream Linux kernel [56] with and without the patch. For generality, we use default building configurations.

Across the evaluation, all our experiments are conducted on a machine with Intel Xeon CPU E7-4820 2.00GHz and 378GB RAM, running Ubuntu 18.04.2 LTS.

5.2 Evaluation of Effectiveness

In this evaluation, we run PDiff on the 11,511 (image, patch) pairs and seek answers to Q1, Q2 and Q3. To better illustrate the results, we include FIBER for comparison.

5.2.1 Answer to Q1. In Table 5, we summarize the performance of PDiff with both AAarch64 targets and ARM32 targets. Overall, PDiff presents superior effectiveness. The low false negative rate (less than 4.5%) demonstrates that FIBER can highly accurately identify true patches while no false positive proves that PDiff provides high conservativeness. Further, the average similarity score S (recall Equation 3) reported by PDiff is 0.95 for true positives and -0.81 for true negatives. This large gap demonstrates that our approach can provide highly confident testing.

Comparison with FIBER. Table 5 also summarizes the comparison between PDiff and FIBER. In this comparison, we only consider 45 patches and AArch64 images because FIBER can only generate signatures for those patches and only support AArch64. Similar to the overall results, PDiff produces a low rate of false negative (less than 4.5%) and incurs no false positive. To the contrary, FIBER misses 26% of the adopted patches (false negatives) and triggers a group of false positives. Such results demonstrate that PDiff have significant better utilities than the state-of-the-art tool for patch presence testing.

Moreover, a practical solution should produce consistent effectiveness with handling a variety of patches. To compare PDiff and FIBER from this perspective, we also measure the variation of recalls pertaining to different patches. In result, PDiff produces a very low variance (0.55%) across all the patches, indicating a highly consistent utility. In contrast, FIBER has a high recall variance (12.94%) and for certain patches, FIBER can even produce a recall of 0%. As such, we believe PDiff also outperforms FIBER in the dimension of stability.

Analysis of Errors. The majority of PDiff's false negatives (174, 67.70%) are due to inaccuracy in the CFGs produced by Angr (more details are discussed in § 6). The CFG inaccuracy leads PDiff to missing critical patch-affected paths and hence, running into errors. Among the remaining false negatives, most of them are incurred by in-lining behaviors. In those cases, the patched images inline function calls that are inserted by the patches. As a result, PDiff fails to capture those calls using symbols and

Table 5: Effectiveness evaluation of PDiff.

Tool	Patches	TN[1]	TP[2]	FN(FNR)	FP(FPR)	RECALL
PDiff-AArch64	51	4,432	5,906	234(3.96%)	0(0%)	96.04%
PDiff-ARM32	34[3]	643	530	23(4.34%)	0(0%)	95.66%
FIBER	45	4,015	5,662	1,492(26.35%)	15(0.37%)	73.65%
PDiff	45	4,015	5,662	234(4.13%)	0(0%)	95.87%

[1] True Negatives — (image, patch) pairs where the patches are not adopted.
[2] True Positives — (image, patch) pairs where the patches are adopted.
[3] The ARM32 images are only affected by 34 vulnerabilities.

```
01 diff --git a/security/keys/gc.c b/security/keys/gc.c
02 @@ -148,12 +148,12 @@ static noinline void
   key_gc_unused_keys(struct list_head *keys)
03       ...
04 -           key_user_put(key->user);
05             if (key->type->destroy)
06                   key->type->destroy(key);
07 +           key_user_put(key->user);
08             kfree(key->description);
```

Figure 5: The patch to CVE-2014-9529.

wrongly determines that patches are missing. The problem of inlining is largely because we only consider symbols of calls without considering the inter-procedure semantics. As a solution, we can expand the callees and include their semantics in patch presence testing. More details are also discussed in § 6. Only in a few cases, code changes tangle with the patches such that they mask the patch semantics, making patch presence testing ineffective. We present such a case in Figure 5. The patch relocates a function call from line 4 to line 7. PDiff abstracts the patch-related semantics as a function list [A,B,C] in the pre-patch version and [A,C,B] in the post-patch version, where B indicates `key_user_put()`, C refers to `key->type->destory()`, and A means an arbitrary function. However, the commit in the patched version moves line `5, 6` to the beginning of `key_gc_unused_keys`, resulting in a function list [C,A,B]. Since [C,A,B] has similar editing distance to [A,B,C] and [A,C,B], PDiff conservatively considers this an un-patched case.

Answer to Q1: PDiff can provide accurate and reliable patch presence testing in practice use. It demonstrates higher utilities and stability than the state-of-the-art patch presence testing.

5.2.2 Answer to Q2. To understand the impacts of code customization on patch presence testing, we perform another evaluation with a sub-set of our test-cases. Specifically, from the 11,511 (image,patch) pairs, we pick the ones containing patch-related functions that overlap with the 516 customized functions as described in § 2.3. In total, we gather 173 pairs and 105 of them have patches that are affected by the customization. Running the 173 pairs with both PDiff and FIBER, we have the results as shown in Table 6. As noted in the table, FIBER has a significant increase of false negative rate (22.73% *vs.* 36.67%) when dealing with customization-affected patches. By contrast, PDiff demonstrates consistent performance despite the impacts of customization. This indicates third-party customization remains a challenge to the existing approaches while PDiff can nicely address this challenge. To better understand how PDiff's design gains this type of robustness against third-party customization, we categorize the cases that mislead FIBER but not PDiff:

Table 6: Evaluation on impacts of code customization.

Tool	Patch-affected[1]	TN	TP	FN(FNR)	FP(FPR)	RECALL
FIBER	Y	45	60	22(36.67%)	0(0%)	63.33%
FIBER	N	46	22	5(22.73%)	1(2.17%)	77.27%
PDiff	Y	45	60	4(6.67%)	0(0%)	93.33%
PDiff	N	46	22	1(4.55%)	0(0%)	95.45%

[1] Whether the customization affects the patch ("Y" means yes and "N" means not).

```
diff --git a/sound/usb/quirks.c b/sound/usb/quirks.c
@@ -180,6 +180,12 @@ static int
create_fixed_stream_quirk(struct snd_usb_audio *chip,
   ...
+  if (altsd->bNumEndpoints < 1) {
+    kfree(fp);
+    kfree(rate_table);
+    return -EINVAL;
+  }
```

(a) Patch snippet to CVE-2016-2184.

```
Source snippet of create_fixed_stream_quirk in test case
  if (altsd->bNumEndpoints < 1) {
    list_del(&fp->list);
    kfree(fp);
    kfree(rate_table);
    return -EINVAL;
  }
```

(b) Code snippet of patch-related function in the target test kernel. The target has been patched and it also includes code that changes the patch (marked in orange color).

Figure 6: Example of patch presence testing on category-1: when patch is changed during code customization.

Category-1: When Code Customization Changes the Patch. In many cases, the developers change the patches without breaking their semantics. Not surprisingly, this type of change can easily break code-based signatures. Consider Figure 6 as an example. The patch adds a check on `altsd->bNumEndpoints` followed by two calls to `kfree` and a return error code `-EINVAL`. In this case, FIBER creates a signature representing that there are two `kfree` called after the check on `altsd->bNumEndpoints`. However, as shown in Figure 6(b), the target kernel adds `list_del` before two call on `kfree`, which interrupts FIBER's signature. In result, FIBER misses to detect the patch. While the new code also affects the patch semantics, the effects are minor and the target image is still closer to the post-patch version. Therefore, PDiff still successfully captures the patch.

Category-2: When Code Customization Changes Code Nearby the Patching Site. In this category, the downstream vendors introduce changes around the patching site but not to the patch itself. These changes interrupt the contexts that FIBER considers for the signature and hence, mislead FIBER to making errors. Figure 1 presents such an example. As shown in Figure 1(a), the patch inserts two calls to `tlb_gather_mmu()` at line 7 and line 9. FIBER picks the call at line 7 and the adjacent statement at line 8 to synthesize the signature. In the test case shown as Figure 1(b), a call to function `mmu_notifier_invalidate_range_end()` is inserted between the two statements in the patch signature. In this case, FIBER considers line 5 and line 6 for testing, which apparently mismatches the

```
01 diff --git a/security/keys/key.c b/security/keys/key.c
02 @@ -597,7 +597,7 @@ int key_reject_and_link(struct key *key,
03    if (keyring) {
04      if (keyring->restrict_link)
05        return -EPERM;
06      link_ret = __key_link_begin(keyring, &key->index_key,
                                                      &edit);
07    }
08    ...
09 -  if (keyring)
10 +  if (keyring && link_ret == 0)
11              __key_link_end(keyring, &key->index_key, edit);
```

(a) Patch snippet to CVE-2016-4470.

```
01 Source snippet of key_reject_and_link in test case
02   if (keyring)
03     link_ret = __key_link_begin(keyring, &key->index_key,
                                                     &edit);
04   ...
05   if (keyring && link_ret == 0)
06     __key_link_end(keyring, &key->index_key, edit);
```

(b) Code snippet of patch-related function in the target test kernel. The target has been patched and it also includes code changes that change the patch dependent control flow (marked in orange color).

Figure 7: Example of patch presence testing on category-3: when the patch dependent control flow is changed during code customization.

signature. Differing from FIBER, PDiff captures the semantics of the patch as explained in § 3.2 and hence, detects the patch.

Category-3: When Code Customization Changes Patch Dependent Control Flow. In cases from this category, the vendors introduce changes that bring impacts to patch-dependent control flows. Consider Figure 7 as an example. The patch adds a check on local variable `link_ret`. FIBER generates a signature with code containing the new check (line 10 in Figure 7(a)). For higher robustness, it also considers the preceding checks on `link_ret` at line 3-4 as contexts and includes the two statements into the signature. However, in the target kernel, the vendor removes line 4-5. In result, FIBER's signature is interrupted and FIBER cannot detect this patch. Alternatively, PDiff looks at the semantics of constraints on `link_ret`, which are largely preserved (line 2 and 5 in Figure 7(b)). Therefore, PDiff considers the target is closer to the post-patch reference version and deems this as a patched case.

Answer to Q2: PDiff can well tolerate third-party code customization, demonstrating a higher level of practicality than the state-of-the-art.

5.2.3 *Answer to Q3.*

To evaluate the tolerance of PDiff to diversities in building configurations, we pick the (image, patches) pairs where the patches can be affected by macros. In total, we gather 1,080 pairs and 308 of them define macros differently from the mainstream versions (and therefore, their patches are affected). In Table 7, we show the performance of PDiff and FIBER with the 1,080 pairs. When the patches are affected by the macros, FIBER's false negative rate intensively increases from 28.27% to 40.85%. To the contrary, PDiff has a consistent low rate of false negative (6.33% and 1.41%). This verifies that our semantic-based design has high resilience to diversities in building configurations. In the following, we demonstrate how variation in building configurations makes FIBER ineffective and why PDiff is robust against that.

Table 7: Evaluation on tolerance of diverse building configuration.

Tool	Patch-affected[1]	TN	TP	FN(FNR)	FP(FPR)	RECALL
FIBER	Y	298	474	134(28.27%)	0(0%)	71.73%
FIBER	N	237	71	29(40.85%)	0(0%)	59.15%
PDiff	Y	298	474	30(6.33%)	0(0%)	93.67%
PDiff	N	237	71	1(1.41%)	0(0%)	98.59%

[1] Whether the target image and the reference version have identical configurations that affect patches. "Y" means Yes and "N" means No.

```
01 diff --git a/include/linux/mm.h b/include/linux/mm.h
02 @@ -95,7 +105,7 @@ retry:
03     if ((flags & FOLL_NUMA) && pte_protnone(pte))
04         goto no_page;
05 -   if ((flags & FOLL_WRITE) && !pte_write(pte)) {
06 +   if ((flags & FOLL_WRITE) &&
07 +               !can_follow_write_pte(pte, flags)) {
08         pte_unmap_unlock(ptep, ptl);
09         return NULL;
10     }
```

(a) Patch Snippet to CVE-2016-5195.

```
01 #ifndef CONFIG_NUMA_BALANCING
02 static inline int pte_protnone(pte_t pte)
03 { return 0; }
04 #endif
05
06 #ifdef CONFIG_NUMA_BALANCING
07 static inline int pte_protnone(pte_t pte) {
08       return (pte_val(pte) & (PTE_VALID |
09                     PTE_PROT_NONE)) == PTE_PROT_NONE;
10 }
11 #endif
```

(b) Definition of `pte_protnone()` belongs to one source code corresponding to one image in test cases, guarded by `CONFIG_NUMA_BALANCING`.

Figure 8: Example of patch presence testing on category-4: when patch is affected by building configuration.

Category-4: When Building Configurations Affect Patches. Oftentimes, downstream vendors use non-standard building configurations that result in different code layout from the mainstream version. The code-signatures used by FIBER are sensitive to such changes. Figure 8 shows an example in this category. Given the patch shown in Figure 8(a), FIBER considers line 6, 7 as the signature and includes code in-lined from `pte_protnone` as contexts. As `CONFIG_NUMA_BALANCING` is enabled in the reference version, line 8 and line 9 in Figure 8(b) are included as part of `pte_protnone` in the signature. In the target image, `CONFIG_NUMA_BALANCING` is undefined and hence, `pte_protnone` only contains line 3. This leads to a mismatch between the target image and the reference version. Alternatively, PDiff extracts the semantic that the check on `pte_write` was changed to `can_follow_write_pte` on the paths calling `pte_unmap_unlock()`. Such semantics are present regardless of `CONFIG_NUMA_BALANCING` and hence, PDiff can capture the patch with both configurations.

Answer to Q3: PDiff can greatly tolerate diversities in building configurations, addressing the limitations of FIBER.

5.3 Evaluation of Efficiency

We also evaluate the efficiency of PDiff by measuring the time cost incurred by the analysis. Table 8 presents the results. Overall, PDiff spends an average time of 303.79 seconds on one round of testing, demonstrating a high efficiency. In many cases such as the testing on CVE-2016-7911, PDiff has a time cost as small as seconds. We also note that in a few cases (*e.g.*, CVE-2016-2053), PDiff can take over 40 minutes. In those cases, the patch-related functions contain a large group of basic blocks and have complicated CFGs, significantly increasing the complexity of anchor block selection, path digest generation, and patch similarity comparison. Despite PDiff is relatively slow in the complicated cases, it still significantly outperforms manual analysis [20].

We note that PDiff runs slower than FIBER for testing. On average, FIBER needs around 10.47 seconds to complete a test. This, however, does not mean that PDiff has a lower efficiency. Prior to a test, FIBER needs to measure the similarity between the target kernel with a group of references. Our empirical evaluation, using BinDiff [63] suggested by FIBER, demonstrates that on average one round of measurement takes over **3 minutes** (without considering the pre-processing by the disassembler). By default, FIBER requires similarity measurement with 2 reference versions (built with O2 and Os), incurring **6 more minutes** in an end-to-end test.

Table 8: Time cost of FIBER and PDiff (seconds).

Tool	*Min.*	*Max.*	*Ave.*
FIBER	1.23	63.87	10.47
PDiff	8.08	2,453.02	303.79

5.4 Effectiveness of Anchor Blocks

The design of PDiff leverages anchor blocks for better efficiency and accuracy. As such, we also perform an evaluation of anchor blocks from three aspects.

First, we measure the performance of our approach of selecting anchor blocks from the reference versions. The data-set used for our evaluation carries 70 unique patch-related functions and we successfully found anchor blocks in 56 of them (on average 1.46 anchor blocks per function). For the other 14 cases, we used the exit node as an anchor block.

Second, we evaluate our approach of finding anchor blocks in the target kernel. Using the pattern-based approach that relies on global-memory accesses and mnemonics of termination-instruction (recall § 4.1), we averagely find 8 candidates for each anchor block. By further leveraging path digests similarity (§ 4.3), we refine our candidates to only one anchor block. Our analysis of the testing results shows that the refined anchor block never causes incorrectness in patch presence testing, demonstrating a high fidelity of our approach.

Third, we investigate the effectiveness of anchor blocks with reducing patch irrelevant paths and code. Leveraging anchor blocks, on average we found 16 patch-affected paths with an average length of 10 basic blocks from each patch-related function. By alternatively considering the exit node as the anchor block, the average number of patch-related paths increases to 204 and the average length increases to 15 basic blocks. These results demonstrate that anchor blocks are indeed helpful to improve the accuracy and efficiency.

5.5 Findings in Evaluation

Beyond the 398 images used in our evaluation, we also apply PDiff with the remaining cases from the image-set as described in § 2.1. As we have no ground truth of patches for those images, we manually examine the interesting cases (where patches are released before the image but identified as non-adopted by PDiff). Our analysis results show that PDiff makes no errors, further demonstrating the conservativeness of our design.

Throughout the testing on the total 715 images, we have several findings. Most interestingly, downstream vendors are largely delaying the available patches and they are not disclosing reliable patching progress. The details have been presented in § 2.1 and we omit them in this section.

6 DISCUSSION

In this section, we discuss the limitations of our work and the potential future directions.

CFG Construction. PDiff requires intra-procedure CFGs to aid the slicing of patch-related paths. In our current design, we rely on the *CFGEmulated* method in Angr for CFG reconstruction, which leverages symbolic execution to resolve control flow transitions. Theoretically, this method can provide complete and accurate results. However, the current implementation in Angr has less comprehensive supports for resolving jump tables with AArch64 targets. As a consequence, PDiff may generate incomplete patch summaries, leading to less accurate testing results. Addressing this problem requires to examine the issues in Angr and develop corresponding fixes, which we leave as a future work.

Function Inlining. Function inlining can affect the effectiveness of PDiff in two ways. First, once a patch-related function is always inlined, PDiff would fail to locate this function for patch presence testing. To handle such cases, we can opt to expand the patch-affected region to include parent functions of the patch-related functions, albeit with higher time cost. Second, a patch-related function may inline child functions that are, however, not inlined in the reference versions. This unintentionally introduces variances at the code level. PDiff is more tolerable to such cases than the state-of-art approaches, since the comparison to the pre-patch version and post-patch version will be equally affected. To fundamentally solve this problem, a promising idea is to expand our testing to be inter-procedural and therefore, PDiff will similarly cover the child functions in both the target kernel and the reference versions.

Computation Complexity. In our design, PDiff leverages the Kuhn-Munkres algorithm [34] to find the optimal matching between two summaries, which has a computation complexity of $O(n^3)$. Despite this algorithm may result in high time cost in extreme cases, our evaluation shows that it consumes acceptable time in most of the cases under practical settings. Further, we believe new techniques such as deep learning [59] would help reduce the computation complexity, which we will explore in the future.

Availability of Symbols. PDiff relies on symbols in the `KALLSYMS` section to locate patch related functions. The availability of such symbols can greatly affects our approach. To better understand this issue, we analyze the `KALLSYMS` section in the set of 398 images with source code that we collected from the wild. It shows the `KALLSYMS` section is usually preserved in the stripping process and

prevalently exists in all the wild-images. It also unveils that the KALLSYMS section carries symbols for all the non-inlined functions.
Generalizability of PDiff. The design of PDiff is architecture independent. However, our implementation has to address many architecture-related challenges. In particular, the symbolic execution for path digests extraction needs many architecture-specific customization, including identification of function arguments, symbolization of uninitialized variables, and modeling of external functions. The current version of PDiff supports both AArch64 and ARM32. We will extend it to cover more architectures in the future. Besides, our approach is not limited to kernels. Actually, PDiff can be extended to check patch presence on all kinds of downstream binaries, such as OpenSSL libraries, Python interpreters. In this paper, we mainly focus on Linux kernels because they are the most common targets with many vendors and well-sized data-sets.

7 RELATED WORK

This section presents the most related work in the following.
Vulnerability Patching. Un-patched vulnerabilities have long been believed as major threats to computer security [60]. To mitigate this type of threat, past research has made many endeavours from various perspectives. Li *et al.* measure the effectiveness of vulnerability notification and they surprisingly find that there lacks reliable means for developers. Further, Stock *et al.* [54] outline major factors that affect the effectiveness of vulnerability notification. Beyond the understanding of vulnerability notification, recent works also unveil the development process and complexities of security patches [30], the effects that code clones bring to security patch deployment [37], and how end-users behave to security patches [50]. To facilitate timely development and adoption of vulnerability patches, other researchers explore many new techniques for automated patch generation and adoption [5, 11, 25–27, 31, 36, 62], and hot-patch schemes for various types of software (*e.g.*, Android apps [6], Android kernels [7] and Web servers [40]).
Patch Presence Testing. Detecting un-adopted security patches in a timely manner is essential for vulnerability mitigation. Along this line, Duan *et al.* propose OSSPolice [12] to identify un-patched libraries at scale, Jang *et al.* develop ReDeBug [28] that discovers un-patched code in language-independent clones, and Feng *et al.* [18] explore neural networks for automatic patch detection. BScout [9] is a patch presence testing tool for Java executables which performs cross-language-layer code similarity analysis. Different from these works that focus on patch presence testing with user-space code, FIBER [61] aims at patch presence testing with binary-only kernels. As demonstrated by our evaluation, the design of FIBER insufficiently considers the code variance incurred by third-party customization and non-standard building configurations, which consequently limits the effectiveness of FIBER in practice.
Similar Code Search. In our research, we determine the patch presence status based on the similarity of semantics in code. In the literature, there exist many other techniques that search similarity with binary code. BinDiff [63] and BinSlayer [4] compute the similarity between two functions based on the isomorphism of two CFGs. Alternatively, BinHunt [19] and iBinHunt [33] find semantic differences corresponding to functionality changes. Further, discovRE [14] and multi-M [42] extract features from basic blocks for similarity comparison. Specifically, discovRE uses numeric features from basic blocks and multi-M [42] leverages the I/O behavior in basic blocks. For higher efficiency of similarity comparison, Genius [17], Gemini [59], and BinSequence [24] propose new schemes to reduce the search space. Technically, Genius [17] converts the CFG into high-level numeric feature vectors, Gemini [59] combines graph embedding with neural network to speed up the searching process, and BinSequence [24] uses Min-hashing to effectively filter the search space. Similar to PDiff, some of these works endeavors to increase robustness against code variations. For instance, TEDEM [43] leverages tree edit distance and symbolic simplification against syntactic changes.
Kernel Fingerprinting. Kernel fingerprinting is widely used for the identification of key information, which plays a critical role in many security applications. Gu *et al.* [22] differentiate the main kernel code from other parts in the physical memory and generate a unique hash signature for the OS. Roussev *et al.* [48], based on the on-disk representation of the kernel, create the page-sized sdhash similarity digests to identify kernel versions. Beyond just kernels, Ahmed *et al.* [1] develop a new technique to pinpoint relocatable code in a memory snapshot, covering both kernel and remnants of prior executions. Conceptually, kernel fingerprinting can also be applied for patch code testing. It, however, will face two challenges. First, fingerprinting mostly considers similarity among large code pieces (*e.g.*, kernel images). It may not well handle patch-related code regions that are usually small. Second, most of the fingerprinting techniques are based on code signatures, which likely will be less effective when the patches carry code-level variations.

8 CONCLUSION

This work presents a deep understanding of patch presence testing with downstream kernels. It identifies two key challenges with patch presence testing: *third-party code customization* and *diversities in building configuration.* To overcome the challenges, we propose PDiff, a semantic-based approach of patch presence testing. PDiff captures the semantic behind a patch, offering high tolerance to code variance incurred by third-party code customization and diversities in building configuration. PDiff significantly outperforms the state-of-the-art patch presence testing. On average, it achieves a false negative rate lower than 4.5% and zero false positive, even when the above challenges arise. We release PDiff and our data-sets at https://github.com/seclab-fudan/PDiff.

ACKNOWLEDGEMENT

We would like to thank our shepherd Andrea Continella and anonymous reviewers for their helpful comments. This work was supported in part by the National Natural Science Foundation of China (U1636204, U1836210, U1836213, U1736208, 61972099), Natural Science Foundation of Shanghai (19ZR1404800), and National Program on Key Basic Research (NO. 2015CB358800). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

REFERENCES

[1] Irfan Ahmed, Vassil Roussev, and Aisha Ali Gombe. 2015. Robust Fingerprinting for Relocatable Code. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY '15)* (San Antonio, Texas, USA). Association for Computing Machinery, New York, NY, USA, 219–229.

[2] Ionut Arghire. 2018. Android Vendors Regularly Omit Patches in Security Updates. https://www.securityweek.com/android-vendors-regularly-omit-patches-security-updates.

[3] Philip Bille. 2005. A Survey on Tree Edit Distance and Related Problems. *Theoretical Computer Science* 337, 1-3 (June 2005), 217–239.

[4] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW'13)* (Rome, Italy). ACM, New York, NY, USA, Article 4, 10 pages.

[5] G. Chen, H. Jin, D. Zou, B. B. Zhou, Z. Liang, W. Zheng, and X. Shi. 2013. SafeStack: Automatically Patching Stack-Based Buffer Overflow Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 10, 6 (Nov 2013), 368–379.

[6] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[7] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android Kernel Live Patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*. USENIX Association, Vancouver, BC.

[8] Symantec Corporation. 2020. Attack Signatures - Symantec Corp. https://www.symantec.com/security_response/attacksignatures/.

[9] Jiarun Dai, Yuan Zhang, Zheyue Jiang, Yingtian Zhou, Junyan Chen, Xinyu Xing, Xiaohan Zhang, Xin Tan, Min Yang, and Zhemin Yang. 2020. BScout: Direct Whole Patch Presence Test for Java Executables. In *Proceedings of 29th USENIX Security Symposium (USENIX Security'20)*. USENIX Association, 1147–1164.

[10] Darpa. 2019. Rapidly Patching Legacy Software Vulnerabilities in Mission-Critical Systems. https://www.darpa.mil/news-events/2019-10-14.

[11] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. 2019. Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS'19)*.

[12] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (Dallas, Texas, USA). ACM, New York, NY, USA, 2169–2185.

[13] Eklektix. 2020. The LWN.net Linux Distribution List. https://lwn.net/Distributions/.

[14] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23th Annual Network and Distributed System Security Symposium (NDSS'16)*.

[15] Ethanhs. 2018. Clang Python. https://github.com/ethanhs/clang.

[16] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting Conditional Formulas for Cross-Platform Bug Search. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (AsiaCCS'17)* (Abu Dhabi, United Arab Emirates). ACM, New York, NY, USA, 346–359.

[17] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS'16)* (Vienna, Austria). ACM, New York, NY, USA, 480–491.

[18] Qian Feng, Rundong Zhou, Yanhui Zhao, Jia Ma, Yifei Wang, Na Yu, Xudong Jin, Jian Wang, Ahmed Azab, and Peng Ning. 2019. Learning binary representation for automatic patch detection. In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 1–6.

[19] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Proceedings of the 10th International Conference on Information and Communications Security (ICICS'08)* (Birmingham, UK). Springer-Verlag, Berlin, Heidelberg, 238–255.

[20] Andy Greenberg. 2018. How Android Phones Hide Missed Security Updates From You | WIRED. https://www.wired.com/story/android-phones-hide-missed-security-updates-from-you/.

[21] Roger A. Grimes. 2019. Why patching is still a problem – and how to fix it. https://www.csoonline.com/article/3025807/why-patching-is-still-a-problem-and-how-to-fix-it.html.

[22] Yufei Gu, Yangchun Fu, Aravind Prakash, Zhiqiang Lin, and Heng Yin. 2012. OS-Sommelier: Memory-Only Operating System Fingerprinting in the Cloud. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC'12)* (San Jose, California). Association for Computing Machinery, New York, NY, USA, Article 5, 13 pages.

[23] Kelly Jackson Higgins. 2018. Unpatched Vulnerabilities the Source of Most Data Breaches. https://www.darkreading.com/vulnerabilities---threats/unpatched-vulnerabilities-the-source-of-most-data-breaches/d/d-id/1331465.

[24] He Huang, Amr M. Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (AsiaCCS'17)* (Abu Dhabi, United Arab Emirates). ACM, New York, NY, USA, 155–166.

[25] Z. Huang, M. DAngelo, D. Miyani, and D. Lie. 2016. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*. 618–635.

[26] Zhen Huang and Gang Tan. 2019. Rapid Vulnerability Mitigation with Security Workarounds. In *Proceedings of the Workshop on Binary Analysis Research (BAR'19)*.

[27] Zhen Huang, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*.

[28] Jiyong Jang, D. Brumley, and A. Agrawal. 2012. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *Proceedings of the 33th IEEE Symposium on Security and Privacy (S&P'12)*. IEEE Computer Society, Los Alamitos, CA, USA, 48–62.

[29] Duo Lab. 2016. THIRTY PERCENT OF ANDROID DEVICES SUSCEPTIBLE TO 24 CRITICAL VULNERABILITIES. https://duo.com/decipher/thirty-percent-of-android-devices-susceptible-to-24-critical-vulnerabilities.

[30] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (Dallas, Texas, USA). ACM, New York, NY, USA, 2201–2215.

[31] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)* (Singapore). ACM, New York, NY, USA, 329–340.

[32] Mike Malloy. 2013. HTC: HTC needs timely kernel source releases! https://www.change.org/p/htc-htc-needs-timely-kernel-source-releases.

[33] Jiang Ming, Meng Pan, and Debin Gao. 2013. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proceedings of the 15th International Conference on Information Security and Cryptology (ICISC'12)* (Seoul, Korea). Springer-Verlag, Berlin, Heidelberg, 92–109.

[34] Karleigh Moore, Nathan Landman, and Jimin Khim. 2020. Hungarian Maximum Matching Algorithn, Brilliant Math & Science Wiki. https://brilliant.org/wiki/hungarian-matching.

[35] Kendra Morton. 2018. GPL Violations: Learning the Hard Way | Software Composition Analysis. https://blog.flexerasoftware.com/software-composition-analysis/2018/04/gpl-violations-learning-the-hard-way/.

[36] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. 2013. PatchDroid: Scalable Third-party Security Patches for Android Devices. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)* (New Orleans, Louisiana, USA). ACM, New York, NY, USA, 259–268.

[37] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. 692–708.

[38] Karsten Nohl and Jakob Lell. 2018. Mind the Gap: Uncovering the Android Patch Gap Through Binary-Only Patch Level Analysis. https://conference.hitb.org/hitbsecconf2018ams/sessions/mind-the-gap-uncovering-the-android-patch-gap-through-binary-only-patch-level-analysis/.

[39] Paloalto. 2020. Exploit Protection - Palo Alto Networks. https://www.paloaltonetworks.com/features/exploit-protection.

[40] M. Payer and T. R. Gross. 2013. Hot-patching a web server: A case study of ASAP code repair. In *Proceedings of the 11th Annual Conference on Privacy, Security and Trust (PST'13)*. 143–150.

[41] PDIFF. 2020. PDIFF source code and labelled data. https://github.com/seclab-fudan/PDiff.

[42] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. 709–724.

[43] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. 2014. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)* (New Orleans, Louisiana, USA). ACM, New York, NY, USA, 406–415.

[44] Dan Price. 2018. The True Market Shares of Windows vs. Linux Compared. https://www.makeuseof.com/tag/linux-market-share/.

[45] Steve Ranger. 2019. Cybersecurity: One in three breaches are caused by unpatched vulnerabilities | ZDNet. https://www.zdnet.com/article/cybersecurity-one-in-three-breaches-are-caused-by-unpatched-vulnerabilities/.

[46] Kaspar Riesen, Michel Neuhaus, and Horst Bunke. 2007. Bipartite Graph Matching for Computing the Edit Distance of Graphs. In *Proceedings of the 6th IAPR-TC-15 International Conference on Graph-based Representations in Pattern Recognition (GbRPR'07)* (Alicante, Spain). Springer-Verlag, Berlin, Heidelberg, 1–12.

[47] Eric Sven Ristad and Peter N. Yianilos. 1998. Learning String-Edit Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, 5 (May 1998), 522–532.

[48] Vassil Roussev, Irfan Ahmed, and Thomas Sires. 2014. Image-based kernel fingerprinting. *Digital Investigation* 11 (2014), S13–S21.

[49] James Sanders. 2019. 25% of software vulnerabilities remain unpatched for more than a year. https://www.techrepublic.com/article/25-of-software-vulnerabilities-remain-unpatched-for-more-than-a-year/.

[50] Armin Sarabi, Ziyun Zhu, Chaowei Xiao, Mingyan Liu, and Tudor Dumitraş. 2017. Patch Me If You Can: A Study on the Effects of Individual User Behavior on the End-Host Vulnerability State. In *Proceedings of 18th Passive and Active Measurement (PAM'17)*. Springer International Publishing, 113–125.

[51] Roy Schestowitz. 2013. Success: Samsung's GPL Violation and Subsequent Leak Officially Mean exFAT Driver is Being Made Free Software | Techrights. http://techrights.org/2013/08/17/exfat-and-gpl/.

[52] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*. 138–157.

[53] Gary Sims. 2018. Why GPL violations are bad - Gary explains - Android Authority. https://www.androidauthority.com/gpl-violations-bad-834569/.

[54] Ben Stock, Giancarlo Pellegrino, Frank Li, Michael Backes, and Christian Rossow. 2018. Didn't You Hear Me? - Towards More Successful Web Vulnerability Notifications. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[55] Linus Torvalds. 2020. Script that generates the KALLSYMS Section. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/scripts/link-vmlinux.sh.

[56] Linus Torvalds. 2020. Torvalds Linux kernel git repositories. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.

[57] Bernd van der Wielen. 2018. Insights into the 2.3 Billion Android Smartphones in Use Around the World. https://newzoo.com/insights/articles/insights-into-the-2-3-billion-android-smartphones-in-use-around-the-world/.

[58] Wikiversity. 2020. Backus-Naur form. https://en.wikipedia.org/wiki/Backus-Naur_form.

[59] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)* (Dallas, Texas, USA). ACM, New York, NY, USA, 363–376.

[60] Byoungyoung Lee Yeongjin Jang, Tielei Wang and Billy Lau. 2014. Exploiting Unpatched iOS Vulnerabilities for Fun and Profit. https://www.blackhat.com/us-14/archives.html#exploiting-unpatched-ios-vulnerabilities-for-fun-and-profit.

[61] Hang Zhang and Zhiyun Qian. 2018. Precise and Accurate Patch Presence Test for Binaries. In *Proceedings of the 27th USENIX Security Symposium (USENIX Security'18)*. USENIX Association, Baltimore, MD.

[62] Mu Zhang and Heng Yin. 2014. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS'14)*.

[63] Zynamics. 2019. BinDiff Homepage. https://www.zynamics.com/.

A FORMALISM OF PATH DIGEST

We formalize the path digest with the Backus-Naur Form (BNF). Details of our formalism are presented in Figure 9.

```
<PathDigest>        ::= [PathConstraints]','[MemoryStatus]','
                        [FunctionCallList]
<PathConstraints>   ::= 'Set('{<Constraints>','}<Constraints>')'
<MemoryStatus>      ::= 'Set('{<MemoryUnit>','}<MemoryUnit>')'
<FunctionCallList>  ::= 'List('{<Func>','}<Func>')'
<Constraints>       ::= <Expr>
                      | <Expr><cop><expr>
<MemoryUnit>        ::= '(addr='<Expr>',value='<Expr>')'
<Expr>              ::= <FunctionArgument>
                      | <GlobalVariable>
                      | <ExternalCallReturnValue>
                      | <ImmediateValue>
                      | '['<Expr>']'
                      | <Expr><op><Expr>
                      | 'not'<Expr>
```

Figure 9: Abbreviated BNF for path digest.

B CVE DATASET

Table 9 shows the details of 51 CVEs that we used to evaluate PDiff. The 51 CVEs cover different types of vulnerabilities such as race condition, NULL pointer dereference, out-of-bounds read/write. These CVEs also affect different versions of Linux kernels across the past 5 years.

Table 9: Overview of CVE Dataset.

CVE	Affected Versions	Vulnerability Type
CVE-2014-1739	<3.14.6	Uninitialized data
CVE-2014-2523	<3.13.6	Logic bug
CVE-2014-4014	<3.14.8	Logic bug
CVE-2014-9529	<3.18.2	Race condition
CVE-2014-9914	<3.15.2	Race condition
CVE-2015-1421	<3.18.8	Use-after-free
CVE-2015-1465	<3.18.8	Logic bug
CVE-2015-5364	<4.0.6	Logic bug
CVE-2015-8787	<4.4	NULL pointer dereference
CVE-2015-8839	<4.5	Race condition
CVE-2015-8955	<4.1	NULL pointer dereference
CVE-2015-8963	<4.4	Race condition
CVE-2015-9004	<3.19	Logic bug
CVE-2016-0723	<4.4.1	Race condition
CVE-2016-2053	<4.3	Logic bug
CVE-2016-2184	<4.5.1	NULL pointer dereference
CVE-2016-2546	<4.4.1	Race condition
CVE-2016-3955	<4.5.3	Out-of-bounds write
CVE-2016-4470	<4.6.3	Uninitialized variable
CVE-2016-5696	<4.7	Logic bug
CVE-2016-6786	<4.0	Race condition
CVE-2016-7910	<4.7.1	Use-after-free
CVE-2016-7911	<4.6.6	Race condition
CVE-2016-7912	<4.5.3	Use-after-free
CVE-2016-7916	<4.5.4	Race condition
CVE-2016-9120	<4.6	Use-after-free
CVE-2016-10200	<4.8.14	Use-after-free
CVE-2017-7374	<4.10.7	Use-after-free
CVE-2017-8070	<4.9.11	Logic bug
CVE-2017-9074	<4.11.1	Out-of-bounds read
CVE-2017-15868	<3.19	Logic bug
CVE-2017-15951	<4.13.10	Race condition
CVE-2017-16527	<4.13.8	Use-after-free
CVE-2017-16533	<4.13.8	Out-of-bounds read
CVE-2017-16534	<4.13.6	Out-of-bounds read
CVE-2017-16535	<4.13.10	Out-of-bounds read
CVE-2017-16939	<4.13.11	Use-after-free
CVE-2017-17806	<4.14.8	Buffer Overflow
CVE-2017-17857	<4.14.8	Logic bug
CVE-2017-18202	<4.14.4	Use-after-free
CVE-2017-18255	<4.11	Integer overflow
CVE-2018-7480	<4.11	Double free
CVE-2018-10938	4.0-rc1~v4.13-rc4	Logic bug
CVE-2018-11508	<4.16.9	Logic bug
CVE-2018-12232	<4.17.1	Race condition
CVE-2018-13405	<4.17.4	Logic bug
CVE-2018-1000200	4.14.x, 4.15.x, 4.16.x	NULL pointer dereference
CVE-2019-9213	<4.20.14	NULL pointer dereference
CVE-2019-10638	<5.1.7	Logic bug
CVE-2019-10639	4.x~5.0.8	Logic bug
CVE-2019-16994	<5.0	Logic bug