An Investigation of the Android Kernel Patch Ecosystem

Zheng Zhang
UC Riverside
zzhan173@ucr.edu

Hang Zhang
UC Riverside
hang@cs.ucr.edu

Zhiyun Qian
UC Riverside
zhiyunq@cs.ucr.edu

Billy Lau
Google Inc.
billylau@google.com

Abstract

open-source projects are often reused in commercial software. Android, a popular mobile operating system, is a great example that has fostered an ecosystem of open-source kernels. However, due to the largely decentralized and fragmented nature, patch propagation from the upstream through multiple layers to end devices can be severely delayed. In this paper, we undertake a thorough investigation of the patch propagation behaviors in the entire Android kernel ecosystem. By analyzing the CVEs and patches available since the inception of the Android security bulletin, as well as open-source upstream kernels (*e.g.,* Linux and AOSP) and hundreds of mostly binary OEM kernels (*e.g.,* by Samsung), we find that the delays of patches are largely due to the current patching practices and the lack of knowledge about which upstream commits being security-critical. Unfortunately, we find that the gap between the first publicly available patch and its final application on end devices is often months and even years, leaving a large attack window for experienced hackers to exploit the unpatched vulnerabilities.

1 Introduction

open-source software is ubiquitous and often serves as the foundation of our everyday computing needs. Unfortunately, they also contain a large number of vulnerabilities — there are new security patches released weekly for open-source software (e.g., Linux).

It can be tricky to ensure timely delivery of patches for open-source software because of the widespread *reuse* phenomenon where multiple versions or branches of the open-source software co-exist and can be divided into so-called *upstream* and *downstream* ones. Downstream developers *reuse* much of the upstream software and add finishing touches (*e.g.,* customization, stability fixes). More importantly, downstream developers have to take critical security patches from upstream to eliminate vulnerabilities. This is often challenging because upstream and downstream branches are often developed and maintained by different organizations and companies that often have different priorities and goals in mind.

The single most prominent example is the Android ecosystem. The Android open-source Project (AOSP) kernels are derived from Linux kernels (*i.e., reused* in Android) with many features added for mobile devices. In turn, the AOSP kernels are *reused* by chipset vendors such as Qualcomm who add additional hardware-specific changes. A chipset vendor's kernel is then finally *reused* by OEM vendors such as Samsung and Xiaomi. This means that the patches can originate from more than one *upstream* kernels (e.g., Linux, AOSP, and Qualcomm), and the propagation can take multiple steps to finally reach the OEM vendors. Even though Google has been working diligently with OEM vendors on patching, e.g., through its monthly update program [1], the ecosystem is unfortunately so decentralized that it is beyond the control of a single entity.

Motivated by the lack of transparency and understanding of the patching process, we set out to investigate the unique and complex Android kernel ecosystem. Specifically, we are interested in the following high-level aspects:

(1) The relationship between the upstream and downstream kernels, *e.g.,* who is responsible for the initial patch, and how does it propagate?

(2) The timeliness of patch propagation, *e.g.,* what is the typical delay in each step with the patch propagation and where is the bottleneck?

(3) The factors that influence the patch propagation, *e.g.,* what are the current best practices by different entities, and how can we improve the situation?

It is challenging to conduct such a measurement study. Specifically, even though Android kernels inherit

the open-source license from Linux, kernel sources from OEM vendors are often released broken/half-baked, with substantial delays, and only intermittently (*e.g.,* when the phone was initially released) [38, 35, 33]. In contrast, the binary ROMs (*i.e.,* firmware images) are easier to find. Therefore, to be able to analyze closed-source Android firmware images, we build a static analysis tool on top of FIBER [42], a state-of-the-art tool capable of conducting patch presence test in binaries.

By analyzing the patches announced in the Android security bulletin, 20+ OEM phone models, and 600+ kernel images, we delineate many interesting findings that reveal intriguing relationships among different parties as well as the bottleneck of the whole patch propagation process. When fair to do so, we also compare the responsiveness among different parties, *e.g.,* which OEM vendors are more diligent in patching their devices.

We summarize our contributions as follows:

- We investigate the unique Android kernel ecosystem that is decentralized and fragmented. We mine the patch propagation delays across all layers and locate the bottleneck.
- We improve a state-of-the-art source-to-binary patch presence test tool and develop a system on top of it to check the closed-source kernels from OEM vendors. We plan to open-source our system and release the dataset to improve the transparency of the ecosystem.
- We conduct a large-scale measurement that shows nearly half of the CVEs are patched on OEM devices roughly 200 days or more after the initial patch is publicly committed in the upstream, and 10% – 30% CVEs are patched after a year or more.
- Furthermore, by mining the commit methods and correlating them with notification dates published by Google and Qualcomm, we explain the causes of patch delays. We also distill takeaways and potential prescriptive solutions to improve the current situation.

2 Android Kernel Ecosystem

Android is known for its diverse and fragmented ecosystem where multiple variants of the operating system co-exist [21]. On one hand, the scale and diversity of the ecosystem participants definitely contributed to Android's overall success. On the other hand, it is extremely challenging to ensure the consistency and security of every Android variant out on the market. It is especially true for Android kernels which are themselves derived from the upstream Linux kernel.

Hierarchy of Linux/Android kernels. Figure 1 illustrates the typical relationship between the upstream and downstream kernels. At the very top, we have the Linux mainline that moves forward rapidly with all the features and bug fixes. Its kernel versions are tagged as 4.4, 4.5, *etc.* Periodically when appropriate, it gets forked into *stable* (*e.g.,* 4.3.y) or *long term support* (LTS) branches (*e.g.,* 4.4.y) with mostly only bug fixes [30]. The difference between *stable* and LTS branches is that the former is short-lived (a few weeks) while the latter is supported for a few years. For the benefit of longer support, Android common kernels (*e.g.,* 4.4) typically follow the LTS branches. Meanwhile, Google developers will add the necessary changes for mobile devices to turn the Linux kernel into an Android kernel [29]. In addition, the developers will merge the fixes from Linux to ensure that they stay up-to-date and bug-free.

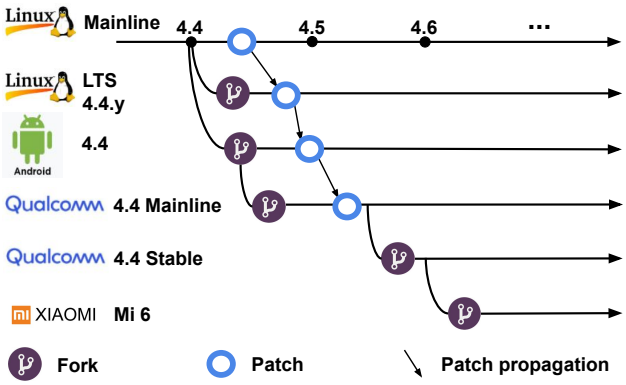

Figure 1: Android ecosystem for kernel version 4.4

In Figure 1, Google's Android common 4.4 is initially forked from Linux mainline 4.4 and in the future merges all the changes from Linux LTS 4.4.y. Then there are branches maintained by SoC vendors such as Qualcomm, MediaTek, and Exynos (out of which only Qualcomm provides the complete history in git repos). Take Qualcomm as an example, when the company decides to ship a new SoC like Snapdragon 830, it may choose to fork a then-recent Android common 4.4.y. In fact, there exists a generic 4.4.y branch and multiple chipset-specific branches all maintained by Qualcomm (simplified in Figure 1). Interestingly, sometimes Qualcomm may choose to fork directly from upstream Linux (*e.g.,* 4.9.y) instead of Android common. Nevertheless, it will still merge significant changes from Android common later on. According to our analysis, SoC vendors typically take fixes and security patches from its direct upstream, Android common, instead of Linux. This practice is reasonable as Google has already done a significant amount of work for the SoC vendors such as patch compatibility tests for Android kernels. However, this also increases the patch propagation delay due to the extra hop.

Finally, at the very bottom of the hierarchy is the OEM vendor kernel. Depending on the device model and its chipset, *e.g.,* a Xiaomi phone using Snapdragon 835, the corresponding branch from the SoC vendor will be forked (Qualcomm's 4.4.y). The OEM vendor may then optionally add new features (*e.g.,* Samsung's kernel hardening [39]) or simply only port bug fixes from the upstream (for smaller OEM vendors). However, when it comes to security patches, OEM vendors tend to have a tighter connection with Google who monthly updates its Android security bulletin since 2015. According to our knowledge, Google serves as the main point of contact notifying OEM vendors about various security vulnerabilities even though the original patch may come from other parties (*e.g.,* Linux or Qualcomm). From Sep 2017, Qualcomm has also established its own security bulletin and independently notifies its customers about Qualcomm-specific vulnerabilities [18, 37], which overlap with the ones on the Android security bulletin.

Android security bulletin is a central location where Google publishes monthly updates on Android security patches and their corresponding CVEs [1]. For the CVEs affecting the open-source Android components (for kernels, most are open-sourced except some proprietary drivers, *e.g.,* by MediaTek), there will be links to the upstream kernel commits representing the patches of the vulnerabilities.

It is worth noting that as Android kernels can be customized by individual OEM vendors, the bulletin may not cover OEM-specific vulnerabilities (*e.g.,* an OEM device may use a custom file system). Nevertheless, it represents Google's best effort to keep track of vulnerabilities that affect the Android common kernel, the upstream Linux kernel, and SOC vendors (primarily Qualcomm). In fact, each CVE has a corresponding link to its patch (*i.e.,* a git commit) that belongs to one of the three kernel repositories.

Before publicizing the vulnerabilities on the Android security bulletin, Google notifies OEM vendors at least one month earlier to ensure that affected devices are patched [2]. In other words, the publication of vulnerabilities on the Android security bulletin represents a major event in the patch management cycle, after which unpatched devices will be in danger. Indeed, our measurement results suggest that OEM vendors are dependent on Google for patching.

3 Measurement Goal and Pipeline

As alluded to earlier, the goal of the measurement is to shed light on the patch propagation in the fragmented Android kernel ecosystem. In this paper, we explicitly assume the knowledge of the affected function(s) and the source-level patch itself, as the upstream Linux/Android kernels do offer detailed patch commits. As a result, our goal is that *given a CVE, we will track the propagation of the initial patch along the chain of upstream-downstream kernels*. Together with the CVE publication time on the Android security bulletin, we can paint a timeline of patch commit and announcement events in the whole patch management cycle.

Before we introduce the measurement pipeline, we first introduce the **three different types of kernels** that are publicly accessible, with increasing degrees of difficulties to analyze.

(1) Type 1: Repository. Kernels made available through git repositories contain complete commit history. They represent the easiest case to analyze as a security patch can be easily located in the commit log — typically they simply copy the commit message and/or reference the commit given in the Android security bulletin's link. *Linux, Android common, Qualcomm and Nexus/Pixel* kernels belong to this category. Unfortunately, other SoC vendors such as Samsung Exynos, MediaTek, and Huawei Kirin do not offer git repositories corresponding their recent chipsets.

(2) Type 2: Source code snapshots. Most OEM vendors prefer to release their kernels in the form of source code snapshots without commit history (Google's own Nexus/Pixel phones are exceptions). It is usually possible to check if a particular CVE is patched in the snapshot via simple source-level function comparison (more details in §4.2). The issue though, is that such snapshots are released with substantial delays and often sporadically, leading to missing data points and inconclusive results.

(3) Type 3: Binary. The most available form of OEM kernels is the binary one – firmware images or ROMs. In fact, there is an abundant supply of Android ROMs on both first-party [9, 10] and third-party websites [7, 8]. These ROMs represent a valuable data source for patch propagation analysis, as long as we can accurately test patch presence in these binaries.

Measurement pipeline. Now we introduce the measurement pipeline (Figure 2) that integrates the analysis of the above three kernel types:

(1) Crawler. Initially, we crawl the kernel-related CVE information from Google's Android security bulletin [1]. This includes CVE numbers, specific patch commits, and the corresponding repositories in which the patches were committed.

(2) Patch locator. This is to analyze type 1 target kernels (*i.e.,* repositories). It attempts to determine if a given patch (or a similar one) exists in a target kernel repository (§4.1). If so, it outputs the corresponding patch commit in the repository, which then also serves as the reference in the patch presence test for type 2 and type 3 kernels.

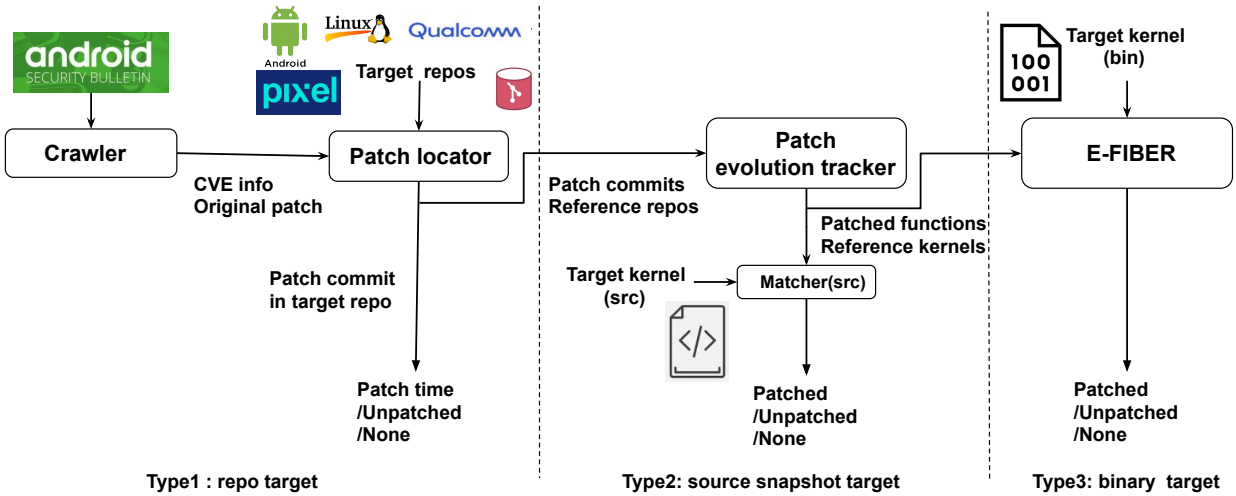

Figure 2: Measurement pipeline.

(3) Patch evolution tracker. The tracker tries to collect all possible versions of a patched function (*i.e.,* the function can continue evolving after the security patch) in the repositories, this can help us reliably test the patch presence in both type 2 (*i.e.,* source snapshot) and type 3 (*i.e.,* binary) kernels.

(4) Source-level matcher. It tries to match each patched function version (identified by the evolution tracker) to the target function in a type 2 kernel, in order to perform a source-level patch presence test (§4.2).

(5) E-FIBER. E-FIBER is capable of translating each patched function version into a binary signature and then matching the signature in type 3 binary kernel as a patch presence test. We build E-FIBER on top of FIBER [42], a state-of-the-art binary patch presence test system. We will articulate the improvements we made over FIBER in §4.3.

4 Patch Presence Test

In this section we will detail the methodology of patch presence tests against the three kernel types.

To better facilitate the discussion of this paper, we call the patch linked in Android security bulletin the "linked upstream patch", which can only be in type 1 kernels (repositories), *i.e.,* Linux, Android commons, Qualcomm. Interestingly, later we find that these may not be the earliest patches.

4.1 Repository Target

When our target is a repository, we search through the commit history using the patch locator to test the presence of an equivalent patch.

Patch locator: We combine various information about the original patch to determine its presence in the target repository. Specifically, we have the following procedure:

1) For each commit, we attempt to perform a simple string match on the commit subject. If it is a patch they borrow from the upstream, the downstream kernels typically retain the original subject. If there are multiple hits, we use the commit message to identify the real match. Typically, the downstream kernels will not only copy the original commit message but also reference the upstream commit, *e.g.,* `cherry picked from commit XYZ`. If no results are found, we perform the second step.

2) When commit subject and message are not retained when applying the same patch in downstream, we search through the commit history of the corresponding patched file, attempting to match the complete source level changes (including both the added/removed lines as well as the context lines) with those in the original patch. If still no match, we move to the next step.

3) It is possible that the downstream kernel has customized the patched function and its context lines no longer match those in the original patch. We therefore also attempt to match the added and deleted lines only (ignoring the context lines). However, if still no results are found, we keep the commits that matched with at least some blocks of added lines (which we call "change sites") in the original patch.

In any of the above steps, if there are multiple results returned, we manually identify the correct one by inspecting the commit message (note that the message is no longer exactly copied else the first step would have caught it). In addition, if no match is ever found after all the steps, we attempt a manual search using parts of the message of the original commit as a last resort. Only if this step fails to locate any commit will we determine the commit is missing. In practice, we find these cases that require manual analysis are small (6.8% in our

experiments).

In addition, there are several special cases we need to pay attention to:

(1) File path/name change: If we cannot find any commits that change the patched file, we extend the search region to files that have the same name but in different directories (sometimes the downstream kernel would decide to rearrange certain source files). If we find any commit that renamed the patched file at some points, we also track the evolution of the renamed file.

(2) Function name change: similar to file names, the name of a function may also change over time. We develop a small script to track the evolution of them too by checking the related commits.

(3) Patched at initialization time: sometimes a kernel repository or branch may choose to copy the entirety of a source file and commit it as a brand new file. In that case, we lose the actual commit that applied the patch. However, we can still match the change sites given in the original patch.

Finally, we note that there can be several reasons when a patch is not found: 1) the patched file/function simply doesn't exist in this branch (*e.g.,* a vulnerable qualcomm driver is not used in Huawei devices), 2) the vulnerability does not affect the particular branch/repository, 3) The vulnerability fails to be patched. In our evaluation, we consider a CVE not applicable for a particular target if it falls under case 1).

4.2 Source Code Target

For kernel source snapshots, we need a way to check its source code against the patched version and infer the patch presence. A naive approach is to match the patched function from upstream against the same function in the snapshot. However, there can be multiple versions of the patched functions (*i.e.,* due to further commits to the same functions), and we do not know which version the target may take (regardless of whether it is source code or binary target). Even worse, the patched function name or patched file may change altogether as mentioned previously.

Our solution to this problem is straightforward. In addition to the single version of a patched function, we choose *multiple versions of the patched function* to represent the patch of a vulnerability. In general, we have two criteria to select the versions we should consider:

(1) Complete. We should be able to discover all patched versions of a function — unless the version is internal to the OEM and not visible in the upstream kernel repositories due to vendor-customization.

(2) Unique. The patched version should not occur in the unpatched version of the kernel. Otherwise, it no longer can distinguish the patched and unpatched cases.

Patch evolution tracker: In order to generate a complete set of patched function versions, we need to pick one or more reference kernels first where we can track the evolution of a function post-patch — this means that we must use kernel repositories with commit history as reference kernels.

In this paper, we choose the repositories from Qualcomm as our reference kernels. This is because Qualcomm has the largest market share as a chipset vendor and therefore is the direct upstream of most Android devices. If a bug is fixed in Linux or Android common kernels, they should also exist in Qualcomm; in other words, Qualcomm has a superset of patches.

Qualcomm maintains different repositories for several major kernel versions (*e.g.,* 4.4 and 4.9). Within each repo, there is typically a "general release branch" (which we simply refer to as mainline) and multiple "stabilization branches" (which we refer to as stable) exist [16]. A stable branch usually corresponds to specific chipsets and OS versions (*e.g.,* Android 8.0) and only port fixes from the mainline. For example, branch `kernel.lnx.4.4.r34-rel` in repo `msm-4.4` has tags sharing a prefix of `LA.UM.7.2.r1` which corresponds to snapdragon 660 and Android 9.0 [17].

As any OEM kernel either forks from or follows a corresponding Qualcomm stable branch (which determines the chipset) and Qualcomm repo (which determines the kernel version), we choose the reference repo according to its kernel version. In practice, this minimizes the differences between the two and improves the accuracy of the patch presence test.

After choosing repositories, we need to determine in which branches to track the patched functions. In principle, we could choose all the branches (including mainline and stabilization) but it may be unnecessary and time-consuming. Instead, we choose the mainline branch only for the following reasons: 1) Generally, vulnerabilities are patched in the mainline first and then propagated to the chipset-specific branches. Due to delays, the patch may not even exist in a chipset-specific branch but we cannot rule the vulnerability out. 2) We prefer to generate generic signatures which are not overly-specific; otherwise there may be too many signatures to generate in the end. In §5, we will show that this strategy produces satisfactory accuracy.

Source-level matcher After collecting the different versions of the patched functions in the corresponding repository, *e.g.,* Qualcomm 4.4, we need to compare them against the function in the target kernel. There are several ways to do so, *e.g.,* hash-based methods [15], a straightforward string match of a few representative lines (*e.g.,* changes made in the patch) in the function, or even a simple string match of the whole function.

We decide to use the most strict and simplest method

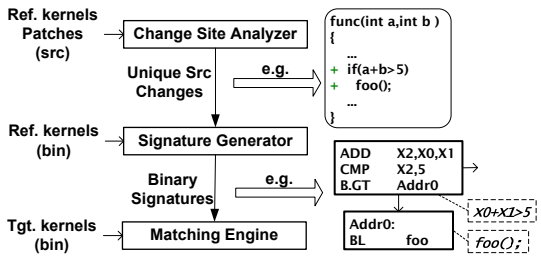

Figure 3: Fiber Workflow

— strict string matching of the whole patched function (using all the evolved versions post-patch) after stripped trailing white spaces for the following reasons: 1) It is strict and never produces any false positive, *i.e.,* if we claim that a function is patched, it must match some version of the patched function (and not any unpatched version). 2) The method is simple and easy to reason about. While it does produce false negatives, *e.g.,* the target kernel may customize the patched function so that it looks different but still patched, we find that these cases are uncommon and we are able to manually analyze them (given that we have the target kernel source).

4.3 Binary Target

If the target is a binary, neither of the previous two methods works. The key challenge is that the patched functions at the binary level are unlikely to be identical even if their sources are the same. This is because of various kernel and compiler options that can influence the compiled binary instructions. Therefore, we choose to generate binary signatures (in the patched function) to test the presence of patch in the target. The signature is what represents the semantics of a patch.

Specifically, we build an improved version of FIBER whose original workflow is illustrated in Figure 3. There are three main steps: 1) it first analyzes a patch (*i.e.,* changes made in one or more places) and checks the uniqueness of each change site. Then it picks a few suitable change sites for signature generation. 2) FIBER compiles the kernel and extracts relevant sequences of instructions (and even symbolic formulas involving the computation of variables) representing the semantics of these change sites. 3) FIBER matches the signatures against a target binary.

Unfortunately, there are several limitations acknowledged and summarized in the original paper: 1) Function inline. (2) Function prototype change (3) Code customization. (4) Patch adaptation. (5) Other engineering issues. We observe that several of these issues share a common root cause: *patched functions evolve over time* and FIBER picks only the initial version of the patched function for signature generation. This means that if the release date of the target kernel and the original patch differ significantly, the generated signature is likely out-of-date for the target kernel. In our preliminary evaluation of FIBER spanning 3 years of reference and target kernels, we find that its accuracy dropped considerably compared to what was reported in [42] due to this issue.

To overcome this limitation, we simply leverage the patch evolution tracker (proposed earlier) to identify the multiple versions of the patched functions so that a more complete set of signatures can be generated. This is especially important when the change sites of the original patch are completely erased during the evolution of the patched function.

In addition, we also address two other technical problems mentioned earlier: (1) the patched function becomes inlined, and (2) the binary signatures look different for the same source due to different compilers and configuration options (FIBER has some degree of robustness but can still be affected as discovered in our preliminary analysis).

Function inlining can cause a direct failure in locating the patched function in the reference binary (missing from the symbol table) and therefore failure in generating the signature.

Our solution is as follows: we try to find the caller of the patched function which should contain the inlined version of the patched function. If the caller is also inlined, then we will recursively locate the caller of the caller until one is found in the symbol table. Since the reference kernels are compiled by E-FIBER, we can make use of debug information to locate the exact sequence of instructions that belongs to the patched function (which is inlined), and generate the signatures (which are now in the context of a caller) accordingly. This signature can then be matched in the target kernel which has the same inlined behavior.

To address the compiler and configuration issues. We vary these configurations ahead of time in generating the binary signature.

(1) Compilers. Most vendors use GCC to compile their source code, however, a few new devices released in 2019 (whose corresponding Linux versions are 4.14) use Clang. Different compilers can yield vastly different binary instruction sequences to the point it becomes hard to semantically test the equivalence of the two. As a result, we use both compilers to compile 4.14 reference kernels and generate two versions of signatures.

(2) Optimization levels. Through sampling a few kernel source snapshots from major OEM vendors, we find that all of them use either Os or O2 as the compiler optimization levels. We, therefore, generate signatures with both optimization levels.

Type of target	Company	Repo (Num of branches) or Phone models (Num of Roms)
Repository	Linux	Linux(mainline, linux-3.18.y, linux-4.4.y, linux-4.4.y, linux-4.14.y)
	AOSP common	Android common(android-3.18, android-4.4, android-4.9, android-4.14)
	Qualcomm	msm-3.18(8), msm-4.4(17), msm-4.9(15), msm-4.14(1)
	Pixel	Android msm (Pixe l, Pixel 2, Pixel 3)
Binary	Samsung	Galaxy S7(78), Galaxy S8(52), Galaxy S9(32), Galaxy Note9(28), Galaxy A9 Star(11), Galaxy A8s(9)
	Xiaomi	Mi 6(84), Mi8 Lite(24), Mi 8(12), Redmi 4(41), Redmi 4pro(38), Redmi Note7(21), Mi Max2(75)
	Huawei	Mate 10(37), P20 pro(31), Honor10(30)
	Oppo	R11s(11)
	LG	V30(10)
	Oneplus	Oneplus5(27), Oneplus6(18)
Source snapshot	Sony	XperiaXZ1(23)
	Samsung	Galaxy S8(1), Galaxy S9(1)
	Xiaomi	Mi 8(1), Mi 9(1), Mi Max2(1), Redmi Note7(1)
	Huawei	Mate 10(1), P20 pro(1)
	Oppo	FindX(1)

Table 1: Dataset of measurements

	repository	**Num. CVEs**
1	Linux	141
2	Qualcomm msm-3.4	12
3	Qualcomm msm-3.10	52
4	Qualcomm msm-3.18	115
5	Qualcomm msm-4.4	63
6	Qualcomm msm-4.9	15
7	AOSP msm	2

Table 2: Corresponding repository of CVE in Android security bulletin

(3) Configuration files. Besides optimization levels, other kernel configuration options (to enable and disable certain kernel components) vary. In the mainline branch of Qualcomm repos (*e.g.,* 4.4 or 4.9), there are typically a few config files. For example, msm-4.9 has 16 config files in total and only 8 of them are specific to Android chipsets, including `sdm845-perf_defconfig` (Snapdragon 845), `msm8937-perf_defconfig` (Snapdragon 430), etc. We pick only the config files that are relevant to the Android devices we are interested in testing. For example, snapdragon 845 is used in Mi 8. Thus `sdm845-perf_defconfig` is used to generate the corresponding signatures.

5 Evaluation

5.1 Dataset

Overall, we collected 402 kernel CVEs released on Android Security Bulletin every month since its inception in Aug 2015 until May 2019. This includes the main bulletin [1] as well as a Pixel bulletin [5]. We summarize the crawled CVEs in Table 2. Clearly, most of them link to Linux and Qualcomm instead of AOSP Android repositories.

We also summarize the target kernels used in our evaluation in Table 1. Overall, we collected 3 levels of upstream kernels as introduced before, *i.e.,* Linux, Android common and Qualcomm). 8 most popular Android brands (Google Pixel, Samsung, Xiaomi, Huawei, Oppo, OnePlus, Sony, LG), covering 26 phone models and 701 released kernel instances (either source or binary). For most phone models, the kernel instances cover a time range of one to two years. We collect these kernels through both official and third-party websites. Our experience is that most official websites supply only the latest ROM for each phone model, and occasional source snapshots. The one exception is that SONY offers all source code snapshots on its websites. To obtain historical versions of ROMs, we rely mostly on third-party websites [11, 12, 7, 8].

We extract compilation dates (*i.e.,* build dates) from these ROMs which are used to compare against various dates such as Android security bulletin release date and patch dates on the upstream. Note that we collect many historical kernel versions (*e.g.,* 78 versions for Samsung Galaxy S7) for the same phone model in order to conduct a longitudinal study on their patching behavior.

To generate robust signatures using E-FIBER (see §4.2 and §4.3), we have used in total 19 different config files from msm-3.18, msm-4.4, msm-4.9, and msm-4.14 Qualcomm repos that represent the chipsets encountered in our OEM devices. We use two compiler optimization

Device	Kernel Version	Source code						Binary					
		Cnt.	TP	TN	FP	FN	Accuracy	Cnt	TP	TN	FP	FN	Accuracy
Samsung S8	4.4.78	351	257	59	0	35	90.03%	246	202	37	0	7	97.15%
Samsung S9	4.9.112	302	293	3	0	6	98.01%	189	180	2	0	7	96.30%
Xiaomi Mi8	4.9.65	232	208	23	0	1	99.57%	168	149	15	0	4	97.62%
Xiaomi Mi9	4.14.83	262	258	3	0	1	99.62%	173	165	1	0	7	95.95%
Redmi Note7	4.4.153	356	342	13	0	1	99.72%	265	255	7	0	3	98.87%
Xiaomi Max2	3.18.31	328	217	88	0	23	92.98%	208	155	45	2	6	96.15%
Huawei P20	4.9.97	137	114	12	0	11	91.97%	83	76	5	0	2	97.59%
Huawei Mate10	4.4.23	147	74	67	0	6	95.92%	86	53	26	2	5	91.86%
Oppo FindX	4.9.65	235	210	19	0	6	97.45%	186	171	12	0	3	98.39%

Table 3: Accuracy of patch presence test

settings: -Os and -O2. We also need to account for patch evolution. In the end, we compiled a total of 2,488 reference kernels all from Qualcomm repos with 11,093 signatures generated in the end (note one compilation allows multiple signatures to be generated).

5.2 Accuracy

In this section, we will describe the accuracy of patch presence test against three types of kernel targets presented in §4.

First of all, for kernels that are in the repository form, since we have conducted both automated and manual analysis (for the few subtle cases) exhaustively on every CVE and every branch, we treat the results as ground truth.

For kernels that are in source snapshots or binary ROMs, we sample a number of them to evaluate the accuracy of the patch presence test at both the source and binary level. Specifically, we picked 9 kernels, each from a different phone model covering 4 different brands. These 9 kernels are available in both source snapshot and binary, which allows us to verify the results of binary patch presence test using the corresponding source code. The results are summarized in Table 3. Generally, our solution works well for both source and binary targets with an average accuracy of more than 96%. To give more details, we also analyzed the sources of inaccuracies.

In the case of source snapshot targets, since we consider a function patched only when a strict string match of the full function is found, it leads to no false positives but some false negatives are observed, which are due to customization of the patched functions. The results suggest that Huawei and Samsung have more customization than others. This is consistent with the fact that Samsung and Huawei are the top 2 players in the Android market and have the strongest product differentiation.

In the case of binary targets, the inaccuracies come from 1) Customization of the patched function. 2) Even when source code is the same, the binaries may look different due to vendor customization of compiler's config options, which we do not have complete access to (other than those from the periodic source snapshots). Interestingly, we can see generally comparable and even lower false negative rates compared to the source snapshot targets. This is because the source-level patch presence test is based on strict string matching of the whole patched function (and will fail to match any vendor customized functions). On the other hand, FIBER by design has some resistance against customization as the generated signatures only characterize a small (but key) portion of the patched function.

Besides, the number of CVEs and their corresponding patches that we can track for binary kernel targets is smaller. One common reason is that many vulnerable drivers are included in the source snapshot but are not compiled into the binaries. Other technical reasons are: 1) FIBER was not able to generate signatures for certain cases. 2) Generation/Matching of signatures costs too much time (over a threshold of 2 hours, which is determined by the distribution of time consuming we observed). These cases attribute to about 10% of the CVEs and were excluded from the binary patch presence test.

Overall, the patch presence test accuracy result gives us confidence in the measurement study in §5.4. We also note that patch presence test in upstream source repos is independently done through patch locator as described in §4.1.

5.3 Patch Propagation in Upstream kernels

In this section, we focus on analyzing the patch propagation in the upstream kernel repos using the patch locator described in §4.1. With the exact time and date of individual commits, we are able to track the patch

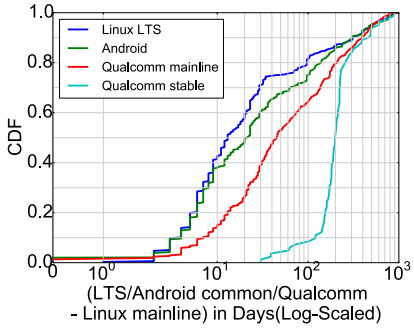

Figure 4: Upstream patch delays (Linux CVEs)

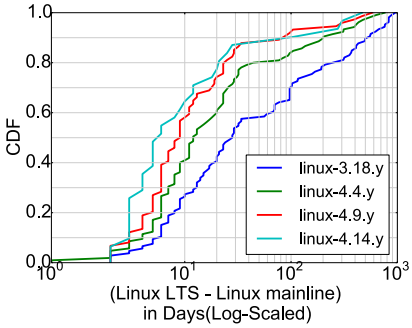

Figure 5: Linux mainline to LTS (Linux CVEs)

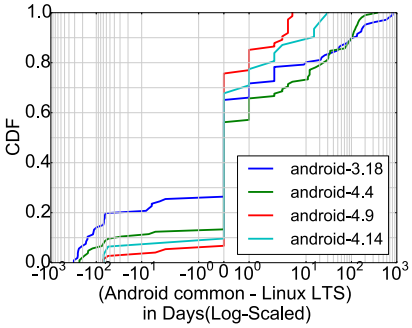

Figure 6: LTS to Android common (Linux CVEs)

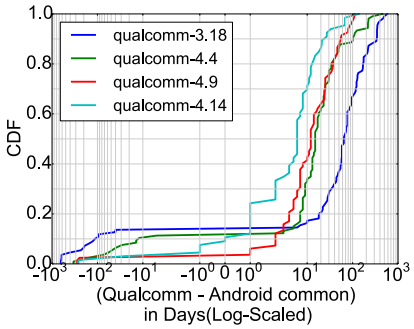

Figure 7: Android to Qualcomm mainline (Linux CVEs)

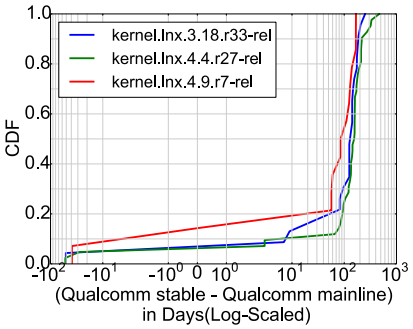

Figure 8: Qualcomm mainline to stable (Linux CVEs)

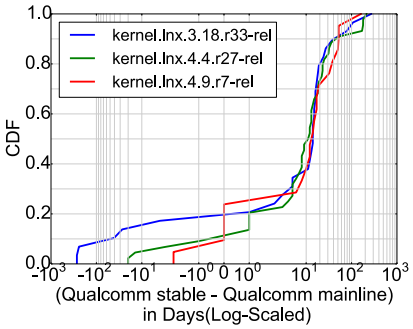

Figure 9: Qualcomm mainline to stable (Qualcomm CVEs)

propagation precisely and make a number of interesting observations about both Linux and Qualcomm vulnerabilities.

Figure 4 gives an overview of the cumulative patch delays observed at each layer with respect to Linux mainline (here all included CVEs affect Linux). As we can see, Linux internally (mainline $\rightarrow$ LTS) already has a substantial delay, with 20% of the patches being 100 days or longer. On the other hand, Google does a good job in tracking Linux vulnerabilities, as the line representing the Android common's patch delays is closely aligned with that of Linux LTS. Qualcomm's mainline is noticeably slower in picking up patches from its upstream (note the log-scale nature of the X-axis). Finally, we find that Qualcomm can be considered the bottleneck as it is extremely slow in propagating most of its patches from mainline to stable branches. For about half of the cases, the Qualcomm-internal propagation delay is at least 2 to 3 months. From the end-to-end point of view, the majority of patches take over 100 days for them to propagate from Linux mainline all the way to Qualcomm stable. About 15% of the patches took 300 or more.

If we break the result down further layer by layer, Figure 5 shows the delay incurred in Linux internally (mainline $\rightarrow$ LTS) across all four major kernel versions 3.18, 4.4, 4.9 and 4.14. We see 5% to 25% of patches experience a delay of 100 days or longer (with 3.18 being the worst). In extreme cases, after patched in Linux mainline, CVE-2017-15868 is not patched in Linux LTS 3.18 until 954 days later. Not too long ago, a critical vulnerability CVE-2019-2215 was not patched in Linux LTS 4.4 until about 600 days later, ultimately leaving most downstream OEM kernels such as Pixel2 and Samsung S8/S9 vulnerable [25].

The case for Linux LTS $\rightarrow$ Android common (Figure 6) is different and interesting. The delays are much smaller where more than half of the CVEs are patched in Android common the same day as Linux LTS or earlier. When we look into the reason, we find that the maintainer of Linux LTS, Greg Kroah-Hartman, also helps maintain the Android common repository (note the large fraction of 0-day delay cases). After merging commits from mainline to LTS, he usually merges commits from LTS to Android common repository right away. The other thing worth noting is that about 10% – 20% of the patches are applied in Android common first and then appear in LTS, exhibiting negative delays. This is because Google has been diligently scouting for important security patches everywhere, sometimes picking up patches from Linux mainline directly and bypassing the slow Linux LTS. Google is capable of

doing this because (1) they hire many engineers who are also Linux maintainers, and (2) Google offers a bug bounty program and thus many Linux bugs are reported to Google first who typically tries to get Linux mainline to patch first and then port it immediately (according to the feedback we received from Google).

The case for Android common → Qualcomm mainline (shown in Figure 7) is similar in the sense that also about 5% – 20% of the patches are observed in Qualcomm first and then Android common. Similar to Google, Qualcomm also independently ports patches from Linux mainline. Interestingly, this means that even after Google picked up patches from Linux mainline directly, there are additional mainline patches missed by Google which are picked up by Qualcomm directly.

The last step in the pipeline is about the Qualcomm mainline branch (e.g., 3.18) to its corresponding stable. As shown in Figure 8, we pick three representative stable branches that correspond to the Android devices and OS versions we will analyze (recall that stable branches are specific to chipsets and Android OS versions). We note that other branches yield similar results (except those ones with insufficient history). We excluded all 4.14 stable branches because they are too new to have sufficient history. Overall, we can see that the delay is very substantial compared to the earlier steps. For 4.4, about 80% of the patches are delayed for 100 days or longer and 20% delayed for 200 days or longer. 4.9 is somewhat better than 4.4 with 80% of the patches delayed for 60 days or longer. Both are far worse than the internal delays in Linux (Figure 5). Interestingly, the 3.18 stable branch shows a comparable delay to 4.4 (and even slightly better) — a sharp contrast with the previous step that the Qualcomm 3.18 mainline being the slowest among all other mainlines (shown in Figure 7). Upon closer inspection, this is due to an older patching practice for the Qualcomm 3.18 repo which we will discuss in detail in §6.

In summary, for vulnerabilities that originate in Linux, we pinpoint the *internal propagation delays* within Qualcomm and Linux (*i.e.,* mainline to stable/LTS) to be clear bottlenecks. In addition, we find that newer kernel versions (from 3.18 to 4.14) generally correspond to more timely patch propagation across all these layers. The improvement however appears to have stabilized since 4.9.

Finally, we also inspect vulnerabilities that originate in Qualcomm — they constitute more than 60% of the CVEs as shown in Table 2. Surprisingly, as shown in Figure 9, the patch delays seem abnormally small compared to the Linux vulnerabilities (Figure 8). We suspect this is because Qualcomm is much more aware of the vulnerabilities specific to its own code, *i.e.,* triaged and analyzed internally, and thus can react faster. We will provide more evidence to support this in §6.

5.4 Patch propagation to Android OEM phones

In this section, we follow the patch propagation pipeline to OEM vendors using a variety of Android devices as described in §5.1. We are primarily interested in measuring the patch delay and understanding generally whether OEM delays represent the bottleneck in the end-to-end patch propagation. In addition, these Android devices are produced and maintained by different companies, marketed as high-end or low-end phones, and released in diverse geographic regions. We therefore also examine how these factors may influence the patching behavior. For most phones, we are able to retrieve a continuous stream of firmware images (one image per month according to build dates). Thus we can pinpoint when a patch is applied.

Figure 10 shows the patch propagation delay from Qualcomm stable to OEM phones (aggregated over all the phones). For every OEM phone, we pick one or more corresponding Qualcomm stable branches as upstream with the matching chipset and Android OS versions (note a phone may upgrade its Android OS version during its lifetime). As we can see, for Qualcomm-specific vulnerabilities (in dotted lines), OEM phones fall behind Qualcomm stable significantly — the delay is 100 days or more for 70 - 90% of CVEs. On the other hand, for vulnerabilities that originated in Linux, we find that the delays are noticeably smaller. This is due to Linux vulnerabilities being patched much earlier in upstream (Linux and Google's Android common) and therefore OEM vendors do not necessarily need to wait for patches to propagate to Qualcomm stable. For example, they could be notified by Google earlier.

Next, we also plot the end-to-end delay in Figure 11 by adding up delays in each propagation layer in the whole ecosystem. Here the earliest patch is either Linux mainline or Qualcomm, depending on whether the vulnerability is originated from Linux or Qualcomm. Generally, both cases incur significant delays with Linux vulnerabilities being generally worse. This is understandable because a Linux patch naturally has a longer propagation chain compared to a Qualcomm patch. As we can see, more than half of the Linux CVEs are delayed for 200 days or more, and 10% to 30% of CVEs are delayed for more than a year. This is an unacceptably long delay that allows experienced hackers to craft exploits against unpatched OEM devices. CVE-2019-2215 is one such example [24].

Next, we analyze a number of factors that might influence the patch delays in OEM phones.

- **Vulnerability severity.** Intuitively, more severe

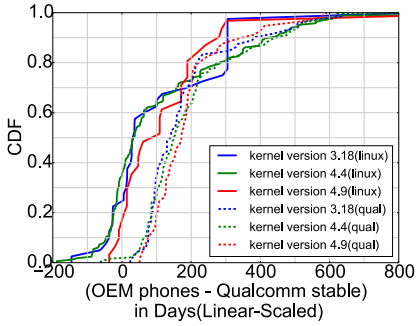

Figure 10: Delay between Qualcomm stable and OEM phones

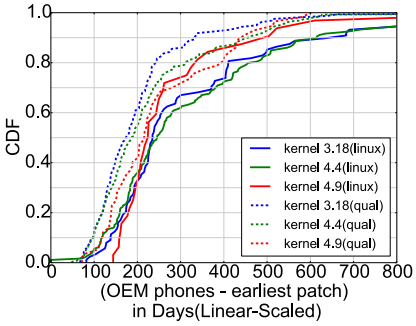

Figure 11: End-to-end delay between earliest patch and OEM phones

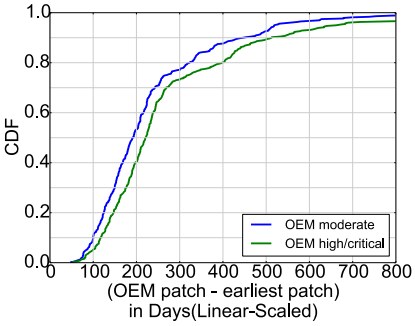

Figure 12: End-to-end delay between earliest patch and OEM phones

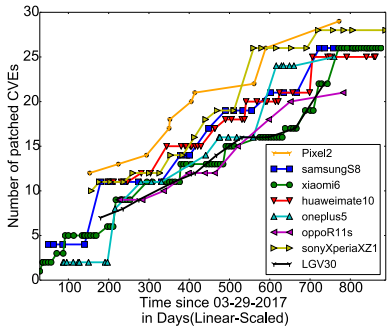

Figure 13: Different OEM vendor comparison

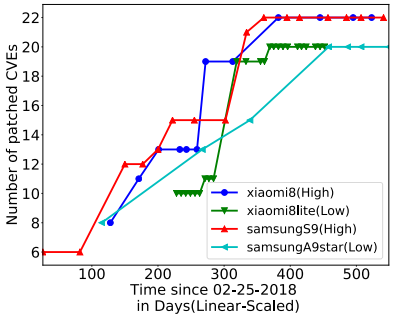

Figure 14: High/low-end phone comparison

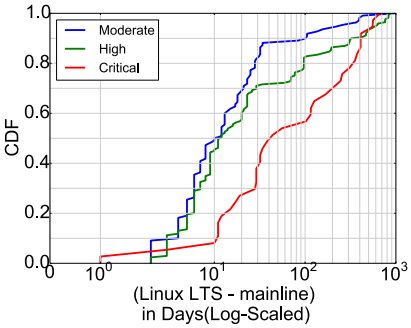

Figure 15: Patch delays from Linux mainline to LTS (by severity)

vulnerabilities should be patched sooner rather than later by OEM vendors (or upstream). However, as shown in Figure 12, the result is not supportive. Specifically, we plot the distribution of end-to-end patch propagation delays by vulnerability severity levels. In §6, we will offer a much more detailed explanation of the phenomenon (after reaching out to Google). Note that there are only 33 critical CVEs from the security bulletin, and 30 of them are very old (originally patched before 2017) not applicable to many of the new OEM devices. Thus we combine them with high severity CVEs.

• **Name brand.** To do a fair comparison, we sample 8 phones from 8 first-tier companies which are all high-end and released in 2017: Google Pixel2, Samsung S8, Xiaomi Mi 6, Huawei Mate 10, Oneplus 5, Oppo R11s, SONY Xperia XZ1 and LG V30. Their corresponding kernel versions are also the same — 4.4.y. We only compare the CVEs that affected all target phones and ignore the CVEs patched beforehand. As seen in Figure 13, the results show that Google Pixel 2 and SONY clearly did the best. In contrast, Xiaomi, Oppo, and LG are the slowest.

• **High-end vs. Low-end.** This may be an expected result as companies tend to devote more resources to their flagship phones. Figure 14 shows the comparison between high-end phones (Mi 8, Galaxy S9) and low-end phones (Mi8 Lite, Galaxy A9 star) in Samsung and Xiaomi.

• **Geographic locations and carriers.** We did a small sample analysis of Samsung and Huawei phones, and the results show that the same kind of phone (only with minor adjustments, *e.g.,* for local carriers) in different regions got patched at the same time in most cases, with about only 10 percent of the cases being slightly different.

• **Time after release.** Android devices are known to have a relatively short support lifetime, *e.g.,* Google phones now offer mostly 3 years of security updates [22]. In practice, most phones (especially high-end ones) do indeed enjoy at least 2 years of support. A major exception is Xiaomi's Redmi 4, a popular low-end phone popular in China and India. It was released in 2017 and still had some updates (*i.e.,* new firmware images) until March 2019. However, surprisingly it stopped patching any security vulnerabilities since early 2018 (less than a year).

6 Causes of Patch Delays

So far, we have quantified the patch delays in the Android kernel ecosystem mostly in a "blackbox" manner. However, other than blaming the long chain of patch propagation, we have not explored the reasons why the delays are so profound. This can be illuminating for future improvements in patching practices.

To achieve this goal, we collect additional information to help explain the rationale behind the patching practices by each participating party in the ecosystem. Specifically, we will analyze the security bulletins released by more organizations (Qualcomm), extract more details related to each patch commit, and reach out for information to the various parties including Google, Qualcomm, and Samsung.

From an intent point of view, a security patch can be applied in either of the two ways: *knowingly* or *unknowingly*. For example, an OEM vendor may be notified by Google about a serious security vulnerability and knowingly look for patches from upstream. On the other hand, Google may be blindly applying all upstream commits from Linux LTS to Android common branches, not knowing which are important security patches. Understanding the intent will provide valuable insight into the patching delays.

Based on this basic framework, we propose the following hypotheses to explain the slow patching.

(1) Even though the Android kernel ecosystem is largely open-source, the "knowledge of a security vulnerability" is often lacking and does not traverse the ecosystem fast enough, preventing security patches from being recognized and "knowingly" picked up by those who are affected (e.g., OEM vendors).

(2) A downstream kernel branch may have drifted from the upstream (*e.g.,* customization in downstream), it is not always possible to blindly apply all upstream commits (conflicts can arise). This may cause some kernels to lower the frequency to "sync" with upstream kernel branches, reducing the possibility of "unknowingly" patching a vulnerability in time.

To validate the hypotheses, we look into detailed commit log of kernel repositories. As all kernel repos (*i.e.,* Linux, Qualcomm, and Android common) are managed by git, we are able to differentiate through the commit log whether an upstream patch is knowingly "cherry-picked" or unknowingly "merged" (together with a stream of commits) into a downstream kernel branch. They correspond to the command `git cherry-pick <upstream-commit>` and `git merge <upstream-commit>` respectively. The semantic of cherry-pick is to pick a specific upstream commit and port it over to downstream, whereas merge pulls all the commits since last divergence up to `<upstream-commit>`.

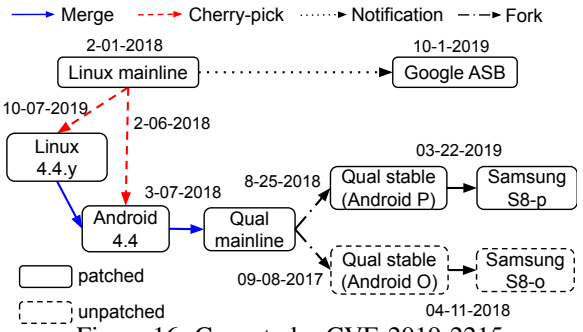

Figure 16: Case study: CVE-2019-2215

Cherry-pick is more flexible as it can patch specific vulnerabilities without influencing other features. However, it requires knowledge about which upstream commit corresponds to an important security patch. In other words, the downstream must either be notified about the patch or identify the security issue proactively.

Merge treats all upstream commits equally and does not differentiate between security patches (severe or not) and other bug fixes. If done frequently enough, patch delays can be effectively reduced. The drawback is that manual resolution is needed when merge conflicts occur.

Similar to merge, fork is sometimes used by a downstream to become a clone of an upstream. This way, the downstream automatically inherits all the patches applied in the upstream at the time of fork. The drawback is if any customization is made in downstream, however, it needs to be ported over to the newly forked branch.

Next, we use a case study of a known CVE to demonstrate when these patch operations are performed, and how they can help explain the patch delays.

Case study. In Figure 16, we illustrate the above patch operations using CVE-2019-2215, a serious vulnerability that allows rooting [25] which was originally patched in Linux mainline on 2/1/2018. The cherry-pick by Linux 4.4 LTS occurred on 10/7/2019 with a long delay. Notably, Google's Android common 4.4 branch proactively cherry-picked the patch from Linux mainline on 2/6/2018 (bypassing its direct upstream). Unfortunately, Google does not appear to be aware of how serious the vulnerability is, evident by the extremely late Android security bulletin announcement on 10/5/2019 (an 18 months delay) and Google's public statement admitting them being informed by the project zero team on 9/26/2019 [24]. It is also worth noting that no CVE was issued prior to the point. During this time, Qualcomm was uninformed about the vulnerability either. Its stable branch `kernel.lnx.4.4.r27-rel` did not cherry-pick the patch, leaving the corresponding Samsung S8-Oreo (Android 8.x) to be vulnerable all

this time [25].

On the other hand, Qualcomm stable branch `kernel.lnx.4.4.r35-rel`, representing the same chipset with an upgraded Android Pie (9.x) had been merging updates from android-4.4 periodically (merge is preferred in Qualcomm stable prior to its release), thus patching the vulnerability on 3/7/2018. Luckily, when Samsung S8 upgraded its OS from Oreo to Pie, it forked from this stable branch, inheriting the patch unknowingly. Unfortunately, other OEM phones using the same chipset (and staying on Android Oreo) will remain vulnerable unless they cherry-pick patches elsewhere. In fact, we have checked that `kernel.lnx.4.4.r27-rel` never bothered to apply the patch until the end of its lifetime on 1/22/2020.

The case study gives us good insight on how the patching process is like in the ecosystem. Next, we will generalize the insight by analyzing each step of the propagation closely and offer takeaways and suggestions on how to improve the ecosystem.

1. Linux community. Linux vulnerabilities are always first patched in Linux mainline and then cherry-picked by downstream branches. Since Linux stable/LTS branches aim to operate as reliably and stably as possible, there is a formal set of rules guiding the cherry-pick of upstream patches [3], *e.g.,* "it cannot be bigger than 100 lines, with context; it must fix a real bug that bothers people, ... a real security issue".

Thanks to the close collaboration between Linux mainline and stable maintainers and the fact they belong to the same community, patch delays between the two are generally small. The outlier 3.18.y was noticeably slower than others. It turns out that other than the fact that it is an older branch, it was never meant to be an LTS branch. However, due to popular demand from Android kernels which decide to fork from 3.18.y, it remains actively maintained for much longer than originally intended. This may partially explain the slow cherry-pick of upstream patches. In other LTS branches, patch delays are generally small despite a long tail.

Unfortunately, due to the general principle followed by Linux that "a bug is a bug" [6], oftentimes the Linux community does not realize whether a bug is truly an exploitable security bug until much later. By convention, security patches in Linux are not labeled as such in the public commit logs [23]. This creates a situation where Linux LTS maintainers are not even aware of the impact of those vulnerabilities. As supporting evidence shown in Figure 15, counterintuitively, CVEs that are (later) rated as critical and high by Google turn out to take noticeably longer time for Linux to patch, indicating the lack of knowledge by Linux. In fact, we find 17 out of 37 patches for critical vulnerabilities were initially missed in the initial "train" of cherry-picked patches, as they appear "out-of-order" with respect to other cherry-picked patches.

Propagation step	3.18	4.4	4.9	4.14
LTS ->Android	63/106	74/105	70/74	30/31
Android ->Qualcomm	26/95	93/109	72/74	61/66

Table 4: The ratio of CVEs patched by merge

Even when Linux is aware of a security vulnerability, *e.g.,* notified by an external party via the private vulnerability reporting mailing list, `security@kernel.org`, this knowledge may or may not propagate internally to Linux LTS maintainers. In addition, as Linux's commits are often intentionally opaque [23], the knowledge is almost definitely lost outside of Linux, preventing downstream kernels from cherry-picking the corresponding patches timely. The only publicly available mechanism to document such knowledge is the CVE database. However, it is known to be incomplete and takes a long time to assign a CVE number and to update the entry [6].

Therefore, a better mechanism to track security issues is needed. Specifically, for the vulnerabilities that are reported to Linux through its private mailing list, we argue that it is a big missed opportunity where Linux has already triaged the bug and can clearly label the corresponding fixes as security-critical to help the downstream kernel (this is much more efficient than the CVE mechanism). For other bug fixes, we call for better tools to automatically reason about the nature of a bug and determine if it has serious security implications — a recent tool has been developed by Wu et al. [36].

2. Google. Android common kernels are forked from Linux stable/LTS initially and then add Android-specific changes on top (sometimes referred to as "out-of-tree" code). Over the years, Google has been upstreaming much of its code to Linux mainline and reducing such "out-of-tree" code [28]. This allows Android common kernels to merge patches from Linux LTS with a delay of 0 day, a week, to a month sometimes, and only occasionally cherry-pick from Linux mainline directly for important security patches. This is evident in Table 4 which shows the exact numbers of patches merged vs. cherry-picked. Note that 3.18 and 4.4 are exceptions as most of the patches in the beginning were cherry-picked from Linux mainline where the delays are less predictable (some are creating negative delays compared to Linux LST).

In addition to keeping its own Android common kernels up-to-date, Google has another important responsibility to notify OEM vendors about security patches. While the exact notification date is mostly not made public, according to Google, it typically goes out

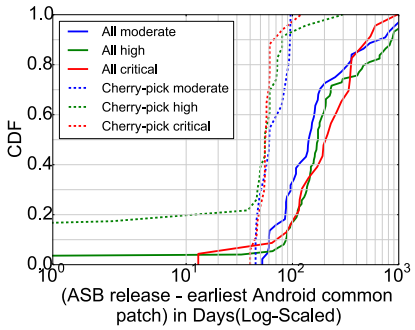

Figure 17: Notification delays of Linux CVEs (by severity)

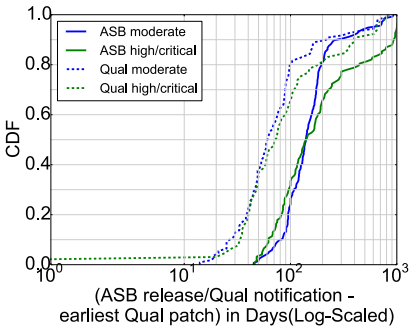

Figure 18: Notification delays of Qualcomm CVEs (by severity)

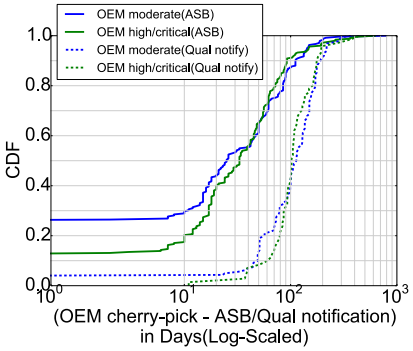

Figure 19: Post-notification delays of cherry-picked patches (by severity)

at least a month prior to the information appearing on the security bulletin [2]. Surprisingly, as Figure 17 shows, in the majority of the CVEs, it takes anywhere from 100 to 500 days for the details to appear on the security bulletin (note that the actual notification should be at least 30 days earlier). In the extreme 20% of the CVEs, it takes 500 days or more. We believe this is due to the fact that Google is not really aware of which of the merged patches are security-critical — indeed the delays shown in the figure do not appear correlated with the severity of vulnerabilities.

In the same figure, we also show the notification delays of CVEs where Google knowingly cherry-picked important security patches. Indeed, the delays are noticeably smaller. This indicates the lack of knowledge is the culprit again, supporting our hypotheses. There is still not too much difference based on vulnerability severity levels. After finishing the analysis, we also confirmed with Google that this is expected as their pipeline does not distinguish severity levels by design. Every month, all issues rated above the threshold and known to Google, *e.g.,* moderate and above, are worked on together in a batch. Exceptions occur only under extraordinary circumstances where disclosure of a serious vulnerability is imminent.

In general, for vulnerabilities that originate in Linux, better and more automated vulnerability triage seems to be a key capability that can benefit Google. Manually sifting through merged upstream commits and narrowing down to the handful that eventually appears on the Android security bulletin can be prohibitively expensive. Alternatively, if Linux has done the triage already, Google can benefit directly from the knowledge, *e.g.,* through tighter collaboration.

For vulnerabilities that originate in Qualcomm, Google should have the first-hand knowledge already — they are almost always informed by either Qualcomm or external parties about the specifics. In such cases, the notification to OEM vendors should be as swiftly as possible, which unfortunately is not the case as we will discuss later in the section.

3. Qualcomm. Qualcomm maintains many more branches compared to Linux and Google and the overhead of patch tracking and management goes up. However, we find its mainline branches are maintained in a similar fashion to Android common. As seen in Table 4, mainlines primarily merge commits from Android common and only occasionally cherry-picks patches from Linux directly. One difference is the merge frequency is generally lower than that of Android common, resulting in longer delays as shown in Figure 7.

On the other hand, Qualcomm stable branches are maintained differently. After they are forked from a mainline and labeled as “release”, only cherry-picks are performed. This creates the same paradox that even though Qualcomm mainlines merge patches relatively timely, the developers are not aware of the security-critical nature of these patches. As a result, it can take Qualcomm stables a long time to cherry-pick the patches. Indeed, Figure 8 illustrates the dramatic delay. Shockingly enough, after we reach out to Qualcomm about the delays, their response indicates that this is because stable branches often receive Linux-specific patches only when customers ask for them explicitly.

In principle, even if Qualcomm is interested in proactively patching Linux vulnerabilities, the knowledge gap needs to be bridged by Linux (*e.g.,* labeling the security nature of a patch). However, Qualcomm can do its part by merging more patches to stable branches without distinguishing their nature, despite the fact that Qualcomm stables are designed to include bug fixes only. This is because Qualcomm stables are already based on Android common branches and indirectly from Linux stable/LTS, which commit

necessary bug fixes only (no new features). Interestingly, we observe two recent stable branches based on Android 10, namely `kernel.lnx.4.9.r34-rel` and `kernel.lnx.4.9.r30-rel` in Qualcomm follow this very strategy.

In contrast, for vulnerabilities that originate in Qualcomm kernels, we know that they are patched much more timely in stable branches (see Figure 8). In such cases, Qualcomm is likely already aware of the nature of the bugs — most are described as externally reported or internally discovered during auditing. Thus Qualcomm should be able to notify OEM vendors as soon as patches are available. Unfortunately, after collecting data from Qualcomm's security bulletin (released monthly since Sep 2017), we found that the delay between the earliest patch and its own notification date is not ideal (median delay: 63 days, mean delay: 130 days), as shown in Figure 18 (surprisingly indiscriminative of the vulnerability severity again). Note that we combine high/critical CVEs into one line here because there are only three critical Qualcomm kernel CVEs since the inception of Qualcomm's security bulletin.

After confirming with Qualcomm, we know that the customer notification is sent out (to all OEM vendors) only after fixes have been widely propagated on affected branches. However, we believe the notification process can be more agile — a subset of OEM vendors can be notified as soon as their corresponding branches have the patches ready. Even better, oftentimes the patches are not really different across branches, Qualcomm can simply notify all customers as soon as the earliest patch is ready and OEM vendors can make an early decision (*e.g.,* testing the patch independently before applying). This way, the major bottleneck of late notification can be mitigated.

According to the same figure, there is another delay of two to three months before Google publishes these CVEs on its security bulletin. Since most OEM vendors follow Google's monthly schedule to update security patch level, OEM patches will be unnecessarily delayed.

4. OEM phones. To understand how patching is performed on OEM kernels, we refer to the Pixel source branches as well as an Oneplus repo that happened to contain the complete commit history. We observe that these kernels cherry-pick patches from Qualcomm (either mainline or stable) and even Linux sometimes. In addition, when OEM vendors decide to upgrade the Android OS (*e.g.,* Android Oreo to Android Pie), they usually abandon the old branch and develop another stable branch (forking from upstream) that corresponds to the new Android OS (as the case study about Samsung S8 showed). We can infer that other OEM vendors follow the same strategy of (1) cherry-picking instead of merging, and (2) forking when upgrading. This is because (1) the firmware images often skip upstream patches (so it is unlikely performing `git merge`), and (2) OS upgrades always happen together with the kernel version updates, which is also the case with Qualcomm stable branches — OS upgrades lead to a new stable branch with an advanced kernel version. In addition, we always observe a large number of kernel patches applied when the firmware is upgraded to a new Android OS.

Specifically, depending on the exact phone model, 30% to 75% of CVEs can be patched through forking a new branch from upstream. This is not a healthy number because Android OS upgrades usually happen on a yearly basis and not to mention that there are often additional delays for these upgrades to reach user devices (*e.g.,* carrier delays). Clearly, more patches should have been cherry-picked in between upgrades.

For the cherry-picked patches, we consider them timely if they are applied within a reasonable amount of time after Google or Qualcomm notify the OEMs, which is typically expected to be a month or two. Unfortunately, OEM vendors are often significantly behind the schedule. As Figure 19 shows, 80% of the Qualcomm CVEs take OEMs 100 days or more to deploy corresponding patches. This is likely because OEM vendors ignore Qualcomm's notifications and prefer to follow the monthly updated security patch level set by Google. We contacted Samsung and confirmed that OEMs are bound to follow Android's monthly bulletin while no such strict requirements exist for Qualcomm. This is reflected in the figure where more than 50% of the CVEs take OEMs less than a month (sometimes even beforehand) to patch after the Android security bulletin publication (which is within the expectations [13]). As we can see, Google's notification plays a huge role in getting OEMs to patch.

We note that there is a small fraction of patches (roughly 5%) delayed for 200 days or more after Google's security bulletin is published. This is not only due to slow and infrequent security updates by some devices but also occasionally skipped CVEs (out of the ones published together in a month). For example, we find that Samsung S8 has skipped nothing but CVE-2018-13900 from Google's Feb 2019's security bulletin, which interestingly got patched eventually in 2020. Finally, from Figure 19, we do not find significant correlation between the severity of vulnerabilities and timeliness of patches being cherry-picked by OEMs. Note that the number of critical cherry-picked patches by OEMs is very limited, especially for some new phones, thus we combine high and critical ones into a single line. In fact, CVE-2018-13900 is a high severity

vulnerability yet skipped by Samsung S8.

To improve the situation, OEM vendors should obviously react more timely to the earliest notification, *e.g.,* Qualcomm. Furthermore, similar to what we suggest for Qualcomm, OEM vendors can consider merging patches directly from upstream instead of cherry-picking them. We also hope that high-end and low-end phones can be treated equally, as we show low-end phones tend to receive patches more slowly in Figure 14. At the end of the day, we believe a better and more automated patching/testing process will help.

Summary. Overall, the analysis supports our hypothesis and we propose three general areas that need improvement.

More efficient triage systems. The triage process of security vulnerabilities today is largely manual. This is evident in the case study where the initial bug fix made in Linux mainline was never treated seriously enough by the rest of the ecosystem (Linux LTS failed to cherry-pick it also). Better automated reasoning tools (*e.g.,* [36]) can assist the developers in identifying security-critical bugs and take actions accordingly.

More efficient knowledge propagation. Unfortunately, even when the knowledge of an important security vulnerability does become available in one party, it either does not have a good mechanism to propagate the information (*e.g.,* Linux), or propagate the information in a delayed manner (*e.g.,* notification by Google and Qualcomm). In addition, sometimes it is beneficial to propagate the knowledge in the reverse direction (*e.g.,* some patches shown to be applied in Google before Linux LTS). Ideally, this process should be more automated to reduce delay.

Cleanly separate the changes made in downstreams. Current patching practices in downstreams largely rely on cherry-picking, *i.e.,* Linux LTS, Qualcomm stables, and OEMs. If a downstream kernel can cleanly separate its customization code from the upstream, or even better, upstream its customization (as is the case with Google[28]), the responsibility of patching upstream vulnerabilities can be completely automated with merging, *i.e.,* Android common and Qualcomm mainlines. A downstream kernel can simply merge automatically and fix security issues unknowingly.

7 Discussion

Unpatched kernels. By design, patch presence test is unable to equate the absence of patches with the target "being vulnerable". Throughout our measurements, we observe many cases where the downstream kernels never apply patches from upstream. However, this could simply mean that the downstream kernel is not affected by the upstream vulnerability, *e.g.,* due to customization of the vulnerable function. This is why we focus on the patched cases only, because it implies the downstream kernels are affected.

Further delays after the OEM patches. Our patch propagation measurement stops at the kernel compilation (build) dates. However, in practice, there are additional delays before the OEM updates can arrive at a user device. They include carrier certification delays (for carrier-locked phones), and users intentionally delaying the firmware update even if it is already available through OTA. Unfortunately, such delays are hard to quantify and we consider them out of scope. To get a basic sense of carrier certification delays, we manage to find the LG V30/Samsung S7/Samsung S8 on T-Mobile websites and SamsungS7/SamsungS8 on ATT websites that appear to publish the firmware release date. The average delay between built and release is about 20 days. To draw any meaningful conclusions though, a large-scale analysis needs to be done across more devices and carriers.

Chipset vendors other than Qualcomm. In addition to Qualcomm, other major SoC vendors include MediaTek, Kirin, and Exynos. Unfortunately, none of these vendors provides the complete git repositories for their recent chipsets. In addition, the CVEs specific to Kirin and Exynos chipsets are published only on Huawei's and Samsung's official websites but no links exist to the corresponding patches. Together, they represent a hurdle for any external party to track their patches. We suspect reverse engineering on the firmware images will be the only way to analyze the presence and absence of patches.

8 Related Work

Code similarity at the source and binary level. To conduct our measurement we need the ability to accurately test the patch presence at both source level (*e.g.,* the source code of the phone kernel is released) and binary level (*e.g.,* only ROMs are available for the target phone). There exist a large body of work aiming to compute the source/binary code similarity (*e.g.,* to find similar functions as a given vulnerable one), using a variety of source and binary level features [14, 27, 26, 34, 41].

In theory, these work can be used to test the patch presence by computing a target function's similarity to the patched/unpatched functions). Unfortunately, similarity-based approaches are fundamentally fuzzy and not suitable to capture the essence of a security patch which often makes only very small changes to patched functions and can still look similar to the unpatched version of the function. Tuning the similarity-based approach for patch presence test is an interesting but orthogonal problem.

Binary patch presence test. FIBER [42] is a state-of-art open-source tool to test the patch presence in binaries with the aid of the fine-grained source level patch information. It generates binary signatures that accurately capture the syntax and semantic information of the patch change sites, and then matches them in the target binary. It suits our needs perfectly and therefore we leverage and build on top of FIBER to test the patch presence for over 600 Android ROMs. To ensure that it works well in our large-scale measurement, we enhance the original FIBER to overcome several of its technical weaknesses as detailed in §4.3.

Android security patch investigation. Farhang *et al.* [19] have recently conducted a measurement on Android security patches, including both user and kernel components, with some minor overlap with this paper. In particular, they also analyzed the delay from the patch date (linked from the security bulletin which we now know is often not the earliest date) to the release date on the bulletin and observed a large delay. However, this represents only a small part of the picture of the end-to-end patch propagation in the ecosystem all the way from the upstream Linux to the end Android devices. Specifically, they do not attempt to locate patches in the source or binary at all. Thus they cannot find the bottleneck of patch delay. On the other hand, we not only showed where the bottleneck is but also explained why they exist with actionable insights and takeaways. More importantly, we also give suggestions on how to improve the patch propagation in the ecosystem.

Patch and vulnerability lifecycle analysis. There exist a number of measurement studies focusing on various aspects of patch propagation in open-source software. Li *et al.* [31], Shahzad *et al.* [40] and Frei *et al.* [20] performed large-scale measurements regarding the vulnerability lifecycle and the patching timeliness, based on publicly available information collected from data sources like CVE databases [4] and open-source repositories. Some of them focus on specific open-source projects, like Farhang *et al.* [19] focusing on Android and Ozment *et al.* [32] targeting FreeBSD. No analysis has been dedicated to the Android kernel ecosystem which involves the analysis of multiple parties in depth and the analysis of source and binary kernels.

9 Conclusion

In this paper, we delved deep into the Android kernel patch ecosystem, revealing the relationship among different parties as well as the bottleneck in patch propagation. This represents a first data point to measure such a huge, decentralized, fragmented, and yet collaborative project. We also analyze that the study is worthwhile in identifying deficiencies and opportunities to better manage such a project in the future.

References

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

[2] Android Security Bulletin—January 2020. `https://source.android.com/security/bulletin/2020-01-01`.

[3] Linux stable kernel patch rules. `https://www.kernel.org/doc/Documentation/process/stable-kernel-rules.rst`.

[4] National Vulnerability Database. `https://nvd.nist.gov/`.

[5] Pixel Update Bulletins. `https://source.android.com/security/bulletin/pixel`.

[6] What to do about CVE numbers. `https://lwn.net/Articles/801157/`.

[7] Huawei-firmware. `http://huawei-firmware.com/phone-list/`, 2019.

[8] Latest Official Android ROMs. `https://www.cnroms.com/`, 2019.

[9] MIUI Global ROM. `http://c.mi.com/oc/miuidownload/index`, 2019.

[10] Oppo Software Updates. `https://oppo.custhelp.com/app/soft_update`, 2019.

[11] Sammobile. `www.sammobile.com`, 2019.

[12] Stock ROM files. `https://stockromfiles.com/`, 2019.

[13] Adam Conway. How Monthly Android Security Patch Updates Work. `https://www.xda-developers.com/how-android-security-patch-updates-work/`.

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

[15] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. ICSM'98.

[16] Code Aurora. Android for MSM Project. `https://wiki.codeaurora.org/xwiki/bin/QAEP/`.

[17] Code Aurora. Android releases. https://wiki.codeaurora.org/xwiki/bin/QAEP/release.

[18] Code Aurora. Security Bulletin. https://www.codeaurora.org/category/security-bulletin/page/3.

[19] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags. Hey google, what exactly do your security patches tell us? a large-scale empirical study on android patched vulnerabilities. 2019.

[20] S. Frei, M. May, U. Fiedler, and B. Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138. ACM, 2006.

[21] Google. Distribution dashboard. https://developer.android.com/about/dashboards.

[22] Google. Learn when you'll get Android updates on Pixel phones Nexus devices. https://support.google.com/pixelphone/answer/4457705?hl=en.

[23] Google. Stable Kernel Releases Updates - Security. https://source.android.com/devices/architecture/kernel/releases#security.

[24] Google Project Zero. Bad Binder: Android In-The-Wild Exploit. https://googleprojectzero.blogspot.com/2019/11/bad-binder-android-in-wild-exploit.html.

[25] Google Project Zero. Issue 1942: Android: Use-After-Free in Binder driver. https://bugs.chromium.org/p/project-zero/issues/detail?id=1942.

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

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

[28] Jonathan Corbet. Bringing the Android kernel back to the mainline. https://lwn.net/Articles/771974/.

[29] A. Kernel. How Android common kernels developed. https://source.android.com/devices/architecture/kernel/android-common, 2019.

[30] L. Kernel. How the development process works. https://www.kernel.org/doc/html/latest/process/2.Process.html, 2019.

[31] F. Li and V. Paxson. A large-scale empirical study of security patches. CCS'17.

[32] A. Ozment and S. E. Schechter. Milk or wine: does software security improve with age? In *USENIX Security Symposium*, pages 93–104, 2006.

[33] I. Patel. Xiaomi Still Hasn't Released Kernel Sources for the Mi A1. https://www.xda-developers.com/xiaomi-not-released-kernel-sources-mi-a1/, 2018.

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

[35] piunikaweb. Asus releases botched up kernel sources for Zenfone Max M2 family on launch day. https://piunikaweb.com/2018/12/12/asus-releases-botched-up-kernel-sources-for-zenfone-max-m2-family-on-launch-day/, 2018.

[36] S. M. Qiushi Wu, Yang He and K. Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. NDSS, 2020.

[37] Qualcomm. Security Bulletin. https://www.qualcomm.com/company/product-security/bulletins.

[38] reddit. Samsung issues related to kernel source. https://www.reddit.com/r/Android/comments/94ol07/samsung_issues_related_to_kernel_source/, 2018.

[39] Samsung. Knox Deep Dive: Real-time Kernel Protection (RKP). https://www.samsungknox.com/en/blog/knox-deep-dive-real-time-kernel-protection-rkp, 2019.

[40] M. Shahzad, M. Z. Shafiq, and A. X. Liu. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 771–781. IEEE, 2012.

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

[42] H. Zhang and Z. Qian. Precise and accurate patch presence test for binaries. USENIX Security, 2018.