Adaptive Android Kernel Live Patching

Yue Chen
Florida State University

Yulong Zhang
Baidu X-Lab

Zhi Wang
Florida State University

Liangzhao Xia
Baidu X-Lab

Chenfu Bao
Baidu X-Lab

Tao Wei
Baidu X-Lab

Abstract

Android kernel vulnerabilities pose a serious threat to user security and privacy. They allow attackers to take full control over victim devices, install malicious and unwanted apps, and maintain persistent control. Unfortunately, most Android devices are never timely updated to protect their users from kernel exploits. Recent Android malware even has built-in kernel exploits to take advantage of this large window of vulnerability. An effective solution to this problem must be adaptable to lots of (out-of-date) devices, quickly deployable, and secure from misuse. However, the fragmented Android ecosystem makes this a complex and challenging task.

To address that, we systematically studied 1,139 Android kernels and all the recent critical Android kernel vulnerabilities. We accordingly propose KARMA, an adaptive live patching system for Android kernels. KARMA features a multi-level adaptive patching model to protect kernel vulnerabilities from exploits. Specifically, patches in KARMA can be placed at multiple levels in the kernel to filter malicious inputs, and they can be automatically adapted to thousands of Android devices. In addition, KARMA's patches are written in a high-level memory-safe language, making them secure and easy to vet, and their run-time behaviors are strictly confined to prevent them from being misused. Our evaluation demonstrates that KARMA can protect most critical kernel vulnerabilities on many Android devices (520 devices in our evaluation) with only minor performance overhead (< 1%).

1 Introduction

Android is a popular mobile operating system based on the Linux kernel. The kernel, due to its high privilege, is critical to the security of the whole Android system [4]. For example, Android relies on the Linux kernel to enforce proper isolation between apps and to protect important system services (e.g., the location manager) from unauthorized access. Once the kernel is compromised, none of the apps in the system can be trusted. Many apps contain sensitive personal data, such as bank accounts, mobile payments, private messages, and social network data. Even TrustZone, widely used as the secure keystore and digital rights management in Android, is under serious threat since the compromised kernel enables the attacker to inject malicious payloads into TrustZone [42, 43]. Therefore, Android kernel vulnerabilities pose a serious threat to user privacy and security.

Tremendous efforts have been put into finding (and exploiting) Android kernel vulnerabilities by both white-hat and black-hat researchers, as evidenced by the significant increase of kernel vulnerabilities disclosed in Android Security Bulletin [3] in recent years. In addition, many kernel vulnerabilities/exploits are publicly available but never reported to Google or the vendors, let alone patched (e.g., exploits in Android rooting apps [47]). The supply of Android kernel exploits likely will continue growing. Unfortunately, *officially* patching an Android device is a long process involving multiple parties with disparate interests: Google/the vendor verifies a reported vulnerability and creates a patch for it. The patch is then thoroughly tested and released to carriers; carriers test the update again for compatibility with their networks and release it to their users as an over-the-air (OTA) update. Many updates may queue up at the carriers waiting to be tested [33]; finally, the user may or may not install the update promptly. Arguably, device vendors and carriers have little incentive to keep user devices updated and secure. They instead prefer users to buy new devices. For example, phone vendors usually move to new products and stop updating older devices within one year. Consequently, many Android phones become obsolete shortly after they get into the customers' hands. There also exist lots of small vendors that do not have necessary resources to keep their phones updated. This dire situation is faithfully reflected in the vulnerable phones in use. Table 1 lists the statistics of two infamous kernel vulnerabilities: CVE-2015-3636 ("PingPong Root") [16] and CVE-2015-1805 [15] (data collected from 30 million devices [1]). After months

[1] With user consent, we collected kernel versions and build dates from devices with the Baidu app installed and compare them to each

CVE ID	Release Date	Months	% Vulnerable Devices
CVE-2015-3636	Sep. 2015	14	30%
CVE-2015-1805	Mar. 2016	8	47%

Table 1: Devices vulnerable to two infamous root exploits as of Nov. 2016. The second column lists the dates when they are disclosed in Android Security Advisory.

since their public disclosure, there are still a significant portion of devices vulnerable to them. Hence, it is unsurprising that Android malware with years-old root exploits can still compromise many victim devices worldwide [5,17,18,21]. In light of these serious threats, there is an urgent need for *third-parties* to promptly provide patches for these out-of-date devices, without involving vendors or carriers.

Android's fragmented ecosystem poses a significant challenge to a third-party kernel patching system: there are thousands of Android vendors that have produced and keep producing tens of thousands of devices [1], and Google releases new versions of Android at a regular base. This combination creates a mess of Android devices with all kinds of hardware and software configurations. For example, Android Lollipop (Android 5.0) was released in November 2014; as of September 2016, 46.3% of Android devices still run an older version of Android with little hope of any future updates [2]. Even worse, many Android vendors, small and large ones alike [19], indefinitely "delay" releasing the kernel source code despite the fact that the kernel's license (GPL) demands it. As such, existing source-code based patching systems [22,23,25,27] can only cover a limited number of devices; a binary-based approach would work better for a *third-party* solution. However, kernel binaries in these devices could differ significantly in details. For example, they may use different build systems, different versions of the compiler, and different optimization levels. An effective solution must accommodate thousands of similar yet very different kernels, a challenging goal.

To achieve our goal, we first quantified the Android fragmentation by systematically studying and measuring 1,139 Android kernel binaries. We formulated three key observations that allowed us to effectively tackle this problem. We also analyzed all the recent critical Android kernel vulnerabilities. Armed with these insights, we propose KARMA, a multi-level adaptive patching model that can overcome the Android fragmentation issue. KARMA stands for Kernel Adaptive Repair for Many Androids [2]. It protects kernel vulnerabilities by filtering malicious inputs to prevent them from reaching the vulnerable code. KARMA's patches are written in a high-level memory-safe language. To prevent patches from being misused, KARMA strictly confines their runtime behaviors so that the kernel remains as stable and consistent as possible under attack. Adaptiveness is a key distinguishing feature of KARMA from other live patching systems. It allows KARMA to scale to many Android devices. Specifically, given a reference patch and a target kernel, KARMA automatically identifies whether the target kernel contains the same vulnerability and customizes the reference patch for the target kernel if so. Therefore, KARMA's patches are easy to vet, secure, and adaptive. Like other kernel patching systems, KARMA requires privileged access to the devices it protects. It can either be pre-installed in the device's firmware or installed afterwards [7]. The implementation of KARMA supports all major Android platforms, and we are currently working with various Android vendors to pre-install KARMA in their future devices.

The main contributions of our paper are four-fold:

- We analyzed the fragmentation issue that hinders existing kernel live patching solutions to be ubiquitously applied on Android devices, and brought the need of an adaptive Android kernel patching solution to light.
- We studied 1,139 Android kernels from popular devices and 76 critical Android kernel vulnerabilities in the last three years. Based on these insights, we propose KARMA, a multi-level adaptive patching model that can be applied to the fragmented Android ecosystem.
- We implemented KARMA with the framework and primitives enabling memory-safe adaptive live patching. The implementation can support all the current Android kernel versions (from 2.6.x to 3.18.x) and different OEM vendors.
- We comprehensively evaluated KARMA against all the recently reported critical kernel vulnerabilities. Our evaluation shows that KARMA can both adaptively and effectively handle the majority of these vulnerabilities with negligible overhead ($< 1\%$).

The rest of the paper is organized as follows. We first state the problem and present the design of KARMA in Section 2. We then evaluate the applicability, adaptability, and performance overhead of KARMA in Section 3. Section 4 discusses the potential improvements to KARMA, and Section 5 compares KARMA to the closely related work. We conclude the paper in Section 6.

vulnerability's disclosure date to decide if it is potentially vulnerable.

[2] KARMA is a part of the OASES (Open Adaptive Security Extensions, `https://oases.io`) project, an initiative founded by Baidu to enable fast and scalable live patching for mobile and IoT devices.

2 System Design

In this section, we first present our key observations on the Android fragmentation problem and then describe the design of KARMA in detail.

2.1 Measuring Android Fragmentation

Designing a live kernel patching system that can scale to lots of devices is a challenging task. However, three key observations we gained from systematically measuring the Android fragmentation render this task feasible and manageable. These observations can serve as a foundation for future systems tackling this problem.

Observation A: *most kernel functions are stable across devices and Android releases.*

Android (Linux) kernel is a piece of large and mature software. Like other large software, evolution is more common and preferred than revolution – bugs are fixed and new features are gradually added. Complete rewrite of a core kernel component is few and far between. A patch for one kernel thus can probably be adapted to many other kernels. Adaptiveness is a key requirement for protecting the fragmented Android ecosystem.

To measure the stableness of Android kernels, we collected $1,139$ system images from four major vendors (Samsung/Huawei/LG/Oppo, $1,124$ images) and Google (15 images). These four vendors together command more than 40% of the Android smartphone market, and Google devices have the newest Android software. This data set is representative of the current Android market: these images come from 520 popular old and new devices, feature Android versions from 4.2 to 7.0, and cover kernels from $2.6.x$ to $3.18.x$. The statistics of these images are shown in Table 2 and 3.

After collecting these images, we extracted symbols from their kernels. There are about $213K$ unique functions, and about $130K$ of them are shared by more than 10 kernels. We wrote a simple tool to roughly analyze how many different revisions each of these shared functions has. Specifically, we abstract the syntax of each function by the number of its arguments, the conditional branches it contains, the functions called by it, and non-stack memory writes. We then cluster each function across all the images based on these syntactic features. Each different cluster can be roughly considered as a revision of the function (i.e., each cluster potentially requires a different revision of the patch). The results are shown in Fig. 1 and 2. Specifically, Fig. 1 shows how many clusters each shared function has. About 40% of the shared functions have only one cluster, and about 80% of them have 4 clusters or less. Fig. 2 shows the percentage of the kernels in the largest cluster for each shared function. For about 60% of shared functions, the largest cluster contains more than 80% of all the kernels that have this function. These data show that most kernel functions are indeed stable across different devices. Vulnerabilities in shared functions should be given a higher priority for patching because they affect more devices.

Observation B: *many kernel vulnerabilities are triggered by malicious inputs. They can be protected by filtering these inputs.*

Kernel vulnerabilities, especially exploitable ones, are often triggered by malicious inputs through syscalls or external inputs (e.g., network packets). For example, CVE-2016-0802, a buffer overflow in the Broadcom WiFi driver, can be triggered by a crafted packet whose size field is larger than the actual packet size. Such vulnerabilities can be protected by placing a filter on the inputs (i.e., function arguments and external data received from functions like `copy_from_user`) to screen malicious inputs. We surveyed *all* the critical kernel vulnerabilities in the Android Security Bulletin reported in 2015 and 2016 and found that 71 out of 76 (93.4%) of them could be patched using this method (Table 6).

Observation C: *many kernel functions return error codes that are handled by their callers. We can leverage the error handling code to gracefully discard malicious inputs.*

When a malicious input is blocked, we need to alter the kernel's execution so that the kernel remains as consistent and stable as possible. We observe that many kernel functions return error codes that are handled by their callers. In such functions, a patch can simply end the execution of the current function and return an error code when a malicious input is detected. The caller will handle the error code accordingly [34]. Linux kernel's coding style recommends that functions, especially exported ones, returning an error code to indicate whether an operation has succeeded or not [24]. If the function does not normally return error codes, it should indicate errors by returning out-of-range results. A notable exception is functions without return values. Most (exported) kernel functions follow the official coding style and return error codes — even kernel functions that return pointers often return out-of-range "error codes" using the `ERR_PTR` macro.

Based on these observations, our approach is as follows: for each applicable vulnerability, we create a patch that can be placed on the vulnerable function to filter malicious inputs. The patch returns a selected error code when it detects an attack attempt. The error is handled by the existing error handling code, keeping the kernel stable. This patch is then automatically adapted to other devices. Automatic adaptation of patches can significantly reduce the manual efforts and speed up the patch deployment.

Vendor	#Models	#Images
Samsung	192	419
Huawei	132	217
LG	120	239
Oppo	74	249
Google Nexus	2	15
Total	520	1139

Table 2: Images obtained from popular devices.

Category	Statistics
Countries	67
Carriers	37
Android Versions	4.2.x, 4.3.x, 4.4.x, 5.0.x, 5.1.x, 6.0.x, 7.0.x
Kernel Versions	2.6.x, 3.0.x, 3.4.x, 3.10.x, 3.18.x
Kernel Architectures	ARM (77%), AArch64 (23%)
Kernel Build Years	2012, 2013, 2014, 2015, 2016

Table 3: Statistics of the obtained Android kernels.

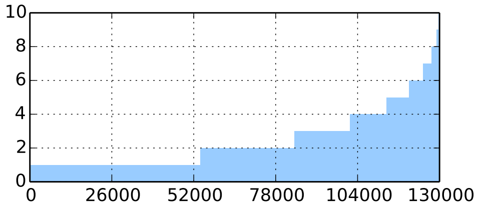


Figure 1: Number of revision clusters for each shared function, sorted by the number of clusters.

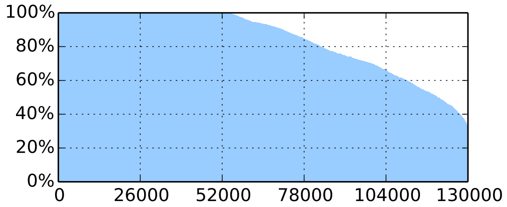


Figure 2: Percentage of kernels in the largest cluster for each shared function.

2.2 Adaptive Multi-level Patching

KARMA features a secure and adaptive multi-level patching model. The security is enforced by the following two technical constraints:

Rule I, *a patch can only be placed at designated locations, and its patched function must be able to return error codes or return void (i.e., no return value).*

KARMA protects kernel vulnerabilities by preventing malicious inputs from reaching them. For security reasons, a patch can only be placed at the designated levels. Specifically, *level 1* is the entry or return points of a vulnerable function; *level 2* is before or after *call instructions* to a callee of the vulnerable function. Note that we do not patch the callee itself but rather hook call instructions in order to avoid affecting other callers of this callee. A typical example of callees hooked by KARMA is `copy_from_user`, a function dedicated to copy untrusted user data into the kernel. `copy_from_user` is a perfect checkpoint for malicious inputs because the kernel calls it whenever the kernel needs to read the user data; Level 3 is similar to the existing binary-based patches [22,23,27]. *Level-3* patches are more flexible but potentially dangerous because they are (currently) unconstrained. If a vulnerability is difficult to patch at level 1 and level 2, we fall back to level 3. Level-3 patches have to be manually scrutinized to prevent them from being misused. Our experiment with 76 critical kernel vulnerabilities shows that level 1 can patch 49 (64%) vulnerabilities, level 2 can patch 22 (29%) vulnerabilities, and we have to fall back to level 3 in only 5 cases (7%). This multi-level design allows KARMA to patch most, if not all, Android kernel vulnerabilities. In the following, we focus on the level-1 and level-2 patches since level-3 patches (i.e., binary patching) have been studied by a number of the previous research [22, 23, 27].

A patch can indirectly affect the kernel's control flow by returning an error code when a malicious input is intercepted. This immediately terminates the execution of the vulnerable function and passes the error code to the caller. We require a patched function to return error codes on fault in order to leverage the existing error handling code of the kernel to gracefully fail on malicious inputs. Allowing a patch to return arbitrary values (i.e., other than error codes) may have unintended consequences. Fortunately, many kernel functions return error codes on fault, following the guidelines of the official coding style. Similarly, we allow functions that return void to be patched.

Rule II, *a patch can read any valid kernel data structures, but it is prohibited from writing to the kernel.*

Even though KARMA's patches are vetted before deployment, they may still contain weakness that can be exploited by attackers. To control their side effects, patches are only allowed to read necessary, valid kernel data structures (e.g., registers, stacks, the heap, code, etc.), but they are prohibited from writing to the kernel. Allowing a patch to change the kernel's memory, even one bit, is dangerous. For example, it could be exploited to clear the U-bit (the user/kernel bit) of a page table entry to grant the user code the kernel privilege. Without the write permission, patches are also prevented from leaking kernel information to a local or remote adversary. This rule is enforced by providing a set of restricted APIs as the only interface for the patches to access the kernel data.

By combining these two rules with a careful vetting process and the memory-safety of the patches, we can strictly confine the run-time behaviors of patches to prevent them from potential misuse.

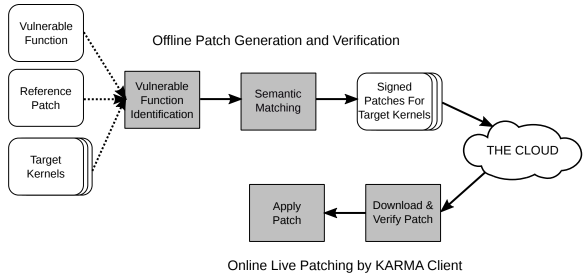

Vulnerable Function
Reference Patch
Target Kernels
Offline Patch Generation and Verification
Vulnerable Function Identification
Semantic Matching
Signed Patches For Target Kernels
THE CLOUD
Download & Verify Patch
Apply Patch
Online Live Patching by KARMA Client

Figure 3: Workflow of KARMA

```
function kpatcher(patchID, sp, cpsr, r0, r1,
    r2, r3, r4, r5, r6, r7, r8, r9, r10, r11,
     r12, r14)
    if patchID == 0xca5269db50f4 then
        uaddr1 = r0
        uaddr2 = r2
        if uaddr1 == uaddr2 then
            return -22
        else
            return 0
        end
    end
end
kpatch.hook(0xca5269db50f4,"futex_requeue")
```

Figure 4: A simplified patch in Lua for CVE-2014-3153

2.3 Architecture and Workflow

KARMA works in two phases as shown in Figure 3. The offline phase adapts a reference patch ($\mathcal{P}_r$) to all the devices supported by KARMA. The reference patch often comes from an upstream source, such as Google and chipset manufacturers. It targets a specific device and kernel (named as the reference kernel, $\mathcal{K}_r$) and is not directly applicable to other devices. To address that, KARMA employs an automated system to customize $\mathcal{P}_r$ for each target kernel ($\mathcal{K}_t$). Specifically, KARMA first roughly identifies potentially vulnerable functions in kernel $\mathcal{K}_t$, and applies symbolic execution to compare the semantics of each candidate function ($\mathcal{F}_t$) against reference function $\mathcal{F}_r$. If these two functions are semantically equivalent, KARMA further adjusts the reference patch for kernel $\mathcal{K}_t$, signs it, and deposits it to the cloud. To prevent malicious patches from being installed by user devices, reference patches are carefully vetted and all the patches are signed. User devices only install signed patches. Matching semantics with symbolic execution can abstract syntactic differences in function binaries (e.g., register allocation). Semantic matching decides whether candidate function $\mathcal{F}_t$ is semantically equivalent, or very similar to, reference function $\mathcal{F}_r$, and whether $\mathcal{F}_t$ has been patched or not. In other words, it is responsible for locating a function in the target kernel that can be patched but has not been patched yet. Semantic matching also provides a scheme to customize reference patch $\mathcal{P}_r$ for target kernels.

In the second phase, the KARMA client in the user device downloads and verifies the patches for its device and applies them to the running kernel. Specifically, the client verifies that each downloaded patch is authentic by checking its signature and that it is applicable to this device by comparing the device model and the kernel version. If a patch passes the verification, it is cached in a secure store provided by Android. The client then applies the patch to the running kernel. An applied patch immediately protects the kernel from exploits without rebooting the device or user interactions. In the unlikely event that a patch causes the device to malfunction, the user can reboot the device and skip the problematic patches by holding a hardware key. Currently, KARMA's patches are written in the Lua language. We choose Lua for its simplicity, memory-safety, and easiness to embed and extend (in security, simplicity is a virtue). Lua provides sufficient expressive power for KARMA to fix most kernel vulnerabilities. Other kernel scripting languages, such as BPF [8], can also satisfy our requirements. To execute these patches, we embed a restricted Lua engine in the kernel. The engine strictly enforces the security rules of KARMA (Section 2.2).

In the rest of this section, we first illustrate KARMA's patches and then present these two phases in detail.

2.4 KARMA Patches

Patches in KARMA are written in the Lua programming language. Lua is a simple, extensible, embedded language. It has only eight primitive types, such as `nil`, `boolean`, `number`, `string`, and `table`. Tables are the only built-in composite data type. Most user data structures are built on top of tables. Lua is a dynamically typed language, and all the data accesses are checked at the run-time. This reduces common memory-related flaws like buffer overflows. Lastly, Lua creates an isolated environment to execute patches. This prevents patches from directly accessing the kernel memory. Instead, the kernel data can only be accessed through restrictive APIs provided by KARMA.

Figure 4 shows a simplified patch for CVE-2014-3153, exploited by the infamous Towelroot. CVE-2014-3153 is a flaw in function `futex_requeue`. It fails to check that two arguments are different, allowing a local user to gain the root privilege via a crafted `FUTEX_REQUEUE` command [14]. To fix it, we just check whether these two arguments (in register `r0` and `r1`, respectively) are different and return an error code (`-22` or `-EINVAL`) if they are the same. As shown in Fig. 4, each hooking point has a unique ID. The patch can check this ID to ensure that it is called by the correct hooking points. When invoked, the patch receives the current values of the registers as arguments. They allow the patch to access function argu-

```
  static int sock_diag_rcv_msg(struct sk_buff *
      skb, struct nlmsghdr *nlh)
  {
      ...
      switch (nlh->nlmsg_type) {
          ...
          case SOCK_DIAG_BY_FAMILY:
              return __sock_diag_rcv_msg(skb,
                  nlh);
          ...
  }

  static int __sock_diag_rcv_msg(struct sk_buff
        *skb, struct nlmsghdr *nlh)
  {
      int err;
      struct sock_diag_req *req = NLMSG_DATA(
          nlh);
      struct sock_diag_handler *hndl;
      if (nlmsg_len(nlh) < sizeof(*req))
          return -EINVAL;
+     if (req->sdiag_family >= AF_MAX)
+         return -EINVAL;
      hndl=sock_diag_lock_handler(req->
          sdiag_family);
      ...
  }
```

Figure 5: Source-code patch for CVE-2013-1763

ments and other necessary data by using the APIs provided by KARMA. The last line of the patch installs itself at the futex_request function with a patch ID of 0xca5269db50f4. Next, we use a few examples to demonstrate how to convert a regular source-code based patch to a reference patch for KARMA.

CVE-2013-1763: Figure 5 shows the original source code patch for CVE-2013-1763. Each "+" sign marks a new line added by the patch. The added lines validate that the protocol family of the received message (req->sdiag_family) is less than AF_MAX and returns -EINVAL otherwise. This patch can be easily converted to a reference patch for KARMA. However, since __sock_diag_rcv_msg does not appear in the kernel's symbol table (because it is a static function), KARMA instead hooks the entry point of its parent function and screens the arguments there.

CVE-2013-6123: this is a vulnerability in function msm_ioctl_server, which reads an untrusted data structure (u_isp_event) from the user space with copy_from_user. However, it fails to check that the queue_index field of the input is valid. This vulnerability is fixed by line 10-17 in Fig. 6. To patch this vulnerability in KARMA, we cannot hook the entry point of msm_ioctl_server because the malicious input data is not available yet. Instead, we should hook the return point of copy_from_user and filter the received data. copy_from_user returns status codes; therefore it can be hooked by KARMA. If the patch detects a malicious input, it returns the error code of -EINVAL. This terminates the execution gracefully.

```
1    static long msm_ioctl_server(struct file *
         file, void *fh, bool valid_prio, int cmd,
          void *arg)
2    {
3        ...
4        if (copy_from_user(&u_isp_event,
5            (void __user *)ioctl_ptr->ioctl_ptr,
6            sizeof(struct msm_isp_event_ctrl))) {
7            ...
8        }
9        ...
10 +     if(u_isp_event.isp_data.ctrl.queue_idx<0
11 +     || u_isp_event.isp_data.ctrl.queue_idx >=
12 +         MAX_NUM_ACTIVE_CAMERA) {
13 +         pr_err("%s: Invalid index %d\n",
14 +             __func__, u_isp_event.isp_data.
         ctrl.queue_idx);
15 +         rc = -EINVAL;
16 +         return rc;
17 +     }
18       ...
19   }
```

Figure 6: Source-code patch for CVE-2013-6123

CVE-2016-0802: this is a buffer overflow in the Broadcom WiFi driver, caused by the missing check that the packet data length is less than the packet length. This vulnerability represents an interesting challenge to KARMA: the source-code is patched in several functions, and a new argument is added to function dhd_wl_host_event and dngl_host_event. The error condition is finally checked in function dngl_host_event. Apparently, this type of fix (i.e., adding new arguments to functions) cannot be translated directly in KARMA because patches are not allowed to write the kernel memory. To address that, we need to hook both dhd_rx_frame and dngl_host_event functions. The first hook saves the packet length, and the second hook compares the packet length to the data length. If the data length is larger than the packet length, the patch returns the error code of BCME_ERROR. This is an example of KARMA's multi-invocation patches (also called stateful patches). Both patches bear the same patch ID. The variables at the first hook are made accessible to the second hook by KARMA's Lua engine. An alternative fix is to hook only dhd_rx_frame and manually extract the data length from the packet. However, this fix is less favorable because the patch has to parse the packet structure by itself and it is placed differently from where the source-code patch modifies the control flow, i.e., where the error handling is guaranteed to work.

2.5 Offline Patch Adaptation

KARMA's offline component adapts a reference patch for all supported devices. It first identifies the vulnerable function in a target kernel through structural and semantic matching; then it uses the information from semantic matching to customize the patch for the target kernel. In the following, we describe these two steps in detail.

```
void dhd_rx_frame(...)
{
    ...
    dhd_wl_host_event(dhd, &ifidx,
                      skb_mac_header(skb),
                      skb->mac.raw,
+                     len - 2,
                      &event, &data);
    ...
}

static int dhd_wl_host_event(...)
{
    ...
-   if (dngl_host_event(dhd_pub, pktdata) ==
    BCME_OK) {
+   if (dngl_host_event(dhd_pub, pktdata,
    pktlen) == BCME_OK) {
    ...
}

int dngl_host_event(...)
{
    ...
+   if (datalen > pktlen)
+       return (BCME_ERROR);
    ...
}
```

Figure 7: Source-code patch for CVE-2016-0802

2.5.1 Syntactic Matching

Given a target kernel $\mathcal{K}_t$, we first identify candidate functions ($\mathcal{F}_t$) in $\mathcal{K}_t$ that may contain the same vulnerability as reference function $\mathcal{F}_r$. However, this task is not as simple as searching the kernel symbol table. There are a number of challenges. *First*, function $\mathcal{F}_t$ might have different semantics than $\mathcal{F}_r$ even though their names are the same. Accordingly, the patch cannot be applied to $\mathcal{K}_t$. KARMA addresses this problem by further matching their semantics. *Second*, $\mathcal{F}_t$ may have a (slightly) different name than $\mathcal{F}_r$ even though their semantics is the same. For example, CVE-2015-3636 [30], exploited by PingPong root, exists in function `ping_unhash` in the Google Nexus 5 kernel but `ping_v4_unhash` in some other kernels. *Third*, $\mathcal{F}_t$ could have been inlined in the target kernel and thus does not exist in the symbol table. To address these challenges, we assume that most (other) functions are not changed or renamed across different kernels. This assumption is backed by our first observation (Section 2.1).

To find matches of function $\mathcal{F}_r$ in target kernel $\mathcal{K}_t$, we first extract the symbol table from $\mathcal{K}_t$'s binary [3] and search in it for the name of $\mathcal{F}_r$. If an exact match is found, we consider this function to be the only candidate. Otherwise, we try to identify candidate functions by call relations. Specifically, we first extract the call graphs from the target and the reference kernels. We collect callers and callees of function $\mathcal{F}_r$ in the reference kernel's graph, and try to locate nodes in the target kernel's graph that have similar call relations to these two sets of functions. We may find a unique matching node if the function has been simply renamed. If the function has been inlined, the target kernel's call graph contains direct edges from the caller set to the callee set (instead of connected through $\mathcal{F}_r$). Accordingly, we use the containing function as the candidate. Multiple candidate functions may be identified using this approach. The semantics of these candidate functions is then compared to that of function $\mathcal{F}_r$ to ensure that the patch is applied to correct functions.

2.5.2 Semantic Matching

In this step, KARMA uses semantic matching to decide whether a function should be patched and whether a given reference patch can be adapted to it. For two Android kernels, the same source code could be compiled into different binaries – they may vary in register allocation, instruction selection, and instruction layout. In addition, the positions of structure members may have shifted, and the stack may contain different temporary variables (e.g., because of differences in the register spilling). Therefore, simple syntactic comparison of functions is too restrictive and may reject functions that could otherwise be patched. To this end, we leverage symbolic execution to compare semantics of the candidate function ($\mathcal{F}_t$) and the reference function ($\mathcal{F}_r$).

Path explosion is a significant obstacle in symbolic execution. The situation is even more serious in the Linux kernel because many kernel functions are highly complicated. Even if the vulnerable function looks simple, it may call complex other functions. This can quickly overwhelm the symbolic execution engine. In KARMA, we assume that functions called by $\mathcal{F}_t$ and $\mathcal{F}_r$ have the same semantics if they share the same signature (i.e., function name and arguments). Therefore, we can use non-local memory writes (i.e., writes to the heap or global variables), function calls, and function returns as checkpoints for semantic comparison. Non-local memory writes, function calls, and returns make up the function's impacts to the external environment. We consider two functions having the same semantics if their impacts to the environment are the same. We do not take stack writes into consideration because the same function may have different stack layouts in two kernels.

To compare their semantics, we symbolically execute the basic blocks of $\mathcal{F}_r$ and $\mathcal{F}_t$ and generate constraints for memory writes and function calls. For each memory write, we first check whether it is a local write or not (we consider it a local write if its address is calculated related to the stack/base pointer). If it is a non-local write, we add two constraints that the memory addresses and the

[3] The kernel binary often contains the symbol table so that kernel modules can be linked to the kernel. This table may or may not be exported through the `/proc/kallsym` file at runtime.

content-to-write should be equal. For function calls, we first check that these functions have the same name (and arguments if the kernel source is available). If so, we add constraints that the arguments to these two functions should be equal. We handle function returns similarly by adding constraints for register `r0` at the function exits. External inputs to these two functions, such as initial register values, non-local memory reads, and sub-function returns, are symbolized.

KARMA supports two modes of operation: in the strict mode, we require that two matching constraints are exactly the same, except for constants. Constants are often used as offsets into structures or the code (e.g., to read embedded constants in the code). These offsets could be different even for the same source code because of different hardware/software settings (e.g., conditional compiling). We ignore these constants to accommodate these differences. In a relaxed mode, we use a constraint solver to find a solution that can fulfill all the constraints at the same time. We consider two functions to be semantically equivalent if there exist at least one such solution. Moreover, to avoid patching an already-patched function, we compare path constraints for the variables accessed by reference patch $\mathcal{P}_r$ in function $\mathcal{F}_r$ and $\mathcal{F}_t$. If they are more restrictive in $\mathcal{F}_t$ than in $\mathcal{F}_r$ (i.e., conditional checks are added in $\mathcal{F}_t$), the function may have already been patched. Note that since KARMA's patches cannot modify the kernel memory, reapplying a patch is likely safe. If a semantic match is found, the symbolic formulas provide useful information for adapting patch $\mathcal{P}_r$ for the target kernel. For example, we can adjust $\mathcal{P}_r$'s registers and field offsets by comparing formulas of the function arguments. We evaluate the effectiveness of semantic matching in Section 3.2.

2.6 Live Patching

To enable its protection, KARMA needs to run its client in the user device. The client consists of a regular app and a kernel module. The app contacts the KARMA servers to retrieve patches for the device, while the kernel module verifies the integrity of these patches and applies ones that pass the verification.

2.6.1 Integration of Lua Engine

Patches in KARMA are written in the Lua language. They are executed by a Lua engine embedded in the kernel. KARMA extends the Lua language by providing a number of APIs for accessing kernel data structures. Normally, extending Lua with unsafe C functions forgoes Lua's memory safety. KARMA provides two groups of APIs to Lua scripts. The first group is used exclusively for applying patches, and the other group is used by patches to read kernel data. Our vetting process automatically ensures that patches can only use the second group of APIs. As such, the memory safety of Lua is retained because all the APIs that a patch can access are read-only. Table 4 lists these APIs, which provide the following functionalities: 1) symbol searching: return the run-time address of a symbol; 2) function hooking: hook a given function/sub-function in order to execute the patch before/after the function is called; 3) typed read: given an address, validate whether the address is readable and return the (typed) data if so; 4) thread-info fetching: return the current thread information, such as its thread ID, kernel stack, etc. The first two functionalities belong to the first group, and the rest belongs to the second group. Again, the live patcher can use both groups of the APIs, but patches can only use the second one.

API	Functionality
`hook`	Hook a function for live patching
`subhook`	Hook the calls to sub-functions for live patching
`alloc_mem`	Allocate memory for live patching
`free_mem`	Free the allocated memory for live patching
`get_callee`	Locate a callee that can be hooked
`search_symbol`	Get the kernel symbol address
`current_thread`	Get the current thread context
`read_buf`	Read raw bytes from memory with the given size
`read_int_8`	Read 8 bits from memory as an integer
`read_int_16`	Read 16 bits from memory as an integer
`read_int_32`	Read 32 bits from memory as an integer
`read_int_64`	Read 64 bits from memory as an integer

Table 4: The extension to Lua. The first five functions can only be used by the live patcher, not by patches.

2.6.2 Patch Application

To apply a patch, KARMA hooks the target function to interpose the patch in the regular execution flow, as shown in Fig. 8. Specifically, for each hooking point, we create a piece of the trampoline code and overwrite the first few instructions at the hooking point with a jump to the trampoline. At run-time, the trampoline saves the current context by pushing all the registers to the stack and invokes the Lua engine to execute the associated patch. The saved context is passed to the patch as arguments so that the patch can access these registers. Before installing the hook, the live patcher calls the `stop_machine` function and checks whether there are any existing invocations of the target function in the kernel stacks. If so, it is unsafe to immediately patch the function because otherwise the existing invocations will return to the patched version, potentially causing inconsistent kernel states. When this happens, we return an error code to the client which will retry later. As soon as the patch is applied, the vulnerable function is protected from attacks. If no malicious inputs are detected, the patch returns zero to the trampoline, which in turn restores the context, executes

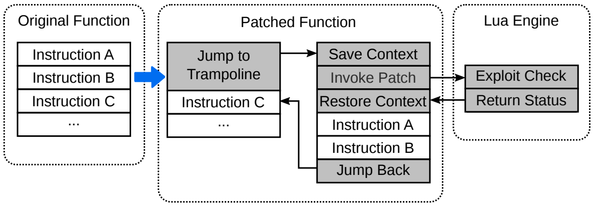


Figure 8: Live patching through function hooking

the overwritten instructions, and jumps back to the original function; If malicious inputs are detected, the patch returns an error code to the trampoline, which ends the execution of the hooked function by jumping to a return instruction.

2.6.3 Patch Dispatching

KARMA supports two methods to dispatch a patch, one for each of the two execution contexts: the interrupt context or the thread (or process) context. In the interrupt context, the Lua engine is directly invoked through the engine's C interface, similar to a regular function call. However, it is expensive to launch a new Lua engine each time a patch is executed. In the thread context, we instead schedule patches to a standalone Lua engine (through a workqueue) and wait for the results. The Lua engine executes in a self-contained kernel thread and processes incoming requests from the workqueue. Each request is identified by the thread ID and the patch ID. This dispatching method cannot be used in the interrupt context because blocking functions (e.g., to acquire a lock) cannot be called in that context. If a vulnerable function is called in both contexts, we dispatch the patch according to the active context (we have not found such cases in practice). Patch dispatching in the thread context is more complex. In the following we give more details about it.

The kernel is a concurrent execution environment, especially with multi-core CPUs, which most Android devices have. A patch accordingly can be executed simultaneous by multiple threads on different CPU cores. These invocations are grouped by their thread ID and patch ID. Specifically, for each distinct combination of thread ID and patch ID, a separate name space is created. Each Lua variable is saved to its associated name space. A name space is not destroyed until the associated thread ends. Therefore, variables of the previous invocations remain available to the subsequent invocations in the same name space [4]. By keeping the states across invocations, KARMA can support multi-invocation patches, i.e., complex patches that need to combine the results of several executions to make a decision. A number of patches we tested require this capability. In the thread context, we can also support multiple Lua engines to improve the throughput of patch execution. Specifically, we can spawn multiple kernel threads to run several instances of the Lua engine. A dispatch algorithm decides which Lua engine a request should be scheduled to. The algorithm must be deterministic so that requests in the same name space will always be scheduled to the same engine, allowing them to access states from previous invocations. When a thread ends, its associated states are cleared from all the Lua engines.

Lua is a garbage-collected language. Patches thus do not need to explicitly manage memory allocation and release. The Lua engine uses a simple mark-and-sweep garbage collector [35]. Kernel patches usually do not need to allocate many memory blocks. The default garbage collector works well for our purpose without slowing down the system.

2.7 Prototype of KARMA

We have implemented a prototype of KARMA. We wrote a number of offline tools for patch adaptation and signing. Our symbolic execution engine was based on the angr framework [6, 44]. We implemented the syntactic and semantic matching by ourselves. Our Lua engine in the kernel is similar to the lunatik-ng project [26]. For example, the Linux kernel does not use floating-point arithmetic. We therefore changed Lua's internal number representation from floating-points to integers. We also removed the unnecessary Lua libraries such as file operations. Furthermore, we added the support to name spaces in our Lua engine and extended the Lua language with the APIs specified in Table 4. We added roughly about $11K$ lines of source code in total to the Android kernel. The added code was compiled as an 800KB kernel module. This kernel module can be pre-installed on Android devices through collaboration with vendors or installed afterwards through rooting, the only choice available. KARMA can support all the known Android kernel versions (from 2.6.x to 3.18.x) and different vendors.

3 Evaluation

The effectiveness of KARMA can be evaluated by its applicability, adaptability, and performance. Applicability quantifies how many existing kernel vulnerabilities can be patched by KARMA, and adaptability quantifies how many devices that KARMA can adapt a reference patch for. In the following, we describe these three aspects of the evaluation in detail.

[4] If the vulnerable function is recursively called, some variable states might be lost. To retain the whole history, we can tag variables with the thread ID, patch ID, and the stack top. However, we have not found any of such cases in practice.

3.1 Evaluation of Applicability

We tested KARMA with all the critical kernel vulnerabilities from Android Security Bulletin and ones used to root Android devices. There are 76 such vulnerabilities in total in the last three years. Remarkably, KARMA can fix 71 of them (93.4%) with level-1 and level-2 patches; i.e., we can create an adaptable KARMA patch for them. Table 6 in Appendix A gives a more complete list of the results. In the following, we describe how KARMA can prevent some interesting kernel vulnerabilities used in one-click rooting apps and recent malware incidents [5,17,18,21]. Appendix A contains a couple of more examples.

CVE-2013-6282 (VROOT): this was one of the most popular vulnerabilities used in the wild to root Android devices, publicly known as “VROOT”. It exists in the `get/put_user` *macros*. They both fail to check that user-provided addresses are in the valid range. The original patches add the necessary checks to these macros and return `-EFAULT` if invalid addresses are detected [12]. However, KARMA cannot patch these two macros because they are expanded by the compiler and thus do not exist in the kernel binary. Instead, KARMA patches their expanded functions (i.e., `__get_user_1/2/4` and `__put_user_1/2/4/8`) with checks of whether user-provided addresses are less than `current_thread_info()->addr_limit-1`. Note that these patches can access the current `thread_info` structure by using the `current_thread` API provided by KARMA. These patches simply return `-EFAULT` if the address is out of the range.

CVE-2013-2595 (Framaroot): this vulnerability was a part of the infamous Framaroot app (the “Gandalf” payload). It exists in the camera driver for the Qualcomm MSM devices [10]. The driver provides an uncontrolled `mmap` interface, allowing the attacker to map sensitive kernel memory into the user space. KARMA can patch this vulnerability by validating whether the memory to be mapped is within the user space.

CVE-2013-2596 (MotoChopper): an integer overflow in the `fb_mmap` function allows a local user to create a read-write mapping of the entire kernel memory and consequently gain the kernel privileges. Specifically, the function has a faulty conditional check:

```
if((vma->vm_end - vma->vm_start + off)>len)
    return -EINVAL;
```

Because `off` is a user-controlled variable, an attacker can pass in a really large number to overflow `(vma->vm_end - vma->vm_start + off)` (the result is interpreted as a negative number) and bypass the validation. Here the original patch adds more checks to prevent this situation [11]. To patch this vulnerability in KARMA, we hook the `fb_mmap` function and extract the needed variables from its argument `vma`. For example, we can calculate `off` as `(vma->vm_pgoff << PAGE_SHIFT)`. The patch then checks whether `(vma->vm_end - vma->vm_start + off)` is negative or not, and return `-EINVAL` if so.

3.2 Evaluation of Adaptability

KARMA is an adaptive kernel live patching system for Android. Its ability to automatically adapt a reference patch is the key to protect a wide variety of devices and reduce the window of vulnerability. In this experiment, we evaluate KARMA’s adaptability with 1,139 Android kernels collected from Internet.

Semantic matching is the key to KARMA’s adaptability. It uses symbolic execution to abstract away syntactic differences in function binaries, such as register allocation, instruction selection, and data offset. To evaluate its effectiveness, we cluster the collected 1,139 Android kernels [5] by syntactic and semantic features for 13 popular vulnerabilities. Specifically, the opcode-based clustering classifies kernel functions by types and frequencies of instruction opcodes; the syntax-based clustering classifies kernel functions by function calls and conditional branches; and the semantic-based clustering classifies kernel functions according to KARMA’s semantic matching results. Table 5 lists the number of clusters and the percentage of kernels in the largest cluster for each clustering method. This table shows that the semantic-based method is the most precise one because it has the smallest number of clusters. Technically, each cluster may need a different adaptation of the reference patch. Therefore, fewer clusters mean a better chance for adaptation to succeed and less manual efforts if automated adaptation fails. Moreover, the largest clusters in the semantic matching often contain the majority of the vulnerable kernels. For example, a single reference patch for the largest cluster of `perf_swevent_init` can be applied to 96.3% of the vulnerable kernels.

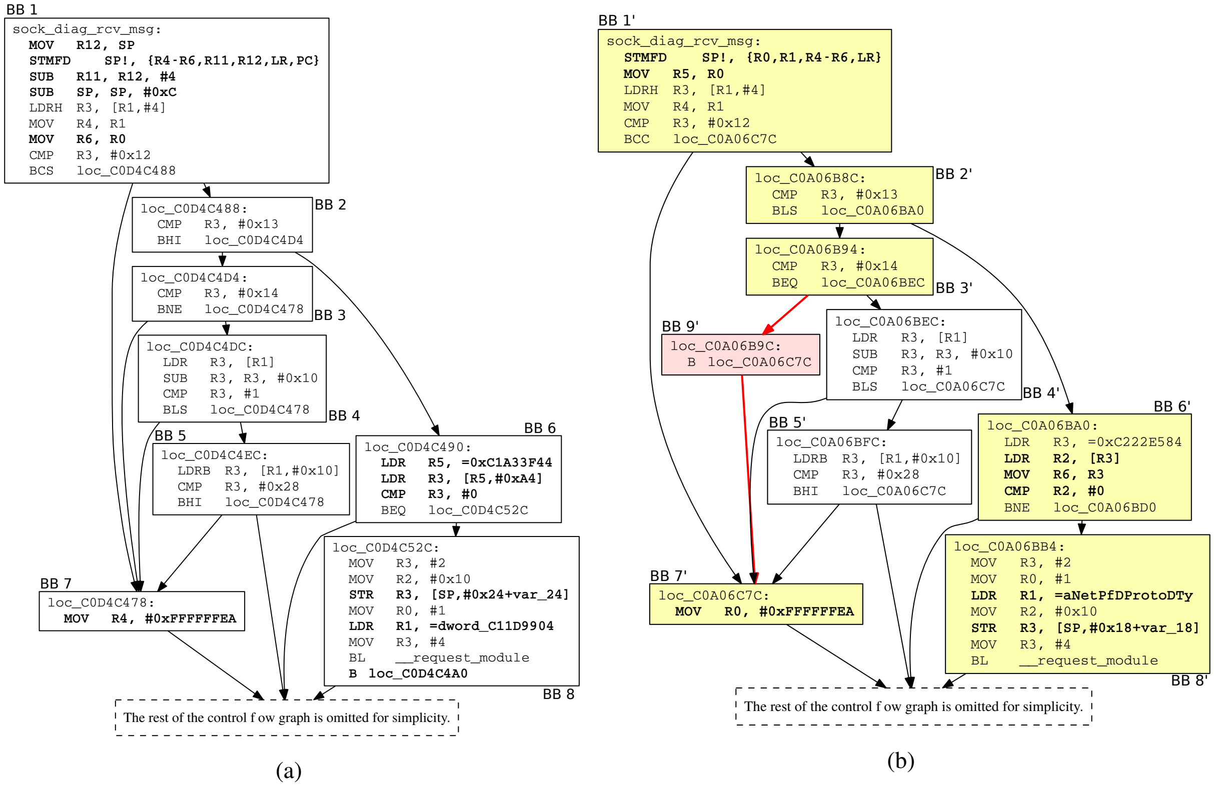
We randomly picked some functions to manually verify the outcome of semantic matching. For example, the source code of `sock_diag_rcv_msg` (the function related to CVE-2013-1763) is exactly the same in Samsung Galaxy Note Edge (Android 5.0.1, Linux kernel 3.10.40) and Huawei Honor 6 Plus (Android 4.4, Linux kernel 3.10.30) [6]. However, its binaries are very different between these two devices because of the different compilers and kernel configurations. Figure 9a and 9b show a part of the disassembly code for these two binaries, respectively. The syntactic differences are highlighted. There are changes to the order of instructions (`BB8` on the left vs `BB8'` on the right), register

[5] Only kernels sharing symbols are considered in the clustering.

[6] Both vendors have released the source code for their devices.

Kernel Function	CVE ID	# of Opcode Clusters	% of the Largest Opcode Cluster	# of Syntax Clusters	% of the Largest Syntax Cluster	# of Semantic Clusters	% of Largest Semantic Cluster	Semantic Matching Time Cost	# of Instructions	# of Basic Blocks
sock_diag_rcv_msg	2013-1763	35	25.0%	7	73.5%	3	75.5%	10.5s	72	16
perf_swevent_init	2013-2094	9	55.9%	5	55.9%	2	96.3%	24.6s	81	22
fb_mmap	2013-2596	26	20.2%	7	44.4%	5	66.9%	12.2s	102	15
__get_user_1	2013-6282	3	92.4%	2	92.4%	2	98.0%	3.2s	6	2
futex_requeue	2014-3153	54	14.8%	9	71.0%	3	99.3%	35.8s	459	107
msm_isp_proc_cmd	2014-4321	42	22.0%	5	66.5%	3	42.8%	8.8s	385	68
send_write_packing_test_read	2014-9878	12	57.6%	4	61.2%	1	100%	4.9s	25	4
msm_cci_validate_queue	2014-9890	6	59.5%	4	84.9%	2	72.4%	6.7s	77	8
ping_unhash	2015-3636	36	12.5%	5	75.7%	3	50.5%	4.6s	54	8
q6lsm_snd_model_buf_alloc	2015-8940	29	34.0%	9	36.6%	5	44.2%	9.9s	104	20
sys_perf_event_open	2016-0819	22	36.3%	6	46.9%	6	84.2%	34.6s	569	118
kgsl_ioctl_gpumem_alloc	2016-3842	16	35.4%	3	88.8%	4	46.0%	4.7s	79	11
is_ashmem_file	2016-5340	6	89.6%	2	93.9%	2	98.1%	0.8s	23	3

Table 5: Clustering 1,139 kernels for each function by syntax and semantics. The last-but-two column lists the time of semantic matching to compare Nexus 5 (Android 4.4.2, kernel 3.4.0) and Samsung Note Edge (Android 6.0.1, kernel 3.10.40). The experiment was conducted on an Intel E5-2650 CPU with 16GB of memory, and the results are the average over 10 repeats. The last two columns list the number of instructions and basic blocks for each function in Nexus 5.



BB 1
BB 2
BB 3
BB 4
BB 5
BB 6
BB 7
BB 8
The rest of the control f ow graph is omitted for simplicity.
BB 1'
BB 2'
BB 3'
BB 4'
BB 5'
BB 6'
BB 7'
BB 8'
BB 9'
The rest of the control f ow graph is omitted for simplicity.

```
sock_diag_rcv_msg:
  MOV   R12, SP
  STMFD     SP!, {R4-R6,R11,R12,LR,PC}
  SUB   R11, R12, #4
  SUB   SP, SP, #0xC
  LDRH  R3, [R1,#4]
  MOV   R4, R1
  MOV   R6, R0
  CMP   R3, #0x12
  BCS   loc_C0D4C488

loc_C0D4C488:
  CMP   R3, #0x13
  BHI   loc_C0D4C4D4

loc_C0D4C4D4:
  CMP   R3, #0x14
  BNE   loc_C0D4C478

loc_C0D4C4DC:
  LDR   R3, [R1]
  SUB   R3, R3, #0x10
  CMP   R3, #1
  BLS   loc_C0D4C478

loc_C0D4C4EC:
  LDRB  R3, [R1,#0x10]
  CMP   R3, #0x28
  BHI   loc_C0D4C478

loc_C0D4C490:
  LDR   R5, =0xC1A33F44
  LDR   R3, [R5,#0xA4]
  CMP   R3, #0
  BEQ   loc_C0D4C52C

loc_C0D4C478:
  MOV   R4, #0xFFFFFFEA

loc_C0D4C52C:
  MOV   R3, #2
  MOV   R2, #0x10
  STR   R3, [SP,#0x24+var_24]
  MOV   R0, #1
  LDR   R1, =dword_C11D9904
  MOV   R3, #4
  BL    __request_module
  B  loc_C0D4C4A0
```

(a)

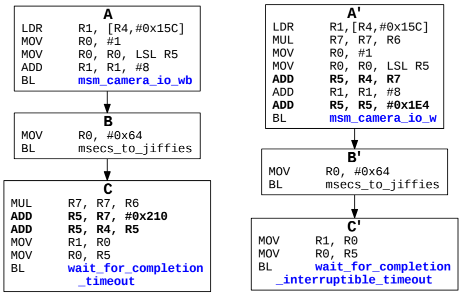
```
sock_diag_rcv_msg:
  STMFD     SP!, {R0,R1,R4-R6,LR}
  MOV   R5, R0
  LDRH  R3, [R1,#4]
  MOV   R4, R1
  CMP   R3, #0x12
  BCC   loc_C0A06C7C

loc_C0A06B8C:
  CMP   R3, #0x13
  BLS   loc_C0A06BA0

loc_C0A06B94:
  CMP   R3, #0x14
  BEQ   loc_C0A06BEC

loc_C0A06B9C:
  B  loc_C0A06C7C

loc_C0A06BEC:
  LDR   R3, [R1]
  SUB   R3, R3, #0x10
  CMP   R3, #1
  BLS   loc_C0A06C7C

loc_C0A06BFC:
  LDRB  R3, [R1,#0x10]
  CMP   R3, #0x28
  BHI   loc_C0A06C7C

loc_C0A06BA0:
  LDR   R3, =0xC222E584
  LDR   R2, [R3]
  MOV   R6, R3
  CMP   R2, #0
  BNE   loc_C0A06BD0

loc_C0A06C7C:
  MOV   R0, #0xFFFFFFEA

loc_C0A06BB4:
  MOV   R3, #2
  MOV   R0, #1
  LDR   R1, =aNetPfDProtoDTy
  MOV   R2, #0x10
  STR   R3, [SP,#0x18+var_18]
  MOV   R3, #4
  BL    __request_module
```

(b)

Figure 9: `sock_diag_rcv_msg` of (a) Huawei Honor 6 Plus (PE-TL10) with Android 4.4 and Linux kernel 3.10.30, compiled by GCC 4.7, and (b) Samsung Galaxy Note Edge (N915R4) with Android 5.0.1 and Linux kernel 3.10.40, compiled by GCC 4.8. Basic blocks and control flows with different syntax are highlighted.

allocation (`BB7` vs `BB7'`), instruction selection (`BB2` vs `BB2'`), and control flow (additional `BB9'` in the Samsung kernel). KARMA's semantic matching can abstract these syntactic differences and put these two binaries of `sock_diag_rcv_msg` into the same cluster. That is, both can be patched by the same CVE-2013-1763 patch discussed in Section 2.4.

Semantic matching can also separate kernel functions that are incorrectly classified together by the syntax matching. For example, the control flow and most instructions of function `msm_cci_validate_queue` (the function related to CVE-2014-9890) are identical in the



```
A
LDR     R1, [R4,#0x15C]
MOV     R0, #1
MOV     R0, R0, LSL R5
ADD     R1, R1, #8
BL      msm_camera_io_wb

B
MOV     R0, #0x64
BL      msecs_to_jiffies

C
MUL     R7, R7, R6
ADD     R5, R7, #0x210
ADD     R5, R4, R5
MOV     R1, R0
MOV     R0, R5
BL      wait_for_completion
          _timeout
```

```
A'
LDR     R1,[R4,#0x15C]
MUL     R7, R7, R6
MOV     R0, #1
MOV     R0, R0, LSL R5
ADD     R5, R4, R7
ADD     R1, R1, #8
ADD     R5, R5, #0x1E4
BL      msm_camera_io_w

B'
MOV     R0, #0x64
BL      msecs_to_jiffies

C'
MOV     R1, R0
MOV     R0, R5
BL      wait_for_completion
  _interruptible_timeout
```

Figure 10: Three semantically different basic blocks of `msm_cci_validate_queue` in Oppo 3007 (left) and Samsung N910G (right). They have different callees and arguments, and thus different semantics.

kernel of Oppo 3007 (Android 4.4.4, kernel 3.10.28) and Samsung N910G (Android 6.0.1, kernel 3.10.40). A simple syntactic matching algorithm would consider them similar. These functions are shown in Fig. 10 (only basic blocks with different semantics are shown). However, KARMA's semantic matching algorithm considers basic block A and A', C and C' to be different because their last instructions call different functions with different arguments. Consequently, KARMA needs to use two patches to fix this vulnerability in these devices. A further investigation shows that KARMA can actually use the same patch for CVE-2014-9890 to fix both kernels because it only needs to validate the arguments, which are the same for both functions.

Finally, KARMA's semantic matching is quite efficient. It simplifies symbolic execution by considering most functions remain unchanged. The last-but-two column of Table 5 lists the time used by semantic matching to compare each listed function in two kernels. The analysis time increases with the complexity of the function, but they are all less than 36 seconds with an average of 12.5 seconds. Without this heuristics, it will take much longer and may never finish in some cases.

3.3 Evaluation of Performance

To evaluate the performance overhead of KARMA, we experimented with both a standard Android benchmark (CF-Bench [9]) and a syscall-based micro-benchmark. Both benchmarks were run on Google Nexus 5 with Android 4.4. Each reported result is the average over 20 measurements. The standard deviation of the results is negligible. Overall, we find that KARMA does not introduce noticeable time lag to regular operations of the test device. Considering the fact that most critical kernel vulnerabilities exist in less-hot code paths (e.g., device drivers' `ioctl` interfaces as shown in Table 6), we consider KARMA's performance is sufficient for real-world deployment.

The first benchmark measures the whole system performance with CF-Bench. We tested the performance of the following four configurations: the original kernel without any patches, the kernel with the patch for Towelroot, the kernel with the patch for PingPong root, and the kernel with both patches. The results are shown in Fig. 11. The measured performance is virtually the same for all four configurations. This benchmark shows that KARMA's kernel engine has minimal impact on the performance if patches are not frequently executed.

To further quantify the overhead of KARMA, we measured the execution time of a syscall with several different patches executed by a single Lua engine. We inserted a hook point in the execution path of a selected syscall (i.e., the patch was always executed for this syscall) and measured the execution time of the syscall under the following conditions:

- The patch simply returns 0. This reflects the run-time cost of the trampoline for function hooking. It takes about 0.42μs to execute.
- The patch contains a set of `if/elseif/else` conditional statements. This simulates patches that validate input arguments. It takes about 0.98μs to execute.
- The patch consists of a single read of the kernel memory. This measures the overhead of the Lua APIs provided by KARMA. It takes about 0.82μs to execution.
- To simulate more complex patches, we created a patch with a mixture of assignments, memory reads, and conditional statements. It takes about 3.74μs to execute.

The results are shown in Figure 12. In each test, the syscall was invoked in a tight loop for a thousand times, and each result is the average of 20 runs. To put this into context, we counted all the syscalls made by Google Chrome for Android during one minute of browsing. The most frequently made syscall was `gettimeofday` for about 110,000 times. This translates to about 0.55 seconds (0.9%) of extra time even if we assume the patch takes 5μs for each invocation. In summary, KARMA only incurs negligible performance overhead and performs sufficiently well for real-world deployment.

4 Discussion and Future Work

In this section, we discuss potential improvements to KARMA and the future work. First, KARMA aims at protecting the Android kernel from exploits because the kernel has a high privilege and its compromise has serious consequences on user security and privacy. An approach similar to KARMA can be applied to the Android framework and user-space apps. In addition, Android O formalizes the interface between the Android framework

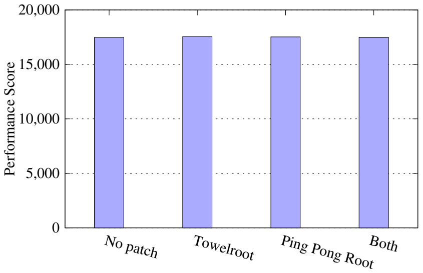


Figure 11: Performance scores by CF-Bench.

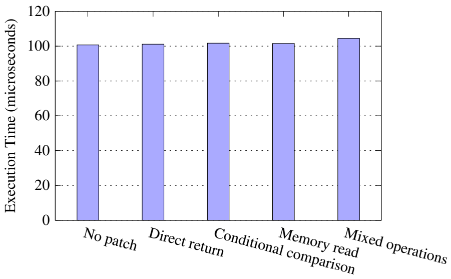


Figure 12: Execution time of `chmod` with different patches.

and the vendor implementation so that, eventually, the Android framework can be updated independent of the vendor implementation (aka. project Treble [20]). This will at least partially address the user-space update problem. However, project Treble does not address the kernel update problem. Android kernels are still fragmented and out-of-date. A system like KARMA is still necessary.

Second, KARMA's patches are written in the Lua programming language. It relies on the Lua engine to strictly confine patches' run-time behaviors. However, this approach increases the kernel's trusted computing base despite the fact that the Lua engine is relatively mature and secure. Executing patches on the Lua engine also negatively impacts the performance, especially if the system is under heavy load (in reality, this is not a concern because most Android kernel vulnerabilities are on the kernel's cold paths, such as device drivers' `ioctl` functions, as shown in Table 6). We are investigating alternative designs that can achieve similar security guarantees, such as BPF [8] and sandboxed binary patches.

Third, KARMA leverages the existing error handling code in the kernel to handle filtered malicious inputs, in order to keep the kernel as stable as possible. However, error handling code has been shown to contain vulnerabilities [36], and this design may leak resources and even cause deadlocks (KARMA does not allow patches themselves to release resource because that requires writing to the kernel). We did not find this to be a constraint during our experiment with *all* the critical Android kernel vulnerabilities. KARMA's reference patch is often a direct translation of the official source-code patch, which should have properly released the resources. If an official patch cannot be translated to a level-1 or level-2 patch, we can fall back to the level-3 (binary) patch. Level-3 patches are more flexible but require careful vetting.

Fourth, KARMA uses symbolic execution to semantically match two vulnerable functions. The approach is sufficient for our purpose in practice because many kernel functions are rather stable across devices and Android releases. In theory, the approach is not sound. It is a trade-off between soundness and scalability. Many systems make a similar trade-off because symbolic execution itself is neither very scalable nor very precise (e.g., how to handle loops). We are improving our method to better identify vulnerable functions and adapt patches. If KARMA's automated adaption cannot find a proper function to patch, we can fall back to the binary patch for this particular vulnerability.

Lastly, KARMA is a third-party kernel live patching system. Patches can be promptly delivered to user devices without the long wait caused by vendors and carriers. However, without testing performed by vendors and carries, its patches could cause stability issues in the user devices. Our implementation allows users to selectively disable a problematic patch. With KARMA's cloud service, we can automatically blacklist such patches from specific device models. We can also work with device vendors so patches can be quickly tested before release.

5 Related Work

Kernel live patching: the first category of the related work consists of a number of kernel live patching systems, such as kpatch [23], kGraft [22], Ksplice [27], and KUP [37]. They assume that the kernel source code is available (a reasonable assumption for their purpose) and create live patches from source code patches. Their patches are however in the binary form. This design does not fit the threat model of KARMA. First, although Android kernel is licensed in GPL, many Android vendors, small and large alike [19], do not (promptly) release their kernel source code. Second, these systems lack a mechanism to automatically adapt a kernel patch to different Android devices. An important design goal of KARMA is adaptiveness so that it can scale to the Android ecosystem. Third, binary patches are prone to misuse because they are hard to understand and vet, and these systems have no strong confinement of patches' run-time behaviors. KARMA has been designed specifically to address all these challenges in a live kernel patching system for Android.

Among these systems, kpatch [23] and kGraft [22] replace a whole vulnerable function with the patched version. They differ in how patches are applied: kpatch stops all the running processes and ensures that none of these processes are running inside the function to be patched (similar to KARMA). kGraft instead maintains two copies of each patched function at the same time and dynamically decides which copy to execute. Specifically, the kernel code active at the time of patching (e.g., system calls, kernel threads, and interrupt handlers) is dispatched to the original version until it reaches a completion point; all other code is dispatched to the patched version. Like kpatch, Ksplice [27] also stops the machine to apply patches. However, Ksplice can patch individual instructions instead of replacing whole functions. These systems share the same limitation that they cannot support patches that "change the semantics of persistent data structures [27]". To address that, KUP [37] employs the process checkpoint-and-restart to implement kernel hot patching. Specifically, it checkpoints all the user processes, replaces the running kernel with the patched version, and then restores these user processes. Because it replaces the whole kernel, KUP can support all kinds of patches. However, restoring external resources (e.g., sockets) is often problematic for checkpoint-and-restore systems, including KUP.

Semantic matching: the second category of the related work includes systems that compare semantics or similarity of two functions [31, 32, 39, 40]. BinHunt [32] first uses symbolic execution to compute semantic similarity of basic blocks and uses a graph isomorphism algorithm to further compare the similarity of CFGs (control-flow graphs). Their follow-up work, iBinHunt [40], extends BinHunt with the inter-procedural control-flow graph comparison. However, whole-program comparison could be very time-consuming. To solve that, iBinHunt runs the program with taint tracking and only compares basic blocks within the same data flows. This approach is not suitable for KARMA because none of the commercial Android devices support kernel dynamic taint tracking or whole-kernel instrumentation. CoP [39] also uses symbolic execution to compute the semantic similarity of basic blocks and uses the longest common sub-sequence of linearly independent paths to measure the similarity of programs. KARMA uses symbolic execution to solve syntax differences in semantically-equivalent functions. In addition, it leverages the fact that most kernel functions remain semantically similar across different kernel versions to significantly speed-up the comparison. DiscovRE [31] takes a different approach by using the syntactic information (i.e., structural and numeric features) to compare function similarities. This can significantly improve the analysis efficiency. KARMA requires a more precise comparison than that can be provided by syntax-based approaches.

Automatic patch/filter generation: the third category of the related work includes systems that aim at automatically generating patches or input filters. For example, Talos [34] is a vulnerability rapid response system. It inserts SWRRs (Security Workarounds for Rapid Response) into the kernel source code in order to temporarily protect kernel vulnerabilities from being exploited. Talos shares a similar goal as KARMA, and both rely on the kernel's error handling code to gracefully neutralize attacks. Talos' source code based approach cannot be applied to the fragmented Android ecosystem. To address the fragmentation problem, KARMA can automatically adapt a patch to other devices and strictly confine the run-time behaviors of its patches. ClearView [41] learns invariants of a program during a dynamic training phase. When program failure happens, it identifies the failure-related invariants and uses them to generate patches for the program. PAR [38] proposes a pattern-based automatic program repair framework. Its generated patches resemble the patterns learned from human-written patches. ASSURE introduces rescue points that can recover software from unknown exploits while maintaining system integrity and availability [45]. ShieldGen [29] is a system for automatically generating vulnerability signatures (i.e., data patches). Signature-based filtering can only block known attacks. To address that, ShieldGen leverages protocol specifications to generate more exploits from an initial sample. Bouncer [28] uses static analysis and dynamic symbolic execution to create comprehensive input filters to protect software from bad inputs. Compared to these systems, KARMA aims at protecting kernel vulnerabilities for many Android systems and have a different design.

6 Summary

We have presented the design, implementation, and evaluation of KARMA, an adaptive live patching system for Android kernel vulnerabilities. By filtering malicious user inputs, KARMA can protect most Android kernel vulnerabilities from exploits. Compared to existing kernel live patching systems, the unique features of KARMA are that it can automatically adapt a reference patch for many Android devices, and it strictly confines the run-time behaviors of its patches. These two features allow KARMA to scale to a large, fragmented Android ecosystem. Our evaluation results demonstrated that KARMA can protect most critical Android kernel vulnerabilities in many devices with negligible performance overhead.

7 Acknowledgments

We would like to thank our shepherd David Lie and the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. Yue Chen and Zhi Wang were supported in part by the US National Science Foundation (NSF) under Grant 1453020 and a grant from Baidu X-Lab. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

[1] Android Fragmentation: There Are Now 24,000 Devices from 1,300 Brands. `http://www.zdnet.com/article/android-fragmentation-there-are-now-24000-devices-from-1300-brands/`.

[2] Android Platform Versions. `https://developer.android.com/about/dashboards/index.html`.

[3] Android Security Bulletins. `https://source.android.com/security/bulletin/`.

[4] Android System and Kernel Security. `https://source.android.com/security/overview/kernel-security.html`.

[5] Android Towelroot Exploit Used to Deliver Dogspectus Ransomware. `https://www.bluecoat.com/security-blog/2016-04-25/android-exploit-delivers-dogspectus-ransomware`.

[6] angr. `http://angr.io`.

[7] Anonymous Citation. This citation is anonymized to avoid leaking the authors' identities.

[8] Berkeley Packet Filter (BPF). `https://www.kernel.org/doc/Documentation/networking/filter.txt`.

[9] CF-Bench. https://play.google.com/store/apps/details?id=eu.chainfire.cfbench.

[10] CVE-2013-2595 Kernel Patch. `https://www.codeaurora.org/patches/quic/la/.PATCH_24430_iwoLuwW321heHwW.tar.gz`.

[11] CVE-2013-2596 Kernel Patch. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=fc9bbca8f650e5f738af8806317c0a041a48ae4a`.

[12] CVE-2013-6282 Kernel Patch. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=8404663f81d212918ff85f493649a7991209fa04`.

[13] CVE-2014-3153 Kernel Patch. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=e9c243a5a6de0be8e584c604d353412584b592f8`.

[14] CVE-2014-3153 (Towelroot). `https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153`.

[15] CVE-2015-1805 Kernel Patch. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=637b58c2887e5e57850865839cc75f59184b23d1`.

[16] CVE-2015-3636 Kernel Patch. `http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=a134f083e79fb4c3d0a925691e732c56911b4326`.

[17] From HummingBad to Worse. `https://blog.checkpoint.com/wp-content/uploads/2016/07/HummingBad-Research-report_FINAL-62916.pdf`.

[18] Ghost Push: An Un-Installable Android Virus Infecting 600,000+ Users Per Day. `http://www.cmcm.com/blog/en/security/2015-09-18/799.html`.

[19] GPLv2 and Its Infringement by Xiaomi. `http://www.xda-developers.com/gplv2-and-its-infringement-by-xiaomi/`.

[20] Here comes Treble: A modular base for Android. `https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html`.

[21] Kemoge: Another Mobile Malicious Adware Infecting Over 20 Countries. `https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html`.

[22] kGraft: Live patching of the Linux kernel. `http://events.linuxfoundation.org/sites/events/files/slides/kGraft.pdf`.

[23] kpatch: Dynamic Kernel Patching. `https://github.com/dynup/kpatch`.

[24] Linux Kernel Coding Style. `https://www.kernel.org/doc/Documentation/CodingStyle`.

[25] Linux Kernel Livepatch. `https://www.kernel.org/doc/Documentation/livepatch/livepatch.txt`.

[26] lunatik. `https://github.com/lunatik-ng/lunatik-ng`.

[27] Arnold, J., and Kaashoek, M. F. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems* (New York, NY, USA, 2009), ACM, pp. 187–198.

[28] Costa, M., Castro, M., Zhou, L., Zhang, L., and Peinado, M. Bouncer: Securing Software by Blocking Bad Input. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), vol. 41, ACM, pp. 117–130.

[29] Cui, W., Peinado, M., Wang, H. J., and Locasto, M. E. Shieldgen: Automatic Data Patch Generation for Unknown Vulnerabilities with Informed Probing. In *Proceedings of the 28th IEEE Symposium on Security and Privacy* (2007), IEEE, pp. 252–266.

[30] CVE-2015-3636. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3636`.

[31] Eschweiler, S., Yakdan, K., and Gerhards-Padilla, E. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Network and Distributed System Security Symposium* (2016).

[32] Gao, D., Reiter, M. K., and Song, D. Binhunt: Automatically Finding Semantic Differences in Binary Programs. In *International Conference on Information and Communications Security* (2008), Springer, pp. 238–255.

[33] How-To Geek. Why Do Carriers Delay Updates for Android But Not iPhone? `http://www.howtogeek.com/163958/why-do-carriers-delay-updates-for-android-but-not-iphone`.

[34] Huang, Z., D'Angelo, M., Miyani, D., and Lie, D. Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. In *Proceedings of the 37th IEEE Symposium on Security and Privacy* (2016).

[35] Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. The Evolution of Lua. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages* (New York, NY, USA, 2007), ACM, pp. 2–1.

[36] Jana, S., Kang, Y., Roth, S., and Ray, B. Automatically Detecting Error Handling Bugs using Error Specifications. In *25th USENIX Security Symposium (USENIX Security '16)* (Austin, August 2016).

[37] Kashyap, S., Min, C., Lee, B., Kim, T., and Emelyanov, P. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference* (2016).

[38] Kim, D., Nam, J., Song, J., and Kim, S. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE, pp. 802–811.

[39] Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2014), ACM, pp. 389–400.

[40] Ming, J., Pan, M., and Gao, D. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Proceedings of International Conference on Information Security and Cryptology* (2012), Springer, pp. 92–109.

[41] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., and Rinard, M. Automatically Patching Errors in Deployed Software. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (October 2009).

[42] Rosenberg, D. QSEE TrustZone Kernel Integer Overflow Vulnerability. In *Black Hat USA* (2014).

[43] Shen, D. Exploiting Trustzone on Android. In *Black Hat USA* (2015).

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

[45] Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., and Keromytis, A. D. ASSURE: Automatic Software Self-healing Using REscue points. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems* (March 2009).

[46] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*. Wiley, 2012.

[47] Zhang, H., She, D., and Qian, Z. Android Root and Its Providers: A Double-Edged Sword. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2015), ACM, pp. 1093–1104.

A Evaluation of Applicability: Additional Case Studies

A.1 CVE-2014-3153 (Towelroot)

this vulnerability is the second most-used one to root Android devices, known as "Towelroot". It lies in the `futex_requeue` function, which takes the addresses of two futexes as arguments. By design, the function should only re-queue from a non-PI (priority inheritance [46]) futex to a PI futex. However, this condition is violated if these two addresses point to the same futex. This leads to an exploitable dangling pointer condition. To fix this bug, Linux simply adds a check to ensure that these two futex addresses are different [13]. This vulnerability can be similarly fixed in KARMA by hooking the `futex_requeue` function, obtaining its arguments, and compare their equality. The patch returns `-EINVAL` if an attack is detected (Figure 4).

A.2 CVE-2015-3636 (PingPong Root)

This is another popular vulnerability used to root Android devices, known as "PingPong Root". It originates in the interaction between the socket and `hlist` functions. Specifically, when `hlist_nulls_del(&sk-> sk_nulls_node)` is called, it assigns `LIST_POISON2` to `sk->sk_nulls_node.pprev`. `LIST_POISON2` is defined as the constant of `0x200200`. If interpreted as an address, address `LIST_POISON2` can be mapped by a malicious app in the user space without any permissions. A second call to `connect` by the attacker will result in a use-after-free on this attacker-controlled address, compromising the kernel. The Linux patch sets the pointer to `NULL` in the `ping_unhash` function [16]. However, this method cannot be applied by KARMA because its patch is prohibited from writing to the kernel memory. Instead, the patch checks if `sk->sk_nulls_node.pprev` equals to `LIST_POISON2`. If so, it returns an error code without freeing the associated memory. This blocks the exploit but leaves the socket object on the list. This patch is not clean, but it works and does not impact the kernel's functionalities. Alternatively, KARMA can hook `connect` in the kernel to prevent reusing the freed socket.

Vulnerability	Hotpatching Using KARMA	Adaptable?
CVE-2016-7117	Hook `__sys_recvmmsg` and its invocation of `fput`. On returning of `fput`, check if `__sys_recvmmsg`'s `err` is not equal to 0 and not equal to `-EAGAIN`. If so, return `err` and skip the rest execution.	✓
CVE-2016-5340	Hook `is_ashmem_file` and check the full path of the input file. Only return `True` if the full path is `/dev/ashmem`. Otherwise return `False`.	✓
CVE-2016-4470	Hook `key_reject_and_link` and its invocation of `__key_link_end`. Check if `link_ret` is 0 before calling into `__key_link_end`. If so, simply return. `key_reject_and_link` is `void` typed so any return value is fine.	✓
CVE-2016-3951	It requires writing to kernel memory, violating KARMA's basic constraint.	Level-3
CVE-2016-3841	Hook `do_ipv6_setsockopt` to avoid concurrent access to the socket options of the same socket fd.	✓
CVE-2016-3775	Hook `aio_setup_single_vector` and check if the input `kiocb->ki_nbytes` exceeds `MAX_RW_COUNT`. If so, return `-EFAULT`.	✓
CVE-2016-3768	It requires to skip some instructions and continue execution afterwards, which is not an allowed operation by KARMA.	Level-3
CVE-2016-3767	Hook `mtk_p2p_wext_discovery_results` etc. functions of which the bodies are deleted by the official patch, and simply return 0.	✓
CVE-2016-3134	Android does not enable `CONFIG_USER_NS` so this should not be a direct threat to Android devices. But KARMA can still fix it by iterating `newpos = pos + e->next_offset` to check if there is a out-of-bound access.	✓
CVE-2016-2503	It requires to reorder the instructions (to change when to take the lock). This is not an allowed operation by KARMA.	Level-3
CVE-2016-2474	Hook `hdd_parse_ese_beacon_req` and check the `tempInt` read from the argument `pValue`. If it exceeds `SIR_ESE_MAX_MEAS_IE_REQS`, return `-EINVAL`.	✓
CVE-2016-2468	Hook `_kgsl_sharedmem_page_alloc` and validate the input `size`.	✓
CVE-2016-2467	Hook `msm_compr_ioctl` and its invocation of `__copy_from_user`. Check if the `params_length` passed into `__copy_from_user` exceeds `MAX_AC3_PARAM_SIZE`. If so, return error code from `__copy_from_user` without executing into it.	✓
CVE-2016-2466	Hook `adm_get_params` and check if `adm_get_parameters[0]` exceeds `ADM_GET_PARAMETER_LENGTH-1` and `params_length/sizeof(int)`. If so, return `-EINVAL`.	✓
CVE-2016-2465	Hook the concerned functions in `drivers/video/msm/mdss/mdss_debug.c` patched in the original patch, and their invocations of `__copy_to_user`. Validate `len` and `count`, and return `-EFAULT` in case of exploit conditions.	✓
CVE-2016-2067	Hook `check_vma` and return `-EFAULT` if `vma->vm_flags & memdesc->flags != memdesc->flags`.	✓
CVE-2016-2062	Hook `adreno_perfcounter_query_group` and its invocation of `kmalloc`. On the entry of `kmalloc`, check if `t` is larger than `count`.	✓
CVE-2016-0844	Hook `ipa_wwan_ioctl` and its invocation of `find_mux_channel_index`. On entry of `find_mux_channel_index`, if the value of `rmnet_index` exceeds `MAX_NUM_OF_MUX_CHANNEL`, return `-EFAULT` directly.	✓
CVE-2016-0843	Hook `msm_l2_test_set_ev_constraint` and check if `shift_idx >= PMU_CODES_SIZE`. Return `-EINVAL` in case of that.	✓
CVE-2016-0820	Hook `priv_get_struct` and its invocation of `__copy_from_user`, check if `prIwReqData->data.length>u4CopyDataMax` and return `-EFAULT` if so.	✓
CVE-2016-0806	Hook `iw_softap_set_channel_range` and check if the caller has the capability `CAP_NET_ADMIN`, return `-EPERM` if not.	✓
CVE-2016-0805	Hook `get_krait_evtinfo` and check if reg exceeds `krait_max_l1_reg`, return `-EINVAL` if so.	✓
CVE-2016-0801	Hook `wl_validate_wps_ie` and check if `subelt_len` exceeds the size of `devname` (100). Hook `wl_notify_sched_scan_results` and its invocation of `memcpy` and check if the buffer length passed in exceeds `DOT11_MAX_SSID_LEN`.	✓
CVE-2016-0758	Hook `asn1_find_indefinite_length` and check if `dp` is larger than `datalen`. Return `-1` if so.	✓
CVE-2016-0728	Hook `join_session_keyring` and iterate the `keyring`. Return error if `keyring->usage` reaches the overflow boundary (0xFFFFFFFF).	✓

Vulnerability	Hotpatching Using KARMA	Adaptable?
CVE-2015-8942	Hook msm_cpp_subdev_ioctl, if the argument cmd equals to VIDIOC_MSM_CPP_IOMMU_DETACH, from its argument sd obtain cpp_dev->stream_cnt and check if it equals to 0.	✓
CVE-2015-8941	Hook msm_isp_axi_check_stream_state and iterate over the input stream_cfg_cmd->stream_handle to see if one exceeds MAX_NUM_STREAM. The other vulnerable functions can be fixed in the same way.	✓
CVE-2015-8940	Hook q6lsm_snd_model_buf_alloc and check if the integer argument len is out of range.	✓
CVE-2015-8939	Hook mdp4_argc_process_write_req and check if the input pgc_ptr->num_r/g/b_stages are out of range.	✓
CVE-2015-8938	Hook msm_isp_send_hw_cmd and check if the ioctl input arguments satisfy the constraints updated by the official patch. The constraint list is long so omitted here.	✓
CVE-2015-8816	Fixing the problem requires locking and increasing the reference of the usb_hub structure, thus the patch needs to write to kernel memory.	Level-3
CVE-2015-6640	Hook the system call prctl and check if the corresponding argument as the end passed to prctl_set_vma_anon_name is out of range.	✓
CVE-2015-6638	Hook PVRSRVSyncPrimSetKM and check if the input psSyncBlk->ui32BlockSize is smaller than another input ui32Index * sizeof(IMG_UINT32).	✓
CVE-2015-6619	The official patch is to remove all .tmpfile handlers. So we can simply hook such handlers and always return -EINVAL.	✓
CVE-2015-2686	Hook sys_sendto/sys_recvfrom and check if the input buff and len/size are out of range.	✓
CVE-2015-0570	Hook __iw_softap_setwpsie and check if ioctl arguments have improper length, same as the official patch. The check list is long so omitted here.	✓
CVE-2014-9902	Hook dot11fUnpackIeCountry and dot11fUnpackIeSuppChannels to validate the value of the input ielen.	✓
CVE-2014-9891	Hook __qseecom_process_rpmb_svc_cmd and validate if the input req_ptr fields passed in from user space are out of range.	✓
CVE-2014-9890	Hook msm_cci_validate_queue and validate if cmd_size extracted from the inputs is larger than 10.	✓
CVE-2014-9887	Hook qseecom_send_modfd_cmd and its invocation of __copy_from_user. Validate req.cmd_req_len obtained from user space.	✓
CVE-2014-9884	Hook qseecom_register_listener etc. handlers to validate pointers passed in from user space, same as the official patch.	✓
CVE-2014-9883	Hook extract_dci_log and check for the integer overflow condition of the input log_length.	✓
CVE-2014-9882	Hook iris_vidioc_s_ctrl. If the input ctrl->id is V4L2_CID_PRIVATE_IRIS_RIVA_ACCS_LEN/_POKE, validate if the copied data length exceeds MAX_RIVA_PEEK_RSP_SIZE.	✓
CVE-2014-9881	Hook iris_vidioc_s_ext_ctrls and perform range/overflow check on the input ctrl.	✓
CVE-2014-9880	Hook vid_enc_ioctl and its invocation of __copy_from_user. Validate seq_header fetched from user space.	✓
CVE-2014-9879	Hook mdp3_histogram_start and validate its input req; hook mdp3_pp_ioctl and validate mdp_pp obtained from user space.	✓
CVE-2014-9878	Hook send_write_packing_test_read and validate its input buffer and count.	✓
CVE-2014-9869	Hook msm_isp_ functions as specified in the official patch, and validate if stats_idx from input exceeds MSM_ISP_STATS_MAX.	✓
CVE-2014-9868	Hook msm_csiphy_release and validate the value of input csi_lane_params->csi_lane_mask.	✓
CVE-2014-9529	Fixing the issue requires to change the instruction order (delay the reference put). This is not a secure operation permitted by KARMA.	Level-3

Table 6: A partial list of recent critical Android kernel vulnerabilities and KARMA's effectiveness to create adaptable patches for them. Some adaptable vulnerabilities are omitted due to the space constraint.