# InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android

Yaohui Chen* Yuping Li† Long Lu* Yueh-Hsun Lin††
Hayawardh Vijayakumar‡ Zhi Wang¶ Xinming Ou†

*Northeastern University †University of South Florida ‡Samsung Research America
¶Florida State University ††JD Research Center

***Abstract*—Hot-patches, easier to develop and faster to deploy than permanent patches, are used to timely (and temporarily) block exploits of newly discovered vulnerabilities while permanent patches are being developed and tested. Researchers recently proposed to apply hot-patching techniques to system programs on Android as a quick mitigation against critical vulnerabilities. However, existing hot-patching techniques, though widely used in conventional computers, are rarely adopted by Android OS or device vendors in reality. Our study uncovers a major hurdle that prevents existing hot-patching methods from being effective on mobile devices: after being developed, hot-patches for mobile devices have to go through lengthy compatibility tests that Android device partners impose on all system code updates. This testing and release process can take months, and therefore, erase the key benefit of hot-patches (i.e., quickly deployable).**

**We propose InstaGuard, a new approach to hot-patch for mobile devices that allows for instant deployment of patches (i.e., "carrier-passthrough") and fast patch development for device vendors. Unlike existing hot-patching techniques, InstaGuard avoids injecting new code to programs being patched. Instead, it enforces instantly updatable rules that contain no code (i.e., no carrier test is needed) to block exploits of unpatched vulnerabilities in a timely fashion. When designing InstaGuard, we overcame two major challenges that previous hot-patching methods did not face. First, since no code addition is allowed, InstaGuard needs a rule language that is expressive enough to mitigate various kinds of vulnerabilities and efficient to be enforced on mobile devices. Second, rule generation cannot require special skills or much efforts from human users. We designed a new language for hot-patches and an enforcement mechanism based on the basic debugging primitives supported by ARM CPUs. We also built RuleMaker, a tool for automatically generating rules for InstaGuard based on high-level, easy-to-write vulnerability descriptions. We have implemented InstaGuard on Google Nexus 5X phones. To demonstrate the coverage of InstaGuard, we show that InstaGuard can handle all critical CVEs from Android Security Bulletins reported in 2016. We also conduct unit tests using critical vulnerabilities from 4 different categories. On average, InstaGuard increases program memory footprint by 1.69% and slows down program execution by 2.70%, which are unnoticeable to device users in practice.**

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23124
www.ndss-symposium.org

## I. Introduction

A daunting challenge for securing the Android ecosystem is device vendors' inability of instantly releasing system security updates to users' devices. For example, the notorious *Quad-Rooter* vulnerabilities [12], affecting more than 900 million Android devices equipped with Qualcomm chipsets, were initially reported to Google in April 2016. However, it was after 5-7 months that large device vendors such as Samsung and HTC started pushing out system updates that finally patched the vulnerabilities. Even today, more than a year since the disclosure of the vulnerabilities, a large number of devices from other vendors have not been patched or will never receive a patch.

To remedy the prolonged security update process that plagues Android OS, researchers proposed to apply vulnerability hot-patching techniques to system programs on Android that cannot be updated till the next OS upgrade [32], [28], [37], [22]. Hot-patching is a class of fast vulnerability mitigation techniques. It dynamically injects code into a vulnerable program as a temporary fix that either disables or replaces the vulnerable code, preventing the vulnerability from being exploited while a permanent patch is being developed and tested. However, despite the wide adoption in desktops and servers [6], [1], [13], hot-patch solutions are not well received on Android platforms.

Teaming up with a major mobile device vendor, we investigated the adoption resistance facing hot-patching mechanisms in the Android ecosystem. We found that the lack of adoption is due to the fact that existing hot-patching techniques, originally designed for conventional computers, are unaware of an important constraint posed by OEMs and mobile network carriers on system code updates: before deployment, all system code updates for mobile devices must go through lengthy compatibility tests performed by each party.

Despite efforts made by both vendors and carriers to reduce the delay in update release [10], the current average delay still lasts 3-4 months. Being system code updates, hot-patches have to undergo this lengthy testing process and cannot be deployed on consumer devices without a significant delay. Therefore, device and OS vendors see little incentive to adopt existing hot-patching systems, knowing that hot-patches for mobile devices would never be instantly deployable as expected.

To bring the benefits of hot-patches to Android devices (i.e., timely protection since vulnerability disclosure) without

breaking carriers' requirements, we propose InstaGuard, a new approach to hot-patching that allows for instant deployment of patches (i.e., "carrier-passthrough") and fast patch development. Although hot-patching has been studied before, no existing method was designed under the requirement that InstaGuard needs to meet, namely enabling hot-patches that are safe to pass-through carriers and at the same time block exploits of various kinds of vulnerabilities.

Unlike existing hot-patching techniques, InstaGuard avoids injecting new code to vulnerable programs or their memory space. Instead, it enforces instantly updatable rules that contain no code to block exploits of unpatched vulnerabilities in a timely fashion. Our design choice of rule-driven hot-patches was inspired by our observation that *carriers allow non-code updates (e.g., new rules) to pass through without regression tests*, as evidenced by SEAndroid policy updates, which vendors can quickly push to consumer devices over-the-air without much involvement of carriers [7].

To design an effective rule-driven hot-patching system, we need to overcome two unique challenges that existing code-injection-based hot-patching techniques did not face. First, an expressive language is needed for composing the rules, which should be capable of instructing the runtime enforcement engine to efficiently block a variety of exploits. At the same time, this language should not be as unrestricted as code for security reasons. Second, the generation of such rules needs to be automated to an extent where human users with high-level knowledge about a newly reported vulnerability can easily produce a hot-patch rule for it.

We propose a simple language for writing hot-patch rules, called GuardRule. This language is based on the basic debugging primitives supported by ARM CPUs, namely `BreakPoint`, `WatchPoint`, and `Assertion`. A GuardRule contains a sequence of statements, each statement representing a use of a debugging primitive confined by necessary conditions. The idea of composing GuardRule using the debugging primitives allows for: $(i)$ constructing hot-patching logic for a wide range of vulnerability (i.e., highly expressive); $(ii)$ instantly releasing new GuardRule to consumer devices (i.e., carrier-passthrough); $(iii)$ efficient enforcement of GuardRule thanks to the hardware-backed debugging support. Additionally, we design GuardRule to be *restrictive*: it restricts, rather than adding new behaviors to, the execution of a vulnerable program, ensuring that a flawed rule cannot cause new security problems or undo the protection provided by other rules. This property makes GuardRule updates even safer in the eyes of stake holders than, for instance, SEAndroid policy updates, which has passed carriers' bar for granting pass-through.

Considering that writing low-level rules can be difficult and error-prone, we built a tool, called RuleMaker, that helps device vendors and security analysts quickly and easily produce GuardRule. RuleMaker takes as input *GuardSpec*, written in a high-level language that we designed for describing vulnerabilities, and outputs GuardRule. Writing GuardSpec does not require any knowledge of InstaGuard's design or its mechanics for blocking exploitations. Our design of GuardSpec was informed by our extensive study of real-world vulnerability reports, where we developed a categorization for Android system vulnerabilities, consisting of 7 buckets. For each vulnerability bucket, GuardSpec supports a simple syntax to describe vulnerabilities in that bucket.

In the event of a newly disclosed vulnerability in Android system, device vendors can quickly write a GuardSpec that describes the vulnerability and then use RuleMaker to synthesize a GuardRule, which can be instantly released over-the-air and deployed on consumer devices. Vendors can then start developing the permanent patch and work with carriers to test it. Even if this process can take months, vulnerable devices are protected by InstaGuard before the permanent patch is installed. On the device side, upon receiving a verified GuardRule, InstaGuard sets up the breakpoints, watchpoints, and assertions accordingly and starts enforcing the hot-patching logic. InstaGuard mitigates various types of vulnerabilities, including memory corruptions, race condition, integer overflows, and logic bugs. When a GuardRule is triggered (i.e., an exploit attempt is caught), InstaGuard either terminates the affected process or logs the event without interrupting the execution, depending on the action defined in the GuardRule. We note that advanced execution recovery or fallback techniques can be used to expand the actions that InstaGuard may take upon a caught exploitation. However, these techniques warrant separate research and are out of the scope for this paper.

Similar to previous hot-patching techniques, InstaGuard is not meant as a replacement for permanent security updates. It aims to provide timely yet temporary mitigation of unpatched vulnerabilities. Filling a void in today's Android security response solutions, InstaGuard can help significantly narrow the vulnerability exploitation window that used to be several months, enabling instant protection of freshly discovered vulnerabilities on Android devices. InstaGuard has the following unique advantages, which distinguishes it from existing hot-patching techniques.

- *Rule-driven*. Hot-patches are released and deployed in the form of enforceable rules, rather than executable code (i.e., no code is carried in the hot-patches; no code is injected to vulnerable programs or their memory space). Moreover, the rules are *restrictive* by design, which ensures that InstaGuard can not be abused as traditional hot-patching mechanisms may. Our rule-driven design allows the patches (or rules) to pass through carriers without delayed, thus achieving instant patch deployment.
- *Vulnerability-generic*. Its rule language is expressive enough to describe 7 different types of critical vulnerabilities.
- *Easy-to-use*. Its users only need to write high-level vulnerability descriptions, which can be easily extracted from vulnerability reports, as opposed to exploit detection logics; such descriptions are then automatically compiled into low-level rules.
- *Efficient*. It uses debugging features for rule enforcement, which incurs very low runtime overhead. Although the general idea of using debugging features was applied to exploit mitigation before [26], InstaGuard advances the idea by hiding the low-level debugging primitives away from patch developers and enabling easy development of hot-patching rules (as opposed to complex code patches that directly deal with debugging primitives).

We evaluated InstaGuard against 30 critical CVEs from the

Android Security Bulletins reported in 2016 [9]. InstaGuard is able to fully mitigate all of them. We also invited security analysts from the collaborating vendor to evaluate InstaGuard and RuleMaker using different kinds of real vulnerabilities. The evaluation results show that on average, each GuardSpec contains only 7-8 lines and takes only a few minutes to write. When enforcing these rules, InstaGuard incurs a mild 1.69% overhead on the memory footprint and 2.70% slowdown during unit tests.

In summary, we make the following contributions:

- We design and implement InstaGuard, the first rule-driven hot-patching system for Android that enables carrier-passthrough patches.
- We design the GuardSpec language that allows for generic description of a wide range of vulnerabilities.
- We build RuleMaker, which automatically compiles high-level, easy-to-write vulnerability descriptions (GuardSpec) into low-level rules (GuardRule) consumed by InstaGuard.
- We examined the coverage of InstaGuard against 30 vulnerabilities released in 2016 and performed unit tests using critical vulnerabilities of 4 different types.

The rest of the paper is organized as follows: In § II we motivate our work by showing the demand for a hot-patching system for Android capable of instant deployment of patches. In § III we present the design details of InstaGuard, GuardRule and RuleMaker. We then discuss the technical challenges we addressed during the implementation of these components in § IV. We report our analysis and empirical evaluation results in § V and in § VI we contrast InstaGuard with the related works. We discuss the potential improvement in § VII and finally, the whole paper is concluded in § VIII.

## II. Background

### A. Delayed Android System Patches

Android is often considered less secure than iOS, not because Android is more vulnerable, but due to the much longer system update cycle of Android devices. In fact, vulnerabilities discovered in iOS are not fewer than those discovered in Android [37]. Unlike iOS devices, which is solely developed by Apple, the Android ecosystem and device market are highly fragmented [8]. Developing and deploying system software updates for Android requires cooperations among multiple parties, such as OS vendors, device vendors, carriers, etc. The fragmentation and the complex chain of stakeholders cause months, or even years, of delays in security patch and update deployment, which often leave a large number of vulnerable devices open to attacks, despite that many vulnerabilities have already been fixed by upstream vendors.

We conducted an empirical survey on the life cycles of vulnerabilities in Android system programs installed on a mainstream product line of Samsung. As shown in Table I, the average time needed for completely resolving these high-severity security vulnerabilities (i.e., from the initial disclosure to the deployment of permanent patch) is 7.6 months, which are dangerously long. We found that third-party libraries, such as OpenSSL, tend to have even longer patch delays than first-party programs. More alarmingly, there are two critical vulnerabilities shown in the table (CVE-2015-1539 and CVE-2015-3836) that still remain unpatched to date. In fact, it is not uncommon to see vendors or carriers decide not to develop or deploy a vulnerability patch because of high cost or out-of-support devices.

***TABLE I:*** *Severe vulnerability life cycles and patch delays. OEMs normally receive upstream patches from google 1 month earlier before the bulletin goes public.*

| | CVE# | Bug reported date | 3rd-party libs website release patch | Google releases patch to OEM | OEM releases patch | Carrier releases OTA update | Delayed months |
|---|---|---|---|---|---|---|---|
| openssl | 2016-2108 | 4/18/2015 | 6/11/2015 | 6/2016 | 7/2016 | Roughly one month. varying across carriers. | 16 |
| libstagefright | 2015-3824 | 5/4/2015 | N/A | 7/9/2015 | 10/2015 | | 6 |
| openssl | 2016-2107 | 4/18/2015 | 6/11/2015 | 6/2016 | 7/2016 | | 16 |
| mediaserver | 2016-2428 | 1/22/2016 | N/A | 5/2016 | 6/2016 | | 6 |
| libstagefright | 2015-3824 | 5/4/2015 | N/A | 8/2015 | 10/2015 | | 6 |
| libstagefright | 2015-1539 | 4/08/2015 | N/A | 8/2015 | no patch planned | | ∞ |
| sonivox | 2015-3836 | 5/14/2015 | N/A | 8/2015 | no patch planned | | ∞ |
| mediaserver | 2016-0835 | 12/6/2015 | N/A | 3/2016 | 4/2016 | | 6 |
| mediaserver | 2016-2416 | 2/5/2016 | N/A | 3/2016 | 4/2016 | | 3 |
| libstagefright | 2015-3823 | 5/20/2015 | N/A | 9/2015 | 10/2015 | | 6 |
| aac | 2016-2428 | 1/22/2016 | N/A | 5/2016 | 6/2016 | | 6 |
| libmedia | 2016-2419 | 12/24/2015 | N/A | 3/2016 | 4/2016 | | 5 |
| Avg. delay | | | 1.6 | 3.4 | 1.3 | 1 | 7.6 |

The key observation from our survey is that security patches for Android system programs and libraries are often delayed by months, mostly due to the extended time needed for developing, testing, and deploying permanent patches. To timely protect vulnerable Android system programs while their permanent patches are in the making, a hot-patching technique is needed that allows device vendors to easily and quickly generate hot-patches and instantly deploy them through carriers to user devices.

### B. Hot-patching and Its Adoption Obstacle on Android

Despite the obvious need for hot-patching on Android and the availability of existing techniques [1], [13], [6], mobile device vendors rarely introduce hot-patching to Android. We collaborated with a major vendor to investigate this problem. We concluded that the existing hot-patching techniques, originally designed for computer programs, do not consider a unique constraint posed by mobile carriers and other stakeholders, and consequently, cannot be deployed instantly.

As explained in [5], any code patch for system binaries on Android devices must be approved by all parties in the operation chain, including carriers, before the patch can be released and deployed. This requirement allows newly added system code to be fully tested by all parties for potential compatibility and security issues, despite that these tests can take weeks or months. However, existing hot-patching techniques are at odds with this requirement. To disable or replace vulnerable code,

these techniques need to inject new code to either program binaries or program memory spaces. Therefore, if applied to Android system programs, these hot-patches (i.e., new code to be injected to vulnerable programs) must go through the lengthy tests, and therefore, cannot be instantly deployed, which makes it pointless for vendors to adopt existing hot-patching techniques.

### C. Carrier-passthrough Updates

During our investigation, we came across a type of updates that carriers and stakeholders waive their tests and allow to pass through without interruption. We refer to these updates as *carrier-passthrough updates*. We found carriers created this exception to let time-sensitive, *non-code* updates be pushed to user devices over-the-air without delays, under the good faith that non-code updates are very unlikely to cause compatibility or safety issues. One example is the SEAndroid policy updates.

This finding inspired us to design a new hot-patching technique that can leverage the carrier-passthrough update channel and enable instantly deployable hot-patches for Android devices. Next, we identify the requirements for designing such a technique.

### D. Design Requirements for Android Hot-patching System

Informed by our empirical study, we set out to design an effective and practical hot-patching system for Android. This system should meet the following requirements:

**R1 - Non-code patches** Hot-patches should not carry code in any form. Further, the patching should not involve injecting new code to vulnerable program binaries or program memory. This requirement ensures that hot-patches do not contain code or introduce new code to to-be-patched programs, and therefore, do not warrant carrier-imposed tests meant for code patches (SEAndroid policy updates already set a precedent for the test waiver).

**R2 - Restrictive patching** We call a hot-patching system "restrictive" when its patches can only *restrict* program execution (e.g., forbidding certain execution paths), rather than amplifying it (e.g., add new execution paths or permissions), which may lead to abuse of the hot-patching system itself as previously reported [15], [17]. When hot-patches meet this requirement, carriers and other stackholders can more confidently grant pass-through on hot-patches (SEAndroid do not meet this requirement because it supports permissive rules).

**R3 - Wide vulnerability coverage** The hot-patching system should be able to block various kinds of vulnerabilities, including the memory corruption bugs, race condition, logic bugs, etc.

**R4 - Ease of use** The system should allow human developers and analysts to easily and quickly compose hot-patches without requiring more knowledge than the high-level understanding of the vulnerability being patched.

A hot-patching system for Android needs to meet all above requirements to be effective, practical, and safe, which means it supports instantly deployable patches, mitigates various kinds of vulnerabilities, and allows fast development of patches. Next, we discuss our design of InstaGuard and show how it meets all the requirements.

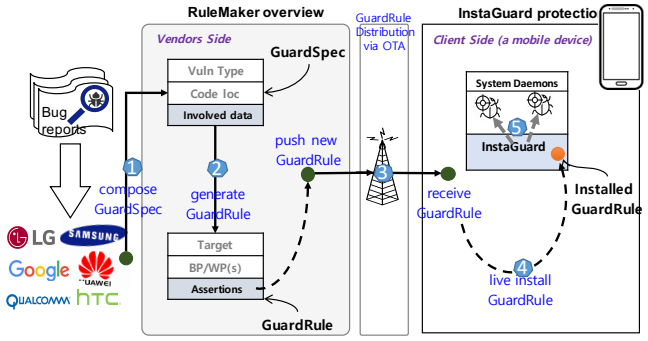


***Fig. 1:*** *Vendors generate GuardRule for their vulnerable system components (left); InstaGuard mitigates reported vulnerability based on GuardRule (right).*

## III. System Design

### A. Overview

Our goal of designing InstaGuard is to provide the first instantly deployable defense against unpatchable or to-be-patched system vulnerabilities in Android. InstaGuard blocks exploitations of vulnerabilities by enforcing GuardRule (a new type of hot-patch), which vendors generate and distribute as soon as they discover new vulnerabilities whose full patches cannot be developed or deployed immediately. GuardRule is designed to meet $R1$–$R2$ (as shown in § III-F), and therefore, can pass-through carriers and be deployed to user devices over-the-air (OTA). To ease and, to a large extent, automate the rule generation process, we designed GuardSpec and RuleMaker. GuardSpec is a high-level, short vulnerability description written in a simple language we designed. RuleMaker is a tool that synthesizes GuardRule from GuardSpec.

Figure 1 shows the system overview. Upon receiving bug reports or vulnerability disclosures, vendors compose GuardSpec (Step ① ) and then use RuleMaker to synthesize GuardRule (Step ② ), which is then instantly distributed to user devices via OTA (Step ③ ). After receiving GuardRule, InstaGuard on a user device installs the rule after necessary integrity checks (Step ④ ) and starts enforcing the rule (Step ⑤ ), closing the exploitation window within hours, if not sooner, since the initial vulnerability report.

InstaGuard enforcement mechanism builds upon three basic debugging primitives: *breakpoint*, *watchpoint* and *assertion*. Breakpoint (BP) allows InstaGuard to intercept a program execution reaching a code location where a vulnerability can be triggered immediately. Watchpoint (WP), on the other hand, allows InstaGuard to efficiently capture memory data accesses potentially causing vulnerability exploitations. Assertion (AS), when used in combination with the first two primitives, allows InstaGuard to perform necessary checks to determine if vulnerability triggering conditions are met. A GuardRule uses these primitives, in a certain sequence with concrete parameters (e.g., BP/WP addresses and AS expressions), to detect and block the exploitations of a vulnerability. We will show in § III-D that these basic primitives and the expressive syntax of GuardRule allow InstaGuard to cover a wide range of critical vulnerability types, including logic bugs, integer overflow, out-of-bound access, format string abuse, race condition, and user-after-free bugs.

```
1  ssize_t utf16_to_utf8_length(const char16_t *src,
     ↪  size_t src_len)
2  {
3    ...//sanity checks
4    size_t ret = 0;
5    const char16_t* const end = src+src_len;
6    while (src < end) {
7      if ((*src & 0xFC00) == 0xD800
8          && (src + 1) < end
9          && (*++src & 0xFC00) == 0xDC00) {
10       // surrogate pairs are always 4 bytes.
11         ret += 4;
12         src++;
13     } else {
14         ret += utf32_codepoint_utf8_length
             ↪  ((char32_t) *src++);
15     }
16   }
17     return ret;
18 }
```

***Fig. 2:*** `system/core/libutils/Unicode.cpp` – *CVE-2016-0836, the logic bug on Line 9 can cause buffer overflow.*

```
1  <rules>
2    <rule cve="CVE-2016-3861">
3      <module_name>libutils.so</module_name>
4      <decision>BLOCK</decision>
5      <binary_path>/system/bin/mediaserver</binary_path>
6      <break_points>
7        <break_point first=true, id=0>
8          <!--binary address corresponding to line 9
               in Listing 1-->
9          <address>0x08055000</address>
10         <!--next action: activating assertion
               primitive with id 0-->
11         <action> VERIFY AS#0</action>
12       </break_point>
13     </break_points>
14     <assertions>
15       <!--if assertion evaluate to true InstaGuard
             BLOCK the execution as node ?decision?
             speicify-->
16       <assertion id=0, action=decision>
17         <data_constraints>
18           <data_constraint>
19             <ops>NE</ops>
20             <left_exp>
21               <!--retrieval rule for *src-->
22               <node id=0>reg_2_32</node>
23               <node id=1>const_0xFC00</node>
24               <node id=2>bitwise_and</node>
25             </left_exp>
26             <right_exp>
27               <node>const_0xDC00</node>
28             </right_exp>
29           </data_constraint>
30         </data_constraints>
31       </assertion>
32     </assertions>
33   </rule>
34 </rules>
```

***Fig. 3:*** *Simplified GuardRule example. It stops program execution at the converted binary address of line 9 in the source code, and use assertion rules to check if *src & 0xFC00 == 0xDC00 as specified in data_constraint nodes.*

To better demonstrate how our system works, we now describe in detail the steps it takes to mitigate a real-world vulnerability (`CVE-2016-3861`). This is a logic bug in *libutils.so*, which is used by several critical system daemons on Android, including *mediaserver*. As shown in Figure 2, when the expression on Line 9 evaluates to false, the string pointer `src` is mistakenly advanced twice (on Line 9 and Line 14), causing the expected length of the string to be shorter than the actual string length, and later, leading to a buffer overrun.

InstaGuard can stop any exploitation of this logic bug using the GuardRule shown in Figure 3. The rule instructs InstaGuard to set a BP at the binary location corresponding to Line 9 in Figure 2 and, when the BP is reached, use an AS to check the vulnerability triggering condition, namely `*src & 0xFC00 == 0xDC00`. The use of the BP and the AS is defined on Line 7-12 and Line 16-31 in Figure 3, where the BP is chained to the AS as a predecessor via the "action" defined on Line 11. If the AS is true (i.e., the bug is about to be triggered), the action for the AS is performed, which in this case simply blocks the program execution, as specified on Line 16 and 4 in Figure 3. While we defer the complete discussion of GuardRule syntax and other covered vulnerability types to § III-D, we note that the expressiveness of GuardRule and the wide vulnerability coverage of InstaGuard are enabled by the carefully designed GuardRule syntax, which allows, among other things, primitive chaining via actions and primitive placement at sub-function level.

```
1  [common]
2  ID = CVE-2016-3861
3  binary_path = /system/bin/mediaserver
4  module_name =libutils.so
5  decision = BLOCK
6
7  [logic bug]
8  vul_location = system/core/libutils/Unicode.cpp |
     ↪  utf16_to_utf8_length | 411
9  lexp = *src & 0xFC00
10 rexp = 0xDC00
11 relation_op = NE
```

***Fig. 4:*** *Image-independent and human-friendly GuardSpec file composed by security analytics. Line 8 reflects the real source code location.*

```
1  [common]
2  ID = CVE-2016-3871
3  binary_path = /system/bin/mediaserver
4  module_name =libstagefright_soft_avcenc.so
5  decision = BLOCK
6
7  [buffer overflow]
8  buf_name = outHeader->pBuffer
9  buf_size = outHeader->nAllocLen
10 vul_location =
     ↪  libstagefright/codecs/mp3dec/SoftMP3.cpp
     ↪  |SoftMP3::internalGetParameter | 303
```

***Fig. 5:*** *The vulnerability description style GuardSpec for CVE-2016-3871 (buffer overflow)*

We recognize that manually composing a GuardRule can be slow and difficult due to the required knowledge about InstaGuard internals and binary-level characteristics of vulnerabilities. Therefore, we do not expect human to write GuardRule directly. Instead, we introduce a simple language called GuardSpec for describing vulnerabilities in a programmer-friendly way. We build a tool called RuleMaker to automatically synthesize GuardRule using GuardSpec as input. To write a GuardSpec, one needs no more than some high-level knowledge about a given vulnerability, which can be easily extracted from typical bug or vulnerability reports. Figure 4 shows the GuardSpec that corresponds to the GuardRule in Figure 3—the former is much simpler and readable than the latter. As another example, Figure 5 shows the GuardSpec for mitigating `CVE-2016-3871`, which is a buffer overflow vulnerability. In this GuardSpec, the human analyst only need to describe the overrun buffer and the culprit code location, RuleMaker will take over the work and synthesize a fully functional GuardRule, which may be directly deployed to end user devices. RuleMaker hides the details about InstaGuard primitives and mechanisms away from users of the system. It allows security analysts and vendors to automatically generate GuardRule by providing GuardSpec, which they can easily

write while investigating reported vulnerabilities. The GuardSpec format is discussed in § III-D.

In the rest of the section, we first discuss the threat model and then the design details of InstaGuard, GuardRule and RuleMaker. We finally examine our proposed design against the requirements (R1-R4).

### B. Usage Scenario and Threat Model

InstaGuard's intended users include Android OS and device vendors. InstaGuard aims to significantly reduce the Android vulnerability exposure window from months down to days, if not sooner. It allows vendors to quickly develop and instantly deploy GuardRule for stopping exploitations of critical vulnerabilities in Android framework programs, whose full patches usually are not available until the OS or firmware is updated months after vulnerability discovery.

Under this usage scenario, we adopt a realistic threat model that is common among vulnerability hot-patch solutions. We trust the OS kernel and rely on it as the TCB for InstaGuard. However, we assume that system daemons and libraries may contain critical vulnerabilities (e.g., stagefright). We expect skilled attackers to attempt exploitations of various kinds. The goal of InstaGuard is to prevent exploitations of vulnerable system components in Android. We note that, as a rule-driven exploit mitigation system, InstaGuard cannot react to exploitations of unknown vulnerabilities for which no rule is defined. InstaGuard serves as the first line of defense against exploitations. It is ineffective on a program that has already been compromised or become malicious.

### C. InstaGuard: Rule-driven Vulnerability Mitigation

*1) InstaGuard Components and Workflow:* The InstaGuard system consists of both the user-space and kernel-space components. The user-space components receive GuardRule updates and collaborate with the kernel-space component to dynamically load and enforce updated rules without rebooting the device. Figure 6 shows the the major components of InstaGuard and their workflow.

- **InstaGuard Daemon** (iDaemon) is a user-space daemon process which instructs iMonitor (discussed shortly) to initiate rule installation or removal upon rule update and process restart.
- **InstaGuard Runtime Monitor** (iMonitor) is a library loaded in each monitored process (i.e., a process being protected by InstaGuard). It is in charge of (*i*) Parsing GuardRule; (*ii*) Requesting iDriver to register BP and WP as needed by GuardRule. (*iii*) Verifying assertion primitives.
- **InstaGuard Kernel Driver** (iDriver) is the kernel-space component responsible for (*i*) handling BP and WP registration requests from iMonitor; (*ii*) coordinating monitored processes to timeshare the hardware debug registers; (*iii*) notifying iMonitor upon hardware BP/WP exceptions; (*iv*) silently dismiss any exceptions/violations caused by iMonitor due to a buggy GuardRule, if any.

We split the system into three components for better security and efficiency. The kernel-level iDriver is kept minimal (thus a small TCB) and used for performing privileged

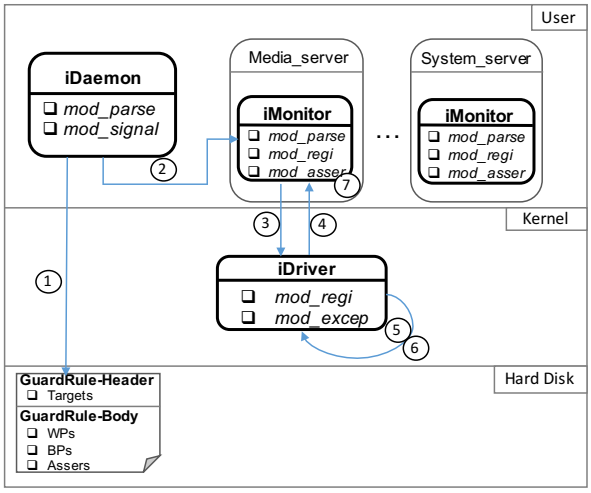


***Fig. 6:*** *InstaGuard components and workflow. While components are marked with rounded rectangle and the arrows indicate the information flow direction.*

operations (e.g., configuring the debug hardware). The user-space iMonitor performs the program- and context-specific operations, such as assertion evaluation and retrieval of memory content. Since iMonitor operates directly in the same context of the monitored program, there is no semantic gap problem and is also more efficient when trying to evaluate the process state. iDaemon is event-driven and mostly remains dormant due to the infrequent update of GuardRule.

The workflow of InstaGuard mechanism is shown in Figure 6. Upon a GuardRule update, iDaemon wakes up to parse the rule headers (Step ①). According to the headers, it then signals the iMonitor in affected processes about the new rule (Step ②). After being notified, the iMonitor parses the rule body. For efficiency, we do not sandbox the parser, since the rule source is trusted and can be verified. As per the rule, iMonitor then requests iDriver to register needed BP and WP (Step ③). After verifying the identity of the requesting iMonitor, iDriver initializes the BP and WP bookkeeping data in the PCB (Process Control Block) of the requesting process. Later, when a monitored process with registered BP/WP gets the CPU and is about to start/resume its execution, iDriver populates the saved BP/WP setup information from the PCB to the available hardware debug registers (Step ⑤ and ⑥). Note that when serving the BP/WP registration requests, iDriver loops through all the PCBs with the same thread group ID and registers the rule for each one of them. This ensures that all threads in a vulnerable program are properly protected.

When a BP or WP debug exception is triggered, iDriver raises a signal to the corresponding process (Step ④). iMonitor later receives the pending signal when the execution returns from kernel to user mode. Since kernel will always check if a process has pending signals before return the control back to user mode. The raised signal will guarantee iMonitor to take over the execution immediately. iMonitor reacts to the signal by following the actions defined in the corresponding GuardRule (Step ⑦), such as (de)registering BP/WP or performing assertion evaluation.

*2) Security and Safety of InstaGuard:* We establish a chain of trust among the InstaGuard components, rooting in the trusted OS. This chain of trust minimizes the attack surface of InstaGuard and prevents attacks against InstaGuard itself. Specifically, iDaemon verifies the source and integrity of every new GuardRule by signature checking. Every received GuardRule is kept read-only on user devices and is only updatable by iDaemon, enforced using SEAndroid policies. Before responding to a signal, iMonitor checks its source PID to ensure it indeed comes from iDaemon. Similarly, iDriver verifies that incoming requests are always from iMonitor.

Our design prevents InstaGuard from malfunctioning or being abused when a buggy or broken GuardRule is deployed (i.e., flawed rules do not pose security threats to the system). This benefit is enabled by the restrictive nature of GuardRule and the separation between rules and mechanisms. Specifically, GuardRule never introduces new code or data into user devices and InstaGuard only impose more, and cannot uplift any, restrictions on program executions. Therefore, a faulty rule cannot create new execution paths in affected programs. In contrast, previous hot-patch solutions all face a security concern that these techniques themselves, when failed, can easily allow disruptive or even malicious code being introduced to protected programs.

### D. GuardRule: Generically Blocking Exploits

GuardRule is a low-level, XML-based rule that, when pushed to the client-side, instructs the InstaGuard mechanism to block exploitations of a vulnerability. The major components and fields of a GuardRule are shown in Table II.

At a high level, each rule consists of a header and a body. The header contains the rule ID, the vulnerable module identifier, and the action upon violation (or alert decision). Our current prototype supports two kinds of alert decision, namely "BLOCK" and "AUDIT", which will terminate the faulting program or simply record the alert event, respectively. These two alert decisions are sufficient for our purpose of demonstrating and testing the InstaGuard prototype because most of the monitored programs (e.g., the Android framework daemons and services) are capable of self-recovering from a crash or abrupt termination. Nevertheless, it is possible to extend the decisions to support more actions such as rollback to benign state, we discuss this more in § VII.

In the rule body, three lists of breakpoint (BP), watchpoint (WP) and assertion (AS) could be specified. A BP is defined by the following fields: <`first`, `address`, `action`>. The `first` field indicates whether the BP should be registered immediately when installing the rule. The `address` field denotes where the BP should be placed. The `action` field references to the next primitive (e.g., an AS or another BP) to activate when this BP is triggered. Each rule should contain at least one BP as the primitive chain initiator, expressing at which point during the program execution should InstaGuard pause the program and take an `action`.

A WP is used in a rule when, for example, an in-memory variable needs to be tracked. A WP contains the three fields that a BP has plus a `size` field, which specifies the memory range to be watched.

An AS contains a `relation` field that expresses the relationship among defined data constraints. Currently, we support "AND" and "OR" relation operators for combining multiple data constraints. A `data constraint` contains a left-hand-side expression, a right-hand-side expression, and an operator connecting the expressions. An example of an AS is shown on Line 16–31 in Figure 3.

The basic primitives and the GuardRule syntax allow InstaGuard to intervene in program execution at instruction level, track in-memory data access, and evaluate assertions and runtime conditions. Via the `action` field for each primitive, the GuardRule syntax further allows the primitives to be chained and triggered in order under defined conditions. As a result, GuardRule is expressive and flexible enough to define mitigations for a wide range of vulnerabilities, including logically complicated use-after-free bugs, which is later demonstrated in § V.

***TABLE II:*** *Major GuardRule components and corresponding fields*

| Component | Fields |
|---|---|
| GuardRule-Header | Rule ID: <Rule id, Vulnerability type><br>Target Module: <Binary path, Module name><br>Alert Decision: <BLOCK \| AUDIT> |
| GuardRule-Body | Breakpoints: [<first, address, action>, ...]<br>Watchpoints: [<first, address, size, action>, ...]<br>Assertions: <relation, [<constraints>, ...], action> |

### E. RuleMaker: Rule Generation Assistance

*1) Vulnerability Categorization:* Informed by our study of the real-world bugs reports, we organized common Android vulnerabilities into 7 buckets, shown in the left column of Table III. Note that, in practice, vulnerabilities are typically caused through a chain of bugs. For instance, most of the reported buffer overflow vulnerabilities related to *libstagefright* are caused by integer overflows. A large amount of reported race condition and use-after-free (UAF) vulnerabilities are actually caused by combinations of logic bugs. For example, CVE-2016-8655 is labeled as race condition and CVE-2016-6707 is reported as a UAF bug, while they all find their roots in various logical errors.

*2) GuardSpec and RuleMaker:* We design the GuardSpec format for InstaGuard users to describe to-be-patched vulnerabilities at ease. We build RuleMaker to automatically synthesize GuardRule from GuardSpec. A GuardSpec is a high-level description of a vulnerability using a simple syntax. An example GuardSpec is shown in Figure 4.

A concise guideline for composing GuardSpec for common types of vulnerabilities is given in Table III. For each type of vulnerability, we list the required fields in a GuardSpec. The fields are intuitive. For instance, the *vul_location* field specifies the source-level location of the bug. It is a 3-tuple containing the source code line number, the function name, and the source file. The information needed for filling these fields can be collected from a regular bug report (i.e., security analysts can easily collect such information and compose GuardSpec while investigating bug reports). The Appendix has more GuardSpec examples defined for each vulnerability category in Table III. It is worth noting that, for UAF vulnerabilities, we provide two options to block them (i.e., two ways to write GuardSpec for a UAF): one blocks the faulty free operation, the other blocks

the faulty use operation. The difference is that, blocking faulty free is generally more lightweight than blocking faulty use; however, it is not always easy or possible to discern a faulty free from a legitimate one due to memory aliasing, in which case blocking faulty use is the only option.

RuleMaker takes GuardSpec as input and generates GuardRule, hiding from users InstaGuard's low-level primitives and undertaking the tedious and error-prone tasks, such as symbol address resolution. During the synthesis process, fields in the GuardSpec-header are directly converted to the top-level XML components of the resulting GuardRule. However, the selection and activation order of primitives are not as straightforward. We developed the synthesis templates, one for each vulnerability type, based on our empirical experiences. As shown in Table IV, the templates guide RuleMaker, for each vulnerability type, to generate the optimal sequence of primitives needed for mitigating the vulnerability and concretize the parameters using data extracted or inferred from the fields in input GuardSpec.

As for symbols, such as variable names, RuleMaker translates them into a series of retrieval routines. InstaGuard supports register and memory based retrieval routines, expressed as `node` in the XML-based GuardRule. They are in the format of <`reg_$regid_$length`> and <`mem_$baseid_$offid_$length`>, respectively. A register retrieval routine is intuitive. It specifies which register to read and what is the size. All general-purpose registers can be retrieved. For a memory retrieval routine, the fields `baseid` and `offid` are node ID(s) that reference to nodes of type register, constant, or memory (for multilevel memory accesses). This scheme allows InstaGuard to cover both direct and indirect memory accesses, which enhances the expressiveness of GuardRule. For instance, when an interested variable is stored in a temporary register (e.g, Figure 3), InstaGuard can efficiently retrieve that value and support GuardRule referencing that temporary variable.

For generating these resolving rules, RuleMaker relies on full debug information of the reported vulnerable binary, the debug information can be in-house prepared since vendors are in control of the source code. More resolving challenges are further described in § IV.

### F. Satisfied Requirements

We now examine if our design of InstaGuard satisfies the practical requirements (R1-R4) listed in § II-D. InstaGuard naturally meet R1 (**non-code patches**), we show that InstaGuard is solely driven by GuardRule, and it does not introduce new code to the to-be-patched system. This is needed to bypass the lengthy regression tests required by carriers. The test is mandatory for all system code updates. In addition, more GuardRules always impose more restrictions and checks on the execution of the target program, had anything goes wrong, attackers can not do arbitrary things by introducing (malicious) rules. This, combined with the non-code update, can guarantee that InstaGuard as a hot-patch system will not be abused at all times. Moreover, InstaGuard is robust against buggy rules, it can recover from any access violations caused internally and prevent the program execution from being disrupted. Thus, R2 (**restrictive patching**) is satisfied.

***TABLE III:*** *GuardSpec defined vulnerability types and corresponding required fields, which can be extracted from bug reports.*

| Vulnerability type | Required fields |
|---|---|
| Logic bug | vul_location: <func_name, func_line, file_name><br>vul_content: <relation_op, lexp, rexp> |
| Integer overflow | vul_location: <func_name, func_line, file_name><br>vul_content: <overflow_direction, exp, value> |
| Out-of-bound access | vul_location: <func_name, func_line, file_name><br>vul_content: <index_var, buf_size_var> |
| Buffer overflow | vul_location: <func_name, func_line, file_name><br>vul_content: <buf_name, buf_size> |
| Format string | vul_location: <func_name, func_line, file_name><br>vul_content: <str_var> |
| Race condition | vul_location: <racer_1, racer_2><br>vul_content: <var1_in_racer1, var2_in_racer2><br>racer_1 <func_name, func_line, file_name><br>racer_2 <func_name, func_line, file_name> |
| Use-after-free (UAF) | (Block faulty use):<br>vul_location: <free_loc, use_loc><br>vul_content: <free_buf, buf_size><br>free_loc <func_name, func_line, file_name><br>use_loc <func_name, func_line, file_name><br>(Block faulty free):<br>vul_location: <func_name, func_line, file_name><br>vul_content: <relation_op, lexp, rexp> |

Combining with the primitive coordination, InstaGuard is capable to fix the commonly seen vulnerabilities such as memory corruptions (including both the spatial and temporal ones), integer overflow as well as the more generic logic bugs. Hence R3 (**comprehensive coverage**) is met.

To use InstaGuard, one only needs to prepare RuleMaker with the high-level information based on the vulnerability type, the fixing GuardRule will be generated automatically, freeing the users from dealing with the low-level machine-facing primitives, which, suggests R4 (**ease of use**).

## IV. System Implementation

We have built a prototype consists of InstaGuard based on LG Nexus 5X, which is equipped with Qualcomm Snapdragon 808 and 2GB RAM. The CPU features the ARM big.LITTLE architecture with two high-performance Cortex-A57 cores and four slower Cortex-A53 cores. The Android version we used is based on AOSP_6.0.1_r8 (Marshmallow) with Linux kernel v3.10 (64-bit). In the rest of the section, we discuss the implementation challenges and how we addressed them.

### A. InstaGuard Implementation Challenges

**InstaGuard primitive implementation:** The breakpoint and watchpoint primitives in InstaGuard are enabled by the ARM hardware debug unit [4]. This allows InstaGuard to enable the protection without touching the process memory. On the other hand, this choice puts a limit on the number of vulnerabilities InstaGuard can mitigate simultaneously in a thread/process. Nevertheless, we find the number of available hardware debug registers in an ARMv8 CPU is largely sufficient for our purpose: Figure 7 shows the life cycle of *all* the critical vulnerabilities in the Android system binaries from Android Security Bulletin 2016. At any point of time, the total number of vulnerabilities in a specific Android system program is less than the number of hardware debug registers, i.e., InstaGuard can cover the whole set of the vulnerabilities. Note that we save and restore the hardware debug registers during the context

***TABLE IV:*** *GuardSpec to GuardRule synthesis rules. (i)Numbers following the primitives indicate their IDs and are used to establish the connection; (ii)The fields following symbol "@" are translated to machine-oriented addressing rules using registers, memory accesses or simply constant value.*

| Vulnerability | Synthesis rules | Comments |
|---|---|---|
| Logic bugs | • BP(1) <address = @vul_location, first = true, action = AS(1)><br>• AS(1) <action = decision, constraints = [<@relation_op, @lexp, @rexp>, ...]> | self-explainatory |
| Integer overflow | • BP(1) <address = @vul_location, first = true, action = AS(1)><br>• AS(1) <action = decision, constraints = [<overflow_direction, @expression, @value>, ...]> | Can cover overflow and underflow. |
| Out-of-bound access | • BP(1) <address = @vul_location, first = true, action = AS(1)><br>• AS(1) <action = decision, constraints = [<LT, @index_var, @buf_size_var>, ...]> | LT as "less than". |
| Buffer overflow | • BP(1) <address = @(vul_location - 1), first = true, action = INSTALL WP(1)><br>• WP(1) <address = @buf_name+@buf_size, size = WORD, first = false, action = AS(1)><br>• AS(1) <action = decision, constraints = [<W/I, @reg_pc_64, @vul_location>, ...]><br>• BP(2) <address = @(vul_location + 1), first = false, action = REMOVE WP(1)> | Context-binding watchpoint (see § IV-B). |
| Format string | • BP(1) <address = @vul_location, first = true, action = AS(1)><br>• AS(1) <action = decision, constraints = [<IN, "%", @str_var>, ...]> | IN as set membership operator. |
| Race condition | • BP(1) <address = @(racer(a)_start_location), first = true, action = flag_set? AS(1): SET flag><br>• BP(2) <address = @(racer(a)_end_location), first = true, action = UNSET flag><br>• BP(3) <address = @(racer(b)_start_location), first = true, action = flag_set? AS(1): SET flag><br>• BP(4) <address = @(racer(b)_end_location), first = true, action = UNSET flag><br>• AS(1) <action = decision, constraints = [<EQ, @AddrOf(var1), @AddrOf(var2), ...]> | Simulate locking operations. |
| Use-after-free (Block faulty use) | • BP(1) <address = @free_location, first = true, action = INSTALL WP(1)><br>• WP(1) <address = @freed_buf, size = @buf_size, first = false, action = AS(1)><br>• AS(1) <action = decision, constraints = [<W/I, @reg_pc, @use_location>, ...]> | W/I as "within range". reg_pc as "value in pc register". |
| Use-after-free (Block faulty free) | • BP(1) <address = @vul_location, first = true, action = AS(1)><br>• AS(1) <action = decision, constraints = [<@relation_op, @lexp, @rexp>, ...]> | self-explainatory |

switch. Therefore, each thread can use all the hardware debug registers.

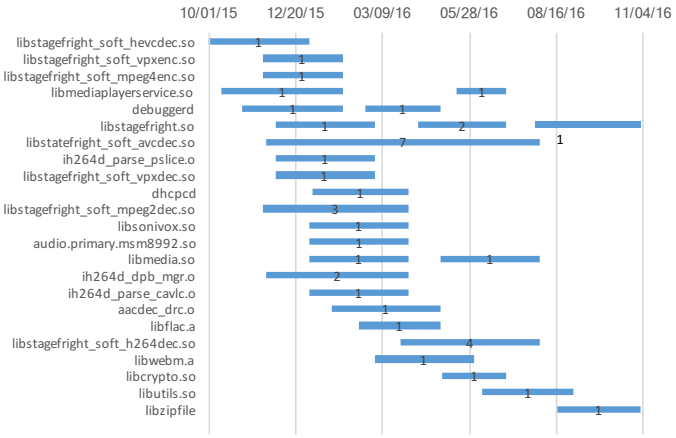


***Fig. 7:*** *Lifespan for all system binaries that had critical vulnerabilities reported in Android Security Bulletin 2016 [9].*

Hardware debug registers are often used by Linux's `perf` subsystem. `perf` is an architecture-independent performance profiling framework in Linux. This is achieved by using multiple layers of abstraction, leading to poorer performance. Therefore, our prototype directly operates on the hardware debug registers, bypassing the `perf` subsystem. Moreover, the hardware debugging module is, *by default*, not compiled into the Android kernel for performance reasons [1]. Therefore, debuggers relying on perf-based interfaces, such as ADB and GDB, are unaware of the hardware debug registers. They instead use software instruction breakpoints. This allows ADB and InstaGuard to co-exist.

**Hardware exception pitfall:** We use hardware debug registers to implement InstaGuard's breakpoint/watchpoint primitives. When a break/watch point is hit, the CPU triggers a hardware debug exception before the exception-triggering instruction is executed. The exception is reflected back to the monitored process (as a signal) for handling. Note that this scenario is different from how traditional debuggers (e.g, gdb) handle breakpoints: traditional debuggers are standalone processes whose own execution will not interfere with the debugged process. In our case, iMonitor is both the debugger and the debugged. This creates a dilemma: we cannot simply resume the execution of the monitored process because that will re-trigger the breakpoint; we also cannot simply disable the breakpoint and resume the process because this removes the breakpoint (i.e., the next invocation of the function cannot be protected). This problem can be easily addressed in a traditional debugger by enabling single-step execution when the breakpoint is hit. However, this solution is not feasible in InstaGuard, otherwise every instruction executed by the breakpoint handler (iMonitor) will be trapped to the kernel. To address that, we temporarily disable the firing breakpoint and allow iMonitor to process the breakpoint (as a signal). After handling the exception, iMonitor calls `sys_rt_sigreturn` to notify the kernel to resume the execution of the target process. The kernel then enables the single-step execution to allow execution of the faulting instruction. The CPU executes the faulting instruction and triggers a single-step exception. In the single-step exception handler, we reenact the previously disabled breakpoints and disables the single-step execution. This way, both the functionality and security are preserved.

**Address Space Layout Randomization:** GuardRules require absolute addresses to enable protection. However, RuleMaker can only encode the offsets in a GuardRule because the translation of GuardSpec to GuardRules happens off-line. To get the correct run-time addresses, we need to know the base address when the binary is (randomly) loaded. To this end, iDriver traverses the process' list of the VMA (virtual memory area) structures to find the load address of the binary and adds

[1]The flag $CONFIG\_HAVE\_HW\_BREAKPOINT$ is not set in the kernel configure file.

it to the offsets to finalize the GuardRules.

**Communication between components:** InstaGuard's components communicate using the signal IPC mechanism. After a system-wide survey, we choose the `SIGUSR2` signal for this purpose because `SIGUSR2` is not used by any Android system components. We modify the bionic linker to load iMonitor in target processes and registers iMonitor's function as the handler for *SIGUSR2*. Moreover, we hook the signal registration routine in the kernel to protect our handler for `SIGUSR2`.

### B. RuleMaker Translation Challenges

**Variable resolution:** One critical support we provide to InstaGuard users is variable resolution so that they can use source code symbols to write GuardSpec. We can resolve global, local, and heap variables. To achieve that, we utilize the debug information [2]. The DWARF debug format provides the useful information about variable types, their scope, and the steps to locate a variable in the given scope. Scope is important. RuleMaker will prompt error if the variable to lookup is not active in the scope of the target breakpoint. The variable resolution module of RuleMaker is built on top of $pyelftools$ [3]. It uses the `DW_AT_LOCATION` and `DW_AT_TYPE` fields to locate variables and their types. The variable retrieval rules are encoded in GuardRule and interpreted by iMonitor at runtime, as described in GuardRule (§ III).

To generate full debug information, we added some extra flags to the compilation options[2] when building the Android images. However, there are rare cases that the locations of some variables are missing due to aggressive compiler optimizations. For instance, the local variable `numRects` of the function `Region::flatten` in `libui.so` is optimized away. There is no `DW_AT_LOCATION` tag for this variable in the debug information (see § V). In this situation RuleMaker prompts the user for assistance to locate the equivalent variable.

We want to point out that the released Android binaries can have the debug information stripped because RuleMaker is run offline. Only the developer needs to have access to the debug information to translate GuardSpec to GuardRules.

**Context-binding watchpoints:** Unlike breakpoints which are associated with code addresses, watchpoints are associated with variables. However, it is possible that a watched variable have multiple versions alive at the same time. For example, to protect a stack-based buffer overflow vulnerability, InstaGuard needs to monitor the local buffer variable. If the vulnerable function is called recursively, there is a different version of the buffer for each stack frame of this function. The available hardware watchpoints will be quickly exhausted in this case. To address this problem, we propose a technique called *context-binding watchpoint*. The basic idea is to associate a WP with the current execution context and save/restore the WP when the stack frame changes. By doing so, we can reuse the watchpoints for each function activation. We implement this technique with the assistance of BPs, which are naturally context-associated. Specifically, we use BPs to monitor the faulting code execution, install and remove the WP when entering and leaving the faulting code region, respectively.

**Source code location translation:** Sometimes, RuleMaker needs to generate rules to verify whether certain memory access comes from a specified source code location, for example, to detect buffer overflow (Table II). However, the source code to machine addresses translation is not always a one-to-one mapping in the compiled binary. In that case RuleMaker encodes the code ranges in the assertion primitives.

## V. Analysis and Evaluation

### A. Security Analysis

We now analyze our system design against bypass and manipulation attempts from attackers. We assume the attacker has already gained a foothold in an untrusted app running on the user device. Her goal is to exploit a known vulnerability protected by InstaGuard in a high-privilege system daemon.

There are multiple attack surfaces the attacker may try to abuse. Firstly, she may try to trick iDaemon into initiating the GuardRule removal process in the target process. This attempt will be foiled because iDaemon does not accept rules whose cryptographic signature does not match the one from the trusted source. This protection is similar to how the SEAndroid rules are verified when an OTA update is released to user devices. Secondly, the attacker may try to directly signal the target iMonitor. This will not succeed either because iDriver will kill any sender process of `SIGUSR2` other than iDaemon. Last but not least, the attacker may try to abuse the interface between iMonitor and iDriver (similar to syscalls) from a compromised process to remove the rule enforcement in the target process, a confused deputy attack. This attack is prevented because iDriver only serves the iMonitor requests in the context of requesting process.

### B. Empirical Evaluation

In our empirical evaluation, we try to answer the following questions: (*i*) Is GuardSpec and GuardRule expressive enough to cover a wide range of vulnerabilities in the different categories? (*ii*) How much effort it requires for security experts who do not know InstaGuard internals to compose an *effective* GuardSpec? and (*iii*) What is the performance and memory impact introduced by the installed rules?

**Vulnerability coverage:** We sampled 30 critical vulnerabilities in the native code of the Android framework from the year of 2016. These vulnerabilities can be roughly categorized into integer overflows, buffer overflows, out-of-bound accesses, and logic bugs. All these vulnerabilities can be expressed in GuardSpec and successfully lowered to GuardRule, as shown in Table V. In the following, we report our observations in writing rules for these vulnerabilities.

First, it seems that spatial memory corruptions and logical bugs prevail in the native framework code, while vulnerabilities like format-string vulnerabilities are rare in the Android framework probably due to the improved static-analysis tools used in

[2] *-ggdb -fstandalone-debug* for CLANG, `-fvar-tracking -ggdb -fvar-tracking-assignments` for GCC.

***TABLE V:*** *Covered Vulnerabilities. Entries with (*) are done by security experts from collaborating vendor*

| CVE number | Vulnerability type | Affected module name | Bug description | # of lines in GuardSpec | Example GuardSpec (Appendix) |
|---|---|---|---|---|---|
| 2016-0811 | integer overflow | libmedia.so | The sum of offset and totalSize could wrap around | 9 | |
| 2016-3822 | integer overflow | libjhead.so | Integer overflow in ProcessExifDir | 9 | |
| 2016-0803 | integer overflow | libstagefright_soft_mpeg4enc.so | Product of mWidth and mHeight could exceed INT32_MAX / 3 | 9 | |
| 2016-3744 | integer overflow | libdeqp.so | Integer overflow in create_pbuf | 9 | |
| 2016-0815 | integer overflow | libstagefright.so | The sum of dstOffset and mBuffer->size() can wrap around | 9 | |
| 2016-0837 | integer overflow | libstagefright.so | Missing bound check and integer overflows in MPEG4Source::read | 15 | A.1 |
| 2016-2463 | integer overflow | libstagefright.so | Product of pStorage->picSizeInMbs and u32 could overflow a 32-bit integer | 9 | |
| 2016-0827 | integer overflow | libreverbwrapper.so | SIZEOF(effect_param_t) + p->psize could exceed SIZE_MAX | 9 | |
| 2016-0849 | integer overflow | libminzip.so | range_count * SIZEOF(MappedRange) could overflow a 32-bit integer | 9 | |
| 2016-2428 | integer overflow | libFraunhoferAAC.so | The number of threads could exceed MAX_DRC_THREADS | 9 | |
| 2016-3895* | integer overflow | libui.so | Product of numRects and Rect could overflow a 32-bit unsigned integer | 9 | |
| 2016-3872 | integer overflow | libstagefright_soft_vpxdec.so | Integer overflow in SoftVPX::outputBuffers | 9 | |
| 2016-3819 | integer overflow | libstagefright_soft_h264dec.so | Product of picSizeInMbs and 384 could exceed UINT32_MAX | 9 | |
| 2016-2451 | buffer overflow | libstagefright_soft_vpxdec.so | Buffer overflow in parameter dst | 7 | |
| 2016-2484 | buffer overflow | libstagefright_soft_amrdec.so | Buffer overflow in outHeader->pBuffer | 7 | |
| 2016-3863 | buffer overflow | libstagefright.so | Stack overflow in AVCC reassemble | 7 | |
| 2016-2485 | buffer overflow | libstagefright_soft_vpxenc.so | Buffer overflow in outHeader->pBuffer | 7 | A.2 |
| 2015-1474* | buffer overflow | libui.so | Buffer overflow in h->data | 7 | |
| 2016-3871 | buffer overflow | libstagefright_soft_avcenc.so | Buffer overflow in outHeader->pBuffer | 7 | |
| 2016-2494 | buffer overflow | sdcard | Buffer overflow in path building | 7 | |
| 2016-0836* | out-of-bound access | libstagefright_soft_mpeg2dec.so | OOB access of pu1_pos with index pi4_num_Coeffs | 7 | A.3 |
| 2016-0840 | out-of-bound access | libstagefright_soft_avcdec.so | OOB access of i2_level_arr with index u4_total_coeff-1 | 7 | |
| 2016-6707 | Use-after-Free | libandroid_runtime.so | Sizes used in mmap and munmap are mismatched (block free) | 8 | A.4 |
| 2016-3861* | logic bug | libutils.so | Incorrect logic when counting length of converted string | 8 | |
| 2016-0835 | logic bug | libmpeg2dec.so | Miss sanity check when processing input | 8 | |
| 2016-2417 | logic bug | libmedia.so | Forgot to sanitize the allocated memory | 8 | |
| 2016-2418 | logic bug | libmedia.so | Incorrect logic as miss sanity check on return value | 8 | A.5 |
| 2016-2419 | logic bug | libmedia.so | Info leak as unsanitized variable | 8 | |
| 2016-3826 | logic bug | libaudioresampler.so | Miss sanity check on cmdCode and replySize before using it | 15 | |
| 2016-0816 | logic bug | libavcdec.so | Incorrect logic when counting decoded bytes | 9 | |

the development. Second, we can compose GuardSpec for each vulnerability by filling in the required fields of the templates in Table III. Most fields can be conveniently extracted from the associated bug reports. Third, most GuardSpec rules are less than 10 lines, with only two exceptions. These two rules contain more assertions. For example, CVE-2016-0837, an integer overflow, can be exploited to cause out-of-bound access and eventually arbitrary-writes. The vulnerable code has the following insufficient bound check, `CHECK(dstOffset + 4 <= mBuffer->size())`. An attacker can bypass this check by overflowing `dstOffset + 4`. Moreover, the code simply forgets to check the bound for a subsequent access to the buffer. InstaGuard can protect this vulnerability by supplementing the original check with two additional assertions to prevent the overflows.

There are some CVEs which are hard to protect without risking availability lost, for instance, CVE-2016-2417 and CVE-2016-2419, the vulnerabilities could cause info leak as a consequence of uninitialized stack and heap variables. For these two vulnerabilities, we switch the `decision` to AUDIT in their GuardSpec files, to preserve the availability of the affected functions. This is an indication that InstaGuard should be applied to vulnerabilities that pose direct security threat (i.e, control flow hijacking) to the to-be-patched programs. A few additional GuardSpec samples for each category are presented in the Appendix.

Lastly, when a POC (Proof-Of-Concept) exploit is available, we installed the corresponding GuardRule and test if InstaGuard can report and block the exploit. When a POC exploit is absent, we studied the released patches to manually check the correctness of the composed GuardSpec. We verified that InstaGuard can capture all the exploits (no false negatives) and preserve the availability of the affected functions.

***TABLE VI:*** *Reported time for composting effective GuardRule.*

| | InC compose time (mins) | InC Line# | InP Line# |
|---|---|---|---|
| *CVE-2016-3895* | 40 | 9 | 52 |
| *CVE-2016-0836* | 20 | 7 | 45 |
| *CVE 2016-3861* | 15 | 8 | 55 |
| *CVE-2015-1474* | 15 | 7 | 94 |
| *Avg.* | 22.5 | 7.75 | 61.5 |

**GuardSpec composing efforts:** To measure the efforts to compose an effective GuardSpec, we selected 4 critical vulnerabilities from Table V, one from each category, and asked four security engineers of our collaborating vendor to write GuardSpec for them. We timed the whole process for them to write GuardSpec and use RuleMaker to translate it to GuardRule. Table VI shows the time required to complete the tasks. On average, they were able to produce an effective and correct GuardSpec and translate it into GuardRule in about 22.5 minutes. Three participants succeeded at the first try. It took more time in the case of CVE-2016-3895 because the debug information did not provide the lookup rule for the related variable `numRects`. Consequently, RuleMaker failed to emit the retrieval rule for that variable. The participant had to manually identify the location of that variable. It turns out that, because $numRects$ only contains the first field of the input buffer, the compiler simply emits the code to read from the buffer. This completely eliminates the local variable $numRects$. To examine the quality of the written rules, we applied them to the test phone and verified that the previously-working related POC exploits no longer work (e.g, [11]).

**Performance and memory overhead:** we further verify that normal operation of the phone is unaffected and measure the

***TABLE VII:*** *InstaGuard performance and memory overheads.*

| | Used Primitives | Root Cause | Memory Overhead(%) | Runtime Slowdown(%) |
|---|---|---|---|---|
| *CVE-2016-3895* | (1x)BP, (1x)AS | Integer Overflow | 0.37% | 2.89% |
| *CVE-2016-0836* | (1x)BP, (1x)AS | Out-of-Bound | 4.11% | 3.27% |
| *CVE 2016-3861* | (1x)BP, (1x)AS | Logic Bug | 1.08% | 2.70% |
| *CVE-2015-1474* | (2x)BP, (1x)WP, (1x)AS | Buffer Overflow | 1.19% | 1.94% |
| *Avg.* | | | 1.69% | 2.70% |

performance overhead caused by InstaGuard. We tested the four GuardRule rules representing 4 different categories of vulnerabilities listed in Table VII. For triggering the vulnerable code path, we used either the test programs released together with the vulnerable library (e.g, *mpeg2dec*) or searched for system services that used the target binary. For instance, *libui.so* is used by the *bootanimation* program, and when we execute *bootanimation* the vulnerable code path in CVE-2015-1474 will be triggered. This way, we make sure all benign executions went through the vulnerable code path at least once; i.e., (part of) the patches are guaranteed to be exercised. Each test was run 20 times. We used *getrusage()* to collect the average execution time with and without GuardRule installed. Table VII shows the runtime and peak memory overheads.

On average, the performance overhead is 2.7%. It is worth noting that, although the required primitives for CVE-2015-1474 are twice as many as the others, its watchpoint was never triggered under this input. Its overhead is mainly caused by the two breakpoints. The overhead of CVE-2016-0836 is higher because its original execution time is shorter, which amplifies the overhead caused by executing the patch. The average memory overhead is 1.69%. The main source of overhead comes from iMonitor. The variable sizes of the rules are negligible. These evaluation results show that InstaGuard incurs unnoticeable overheads in terms of both performance and memory overheads. Furthermore, to measure how performant InstaGuard is when protecting multiple vulnerabilities in the same process, we created a wrapper service that contains the aforementioned 4 types of the vulnerabilities and applied the corresponding GuardRules. We created the test input to trigger the vulnerable code path, each 20 times. The average total performance overhead is 9.73%, which represents the accumulative overheads from each GuardRule.

Overall, we can estimate that the performance overhead caused by one breakpoint is about 0.97% and 2.01% for evaluating an assertion. We note that these overhead is an upper-bound since in practice the end-to-end slowdown depends on how frequently the vulnerable code path is executed. It is far more likely that security vulnerabilities exist in the cold paths as they are less tested [36].

## VI. Related Works

### A. Hot-patching

Many hot patching techniques have been proposed, including those for computer programs [30], [21], [18], [20], [24], [26], for the Linux kernel [35], [6], [19], and more recently, for Android [28], [22]. Despite their different designs and implementations, the existing hot-patching systems all follow the similar basic approach–injecting hot-patches as executable code into target programs in order to disable or replace the vulnerable code. As revealed by our study (§ II), this approach does not consider the unique constraint facing code patches for mobile devices (i.e., all patches to system code are subject to lengthy carrier-imposed tests). Therefore, the existing hot-patching methods see little adoption on mobile devices, despite that mobile devices in fact urgently need hot-patching capabilities.

In contrast, we follow a completely different approach, namely rule-driven hot-patching, when designing InstaGuard. Our approach enables "carrier-passthrough" hot-patches that do not carry or inject code and are restrictive by design. Moreover, to ease and speed up the hot-patch development process, we design a simple vulnerability description language and an automatic rule synthesizer. They allow human developers to quickly produce hot-patches without having to understand the hot-patching mechanics, which is *not* required by previous systems.

IntroVirt [26] injects code predicates to vulnerable programs to block exploitations. These predicates are executable code pieces that use the low-level debugging primitives to intercept executions and check conditions. In comparison, InstaGuard avoids code patches and follows a rule-driven approach. InstaGuard internally also uses the debugging primitives, but unlike IntroVirt, InstaGuard does not expose the low-level debugging primitives to human users and is accompanied by RuleMaker, making hot-patch (or rule) development much less demanding.

For Android apps, PatchDriod [28] uses a dynamic code injector to apply either binary or Dalvik patches in memory. KARMA [22] is a recent system for live patching Android kernels. Its patches are written in an interpretation-based language, Lua, and executed by an interpreter planted in Android kernels. Thanks to the use of the scripting language, KARMA's patches do not need to be binary-compatible with individual device models, which are highly fragmented. KARMA represents a variant of the traditional code-based hot-patching techniques, whereas InstaGuard's rule-driven approach is a departure from the existing techniques and emphasizes carrier-passthrough, patch safety,and fast patch development.

### B. Automatic Bug Detection and Mitigation

This line of research is related to InstaGuard in that they represent parallel methods for reducing the vulnerability response or patching time. DIRA [34] removes certain control-hijacking vulnerabilities from programs via recompilation. Automatic patches were also used to defend against worms [33]. VSEF [29] hardens program binaries by filtering out execution traces that correspond to exploitations of specific vulnerabilities. First-Aid [23] monitors program execution for memory management bugs. When it determines such a bug is triggered, it reverts the program execution to the last checkpoint and allow it to proceed with patched code.

Compared to these systems. InstaGuard can efficiently and effectively catch exploit attempts for known vulnerabilities. instead of detecting and patching after the attack or delaying the protection until the attack happened. Also, InstaGuard does not introduce code changes and ensures patch safety, which is more ideal for the Android ecosystem where traditional code patches cannot be quickly deployed to vulnerable devices.

### C. Rule-driven Security Defense

Program shepherding [27] is a classic example of rule-based protection of program execution. Its rules can specify executable code origins, restrict control transfers, and prevent bypasses of checks. It uses a dynamic code instrumentation tool for policy enforcement. Another common example of rule-based security is program access controls, either discretionary or mandatory. These and other similar works have demonstrated a key benefit of the rule-driven approach—the separation between mechanisms and policies. These works inspired us to design a rule-based approach to hot-patching. On the other hand, we also took an important lesson from previous rule-based systems: without supporting easy rule generation, such a system is unlikely to be used in practice. To enable easy and quick development of GuardRule, we designed the simple GuardSpec language and RuleMaker to automatically synthesize GuardRule from low-level GuardSpec.

Talos [25] is a recent vulnerability mitigation system. It pre-plants a kill-switch in every function in a program. When a function is later found vulnerable, it activates the corresponding kill-switch to disable that function. Talos achieves vulnerability mitigation without using code patches. However, it has to disable entire functions that contain vulnerabilities (i.e., reduced program functionality). This is because Talos only recognizes activation commands for function kill-switches, rather than comprehensive rules (e.g., GuardSpec) that describe the precise triggering conditions for various kinds of vulnerabilities.

## VII. Discussion and Future Work

**Hardware debug registers:** Our current prototype uses the hardware debug unit of ARMv8 to support InstaGuard's breakpoint and watchpoint primitives (§ IV). The current ARMv8 CPUs can support up to 32 break/watch points [4]. This implementation choice might limit the number of total GuardRules that can be supported in a single vulnerable process. However, our survey of all the critical vulnerabilities reported in Android Security Bulletin 2016 shows that the available hardware registers are sufficient to protect every Android system binary (Fig. 7). Note that we save and restore the GuardRules during the context switch. Therefore, each process can use all the hardware debug registers. In addition, we can easily extend the current InstaGuard implementation to support unlimited number of breakpoints with software instruction breakpoints. Moreover, InstaGuard aims at protecting released programs on consumer Android devices, rather than programs running in debugging mode, in which case InstaGuard should be disabled for those programs to avoid interfering with their debugging.

**Response to detected attacks:** Our prototype supports two actions in response to detected attacks: "BLOCK" and "AUDIT". The former terminates the faulting process, while the latter simply records the alert event. This choice is sufficient for our prototype because most of our target programs (Android system daemons and services) can recover from crashes or abrupt termination. Moreover, our framework does allow more sophisticated response to attacks: when a breakpoint is hit, the kernel saves the current process states to the stack. InstaGuard can manipulate the saved states to change the process' control flow or its data. For example, we can set the error code (by changing the saved register) and jump directly to the function's return instruction (by changing the saved program counter) if the function has proper error-handling code, similar to Talos [25] and KARMA [22]. Nevertheless, attack recovery is a challenging problem. We consider it out of the scope for this paper and leave it as the future work.

**Kernel protection and project Treble:** InstaGuard focuses on protecting the Android user space from attacks and considers the kernel is trusted. Existing systems such as KARMA [22] have been proposed to protect the integrity of the Android kernel. On the other hand, InstaGuard can shield the kernel from some exploits originated from the user space by protecting the Android system binaries from attacks. A common way to gain the root privilege on Android is to compromise the vulnerable system daemons [14], [31].

Project Treble is a new feature in the Android O release. It separates the Android OS framework from the hardware-specific vendor implementation so that vendors can more easily update their devices to a new version of Android. Ideally, an updated Android OS framework with bug fixes and new features should run without any problem on the vendor implementation. However, Project Treble is unlikely to significantly reduce the vulnerability life cycle [16]: first, almost all the Android devices are heavily customized (e.g., Samsung's TouchWiz, Huawei's EMUI, LG's UX). Android vendors still need to port these customizations to the new release. Second, the new Android OS is still subject to lengthy testing by the vendor and the carriers. Therefore, InstaGuard is still valuable in quickly closing the vulnerability window.

**Notes on RuleMaker:** In our current design of RuleMaker, we use hardcoded rules to translate GuardSpec into GuardRules. We plan to optimize it by employing an automatic metric-based rule selection tool, for example [36], to improve efficiency. When automatically synthesizing GuardRules from GuardSpec, RuleMaker uses debug symbol information to resolve source-code level references in GuardSpec. Although the intended users of our system (e.g., device or OS vendors) always have access to debug symbols, we note that InstaGuard itself functions without symbol information and allows users to manually generate GuardRule using pre-defined templates if needed.

## VIII. Conclusion

We present InstaGuard, a new approach to hot-patching that allows instant patch deployment on Android platforms. Unlike existing hot-patching techniques, which directly inject code into vulnerable programs or their memory, InstaGuard enforces easy-to-generate rules, called GuardRule, that are expressive enough to mitigate a wide range of vulnerabilities. Moreover, these rules are restrictive by design, preventing InstaGuard from being abused to introduce unsafe behaviors to patched programs. We also build RuleMaker, a GuardRule generation tool that takes as input GuardSpec, a high-level vulnerability description that developers or security analyst can quickly compose solely based on a basic understanding of given vulnerabilities. We collaborate with a major mobile device vendor to evaluate and deploy InstaGuard. Our evaluation

shows that InstaGuard is versatile enough to handle commonly seen real-world vulnerabilities and incurs negligible overhead.

## ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their insightful comments. This project was supported by the National Science Foundation (Grant#: CNS-1652205, CNS-1421824, and CNS-1453020) and the Army Research Office (Grant#: W911NF-17-1-0039). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies. We also thank Xiaoyong Zhou, Tomislav Suchan, Kunal Patel, Mike Grace, Wenbo Shen and Andrea Possemato for the supports and helpful discussions during the development of InstaGuard.

## REFERENCES

[1] "Microsoft hotfix mechanism," https://blogs.technet.microsoft.com/hot/2007/12/26/something-about-hotfix/, 2007.

[2] "Dwarf debugging information, version 4," http://dwarfstd.org/doc/DWARF4.pdf, 2010.

[3] "Python library for parsing elf and dwarf," https://github.com/eliben/pyelftools, 2011.

[4] "Armv8 hardware debugging interfaces," http://infocenter.arm.com, 2012.

[5] "Htc system update," http://www.htc.com/us/go/htc-software-updates-process/, 2013.

[6] "Redhat enterprise kpatch solution," http://rhelblog.redhat.com/2014/02/26/kpatch/, 2014.

[7] "Samsung security policy update," https://play.google.com/store/apps/details?id=com.policydm&hl=en, 2014.

[8] "Android fragmentation visualized," http://opensignal.com/reports/2015/08/android-fragmentation, 2015.

[9] "2016 android security bulletins," https://source.android.com/security/bulletin/2016.html, 2016.

[10] "Android monthly security maintenance release," https://www.android.com/security-center/monthly-security-updates/, 2016.

[11] "Cve-2016-3861 poc," https://bugs.chromium.org/p/project-zero/issues/attachment?aid=249763, 2016.

[12] "Quadrooter: New android vulnerabilities in over 900 million devices," http://blog.checkpoint.com/2016/08/07/quadrooter/, 2016.

[13] "Ubuntu commercialized live patch solution," https://www.ubuntu.com/server/livepatch, 2016.

[14] "Bitunmap: Attacking android ashmem," https://googleprojectzero.blogspot.com/2016/12/bitunmap-attacking-android-ashmem.html, 2017.

[15] "Defeating samsung knox with zero privilege," https://www.blackhat.com/docs/us-17/thursday/us-17-Shen-Defeating-Samsung-KNOX-With-Zero-Privilege.pdf, 2017.

[16] "Google's "project treble" solves one of android's many update roadblocks," https://arstechnica.com/gadgets/2017/05/google-hopes-to-fix-android-updates-no-really-with-project-treble/, 2017.

[17] "Have we seen the end of ios hot patching?" https://www.appthority.com/mobile-threat-center/blog/seen-end-ios-hot-patching/, 2017.

[18] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "Opus: Online patches and updates for security." in *Usenix Security*, vol. 5, 2005, p. 18.

[19] J. B. Arnold, "Ksplice: An automatic system for rebootless kernel security updates," Ph.D. dissertation, Massachusetts Institute of Technology, 2008.

[20] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 35–44.

[21] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 271–281.

[22] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *Proceedings of the 26th USENIX Security Symposium*, August 2017.

[23] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: surviving and preventing memory management bugs during production runs," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 159–172.

[24] M. Hicks and S. Nettles, "Dynamic software updating," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 6, pp. 1049–1096, 2005.

[25] Z. Huang, M. DAngelo, D. Miyani, and D. Lie, "Talos: Neutralizing vulnerabilities with security workarounds for rapid response," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 618–635.

[26] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting past and present intrusions through vulnerability-specific predicates," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 91–104.

[27] V. Kiriansky, D. Bruening, S. P. Amarasinghe *et al.*, "Secure execution via program shepherding." in *USENIX Security Symposium*, vol. 92, 2002, p. 84.

[28] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 259–268.

[29] J. Newsome, D. Brumley, and D. Song, "Vulnerability-specific execution filtering for exploit prevention on commodity software," 2006.

[30] M. Payer and T. R. Gross, "Hot-patching a web server: A case study of asap code repair," in *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*. IEEE, 2013, pp. 143–150.

[31] O. Peles and R. Hay, "One class to rule them all: 0-day deserialization vulnerabilities in android." in *WOOT*, 2015.

[32] G. Russello, A. B. Jimenez, H. Naderi, and W. van der Mark, "Firedroid: hardening security in almost-stock android," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 319–328.

[33] S. Sidiroglou and A. D. Keromytis, "Countering network worms through automatic patch generation," *IEEE Security & Privacy*, vol. 3, no. 6, pp. 41–49, 2005.

[34] A. Smirnov and T.-c. Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks." in *NDSS*, 2005.

[35] SUSE, "kGraft," https://www.suse.com/products/live-patching, 2014.

[36] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 866–879.

[37] Y. Zhang, Y. Chen, C. Bao, L. Xia, L. Zhen, Y. Lu, and T. Wei, "Adaptive kernel live patching: An open collaborative effort to ameliorate android n-day root exploits," in *Proceedings of Black Hat USA 2016*, Las Vegas, NV, 2016.

## APPENDIX

## APPENDIX: GUARDSPEC EXAMPLES

### A. CVE-2016-3895

This is a integer overflow vulnerability in libs/ui/Region.cpp in mediaserver on Android. The uint32_t variable *numRects * sizeof(Rect)* in $Region : flatten$ function can be overflowed, which invalidates existing security checks. It allows attackers to obtain sensitive information via a crafted application.

```
[common]
ID = CVE-2016-3895
binary_path = /system/bin/surfaceflinger
module_name = libui.so
decision = BLOCK

[integer overflow]
involved_vars = numRects,Rect
overflow_exp =  numRects * Rect
overflow_dir = MAX
trigger_value = 0xffffffff
vul_location = frameworks/native/libs/ui/Region.cpp
  ↪  | Region:flatten | 794
```

### B. CVE-2016-3871

This is a buffer overflow vulnerability in codecs/mp3dec/SoftMP3.cpp in libstagefright of mediaserver on Android. The *SoftMP3::onQueueFilled* function uses memset to cleanup memory, but it can accept a large *len* value which leads to heap overflow, it was fixed by adding additional boundary check before calling memset. It allows attackers to gain privileges via a crafted application.

```
[common]
ID = CVE-2016-3871
binary_path = /system/bin/mediaserver
module_name =libstagefright_soft_avcenc.so
decision = BLOCK

[buffer overflow]
buf_name = outHeader->pBuffer
buf_size = outHeader->nAllocLen
vul_location =
  ↪  libstagefright/codecs/mp3dec/SoftMP3.cpp
  ↪  |SoftMP3::internalGetParameter | 303
```

### C. CVE-2016-0836

This is a out-of-bound access vulnerability in decoder/impeg2d_vld.c in mediaserver on Android. The index *pi4_num_coeffs* of buffer pi2_coeffs and pu1_pos within $impeg2d\_vld\_decode$ function can be increased beyond buffer boundary, adding a boundary check within the while loop can fix the problem. It allows remote attackers to execute arbitrary code or cause a denial of service via a crafted media file.

```
[common]
ID = CVE-2016-0836
binary_path = /system/bin/mediaserver
module_name =libstagefright_soft_avcenc.so
decision = BLOCK

[out-of-bound access]
index_var = pi4_num_coeffs
buf_size_var = 64
vul_location =
  ↪  external/libmpeg2/decoder/impeg2d_vld.c |
  ↪  impeg2d_vld_decode | 696
```

### D. CVE-2016-6707

This is a use-after-free vulnerability in graphics/Bitmap.cpp in system server on Android. The memory region size of munmap call within $Bitmap :: doFreePixels$ function can be controlled from user process, which results in unmapping crucial regions of memory in the remote process. It allows an attacker to replace the heap region of the remote process with controlled data and eventually leads to privilege escalation.

```
[common]
ID = CVE-2016-6707
binary_path = /system/bin/app_process64
module_name = libandroid_runtime.so
decision = BLOCK

[use-after-free]
lexp = mPixelStorage.ashmem.size
rexp = VMA_SIZE(mPixelStorage.ashmem.address)
relation_op = NE
vul_location = core/jni/android/graphics/Bitmap.cpp
  ↪  | Bitmap::doFreePixels | 186
```

### E. CVE-2016-2417

This is a logic bug in media/libmedia/IOMX.cpp in mediaserver on Android. The *params* data structure in $BnOMX :: onTransact$ function is not properly initialized, which allow an attacker to obtain sensitive information from process memory, and consequently bypass security measures in place.

```
[common]
ID = CVE-2016-2417
binary_path = /system/bin/mediaserver
module_name = libmedia.so
decision = AUDIT

[logic bug]
vul_location =
  ↪  frameworks/av/media/libmedia/IOMX.cpp |
  ↪  BnOMX::onTransact | 621
lexp = *param
rexp = 0
relation_op = NE
```