University of California
Santa Barbara

Protecting Smart Devices from the Bottom-up

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Aravind K Machiry

Committee in charge:

Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel, Co-Chair
Professor Ben Hardekopf
Professor Michael Hicks, University of Maryland, College Park.

September 2020

The Dissertation of Aravind K Machiry is approved.

Professor Ben Hardekopf

Professor Michael Hicks, University of Maryland, College Park.

Professor Christopher Kruegel, Committee Co-Chair

Professor Giovanni Vigna, Committee Co-Chair

August 2020

Protecting Smart Devices from the Bottom-up

Copyright © 2020

by

Aravind K Machiry

I am dedicating this dissertation to my parents, who always believed in me and supported my academic pursuits.

Acknowledgements

I want to thank and am grateful (in no strict order):

- To Prof. Christopher Kruegel and Prof. Giovanni Vigna for sharing their wisdom and allowing me the freedom to pursue any problem of my interest.
- To all the reviewers for rejecting my papers and consequently resulted in making them better.
- To all the members of SecLab for helping me navigate the various ups and downs during my Ph.D.
- To all my collaborators and co-authors for helping me write good papers.
- To Prof. Mayur Naik for introducing me to academic research.

Curriculum Vitæ

Aravind K Machiry

Education

2020	Ph.D. in Computer Science (Expected), University of California, Santa Barbara, California, USA.
2013	M.S. in Information Security, Georgia Institute of Technology, Atlanta, USA.

Publications

1. C. Salls, **Aravind Machiry**, A. Doupe, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. "Exploring Abstraction Functions in Fuzzing." *Proceedings of the 2020 IEEE Conference on Communications and Network Security (**CNS**)*, **2020**
2. C. Spensky, **Aravind Machiry**, M. Busch, K. Leach, R. Housley, C. Kruegel, and G. Vigna. "TRUST.IO: Protecting Physical Interfaces on Cyber-physical Systems." *Proceedings of the 2020 IEEE Conference on Communications and Network Security (**CNS**)*, **2020**
3. **Aravind Machiry**, N. Redini, E. Cammellini, C. Kruegel and G. Vigna. "SPIDER: Enabling Fast Patch Propagation in Related Software Repositories." *Proceedings of the 41st IEEE Symposium on Security and Privacy (**S&P**)*, **2020**
4. N. Redini, **Aravind Machiry**, R. Wang, C. Spensky, A. Continella Y. Shoshitaishvili, C. Kruegel and G. Vigna. "KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware." *Proceedings of the 41st IEEE Symposium on Security and Privacy (**S&P**)*, **2020**
5. **Aravind Machiry**, H. Touma, R. Chen, M. Hicks. "(POSTER) Automated conversion of legacy code to Checked C." *Proceedings of the IEEE Secure Development Conference (**SecDev**)*, **2019**
6. E. Gustafson, M. Muench, C. Spensky, N. Redini, **Aravind Machiry**, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. E. Choe, C. Kruegel, G. Vigna. "Toward the Analysis of Embedded Firmware through Automated Re-hosting." *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (**RAID**)*, **2019**
7. N. Redini, R. Wang, **Aravind Machiry**, Y. Shoshitaishvili, C. Kruegel and G. Vigna. "BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation." *Proceedings of the 16th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (**DIMVA**)*, **2019**
8. **Aravind Machiry**, N. Redini, E. Gustafson, H. Aghakhani, C. Kruegel and G. Vigna. "Detecting Deceptive Reviews using Generative Adversarial Networks." *Proceedings of the 2nd Binary Analysis Research Workshop (**BAR**)*, **2019**.

9. **Aravind Machiry**, N. Redini, E. Gustafson, Y. Fratantonio, Y. E. Choe, C. Kruegel and G. Vigna. "Using Loops For Malware Classification Resilient to Feature-unaware Perturbations." *Proceedings of the 34th Annual Application Security Application Conference (**ACSAC**)*, **2018**
10. H. Aghakhani, **Aravind Machiry**, S. Nilizadeh, C. Kruegel and G. Vigna. "Detecting Deceptive Reviews using Generative Adversarial Networks." *Proceedings of the 1st Deep Learning and Security Workshop (**DLS**)*, **2018**.
11. A. Bianchi, Y. Fratantonio, **Aravind Machiry**, C. Kruegel, G. Vigna, S. Chung, W. Lee. "Broken Fingers: On the Usage of the Fingerprint API in Android." *Proceedings of the ISOC Network and Distributed System Security Symposium (**NDSS**)*, **2018**.
12. A. Bianchi, K. Borgolte, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, **Aravind Machiry**, C. Salls, N. Stephens, G. Vigna, R. Wang (Authors listed alphabetically). "Mechanical Phish: Resilient Autonomous Hacking." *IEEE Security & Privacy Magazine - **SPSI: Hacking without Humans 2018***.
13. N. Redini, **Aravind Machiry**, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, G. Vigna. "BootStomp: On the Security of Bootloaders in Mobile Devices." *Chaos Communication Congress (**34C3**)*, **2017**.
14. J. Corina, **Aravind Machiry**, C. Salls, Y. Shoshitaishvili, Shuang Hao, C. Kruegel, and G. Vigna. "DIFUZZING Android Kernel Drivers." *Black Hat Europe London, UK December (**BH EU**)*, **2017**.
15. J. Corina, **Aravind Machiry**, C. Salls, Y. Shoshitaishvili, Shuang Hao, C. Kruegel, and G. Vigna. "DIFUZE: Interface Aware Fuzzing for Kernel Drivers." *Proceedings of the 24th ACM Conference on Computer and Communications Security (**CCS**)*, **2017**. Finalist for **CSAW Applied Research Competition**.
16. **Aravind Machiry**, C. Spensky, J. Corina, N. Stephens, C. Kruegel, G. Vigna. "DR.CHECKER: A Soundy Analysis for Linux Kernel Drivers." *Proceedings of the 26th USENIX Security Symposium (**USENIX Security**)*, **2017**. Runner up for **Facebook Internet Defense Prize**
17. N. Redini, **Aravind Machiry**, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, G. Vigna. "BootStomp: On the Security of Bootloaders in Mobile Devices." *Proceedings of the 26th USENIX Security Symposium (**USENIX Security**)*, **2017**.
18. **Aravind Machiry**, E. Gustafson, C. Spensky, C. Salls, N. D. Stephens, R. Wang, A. Bianchi, Y. E. Choe, C. Kruegel, G. Vigna. "BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments." *Proceedings of the ISOC Network and Distributed System Security Symposium (**NDSS**)*, **2017**.
19. R. Wang, Y. Shoshitaishvili, A. Bianchi, **Aravind Machiry**, J. Grosen, P. Grosen, C. Kruegel, G. Vigna. "Ramblr: Making Reassembly Great Again." *Proceedings of the ISOC Network and Distributed System Security Symposium (**NDSS**)*, **2017**. Won **Distinguished Paper Award**.

20. A. Bianchi, K. Borgolte, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, **Aravind Machiry**, C. Salls, N. Stephens, G. Vigna, R. Wang (Authors listed alphabetically). “Cyber Grand Shellphish." ***Phrack***, **2017**.
21. Y. Fratantonio, **Aravind Machiry**, A. Bianchi, C. Kruegel, G. Vigna. “CLAPP: Characterizing Loops in Android Applications." *Proceedings of the ACM Symposium on Foundations of Software Engineering (**FSE**)*, **2015**.
22. Y. Fratantonio, **Aravind Machiry**, A. Bianchi, C. Kruegel, G. Vigna. “CLAPP: Characterizing Loops in Android Applications (Invited Talk)." *Proceedings of the International Workshop on Software Development Lifecycle for Mobile (**DeMobile**)*, **2015**.
23. **Aravind Machiry**, R. Tahiliani, M. Naik. “Dynodroid: An Input Generation System for Android Apps." *Proceedings of the ACM Symposium on Foundations of Software Engineering (**FSE**)*, **2013**. Won **Distinguished Artifact Award**.

Abstract

Protecting Smart Devices from the Bottom-up

by

Aravind K Machiry

Modern systems are mainly composed of IoT devices and Smartphones. Most of these devices use ARM processors, which, along with flexible licensing, have new security architecture features, such as ARM TrustZone, that enables execution of a secure application in an untrusted environment. Furthermore, with well-supported, extensible, open-source embedded operating systems like Android allows the manufactures to quickly customize their operating system with device drivers, thus reducing the time-to-market. Unfortunately, the proliferation of device vendors and race to the market has resulted in poor quality device drivers containing critical security vulnerabilities. Furthermore, the patches for these vulnerabilities get merged into the end-products with a significant delay resulting in the Patch Gap, which causes privacy and security of billions of users to be at risk.

In this dissertation, I will show how the new architecture features can lead to security issues by introducing new attack vectors. Second, I will show that the existing techniques are inadequate to find the security issues in Linux kernel drivers and how, with certain well-defined optimizations, we can precisely find security issues. Third, I will present my solution to the problem of Patch Gap by showing a principled approach to automatically port patches to vendor product repositories.

Finally, I will present our on-going work to automatically port C to Checked C, which provides a low overhead, backward-compatible, and memory-safe C alternative that could be used on resource-constrained modern systems to prevent security vulnerabilities.

Through this work, I presented effective ways to find, fix, propagate, and prevent vulnerabilities in modern system software, thus improving modern systems security.

Contents

Chapter 1

Introduction

Smart devices or IoT devices have become part of the everyday life of billions of people [1]. As market research suggests, their adoption is continuously growing, and the number of devices will reach 20 billion by 2020 [2]. One of the main reasons for this is the availability of well-supported, open-source embedded operating systems like Android. The manufactures usually fork the main repository and quickly customize the operating system with device drivers and reduce the time-to-market.

Smart devices usually contain a RISC processor, predominantly an ARM processor, as one of its main components. Unlike the x86 processor, where Intel and AMD are the leading manufacturers, there are over 200 manufactures [3] of ARM processors, chipsets, and, corresponding devices. One of the reasons for this is the flexible licensing by ARM, which provides a RISC ISA and allows companies to manufacture custom chips that has to at least support the predefined ISA.

The ARM architecture has advanced security architecture features such as TrustZone [4], which provides a secure, isolated execution environment. ARM's TrustZone enables secure applications such as DRM and kernel integrity monitors, which improve the security of smart devices.

1.0.1 Securing from bottom-up

I started my Ph.D. with a noble goal of trying to make smart devices the most secure entities. Being a firm believer in having a secure base, I started to look at the security architecture features of the ARM processor, specifically, ARM's TrustZone. I was fascinated by the fact that TrustZone could prevent the DMA malware problem [5] inherent to x86 processors and also defend against sophisticated attacks that are capable of compromising the operating system (OS) itself.

ARM TrustZone provides a mechanism to have a hardware isolated privileged execution environment known as a trusted execution environment (TEE). The TEE permits the existence of two separate *worlds* on the same system on a chip (SoC), called the *secure world* (i.e., the world inside the TEE) and the *non-secure world* (i.e., the sandboxed world containing the main OS). Each of these worlds contains its dedicated OS and applications, and the software on the system is thus considered to be either *trusted* (i.e., in the secure world) or *untrusted* (i.e., in the non-secure world).

In practice, these two worlds frequently need to communicate with each other (e.g., to encrypt or decrypt data with keys stored inside the TEE). This communication is facilitated by the OSes in both worlds, which leverage specialized memory regions and central processing unit (CPU) registers to establish an application program interface (API) for data exchange. Moreover, most trusted OSes also permit the installation of trusted applications (TAs) to expand functionality and offer services to the untrusted applications in the non-secure world.

The TEE works by facilitating the creation of a non-secure world for untrusted software (e.g., Android and its applications), which is completely isolated from any critical code within the secure world by hardware-enforced mechanisms. Thus, by design, the secure world necessarily has access to all of the non-secure world's memory.

I noticed that this complete isolation between secure and non-secure world results in a *semantic gap*, where the trusted OS is unaware of the security mechanisms of the untrusted OS, i.e., user and kernel level. Furthermore, this semantic gap leads to a type of *confused deputy attack*, wherein a user-level application in the non-secure world can leverage a TA to read from or write to non-secure world memory that it does not own, including the untrusted OS's.

More specifically, a malicious user-level application can send inputs to the TA, which are not properly checked, that will trick the TA into manipulating memory locations that should otherwise be inaccessible to the malicious application. My advisor, Dr. Vigna, named this the BOOMERANG flaw [6], as the untrusted applications attack the untrusted OS through trusted applications giving it a boomerang effect.

Ironically, ARM's TrustZone is supposed to improve the security of the system, but it leads to a new attack vector where a user-level application (e.g., an Android app) could compromise the kernel.

I found BOOMERANG [6] flaw in all the commercial and open-source TEE implementations, which were acknowledged by the corresponding vendors. Furthermore, to demonstrate the severity of BOOMERANG vulnerability, I developed a proof-of-concept exploit to read arbitrary kernel memory on Android. Finally, I developed a novel, low overhead defense, called Cooperative Semantic Reconstruction (CSR), that fixes BOOMERANG flaw by bridging the semantic gap. This work will be presented in Chapter 3.

1.0.2 Poor driver quality

During my exploratory phase of BOOMERANG, I had to go through the kernel sources of various smart devices. One thing I noticed is that the code quality of the kernel drivers was bad, there were a lot of critical security issues where data from user space was blindly

trusted (without validation). This observation is further validated by the growing trend of CVEs [7] in the corresponding kernels.

Most of the existing vulnerability detection tools for Linux kernel are specialized for individual classes of bugs like Integer overflows. The general-purpose tools have many false-positives to be usable. What we need is a general-purpose vulnerability detection tool that is *precise* and reasonably sound. I designed Dr. Checker [8] that is targeted towards finding taint-related vulnerabilities in Linux kernel drivers. Dr. Checker uses an extensible framework that encompasses flow-, context-, and field-sensitive pointer and taint analysis to track user data. Furthermore, Dr. Checker provides support for vulnerability detectors which implement taint verification policies that check if user data is used in sensitive locations.

The novelty of Dr. Checker lies in its ability to analyze only driver code by assuming all the kernel interface APIs as safe. Moreover, the driver code is analyzed in a *soundy* manner where the recursive constructs like loops and recursive calls are analyzed a fixed number of times. Specifically, Dr. Checker has the following core assumptions:

- **Assumption 1.** We assume that all of the code in the mainline Linux core is implemented *perfectly*, and we do not perform any inter-procedural analysis on any kernel API calls.

- **Assumption 2.** We only perform the number of traversals required for a reach-def analysis in loops, which could result in our points-to analysis being unsound.

- **Assumption 3.** Each call instruction will be traversed only once, even in the case of loops. This is to avoid creating additional contexts and limit false positives, which may result in our analysis being unsound.

I implemented Dr. Checker on top of the LLVM [9] framework and evaluated on

nine vendor kernels and found 158 security vulnerabilities with an overall precision of 78%. The details of which will be presented in Chapter 4.

1.0.3 The problem of patch propagation

While reporting vulnerabilities during the evaluation of Dr. Checker, I noticed that few vendors had fixed the vulnerability long back. A careful observation of the patch propagation mechanism used by the vendors revealed that almost all the vendors use a reactive mechanism based on CVEs.

Unfortunately, the CVE databases are known to be ineffective for timely propagation of security patches [10, 11, 12, 13]. In the year 2016, the Android maintainers patched 76 publicly known vulnerabilities (i.e., CVEs) from the year 2014, two from 2013, and two from 2012, which means that 80 disclosed vulnerabilities remained unpatched in the Android code base for more than one year [14].

What we need is a system that can automatically propagate security patches to vendor repositories. Imagine a command like `git rebase` $-security$, that automatically merges all the security patches from the main repository. One of the crucial components in achieving this is a method to automatically identify security patches or in general patches that do not affect the intended functionality of the software. The patches can be called *safe patches* as they do no affect the intended functionality of the software.

The existing systems to identify safe patches are based on commit messages [15, 16, 17] or pattern-based [18]. These systems have the advantage of being fast, lightweight, scalable, and suitable to be used on large code bases. However, either they only match simple patches, or they analyze commit messages, which are often not expressive enough to convey the scope and effect of a change [19, 20, 21]. Other techniques attempt to go a step further and analyze the semantic differences introduced by a patch using static

analysis [22, 23, 24, 25, 26] and symbolic execution [27, 28, 29, 30]. Unfortunately, these techniques suffer from scalability issues. Moreover, some of these approaches also require the exact build environment [31] of the whole code base, restricting their practicality and applicability to complex software, such as the Linux kernel, the VLC player, the OpenBSD OS, etc., as these software components have many possible configurations [32].

To be effective and usable on large code bases, a system to identify safe patches should at least satisfy the following requirements:

- **R1**: Only rely on the original and patched versions of the modified source code file, without any other additional information (e.g., commit message, build environment, etc.)
- **R2**: Be fast, lightweight, and scalable.

I developed SPIDER [33], a system to automatically find safe patches and satisfy the above requirements.

I start with first formally defining the necessary conditions for a patch to be considered safe. These conditions are then converted into first-order logic constraints. Second, using the *Program Dependency Graph (PDG)* of the patched function, we identify all the statements that are affected by the patch. Finally, these statements are converted into symbolic constraints to verify the satisfiability of safe patch conditions.

I evaluated SPIDER on 341,767 patches from 32 large and popular source code repositories as well as on 809 CVE patches. Results show that SPIDER was able to identify 67,408 safe patches and that most of the CVE patches are safe patches. In addition, SPIDER identified 2,278 patches that fix vulnerabilities lacking a CVE; 229 of these are still unpatched in different vendor kernels, which can be considered as potential unfixed vulnerabilities. The details of SPIDER will be presented in Chapter 5.

1.0.4 Achieving memory safety using Checked C

Although there exists mainstream memory-safe C alternative like Rust [34], it is depressing to see that new C code is being developed in the 21st century. One of the main reasons, apart from the natural learning curve of a new language, is the engineering hurdles involved in seamlessly interacting with legacy C code [35]. There exist techniques such as ASan [36], SoftBound+CETS [37, 38] that try to retrofit memory safety to legacy code by automatically adding runtime checks. These techniques suffer from high-performance overhead (both in runtime and memory) and lack backward compatibility as they change the runtime representation of pointers (fat pointers [39]). Furthermore, smart devices have resource constraints that pose a higher impedance to adoption.

Ideally, we need a backward-compatible safe C dialect:

- That should allow safe and unsafe code to co-exist. So that the developers can write the new code in the safe dialect, which could interact with unsafe legacy code.
- The safe dialect should not have a steep learning curve, i.e., should be *very* similar to C.
- There should be very low-performance overhead.

The Checked C [40] is an extension to the C programming language that satisfies all the above requirements. It extends C with type annotations using which it tries to prove spatial memory safety statically. It adds dynamic checks for the cases where safety cannot be proven statically. During the summer of 2019, I did an internship at the PLUM group of the University of Maryland, College Park, under the guidance of Dr. Hicks. I was involved in a project [41] to automatically convert C code to include type annotations supported by Checked C. We are working on a conversion technique that tries to integrate user input into a constraint solving mechanism. This is an on-going

work which will be presented in Chapter 6.

1.1 Permissions and Attributions

1. The content of Chapter 3 is the result of a collaboration with Eric Gustafson, Chad Spensky, Chris Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2017 edition of the Network and Distributed Systems Security Symposium.

2. The content of Chapter 4 is the result of a collaboration with Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna, and has previously appeared in the 2017 edition of the USENIX Security Symposium.

3. The content of Chapter 5 is the result of a collaboration with Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna and has previously appeared in the 2020 edition of the IEEE Symposium on Security and Privacy.

4. The content of Chapter 6 is the result of an on-going collaboration with Michael Hicks, a part of which has previously appeared in the 2019 edition IEEE Secure Development Conference.

Chapter 2

Background and Related Work

Uses of TrustZone. Different works have used TEEs to implement a variety of security systems. For instance, TZ-RKP [42], Sprobes [43], and SKEE [44] use TrustZone to verify and protect kernel integrity. The usage of TrustZone to enforce kernel integrity has also been implemented by Samsung in its Knox platform [45]. Other interesting applications of TrustZone are transparent memory acquisition [46], untrusted world memory introspection [47], and secure one-time password token generation [48].

TrustZone Vulnerabilities. Several researchers have tried to find vulnerabilities in Trusted Execution Environments running on ARM TrustZone. Dan *et al.* [49] presented an exploit for a vulnerability in Qualcomm Secure Execution Environment (QSEE) [50], which is the TEE by Qualcomm. Recently, Gal wrote a series of blog posts [51] explaining his reverse engineering efforts on identifying and exploiting vulnerabilities in QSEE. He further showed that achieving code execution in the context of a TA allows an attacker to compromise the untrusted OS easily. Other TEE implementations have been exploited as well ([52, 53, 54]), resulting in a complete compromise of the secure world. Similarly, Di presented [55] his efforts to exploit vulnerabilities in Trusted Core, which is Huawei TEE.

All the works mentioned above rely on various implementation mistakes, whereas BOOMERANG vulnerability stem from design issues caused by the semantic gap between worlds.

Recently, there has been growing interest in finding side-channel attacks on ARM TrustZone. Consequently, there are attacks based on the cache [56], electromagnetic analysis [57], and, power management [58].

Confused Deputy Attacks. The BOOMERANG attacks are ultimately a form of *confused deputy* problem, as it results from the inability of the TEE to make proper security decisions due to the semantic gap. Other works, such as [59, 60], focused on confused deputy problems in other areas of the system, such as between untrusted code components.

Static vulnerability detection on Kernel drivers. There are many special purpose vulnerability detection tools targeting kernel drivers. Johnson *et al.* [61] proposed a sound CQUAL-based [62] tool, which is context-sensitive, field-sensitive, and precise taint-based analysis that targets a specific class of taint based vulnerabilities, i.e., address-space vulnerabilities, however, this tool also requires user annotations of the source code. APISan [63] uses a symbolic-execution-based approach to find problems of the API-misuse problem in kernel drivers. Similarly, Static Driver Verifier (SDV) [64] identified API-misuse using static data-flow analysis. SymDrive [65] uses symbolic execution to verify the properties of kernel drivers. However, it requires developers to annotate their code and relies heavily on the bug finder to implement proper *checkers*. KINT [66] uses taint analysis to find integer errors in the kernel. While KINT is sound, their techniques are specialized to integer errors. DEADLINE [67] uses symbolic checking to identify double-fetch vulnerabilities in OS kernels.

Unlike all the existing techniques, DR. CHECKER is a general purpose taint analysis technique with a pluggable interface [68] to detect any taint-related vulnerabilities. DR. CHECKER achieves its precision by assuming that all the external code (i.e.,

code not part of the driver) to be safe or implemented correctly. Linux Driver Verification (LDV) [69] is a tool based on BLAST [70] that offers precise pointer analysis; however, it is still a model-checker-based tool [70, 71], whereas we built our analysis on well-known static analysis techniques.

Recently, Yamaguchi *et al.* have done a significant amount of work in the area taint-style vulnerability detection based on graph queries [72, 73, 74, 75], where they use static analysis to parse source code into novel data structures and find known vulnerable signatures. However, their tool is similar to a pattern-matching [76, 77, 78] or model-checking type approach, whereas we are performing general taint and points-to analysis with pluggable vulnerability detectors. VCCFinder [79] also used a similar pattern-matching approach, but automatically constructed their signatures by training on previously known vulnerabilities to create models that could be used to detect future bugs. MECA [80] is a static-analysis framework, capable of taint analysis, that will report violations based on user annotations in the source code, and similarly aims to reduce false positives by sacrificing soundness. ESP [81] is also capable of fully path-sensitive partial analysis using "property simulation," wherein they combine data-flow analysis with a property graph. However, this approach is not as robust as our more general approach.

Boyd-Wickizer *et al.* [82] proposed a potential defense against driver vulnerabilities that leverages x86 hardware features; however, these are unlikely to be easily ported to ARM-based mobile devices. *Nooks* [83] is a similar defense; however, this too has been neglected in both the mainline and mobile deployments thus far due to similar hardware constraints.

Finding unpatched code clones. Finding *unpatched code clones* is the focus of most of the prior research on patches in the security field [84, 13, 85]. SPIDER do not look for code clones but for instances where the function affected by a patch is still equal to the unpatched version. Brumley *et al.* [86], instead, show how to generate exploits for a

vulnerability starting from the corresponding patch.

Easing the patching process. Prior research has been very active in designing approaches and building tools to ease and speed up the process of patching [87, 88, 89], However, most of these techniques target only specific bug classes [90], while SPIDER define generic conditions for a patch to be considered safe independent of the bug classes. Other studies concentrate on helping developers in applying systematic changes [91, 92]. Long *et al.* [93], in contrast with the previously mentioned studies, use machine learning to model correct code and generate generic defects fixes, but do not focus on propagating existing patches targetted by SPIDER. Similar to what we do in this work, Kreutzer *et al.* [94] use AST differencing on changes to extract metrics to help cluster the changes by similarity.

Software evolution. Mining software repositories is a well-known technique to gain insights into the dynamics of software evolution [95, 96]. Perl *et al.* [79] built VCCFinder, a tool that leverages code metrics and patch features (e.g., keywords in commits) to identify vulnerability-contributing changes. However, SPIDER do not rely on the commit messages, and, instead perform a systematic analysis of the patches.

Chapter 3

The perils of absolute isolation

As mentioned in the Chapter 1.0.1, ARM architecture provides an *isolated* and *privileged* execution environment called ARM TrustZone. This is commonly referred to as the Trusted Execution Environment (TEE), as the code that executes in the TEE is trusted, i.e., digitally signed and verified through the chain-of-trust [97]. Although the absolute isolation provides a strong non-inference property, it introduces a problem with semantic-gap. Before understanding the details of that, lets first try to understand how a TEE, specifically ARM TrustZone works.

3.1 What is a Trusted Execution Environment (TEE)

A TEE is a separate execution environment for code and its associated data that requires a higher level of *trust* than the typical operating system. TEEs can be implemented as either a physically separated environment (i.e., dedicated CPU and memory) or on the same SoC as the normal CPU with specialized hardware-isolation mechanisms (e.g., ARM's TrustZone [4]). Because of this strict hardware isolation (e.g., separate registers, memory, and peripheral access), the two execution environments are typically

referred to as different *worlds*: the secure world (i.e., the world within the TEE) and the non-secure world. Because the software in the secure world is assumed to have a higher level of *trust* than the software executing in the non-secure world, we refer to all software in the secure world as *trusted* and the software in the non-secure as *untrusted*. Each world has its own OS, which we refer to as the untrusted and trusted OSes, and each OS runs its own respective accompanying applications, which we refer to as untrusted applications (UAs) and trusted applications (TAs). Similar to traditional execution environments, both the secure and non-secure worlds segregate the applications and their OSes using different execution privileges (i.e., user and supervisor mode).

In TEE implementations where the secure and non-secure worlds exist on the same SoC (e.g., TrustZone), the processor will always start in the secure world. The secure world software is then responsible for initializing the sandboxed, non-secure, world and switching the process state to the non-secure mode. From the non-secure world's perspective, the existence of the secure world is completely hidden, and the hardware architecture presents itself as if the system had just booted, without any evidence of the underlying secure world. However, by virtue of the architecture, the secure world always maintains complete control over and visibility into the non-secure world (similar to a hypervisor and its guests). Furthermore, the hardware enforces isolation between the two worlds through the use of specialized CPU registers and a non-secure (NS) bit. Specifically, the NS bit is used to restrict access to memory and all peripherals accessible on the Advanced eXtensible Interface (AXI) bus. The context switching between the two worlds is handled by a *Secure Monitor* that is instantiated when a secure monitor call (SMC), or a special exception, is issued by either a privileged (supervisor mode) application in the non-secure world or any secure world application. To share information, the worlds can pass a limited amount of information using either registers or memory regions, which can either be dictated by the secure world or passed by pointer reference.

The principal idea of the TEE is to minimize the trusted computing base (TCB), in that the code running in the TEE is intended to be a small, more easily verified subset of the overall system that is used for security-sensitive tasks. However, in practice, there is a strong desire to have the TEE offer rich functionality to the non-secure world (e.g., digital rights management (DRM) [98], Trusted Input [99], or authentication [100]). All of these applications require that a communication channel between the two worlds is established to share data over. This presents a major security risk to the TEE, as it must accept input from the *non-secure* world and its *untrusted* software. The existing implementations still depend on the non-secure world's OS to sanitize any inputs before passing them into the secure world, as sanitization in the secure world is hindered by the semantic gap.

Despite efforts to enforce strict standards (e.g., GlobalPlatform [101]) on TEE interactions, most of the software running inside these TEEs is typically custom-built, and the trusted and untrusted software are commonly developed by completely disjoint entities. For example, on Android devices, while Google is responsible for the untrusted OS, the secure world OS is commonly developed by other parties like Qualcomm[102], Trustonic [103], Nvidia [104], and the open-source community [105, 104].

The lack of well-defined, secure, standards and mechanisms for secure world applications to verify security properties of non-secure memory addresses results in scenarios wherein untrusted applications can convince trusted applications to read or modify the contents of *any* physical memory address within the non-secure world. This is the essence of Boomerang vulnerability.

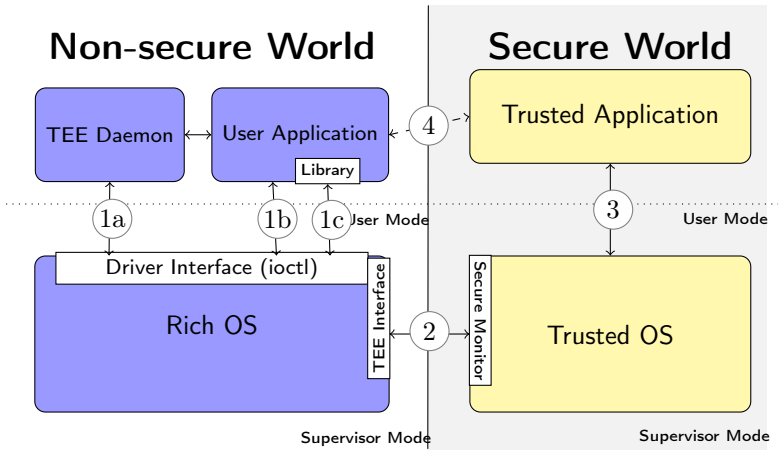

Figure 3.1: High-level interactions when a user-level untrusted application exchanges data with a trusted application in a TrustZone-enabled SoC.

3.2 The BOOMERANG Vulnerability

BOOMERANG exploits the *semantic gap* inherent to the design of all the current TEE implementations, where the secure world and its associated TAs have the ability to read and write to non-secure world memory. However, most TAs have a legitimate need to interact with the non-secure world's memory, and this functionality is routinely offered as a feature of the architecture. While the untrusted OS is able to protect itself and its applications within the non-secure world, all of these security mechanisms can be trivially bypassed from within the secure world. The trusted OS has no inherent ability to determine the provenance or security properties of any non-secure memory regions that are passed from untrusted applications, due to the separation provided by the TrustZone mechanism. More precisely, while the trusted OS can analyze secure world pointer values to protect itself and other secure-world applications, it has no insight into the memory permissions of the non-secure world. Thus, when a TA receives a non-secure world memory address as a parameter to a command, it has no choice but to blindly act on that memory.

The untrusted OS is the most obvious place to implement a defense, as it is already enforcing the non-secure world security mechanisms, and, in fact, all current implementations do employ some form of pointer sanitization (PTRSAN) functions when handling pointers. However, the trusted OSes and their applications frequently define their own structures for the exchange of commands and data, making it impractical for the untrusted OS to determine which values in the data are pointers and need to be sanitized. This semantic gap forces the untrusted OS to obliviously pass unknown data structures to the secure world and similarly forces the secure world to act on non-secure memory without any verification of whether or not the untrusted OS has authorized those actions. Thus, in these scenarios, an untrusted application is able to issue requests to the secure world for memory that it does not own, which the secure world will manipulate, permitting unauthorized reading and writing of another application's memory, including the untrusted OS's kernel. Even when such pointer sanitizations occur in the untrusted kernel, most of the PTRSAN functions are implemented incorrectly, making them easy to bypass, resulting in BOOMERANG vulnerabilities.

We demonstrate this interaction graphically at a high level in Figure 3.1 and briefly walk through a specific example in TrustZone; however, this general data flow holds for all TEE implementations. Note that there are three distinct security and semantic boundaries that must be properly handled: user mode to supervisor mode in the non-secure world (1), supervisor mode in the non-secure world to supervisor mode in the secure world (2), and supervisor mode to user mode in the secure world (3). Since the SMC instruction, which is used to change between the two worlds, is a protected call, the untrusted OS must either implement a long-running service that user applications can use as an arbiter to interact with the secure world (1a) or expose an API to applications and permit interaction with the TEE driver directly (1b) (most vendors provide a library in this case for convenience, shown as (1c)).

All TEE implementations rely on an agreed-upon standard between the untrusted OS and the trusted OS for passing information (2). However, as mentioned previously, there are various trusted OSes in circulation and there is no global standard, as of yet, that has been agreed upon by these trusted OS vendors. Thus, each trusted OS is accompanied by a specialized untrusted kernel driver for interacting with the secure world, each driver using its own unique calling convention. What is worse, while the protocol for exchanging information between the trusted OS and the trusted application (3) is well-defined, the structure of this information is not standardized. Therefore, most TA vendors are required to devise their own unique data structures for sharing data between the untrusted application and the trusted-world application (4). Note that while the untrusted OS can sanitize the memory address of the structure, it has no insight into its contents unless the untrusted application explicitly provides it. Similarly, TAs currently have no way of conferring with the untrusted OS to validate the authenticity of memory pointers and they have no choice but to assume that all pointers have been sanitized.

Because of the isolation between secure and non-secure worlds, the virtual memory addresses that applications use are incomparable as the worlds utilize separate page tables within the memory management unit (MMU). Thus, any reference to memory must be converted to a common entity before being shared with the other world. While it is possible for both worlds to simply use a common memory map, this has been shown to be a major security risk, as it allows the non-secure world to control the execution of secure world by using page faults [106]. Therefore, in practice, this commonly agreed-upon representation is typically either a physical memory address or a shared identifier (e.g., a virtual address in the secure world), which permits each world to access the particular memory region without any insight into the other world's memory mapping. We refer to this translation of memory addresses, and any associated security checks, as

PTRSAN and depict its various implementations in Figure 3.2.

By virtue of the implementation, any data being passed between the two worlds (2) must go through a PTRSAN function, which will convert pointers to this common entity. This PTRSAN step is typically implemented within a hardened application (1a) or within the kernel ((1b) and (1c)) for two reasons: *1)* the specific pointer translation procedure should be transparent to the user application, which increases the modularity of the code; and *2)* the PTRSAN function can perform the appropriate security checks to verify that the pointer indeed belongs to the corresponding application and is safe for applications in the secure world to access. PTRSAN is intended to protect both the untrusted kernel and other untrusted applications from a malicious application. However, amongst the data being handled by PTRSAN, there is TA-specific data, which the PTRSAN application has no insight into. Any pointers within these TA-specific data structures must be explicitly annotated so that the PTRSAN can translate them appropriately. Herein lies the problem, and the core flaw being exploited by BOOMERANG. Specifically, the PTRSAN function has no insight into the protocol agreed upon between user application and the trusted application (4), and thus it is possible for the user application to pass pointers directly, which evade the PTRSAN security checks. This critical semantic gap is fundamentally what makes it so difficult to prevent BOOMERANG attacks in practice.

To demonstrate how memory addresses can evade sanitization in practice, we will briefly walk through an example from Figure 3.2. Note that (4) is the boundary that the data must ultimately cross; however, the architecture does not permit this particular interaction directly. So, the application prepares a data packet destined for the TA in memory, using a data structure that was specified by corresponding TA. When the user application needs to share a large amount of variable length data with the trusted application (e.g., encrypted content), it is desirable to permit the TA to act on this data in place (versus copying it into a separate memory region). The pointer to the data

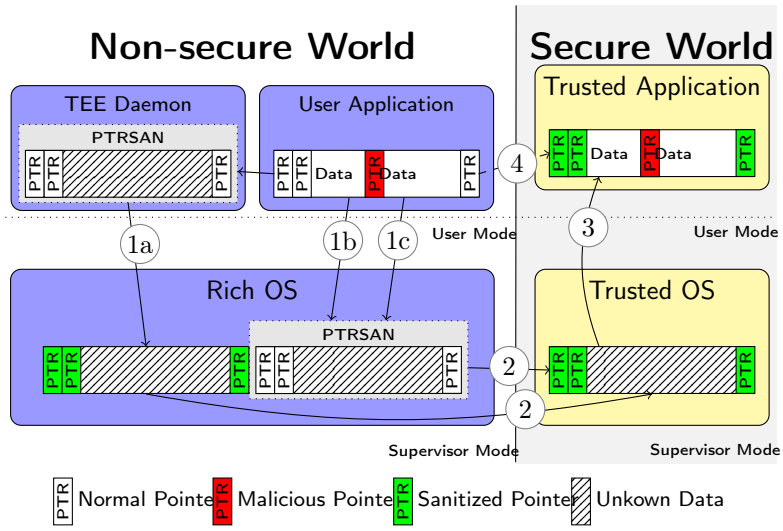

Figure 3.2: An example of BOOMERANG, where a malicious memory pointer is hidden from pointer sanitization, ultimately tricking a TA to act on that memory address.

to be manipulated is annotated using the specific API for the TEE, and the PTRSAN function handles the pointer appropriately. However, in most cases, this annotation can be trivially omitted, permitting the user application to control the pointer value that the trusted application will receive. For example, when physical memory is used as the common entity between the two worlds, the user application could pass a physical address in the TA-specific data structure without reporting this information to PTRSAN (i.e., a *malicious pointer*). The TA has no way of validating these pointers, due to the semantic gap, and thus has no choice but to perform the requested action resulting in a BOOMERANG vulnerability.

To the best of our knowledge, BOOMERANG was previously completely unknown. In fact, the most related security issue that was mentioned in the documentation [101] was a time-of-check vs. time-of-use bug that exists in TEEs, wherein the contents of non-secure memory may be changed while the TEE is operating on the buffer. This limitation could lead to situations where the data could be changed in malicious ways to exhibit unintended behavior or permit untrusted world applications to access each other's data if

the shared memory region is globally readable. As we show in Chapter 3.6.3, our proposed defense, CSR, can be trivially augmented to address this security concern as well.

It is worth noting that there is already a mechanism in place for querying the non-secure world from TAs. In an effort to decrease the TCB within the secure world, any high-level operations (e.g., file operations, networking) that the secure world needs to exercise are typically handled by the non-secure world on behalf of the secure world. In practice, each trusted OS is accompanied by a user space service (i.e., a TEE daemon) that is capable of handling these requests. In some cases, this same daemon is also utilized as the arbiter between untrusted applications and the untrusted kernel driver (1c in Figure 3.1). We show in Chapter 3.6.3, how we were able to leverage this mechanism (i.e., the trusted world requesting information from the untrusted world) to reconstruct the non-secure world semantics and prevent BOOMERANG.

3.3 Assumptions and Attacker Capabilities

To understand how the BOOMERANG vulnerability can be exploited in practice by an attacker, let us first understand the environment and capabilities of the attacker.

We consider Smartphones with ARM TrustZone running Android as the untrusted OS. We assume that an attacker can convince a user to install an app on her phone. We also assume that this app has the ability to interact, using proper system calls, with TEE applications. Depending on the implementation, this requires either no permissions or a single permission to interact with a specific TEE application (e.g., the `ACCESS_DRM` permission to access the DRM application in the trusted world). No `root` or `system` permissions are required for the attacker application in the untrusted world.

The attacker goal here is to raise the privileges of the installed app to *root*. The was attacker achieves this is to convince the code running within the TEE to read and write

non-secure world memory at the attacker's will. Thus thwarting the security mechanisms of the untrusted OS, and, raising the privileges of the app to *root*.

3.4 BOOMERANG on Real World Devices

While BOOMERANG, in general, is applicable across all TEE implementations, it is useful to examine various flavors that appear in real-world implementations. To this end, we have examined the most popular TEE implementations to verify the existence of BOOMERANG. In this chapter, we describe the architecture of each of the examined implementations, highlighting how their specific design choices affect their susceptibility to BOOMERANG.

3.4.1 Qualcomm Secure Execution Environment (QSEE)

Recent studies indicate that around 60% of all Android phones in production are running Qualcomm's QSEE [107], making it an exceptionally high-impact implementation, as any vulnerabilities could potentially lead to a complete compromise of these devices [108].

Untrusted Application and Untrusted OS

QSEE exposes a kernel driver `/dev/qseecom` to untrusted applications (①b and ①c in Figure 3.2). Interactions with this device are carried out using the `ioctl` system call with various commands, which untrusted applications can use to interact with the secure world. Qualcomm also provides a user-space library `libQSEEComAPI.so`, which conveniently exposes the different `ioctl` commands as functions. Data is exchanged between untrusted and trusted applications using a specialized data structure (Figure 3.3). This data structure is then passed through a PTRSAN function to resolve any pointers

to non-secure world memory regions. In QSEE, physical memory addresses are used as the common entity between worlds, and the pointer translation from virtual to physical occurs directly in the provided kernel driver (1b and 1c in Figure 3.2). Sending commands to a TA happens in multiple steps, which are described hereinafter.

First, the untrusted application requests the allocation of a shared memory region using a separate shared memory driver `/dev/ion`[109]. This region will be used for both requests and responses. The shared memory driver returns a shared memory identifier (i.e., `shmid`), an opaque identifier that is used to refer to this memory region, independent of its location. This identifier can then be used to map (i.e., using `mmap`) the allocated memory into the untrusted application's memory space. The shared memory region is then split into two buffers, one for sending data into the trusted world (i.e., `send_buf`) and one for the response (i.e., `resp_buf`).

Second, the application prepares the command to be executed, and stores it in `send_buf` (see Figure 3.3). Pointers stored directly in the driver interface structure will *always* be validated and translated by the pointer translation function. However, the untrusted application can also pass pointers within the body of the request itself that was previously allocated using `/dev/ion` (i.e., within the `send_buf` data). Since the request body is application-specific, these pointers cannot automatically be located or translated. To enable this, the application can supply a replacement vector (i.e., `QSEECom_io_fd_info`), which is a list of offsets in `send_buf` that should contain the pointers together with the corresponding `shmid`s that should be translated and placed there. The final command sent will contain the physical addresses for each shared memory region in the desired locations.

Third, the application either performs an `ioctl` directly on the `/dev/qseecom` device with the `QSEECOM_IOCTL_SEND_MODFD_CMD_REQ` command, or uses the

`QSEECom_send_modified_cmd` command provided by the `libQSEEComAPI.so` library

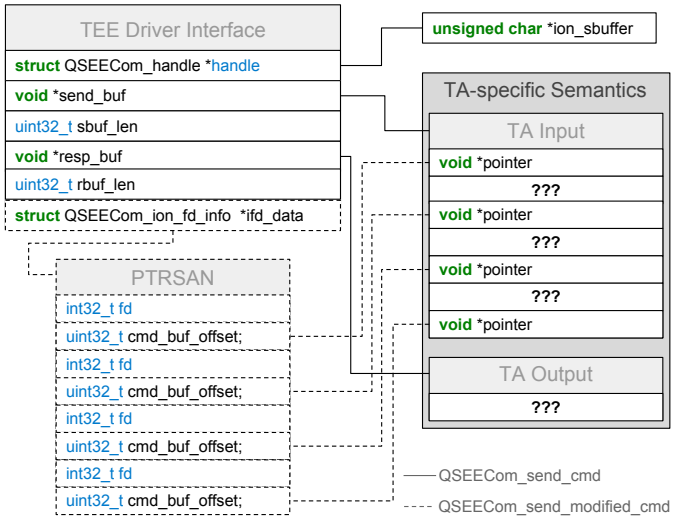

Figure 3.3: The data structure used to communicate with the TEE in QSEE [110].

to trigger the execution of the command. This causes `QSEECOM` to copy the request buffer into a temporary buffer, and optionally perform pointer translation.

Untrusted OS and Trusted OS

The untrusted OS and trusted OS interact using Qualcomm's secure channel manger (SCM), which defines a set of functions that prepare and execute SMC calls with the provided data. All SMC calls are made with four parameters (i.e., send_buf, sbuf_len, resp_buf, rbuf_len), where `send_buf` and `resp_buf` are the buffers passed by the application. All of these parameters are packed into an `scm_command` structure, and the physical address of the packed structure is passed as an argument [111].

Trusted OS and Trusted Application

TAs are executed as user mode applications within the trusted world, with no access to any other secure world memory (e.g., other TAs or the trusted OS). Consequently, TAs must issue system calls to the trusted OS kernel for any privileged tasks that they need to

perform. For example, to access non-secure memory (i.e., a physical address passed from the untrusted world), they must utilize the `qsee_register_shared_buffer()` syscall. In this call, the trusted OS validates that the request memory region is not inside the secure world (e.g., within the trusted OS), to protect itself from the untrusted world. If the physical memory address is indeed within the non-secure world's memory, the kernel will map the requested physical memory region into the TA's memory space. Note that `qsee_register_shared_buffer()` only verifies that the memory is not in the secure world; it cannot verify that this physical address indeed belongs to the untrusted world application that initiated this request [108].

Boomerang on QSEE

As discussed above, the untrusted application makes use of the `QSEECom_send_modified_buffer` function, which updates the `send_buf` with physical addresses before sending it to the TA using the provided replacement vector (i.e., `QSEECom_io_fd_info`). However, this puts the onus on the untrusted application to supply the necessary information for the appropriate pointer translation to occur. A malicious application that wishes to pass arbitrary physical memory addresses could simply insert them into `send_buf` in the proper locations for the victim TA, and exclude them from the replacement vector. Alternatively, the malicious application could simply utilize the un-sanitized `QSEECom_send_cmd` command, which will send commands to the TA without any pointer translation (see Figure 3.3). The trusted OS only checks to confirm that these physical pointers are not mapped into the secure world. Thus, any malicious physical address placed within the `send_buf` buffer, and kept hidden from PTRSAN, will be blindly acted upon by the TA (e.g., decrypted, copied, encoded), resulting in a Boomerang vulnerability. We show in Chapter 3.5.2 how we were able to leverage actual Boomerang vulnerabilities to craft an arbitrary physical memory read exploit.

While both `QSEECom_send_modified_cmd` and `QSEECom_send_cmd` are provided in the `libQSEECom.so` library, where additional checks could be implemented, it would still be possible to perform the same un-sanitized operations on the kernel driver directly. Therefore, any fool-proof solution will require at least some coordination with the secure world to ensure that it cannot be easily bypassed, such as the ones we examine in Chapter 3.6.

3.4.2 Trustonic

Trustonic [103] is another very popular vendor of TrustZone-based TEE technology. Their TEE implementation is widely deployed across consumer hardware (over 400 million devices [112]), with Samsung leveraging it as part of its Knox [113] platform. Trustonic encrypts and signs all of their trusted applications and their trusted OS kernel, which makes it more challenging to audit their functionality, although recent efforts have made headway in recovering the decrypted code [114].

Untrusted Application and Untrusted OS

Trustonic employs a kernel driver `/dev/mobicore`, similar to QSEE, and a service `mcDaemon`, which user applications *must* use to communicate with the secure world. Due to its permissions, unprivileged user applications cannot communicate with the driver `/dev/mobicore` directly, as was possible in the case of QSEE. In Trustonic's implementation, communication with the secure world must go through the `mcDaemon` service using a write-and-notify mechanism known as world-shared memory (WSM). This communication is initiated when an untrusted application registers a buffer, called a *session buffer*, with a TA to open a new *session*. Commands intended for the TA are then sent by writing data into the session buffer, and issuing a notify command through

`mcDaemon`. Once the data is passed to the secure world, the trusted OS will then notify the TA that the contents are ready. Similarly, to receive responses, untrusted applications wait for a notification from the TA (through `mcDaemon`).

In the Trustonic implementation, opaque identifiers are used instead of memory locations (i.e., physical memory pointers). By examining the source of `mcDaemon` [115], we confirmed that the opaque id is actually a virtual address that has been mapped into the memory space of the TA, within the secure world. If an untrusted application wants to share some memory with a TA, it must register the buffer using the `processMapBulkBuf` function in the `mcDaemon` service, which maps the corresponding physical memory region into the TA's memory space and returns an opaque identifier back to the untrusted application. `processMapBulkBuf` also verifies that the pointer being converted is indeed owned by the requesting application, which thwarts the trivial instance of BOOMERANG. From this point on, the only method for the untrusted application to interact with that shared memory region is using this opaque identifier and the `mcDaemon` service (i.e., the untrusted application has no direct control over the pointers that the TA will receive and operate on).

Untrusted OS and Trusted OS

The interaction between the untrusted OS and the trusted OS is performed using the standard SMC TrustZone instruction. Unlike QSEE, where the physical address of a packed structure is passed to the trusted OS, Trustonic's implementation explicitly passes parameters using values stored in registers (current implementations only support up to four unique parameters [116]).

```
void processMapBulkBuf(Connection *connection) {
...
 // Trustonic's PTRSAN function
 uint64_t pAddrL2 = device->findWsmL2(cmd.handle,
     connection->socketDescriptor);
...
 // Map bulk memory to secure world
 // BOOMERANG if the attacker can control pAddrL2
 mcResult_t mcResult = device->mapBulk(connection,
     cmd.sessionId, cmd.handle, pAddrL2,
     cmd.offsetPayload, cmd.lenBulkMem,
     &secureVirtualAdr);
...
 if (mcResult != MC_DRV_OK) {
  writeResult(connection, mcResult);
  return;
 }
 mcDrvRspMapBulkMem_t rsp;
 rsp.header.responseId = MC_DRV_OK;
 rsp.payload.sessionId = cmd.sessionId;
 rsp.payload.secureVirtualAdr = secureVirtualAdr;
 connection->writeData(&rsp,
     sizeof(mcDrvRspMapBulkMem_t));
}
```

Figure 3.4: Code snippet from Trustonic's MobiCore daemon that exhibits a potential BOOMERANG flaw [117].

Trusted OS and Trusted Application

Given that the secure world binaries are encrypted, we were not able to completely reverse-engineer the interaction between TAs and the trusted OS. However, based on our experience with other implementations, we assume that it follows a similar structure, where TAs in the trusted world run as normal user-space applications, with no access to the trusted OS's memory. Similarly, all privileged tasks from TAs are likely handled by system calls to the trusted OS. We hypothesize that they also implement some checks on the pointers (i.e., opaque ids, virtual addresses) passed by the untrusted applications to validate that they indeed belong to the non-secure world, but currently we have no way to confirm this.

BOOMERANG on Trustonic

Although there is no explicit PTRSAN mechanism in Trustonic's implementation, the use of opaque identifiers by `mcDaemon` for shared memory inherently ensures that an

untrusted user application does not have control over the resulting pointers. Figure 3.4 shows the exact code that is enforcing this within the `mcDaemon` service. Note that this construction inherently makes the assumption that all shared memory requests come from `mcDaemon`, and that this daemon is not compromised. However, if an attacker were able to gain access to `/dev/mobicore`, or compromise `mcDaemon`, `pAddrL2` (in Listing 3.4) could be replaced with an arbitrary non-secure world physical memory (just as in QSEE) resulting in a BOOMERANG vulnerability. We have confirmed this issue with Trustonic, and are working with them toward an improved design for future releases.

3.4.3 Open Source Trusted Execution Environment (OP-TEE)

OP-TEE [118] is an open source TEE implementation, which can run on a selection of hardware development platforms. OP-TEE adheres to the GlobalPlatform [119] specification and provides libraries that ease the development of TAs. While OP-TEE has not yet been deployed on consumer hardware, it was valuable for our research, as it provided us with an implementation into which we had complete visibility and a platform for evaluating our defenses.

Untrusted Application and Untrusted OS

Similar to other implementations, the untrusted OS exposes a driver `/dev/tee0` [122], which can be used by applications to interact with the TAs. A client library `libteec.so` [123] is also provided to make it easier for applications to communicate with this driver. All parameters that are passed to the TA are strongly typed. There are two broad types: a *pointer* type and a *value* type (either of which can be input to a TA, output from a TA, or both). Every call to the secure world can only support up to four parameters, which must conform to the strict typing.

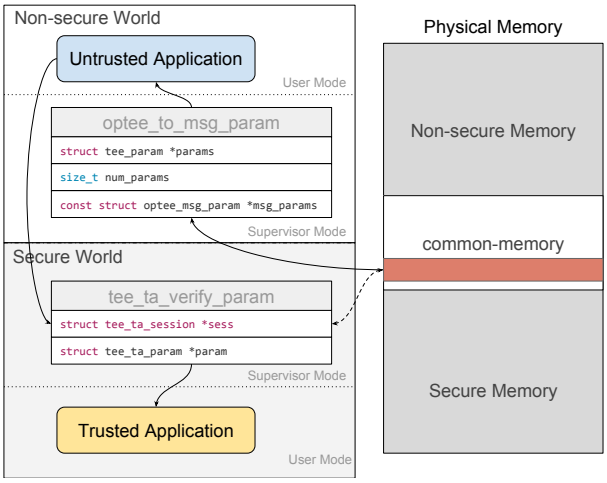

Figure 3.5: Outline of interactions with the TEE in OP-TEE's implementation using `common-memory`.[120, 121]

Untrusted applications again use opaque pointers (i.e., `shmid`s) to refer to memory that is intended to be shared with a TA. To pass a pointer argument, the untrusted application communicates with `/dev/tee0` to request memory of a specific length. The kernel driver then allocates this memory in a dedicated shared memory region (i.e., `common-memory`), pairs it with a `shmid`, and returns it to the client. Untrusted applications can use this `shmid` to map the memory into their address space, where they can then write commands to and read responses from the TA. This shared memory region is accessible by both the non-secure and secure worlds. However, because it is a dedicated memory region, it greatly reduces the risk of BOOMERANG vulnerabilities.

Untrusted OS and Trusted OS

Upon receiving a command from the untrusted application, the untrusted OS will first perform the required pointer translations (i.e., PTRSAN). Next, it packs all of the parameters into an `optee_msg_arg` structure and copies it into a free region in `common-memory`. Lastly, it performs a world-switch using the `SMC` instruction [124], with the physical address of this region as its argument.

Trusted OS and Trusted Application

TAs in OP-TEE run as unprivileged applications within the secure world, each running in its own thread, which are only spawned when a request is issued from the non-secure world. All privileged operations must, again, be performed through system calls into the trusted OS (i.e., supervisor call (SVC) instructions). For each memory parameter passed to a TA from the non-secure world, the physical address is first checked to ensure that is contained within the `common-memory` region, and that this memory region is mapped to the thread that is handling the request. More precisely, the trusted OS will take the physical address that was passed as a parameter and update it with a corresponding virtual address within the memory space of the handling thread (i.e., TA). Thus when the TA accesses any pointer arguments, it can access them as normal pointers (i.e., without any additional verification calls). However, the TA must strictly ensure that the types of all of the arguments are as expected, or else type-confusion attacks could be utilized to exploit the TA or trusted kernel. For example, if a memory pointer could be disguised as a value, bypassing PTRSAN, memory regions outside of the shared memory region could be passed to a TA, which would result in a BOOMERANG vulnerability. This process is shown in Figure 3.5.

BOOMERANG on OP-TEE

Although the use of `common-memory` prevents all TAs from accessing the untrusted OS's memory, the shared memory ids (`shmid`s) assigned to the different untrusted applications are stored in a global structure. This allows a malicious untrusted application to read and write the corresponding `common-memory` assigned to another untrusted application resulting in

BOOMERANG vulnerabilities [125, 126]. As described above, `common-memory` provides a

shared memory communication channel between untrusted applications and TAs, and, depending on the TA, this memory region can contain sensitive information (e.g., DRM decrypted content, passwords, or cryptographic keying material). Moreover, we also found a heap overflow [127] and an out-of-bounds read [128] in the PTRSAN function of the untrusted kernel driver. The OP-TEE developers responded promptly, fixing all of these issues; however, these various bugs demonstrate just how difficult a shared memory management implementation can be to deploy in practice. While shared memory regions can be used to defend against general BOOMERANG vulnerabilities, they present a significant degree of complexity and subtlety that must be overcome. There are also other technical limitations introduced with this approach (e.g., performance, limited parameters), which we discuss in detail in Chapter 5.6.

3.4.4 Huawei

We analyzed the TrustZone implementation from Huawei, with tens of millions of devices in circulation.

Untrusted Application and Untrusted OS

This TEE implementation, like Trustonic, employs a kernel driver `/dev/tc_ns_client` and a service `teecd`, which all user-space applications must use to communicate with the secure world. The permissions are similarly set such that untrusted user applications cannot communicate with the driver directly. Similar to OP-TEE, all parameters in the secure-world interface are one of two broad types: pointers and values, and all calls to secure world support up to four parameters, which can take either of those types.

However, in this instance, untrusted applications *can* directly pass an address with an `offset` as a pointer argument in their commands. The kernel driver attempts to perform

PTRSAN by first checking that the corresponding address is indeed in the requesting application's memory before replacing the address with the corresponding physical address, incremented by the provided `offset`.

Untrusted OS and Trusted OS

The interactions between the untrusted world OS and the trusted world OS are, again, done using the standard `SMC` instruction. All parameters to be passed are packed into a common structure (`TC_NS_SMC_CMD`), and the physical address of this structure is passed as the argument to the SMC call (similar to QSEE).

Trusted OS and Trusted Application

As with other trusted world implementations, each TA runs in an isolated process and interacts with the trusted OS through system calls (using SVC instructions). However, in this instance, *the entire non-secure world memory space is mapped into every TA*, which makes exploiting BOOMERANG vulnerabilities trivial.

BOOMERANG on Huawei

BOOMERANG exists on this implementation for a few reasons. First, PTRSAN fails to validate the `offset` value; a malicious untrusted application can use this to pass an arbitrary physical address to the TA. Second, almost all the TAs we examined do not validate the types of parameters, allowing one to bypass PTRSAN entirely, by misrepresenting the type of an argument to the kernel driver as a non-pointer, while still being correctly interpreted as a pointer by the TA. Type-confusion attacks within the TA are cumbersome to avoid, as each function that handles the parameter must independently verify that the type of the argument is correct, since the parent function has no insight into the ultimate use of each parameter. We found both instances of BOOMERANG (i.e.,

PTRSAN bypass and type-confusion) in different components within this implementation, as we show in Chapter 3.5.1.

3.4.5 Sierraware Trusted Execution Environment (SierraTEE)

SierraTEE is a Trusted Execution Environment developed by Sierraware [129]. They published an open source version of their implementation under the Open Virtualization project [130]. Similar to OP-TEE, this adheres to the GlobalPlatform specification [119] and provides libraries to support development. Although SierraTEE is used in academic projects [131], we were unable to determine whether it is used in any commercial device.

Untrusted Application and Untrusted OS

Similar to OP-TEE, SierraTEE employs a kernel driver `/dev/otz_client` and a client library `libotzapi.so` for ease of development. Applications can either use the driver or library to interact with the TAs. Similar to OP-TEE, all parameters to the TA are strongly typed, with three possible types: pointer, 32-bit value, or array. To pass a pointer, untrusted applications should first use `mmap` on the driver to allocate memory of the required size. The kernel driver then allocates the memory and associates it with the requested address (i.e., `usr_addr`), which can be used by the corresponding application as a shared memory id (`shmid`). Similar to Huawei, a pointer argument is passed as a tuple of (`shmid`, `length`, `offset`).

Untrusted OS and Trusted OS

First, PTRSAN is performed on all the pointer arguments by computing the physical address corresponding to the provided `shmid`. The resulting physical address and its corresponding `length` are packed as the new pointer argument. Next, all the arguments

are packed into an `otz_smc_cmd` structure, and the physical address of this structure is passed as the argument to the `SMC` instruction, and therefore to the trusted OS.

Trusted OS and Trusted Application

Similar to OP-TEE, each TA runs as an unprivileged application within the secure world, in its own thread. Privileged operations must be performed through system calls (SVC instructions), and are handled by the trusted OS. All parameters from the untrusted OS and applications are directly passed to the destination TA. As mentioned above, these take the form of physical memory addresses and region lengths, which must be mapped by the TA prior to use.

Boomerang on SierraTEE

Similar to Huawei, PTRSAN in SierraTEE fails to validate the `offset` for pointer arguments. This allows a malicious untrusted application to pass an arbitrary physical address to the TA leading to a Boomerang vulnerability. Furthermore, we noticed that PTRSAN also fails to verify the `length` parameter, which increases the exploitability of this flaw.

We notified Sierraware of our findings on multiple occasions, beginning in October 2016, and received no reply. We suggest that the users of the open source version of the SierraTEE be aware of this issue, and contact Sierraware to obtain an appropriate fix.

3.4.6 Observed Instances of Boomerang

In summary, we have observed two distinct instances of Boomerang in practice: PTRSAN bypass attacks, where the pointer sanitization function can be bypassed altogether, and type-confusion attacks, where TAs can be tricked into treating a non-pointer

Table 3.1: Summary of the various manifestations of BOOMERANG across the various TEE implementations.

Vendor	Common Entity		
	Physical Address	Shared Memory	Unique Identifier
QSEE	$\mathbb{B}_{Ptr}$		
Trustonic			$\mathbb{B}^*_{Ptr}$
OP-TEE		b_{Ptr}	
Huawei	$\mathbb{B}_{Ptr}, \mathbb{B}_{Type}$		
SierraTEE	$\mathbb{B}_{Ptr}$		

$\mathbb{B}$ - Full BOOMERANG (arbitrary non-secure memory access)

$\mathbb{B}^*$ - Full BOOMERANG, but requires an additional exploit

b - Partial BOOMERANG (access to specific regions of non-secure memory)

$_{Ptr}$ - PTRSAN bypass vector present $_{Type}$ - Type-confusion vector present

value as a pointer. This general flaw (i.e., the secure world's ability to freely influence non-secure memory) exists on each system, regardless of the common entity used for passing memory references between worlds. Table 3.1 demonstrates how the various bugs affect the vendors that we examined. It is worth noting that every analyzed TEE implementation is affected by BOOMERANG to some degree. The table only outlines the bugs that we personally were able to verify; however, we have reasons to believe Trustonic also likely contains a pointer-confusion attack, but we are unable to verify this hypothesis without access to the un-encrypted TAs.

3.5 Finding BOOMERANG Vulnerabilities

To evaluate the severity of BOOMERANG, we explored two very popular commercially available TEE implementations (i.e., QSEE and Huawei) to see if exploitable BOOMERANG flaws existed in deployed TAs. We were unable to perform our analysis

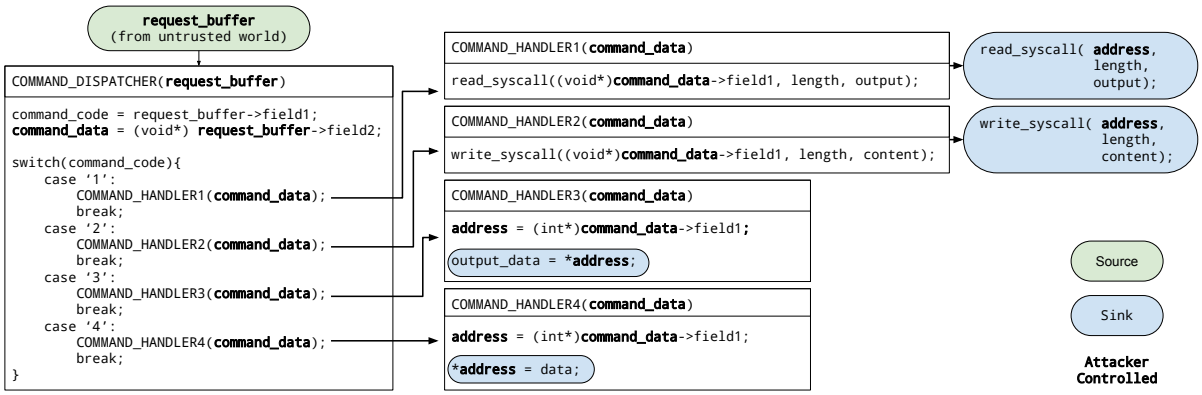

Figure 3.6: Examples of the different types of data flows that our tool would detect as being vulnerable to BOOMERANG.

on Trustonic's implementation because all of their TAs are encrypted. Similarly, we did not evaluate any TAs developed for the OP-TEE and SierraTEE architectures, as they have not been deployed on any commercial devices. We, indeed, found the BOOMERANG vulnerabilities in all of the evaluated TAs that accepted pointers from the non-secure world, some of which we used to craft exploits.

3.5.1 Detecting Potential Vulnerabilities

As we showed in Chapter 3.4, all of the TrustZone implementations that we analyzed will, at some point, pass commands from the untrusted application to the TA through the untrusted OS and the trusted OS. This data usually contains an application-dependent structure, and, in malicious instances, its contents may contain un-sanitized memory pointers. Thus, the general approach to our detection technique is to perform data-flow analysis to track all of the data that is passed from the non-secure world, and annotate any functions that use any portion of this data as a pointer. By capturing any function that dereferences non-secure data as a pointer, an analyst could then trivially use manual analysis to see if that data can be controlled by an untrusted application in a way that

bypasses PTRSAN, which would result in a BOOMERANG vulnerability.

We created a static analysis technique to locate these instances using simulated execution, which we implemented using the *angr* [132] static analysis and reverse-engineering framework. Our analysis works in the following way: First, we analyze the control flow graph and perform function recovery on a given TA, which identifies function entry points based on standard Advanced RISC Machine (ARM) calling conventions. This step requires that the binary is not obfuscated (e.g., encrypted or packed). Next, we locate the source of any input data, by locating the primary *command dispatcher* of the TA. This function is TEE-specific, but can be found easily through reverse engineering (e.g., identifying entry points in the program or using symbols) and is applicable to every TA for that TEE implementation. In QSEE's implementation, we referenced prior work to locate the command dispatcher [108], which accepts 4 arguments, consisting of the input and output buffers and their sizes (i.e., `send_buf`, `send_len`, `resp_buf`, `resp_len`). On Huawei, we were able to locate the symbol referring to the command dispatcher, which takes a list of inputs, and a list of the associated data types for each argument.

Once the command dispatcher function is located, we then perform data-flow analysis (similar to static taint tracking) on the data in the input buffers to detect any instances where any part of the input is used as a pointer. This pointer dereferencing may be done explicitly in the code itself, but could also be delegated to system calls within the trusted OS. Since the semantics of system calls are TEE-specific, we require that an analyst annotates those calls that handle the reading or writing to non-secure memory for each TEE (e.g., cryptographic operations or secure file-system operations). With the given system calls identified, our data-flow analysis can detect and return relevant paths in the TA.

Our analysis starts with the input buffers or argument lists as a *source* and performs a *blanket execution* [133] of the program, where all of the basic blocks in the control

flow graph (CFG) are executed, until the data from the source reaches a *sink* (i.e., an annotated system call or memory operation). TAs usually contains many possible commands, selectable by a TA-specific command identifier included as part of the request, which is typically checked by the TA at the beginning of execution. We can therefore locate a unique "handler" for the different commands (i.e., cases in the main switch-case statement of the command dispatcher), by analyzing all of the call sites in the command dispatcher function. This information is useful when determining exploitability, as it helps to identify the major functionality of the TA that is exercised with the identified vulnerability. Our tool will produce as output the call chain from the input to the memory operation or system call, and whether the final operation is a read or a write. Figure 3.6 provides a high-level overview of our technique.

3.5.2 Vulnerabilities in QSEE

While hundreds of millions of devices use QSEE as their TEE implementation, only a few TAs are actually widely distributed for the platform. We were able to obtain the binaries for KeyMaster, WideVine, and PlayReady, which to the best of knowledge are the only 3 QSEE TAs that accept user input. KeyMaster is the standard cryptographic application that is included on all Android-based devices with a TEE. WideVine is a Google-owned DRM technology, used most prominently in the Netflix and YouTube applications. PlayReady is a similar DRM technology provided by Microsoft, which provides DRM support for Windows Media files, amongst others.

After running our static analysis technique on the three TAs described above, we found that *all* of them were vulnerable to BOOMERANG attacks. KeyMaster contained three separate call-chains that permit an untrusted application to read arbitrary physical memory from within the non-secure world, using functionality within the TA. Similarly,

WideVine and PlayReady both contained call-chains that permit an unprivileged application to decrypt data to arbitrary physical memory within the non-secure world, which could be leveraged for an arbitrary physical memory write.

Proof-of-Concept (Memory Read)

We were able to easily leverage one of the three call chains located in QSEE's KeyMaster to craft a proof-of-concept arbitrary memory leak exploit. Figure 3.7 shows a graphical representation of the discovered path, including the addresses of each function call instruction between the input and a controllable memory operation, as well as the type of memory operation (e.g., "read," "write," or in this case, "syscall"). The tool also indicates the vulnerable "handler," which is the start address of the first unique function seen among the set of all the call chains.

In this case, the call chain terminates in QSEE system call number `0x06`, which was identified as the system call that prepares for memory read operations from the non-secure world. Using manual analysis, we were easily able to determine the purpose of the handler function on our chosen path, at `0x5ac`, which generates cryptographic signatures of data from the non-secure world. While the returned value is signed, the attacker can select the key, cipher, data, and data length. To recover the original non-secure world data, the signature is performed on a single byte, with a known key, and the result checked against a pre-computed table of signatures for all of 256 possible values of a byte with that key. To control the data that is to be signed, we can bypass PTRSAN in the non-secure world using `QSEECom_send_cmd` (as shown in Figure 3.3). The resulting exploit allows a malicious untrusted application, in the non-secure world, to read any amount of memory from an arbitrary location in the non-secure world, including memory of all other applications and the kernel.

We disclosed this vulnerability, and proof-of-concept, to Qualcomm and Google in

June 2016, and received the designation CVE-2016-5349. We are actively working with both companies on a fix and, as of December 2016, this critical patch is still in development. Our tool also identified memory write functionality in the WideVine TA, which could, in theory, be leveraged into a full exploit; however, we did not invest the engineering time at this point to verify this exploit.

3.5.3 Vulnerabilities in *Huawei*

For our analysis of Huawei, we were able to obtain a set of 10 TAs. Using our static-analysis tool, we found out that only 6 of them accepted commands and all of these 6 TAs were vulnerable to BOOMERANG. We were able to locate both arbitrary read and write functionality, which allows us to gain root privileges on any device running this TEE implementation. We use a technique based on ret2dir [134], which allows the execution of code as the root user, by overwriting kernel memory structures to include a malicious return-oriented programming (ROP) payload. This technique has been implemented and tested on Android 5.0.1, and works regardless of Privileged eXecute Never (PXN) protections deployed by the hardware.

These vulnerabilities were reported to Huawei, as part of our submission to the GeekPwn 2016 hacking contest [135], and received the designations CVE-2016-8762, CVE-2016-8763, and CVE-2016-8764. We were able to develop a full exploit, which leveraged BOOMERANG and other techniques to obtain full root privileges, as well as code execution within the TEE itself[1]. Huawei has implemented a fix, and as of December 2016, updates to various Huawei devices are available to address the problem.

[1] A video demonstrating the exploit can be found at `https://www.youtube.com/watch?v=XjbGTZrg9DA`

3.6 Defenses

Before discussing the examined defenses, we first outline the requirements that we set forth to ensure that our proposed defense would be both practical to implement and usable for developers. We identify the following minimum requirements that any solution to BOOMERANG must satisfy to be usable:

- **Independence from the untrusted OS:** The TEE implementation should not be dependent on the untrusted OS (i.e., it should not leverage OS-specific functionality). For example, the trusted OS should be unaffected if the untrusted OS is upgraded or changed entirely. This requirement forces the solution to be generic, rather than depending on a particular feature within the untrusted OS implementation.

- **Minimal or no changes to user applications (untrusted and trusted):** Changes to trusted and untrusted applications should be minimal or none at all. This requirement eases the adoption of the solution and ensures that existing applications will be automatically protected, without burdening the developers to re-write their applications.

- **Minimal changes to the trusted kernel:** No major architectural changes should be required within the secure world. This ensures that the TCB will remain small and that all modifications can be sufficiently audited. Since *minimal* is subjective, we specify that any modifications to the trusted OS abide by a soft ceiling of 100 lines of code.

3.6.1 Page Table Introspection

An obvious and simple method capable of defending against BOOMERANG is to leverage the trusted OS's visibility into the non-secure world to verify the ownership of the

memory being accessed by simply reading the same page tables that are used by the untrusted OS. A variant of this approach is taken by NVIDIA's Trusted Little Kernel (TLK), the TEE used by Tegra processors [104]. This defense requires the trusted OS to have a complete understanding of the page table structure within the untrusted OS. Thus, when an untrusted application passes a memory reference, the trusted OS would first verify that the memory actually belongs to the untrusted application that made the call by doing a page-table walk, and, only then, map that memory into the memory space of the requested TA.

This approach has a few notable advantages. It is entirely invisible to the untrusted OS, as the entire PTRSAN function is implemented within the secure world. Additionally, it does not require any extra memory copy operations, which is an improvement over shared-memory defenses, which we explain in Chapter 3.6.2. However, the Achilles' heel of this approach is the amount of work that must be performed by the trusted OS to interpret the untrusted OS's page table structure, and then make security decisions based on that interpretation. Researchers have shown that page table walks can be extremely dangerous. For example, since the trusted OS is performing a walk on a page table controlled by the untrusted OS, a malicious untrusted OS could potentially craft a malicious page table and obtain arbitrary code execution within the trusted OS [106, 136].

Furthermore, this defense, while relatively easy to implement, does not satisfy our first requirement, as the trusted OS must be aware of the page table structure managed by the untrusted OS. This approach is not generalizable and would likely require a customized trusted OS to accompany each untrusted OS, or at least a different instantiation based on the page table structure. Finally, this defense is likely not possible to implement while satisfying our third requirement of a minimal TCB, as an elegant and correct page table walk requires a considerable amount of code, likely far more than 100 lines.

This approach works well for TLK, where the trusted OS is a derivative of Linux

and is therefore able to manage Linux page tables using the same code as the untrusted OS. However, we do not consider it a viable generic approach since it violates two of our requirements. Thus, we did not evaluate its efficacy in practice in Chapter 3.7; however, we do not discredit its viability as a defense against BOOMERANG, and we believe that it could be a reasonable defense in specific instances.

3.6.2 Dedicated Shared Memory Region

The heart of the BOOMERANG flaw stems from the fact that the secure world can read from and write to any non-secure memory it wishes. In the dedicated shared memory region defense, a special physical memory region (e.g., `common-memory` in the case of OP-TEE) is defined, which is the *only* region of memory that is readable and writable by both worlds. To verify any pointers that are passed from the non-secure world, the secure world then needs only to verify that the memory is within the `common-memory`, which will protect both worlds. Note that this is the exact method employed by OP-TEE (see Figure 3.5).

This defense is easy to implement in the secure world. In fact, this defense actually makes the secure world's PTRSAN function extremely simple, as it needs only to confirm that the memory is within the shared region. Nevertheless, this defense has numerous drawbacks in the non-secure world:

- The untrusted OS is burdened with handling all of the shared memory regions (i.e., sections of `common-memory`) amongst the various untrusted user applications. This memory management can be exceptionally complicated, and, indeed, we found at least 4 bugs [125, 126, 127, 128] in different components of this mechanism in OP-TEE.

- For high-throughput applications (e.g., DRM video decryption), this defense adds an undesirable overhead, since it requires all of the data to be copied into a special buffer,

which is not in the requesting application's memory space. This global memory region also requires a global lock on memory, which can become a serious bottleneck in multi-threaded applications. In our tests (Chapter 3.7.1), this global locking mechanism alone consumed approximately 36% of the total overhead.

- Shared memory makes it extremely difficult, and in some cases impossible, to implement certain types of applications. For example, a popular use of TrustZone is memory integrity checking [43], where an untrusted application requests that a TA monitors its memory, which does not work with shared memory.

- This defense only thwarts the general BOOMERANG attack, but can still permit applications to leverage BOOMERANG to read from and write to arbitrary regions within the shared memory, which may contain sensitive data.

We show in Chapter 3.7.3 how this currently advocated defense compares against our proposed solution.

3.6.3 Cooperative Semantic Reconstruction

Due to the limitations of existing BOOMERANG defenses, we propose a novel defense (CSR), which is capable of bridging the semantic gap between the two worlds with minimal modification and minimal overhead. In this defense, the trusted OS and the untrusted OS both cooperate to verify memory pointers that are passed into the secure world to ensure that the untrusted application indeed has permission to access the referenced memory region. This implementation was based on one key insight: the untrusted OS already adequately implements memory protection mechanisms; however, this information is not currently easily accessible to the trusted OS. Thus, to implement this defense, the untrusted OS needs only to expose a simple callback to the secure world that

permits the trusted OS to query the untrusted OS's PTRSAN function, where the memory address can be trivially verified. This callback can be used from within the secure world any time that non-secure memory is to be accessed, thus thwarting any unintended BOOMERANG vulnerabilities. Fundamentally, this defense bridges the semantic gap by allowing the secure world, which has no insight into the layout of non-secure memory, to query the untrusted OS as a security oracle, which is able to correctly respond. An overview of the approach can be seen in Figure 3.8

In this defense, the untrusted applications prepare requests to TAs exactly as they would without it. The call to the TA would similarly be handled by an exposed kernel driver or TEE (1), which would handle the world switching. Note the there is no proactive PTRSAN necessary by either the daemon or the kernel driver. In fact, the buffer is passed directly into the secure world with the non-secure world virtual memory address intact. The only addition is that the process identification number (PID) of the requesting process (we refer to this as the `req_pid`) is now appended to the request structure by the untrusted OS during the SMC call (2). Now, in the secure world, when a TA needs to access a pointer that was passed as an argument, which is a virtual address that belongs to the untrusted application that initiated the call, the TA must first query the trusted OS to resolve the physical address (4). This query is implemented as a callback to the untrusted kernel with the pointer value (virtual address), the length of the buffer, and the corresponding `req_pid` (5). The untrusted OS kernel can trivially handle the callback by checking that the buffer indeed belongs to the address space of `req_pid` (6). If the verification is successful, the untrusted OS then locks the corresponding pages (to avoid them being paged out) and sends the physical addresses back to the secure world (7). At this point, the trusted OS will then implement its own PTRSAN function to verify that the physical address from the untrusted OS is, in fact, in the non-secure world (8). Then, the trusted OS will map it into the TA's memory space or allow the TA to access the

physical address directly (9). If verification fails, a corresponding error code is returned.

Given that every TEE implementation already has callback support for high-level operations (e.g., file operations, network communication), this exact same channel can be leveraged to implement CSR. Note that CSR provides a generic mechanism to bridge the semantic gap between the two worlds, and that it can also be extended to verify access to files, or other peripherals by the secure world.

At first glance, it may appear that this defense would require modifications to all of the components (i.e., the untrusted application, the untrusted kernel, the trusted kernel, and the trusted application). However, since all of the trusted applications that we observed use a client library, we believe that simply updating this client library would be enough in practice. Similar to untrusted applications, existing TAs would not require any modification, as this defense could be implemented in the trusted kernel functions (e.g., `qsee_register_shared_buffer()` in the case of QSEE) that are already used to access non-secure world memory. The only real modifications that would have to be deployed would be the modifications to the untrusted and trusted kernels, which would add the functionality to handle and perform the required callback, respectively.

The main overhead introduced by CSR is the additional verification path (i.e., (4)-(9)). However, we show in Chapter 3.7.2 that this overhead is minimal and comparable to other defenses.

3.7 Evaluation of Defenses

We evaluated the two most promising proposed defenses: Dedicated Shared Memory Region (DSMR) and CSR. We decided not to include Page Table Introspection (PTI) in our analysis, as it does not satisfy our requirements as a general BOOMERANG defense. Similarly, we did not explicitly compare our defenses against a vanilla TEE implemen-

tation, as we do not see *no defense* as an option. We performed our evaluation on the OP-TEE platform [105], with Linux as our untrusted OS. OP-TEE was chosen because it is completely open source, has a very well-maintained code base with clear documentation, and includes an exhaustive test suite, which we used to evaluate the performance overhead of our defenses.

We chose the HiKey development board (Lemaker Version) as the hardware platform for testing, which is one of the boards recommended by the OP-TEE developers [105]. This board includes a traditional ARM processor and associated hardware, which are almost identical to what would be found on a consumer Android handset [137]. OP-TEE has an extensive test suite with 63 tests called `xtest`[138]. These tests cover both sanity and functionality check of various TAs, TEE benchmarking, and Global Platform compliance. We modified the test driver to record timings for each of the tests as well as profiling information for the different phases of DSMR and CSR. All reported timing data are averaged across 30 runs of `xtest` on the HiKey board, where the system was rebooted between runs to avoid caching-related inconsistencies.

3.7.1 Dedicated Shared Memory Region

As explained in Chapter 3.4.3, OP-TEE's default configuration uses the DSMR method as the only mechanism for passing memory arguments. In this implementation, the untrusted OS's client library handles the allocation of the shared memory region, which consists of assigning an identifier (`shmid`), copying of data to and from the corresponding shared buffer, and ultimately releasing it. Recall that this shared memory management within the untrusted OS is the main overhead in this implementation. There is virtually no overhead in the trusted OS, as it just needs to check that the pointer argument is contained within the `common-memory` region. On average, allocating shared memory took

13.795 μs, releasing memory took 7.982 μs, and the time it took to copy memory contents was negligible. Thus, the total incurred overhead was 21.777 μs per secure-world query. This low overhead is partially attributed to the fact that the maximum size being copied in the tests was only 4,097 bytes; however, we would expect these numbers to rise significantly with larger memory regions.

3.7.2 Cooperative Semantic Reconstruction

As we previously explained in Chapter 3.4.3, in OP-TEE all arguments to TA are typed (i.e., pointer or value), and all pointers are already checked to ensure that they are within the `common-memory` region. Thus, we were able to implement our CSR defense by simply adding a new pointer parameter type, `RAW_PTR`, and modifying the trusted OS to perform the required callback to the untrusted OS for every `RAW_PTR`. We also changed the untrusted OS's client library (i.e., `libteec.so`) to use the `RAW_PTR` as the default type for all pointers. The untrusted kernel driver was similarly modified to handle the callback function. We implemented our PTRSAN function in the callback, which verifies that the argument is a valid virtual address within the appropriate untrusted application (referenced by its PID). Upon verification, we then resolve the corresponding physical memory pages, set them to be *non-pageable*, and return the physical addresses back to the secure world. All of our modifications to OP-TEE are backward-compatible and can easily co-exist with the existing DSMR defense. These modifications resulted in only 91 modified lines of code in the OP-TEE trusted OS (see Table 3.2 to see the modifications per component).

As explained in Chapter 3.6.3, most of the additional overhead introduced by CSR is caused by the callbacks from the trusted OS to the untrusted OS for every `RAW_PTR` argument type. In OP-TEE, all of the pointer arguments are first sanitized by the trusted

Table 3.2: Total modifications required to implement CSR in OP-TEE, measured in lines of code (LOC).

Component	Added LOC	Modified LOC	Total LOC
Trusted OS	88	3	**91**
Untrusted OS	273	2	**275**
Client Library	39	0	**39**

OS before invoking the TA. Hence, all of our results for CSR do not include the calls between the TA and the trusted OS (i.e., ④ and ⑨ in Figure 3.8). Nevertheless, we similarly measured the incurred overhead of CSR by running the `xtest` suite, which made a total of 3,885 callbacks throughout its tests. The average time taken for the trusted OS to confer with the untrusted OS to sanitize pointers (⑤-⑥-⑦-⑧) over all 3,885 callbacks was 26.891 μs, 21.909 μs of which were spent within the untrusted OS doing validation and memory page pinning (⑥). This is almost identical to the 21.777 μs overhead incurred by the DSMR defense.

3.7.3 Comparative Evaluation

To get an idea of the specific performance of memory management operations with the two defenses, we analyzed the profiling data for the various operations performed by both approaches and found that performance for a single memory access with DSMR is slightly better, 5.113 μs faster, than CSR. However, the performance across the entire range of tests is much more interesting.

A summary of the testing data, in terms of the average overhead of CSR over DSMR for each test category, is shown in Table 3.3. Note that a negative value indicates CSR was faster than DSMR for the corresponding category. The *Trusted-Untrusted Communication* category represents CSR's worst performance in terms of the percentage of

Table 3.3: Summary of benchmark results, showing the overhead of CSR over DSMR.

Category	Overhead	
	Avg. %	Avg. Time (ms)
Basic Functionality	-0.58%	-7.168
Trusted-Untrusted Communication	4.45%	0.510
Crypto operations	-1.72%	-901.548
Secure File Storage	0.03%	0.694
Total for all Tests	**-0.0344**	**-189.919**

overhead. There are 14 tests in this category and all of them primarily perform a lot of SMC operations (approximately 200) to test inter-world communication. CSR allocates and deallocates memory-tracking structures during each SMC, as it cannot know ahead of time when memory arguments are to be used. This contributes a very small overhead for each SMC, which is reflected as a larger percentage in these particular tests, although even here, this net overhead in terms of time is still low.

In the context of the other 49 tests performed, the percentage of overhead contributed by CSR versus DSMR is very small. CSR introduces no more than 0.03% overhead in the worst case and improves performance by up to 1.72% in others.

For those tests with non-secure memory operations, we observed that the DSMR overhead varied significantly, whereas the overhead of CSR remained constant for a given number of memory operations. The main reasons for variance in DSMR overhead are:

- *Synchronized access:* The allocation and release of shared memory involves acquiring a global lock. For a multi-threaded application making simultaneous shared memory requests and releases will result in idle tasks as they wait for the global lock, increasing the overhead of DSMR. We observed this in one of the tests of the *Basic Functionality*

category, which creates several threads, all of which make requests to a TA. During this subtest, the overhead for a shared memory allocation went up to 80 microseconds and in total CSR beat DSMR by 11.72 seconds of execution time.

- *Additional copying:* In DSMR, untrusted applications need to copy data to or from shared memory to communicate with the TA. This copying time can be an overhead, if a large amount of data is being exchanged between the untrusted application and the corresponding TA. For example, one of the tests in the *Trusted-Untrusted Communication* category, which passes a large amount of data, suffered a 26% overhead because of this memory copying.

- *Memory Fragmentation:* Depending on how shared memory is allocated and released, it could get severely fragmented. As DSMR in OP-TEE uses a best-fit algorithm to find free regions of shared memory, fragmentation increases the time to find a free chunk, thus increasing the overhead of DSMR.

Although CSR is slightly outperformed by DSMR in some tests, in practice CSR is the best candidate for an all-around defense. CSR offers the best security properties, requires minimal modification for implementation, incurs minimal overall performance overhead, and actually boosts performance for multi-threaded applications. Thus, per our evaluation, CSR appears to be the ideal defense against BOOMERANG.

3.8 Moving up

In addition to the novel security features such as ARM TrustZone, smart devices have extensible software support in the form of open-source system software. The availability of well-supported open-source system software enables vendors to perform quick customizations, e.g., by adding device drivers to the operating systems. Unfortunately,

these customizations are poorly developed, which results in a lot of critical security issues. In the next chapter, I will explain how we can develop efficient static analysis techniques to detect security issues in complex system software, specifically, Linux kernel drivers.

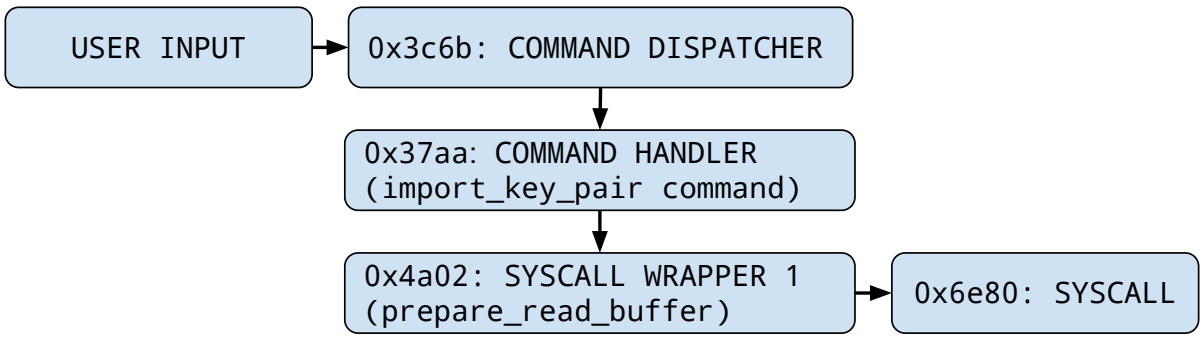

Figure 3.7: One of the three outputs of our data-flow analysis described in Chapter 3.5.1 for the KeyMaster TA on QSEE.

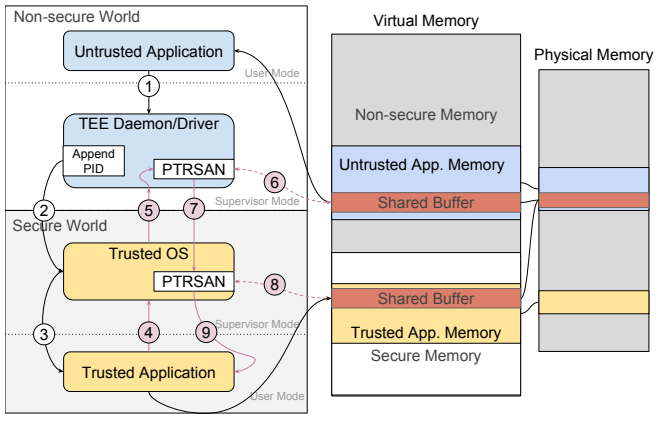

Figure 3.8: Cooperative Semantic Reconstruction data-flow and pointer resolution technique.

Chapter 4

Scalable static analysis of device drivers

The mobile domain has seen an explosion of new devices, and thus new drivers, introduced in recent years. The lack of attention being paid to these drivers, and their potential danger to the security of the devices, has also not gone unnoticed [139]. Recent studies even purport that mobile kernel drivers are, again, the source of up to 85% of the reported bugs in the Android [140] kernel. Yet, we are unaware of any large-scale analysis of these drivers. Bugs in kernel-level code can be particularly problematic in practice, as they can lead to severe vulnerabilities, which can compromise the security of the entire computing system (e.g., Dirty COW [141]).

This fact has not been overlooked by the security community, and a significant amount of effort has been placed on verifying the security of this critical code by means of manual inspection and both static and dynamic analysis techniques. While manual inspection has yielded the best results historically, it can be extremely time consuming, and is quickly becoming intractable as the complexity and volume of kernel-level code increase. Low-level code, such as kernel drivers, introduce a variety of hard problems that must be

overcome by dynamic analysis tools (e.g., handling hardware peripherals). Thus, static source code analysis has long prevailed as the most promising technique for kernel code verification and bug-finding, since it only requires access to the source code, which is typically available.

Unfortunately, kernel code is a worst-case scenario for static analysis because of the liberal use of pointers (i.e., both function and arguments are frequently passed as pointers). As a result, tool builders must make the tradeoff between *precision* (i.e., reporting too many false positives) and *soundness* (i.e., reporting all true positives). In practice, precise static analysis techniques have struggled because they are either computationally infeasible (i.e., because of the state explosion problem), or too specific (i.e., they only identify a very specific type of bug). Similarly, sound static analysis techniques, while capable of reporting all bugs, suffer from extremely high false-positive rates. This has forced researchers to make variety of *assumptions* in order to implement practical analysis techniques.

Bugs in kernel-level code can be particularly problematic in practice, as they can lead to severe vulnerabilities, which can compromise the security of the entire computing system (e.g., Dirty COW [141]). This fact has not been overlooked by the security community, and a significant amount of effort has been placed on verifying the security of this critical code by means of manual inspection and both static and dynamic analysis techniques. While manual inspection has yielded the best results historically, it can be extremely time consuming, and is quickly becoming intractable as the complexity and volume of kernel-level code increase. Low-level code, such as kernel drivers, introduce a variety of hard problems that must be overcome by dynamic analysis tools (e.g., handling hardware peripherals). While some kernel-level dynamic analysis techniques have been proposed [43, 142, 143, 144], they are ill-suited for bug-finding as they were implemented as kernel monitors, not code verification tools. Thus, static source code analysis has long

prevailed as the most promising technique for kernel code verification and bug-finding, since it only requires access to the source code, which is typically available. There are numerous successful tools have been developed (e.g., Coverity [145], Linux Driver Verification [69], APISan [63]), and have provided invaluable insights into both the types and locations of bugs that exist in critical kernel code. These tools range from precise, unsound, tools capable of detecting very specific classes of bugs (e.g., data leakages [146], proper `fprintf` usage [81], user pointer deferences [147]) to sound, imprecise, techniques that detect large classes of bugs (e.g., finding all usages of `strcpy` [77]). However, there is no single tool that uses a generic technique to find all classes of vulnerabilities.

4.1 Vulnerability Detection Technique Based on Soundy Analysis

In this Chapter, we present Dr. Checker, a fully-automated static-analysis tool capable of identifying numerous classes of bugs in Linux kernel drivers. Dr. Checker is implemented as a completely modular framework, where both the types of analyses (e.g., points-to or taint) and the bug detectors (e.g., integer overflow or memory corruption detection) can be easily augmented. Our tool is based on well-known program analysis techniques and is capable of performing both pointer and taint analysis that is flow-, context-, and field-sensitive. Dr. Checker employs a *soundy* [148] approach, which means that our technique is mostly sound, aside from a few well-defined assumptions that violate soundness in order to achieve a higher precision. Dr. Checker, is the first (self-proclaimed) *soundy* static-analysis-based bug-finding tool, and, similarly, the first static analysis tool capable of large-scale analysis of general classes of bugs in driver code. We evaluated Dr. Checker by analyzing nine popular mobile device kernels, 3.1 million

LOC, where it correctly reported 3,973 flaws and resulted the discovery of **158** [149, 150, 151, 152, 153] previously *unknown* bugs. We also compared Dr. Checker against four other popular static analysis tools, where it significantly outperformed all of them both in detection rates and total bugs identified. Our results show that Dr. Checker not only produces useful results, but does so with extremely high precision (78%).

In summary, we claim the following contributions:

- We present the first *soundy* static-analysis technique for pointer and taint analysis capable of large-scale analysis of Linux kernel drivers.
- We show that our technique is capable of flow-sensitive, context-sensitive, and field-sensitive analysis in a pluggable and general way that can easily be adapted to new classes of bugs.
- We evaluated our tool by analyzing the drivers of nine modern mobile devices, which resulted in the discovery of 158 *zero-day* bugs.
- We compare our tool to the existing state-of-the-art tools and show that we are capable of detecting more bugs with significantly higher precision, and with high-fidelity warnings.

4.2 Dr. Checker Analysis Design

Dr. Checker uses a modular interface for its analyses. This is done by performing a general analysis pass over the code, and invoking *analysis clients* at specific points throughout the analysis. These analysis clients all share the same global state, and benefit from each other's results. Once the analysis clients have run and updated the global state of the analysis, we then employ numerous *vulnerability detectors*, which

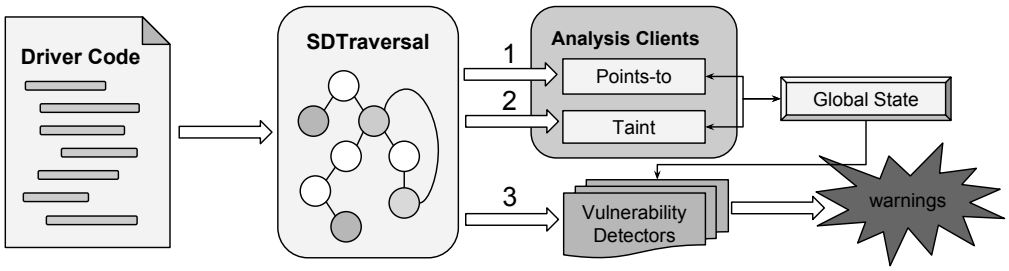

Figure 4.1: Pluggable static analysis architecture implemented by Dr. Checker.

identify specific properties of known bugs and raise warnings (e.g., a tainted pointer was used as input to a dangerous function). The general architecture of Dr. Checker is depicted in Figure 6.1, and the details of our analysis and vulnerability detectors are outlined in the following chapters.

Below we briefly outline a few of our core assumptions that contribute to our *soundy* analysis design:

Assumption 1. We assume that all of the code in the mainline Linux core is implemented *perfectly*, and we do not perform any inter-procedural analysis on any kernel API calls.

Assumption 2. We only perform the number of traversals required for a reach-def analysis in loops, which could result in our points-to analysis being unsound.

Assumption 3. Each call instruction will be traversed only once, even in the case of loops. This is to avoid creating additional contexts and limit false positives, which may result in our analysis being unsound.

4.2.1 Terminology and Definitions

We first define the various terms and concepts used to describe our analysis.

Definition 4.2.1 *A control flow graph (CFG) of a function is a directed graph where each*

node represents a basic block (i.e., a contiguous sequence of non-branch instructions) and the edges of the graph represent possible control flow between the basic blocks.

Definition 4.2.2 *A strongly connected component (SCC) of a graph is a sub-graph, where there exists a bi-directional path between any pair of nodes (e.g., a loop).*

Definition 4.2.3 *Topological sort or ordering of nodes in a directed graph is an ordering of nodes such that, for every edge from node v to u, v is traversed before u. While this is well-defined for acyclic graphs, it is less straightforward for cyclic graphs (e.g., a CFG with loops). Thus, when performing a topological sort on a CFG, we employ Tarjan's algorithm [154], which instead topologically sorts the SCCs.*

Definition 4.2.4 *An entry function, ϵ, is a function that is called with at least one of its arguments containing tainted data (e.g., an* `ioctl` *call).*

Definition 4.2.5 *The context, Δ, of a function in our analysis is an ordered list of call sites (e.g., function calls on the stack) starting from an entry function. This list indicates the sequence of function calls and their locations in the code that are required to reach the given function. More precisely, $\Delta = \{\epsilon, c_1, c_2, ...\}$ where c_1 is call made from within the entry function (ϵ) and for all $i > 1$, c_i is a call instruction in the function associated with the previous call instruction (c_{i-1}).*

Definition 4.2.6 *The global taint trace map, τ, contains the information about our tainted values in the analysis. It maps a specific value to the sequence of instructions (I) whose execution resulted in the value becoming tainted.*

$$\tau : \begin{cases} v \rightarrow \{I_1, I_2, I_3, ...\} & \textit{if TAINTED} \\ v \rightarrow \emptyset & \textit{otherwise} \end{cases}$$

Definition 4.2.7 *An alias object, $\hat{a} = \{\rho, t\}$, is a tuple that consists of a map (ρ) between offsets into that object, n, and the other corresponding alias objects that those offsets can point to, as well as a local taint map (t) for each offset. For example, this can be used to represent a structure stored in a static location, representing an alias object, which contains pointers at given offsets (i.e., offsets into that object) to other locations on the stack (i.e., their alias objects). More precisely, $\rho : n \rightarrow \{\hat{a_1}, \hat{a_2}, \hat{a_3}, ...\}$ and $t : n \rightarrow \{I_1, I_2, I_3, ...\}$. We use both $\hat{a}(n)$ and $\rho(n)$ interchangeably, to indicate that we are fetching all of the alias objects that could be pointed to by a field at offset n. We use $\hat{a}_t$ to refer to the taint map of location $\hat{a}$, and similarly $\hat{a}_t(n)$ to refer to taint at a specific offset. These maps allow us to differentiate between different fields of a structure to provide field-sensitivity in our analysis.*

The following types of locations are traced by our analysis:

1. *Function local variables (or stack locations): We maintain an alias object for each local variable.*

2. *Dynamically allocated variables (or heap locations): These are the locations that are dynamically allocated on the program heap (e.g., as retrieved by* `malloc` *or* `get_page`*). We similarly create one alias object for each allocation site.*

3. *Global variables: Each global variable is assigned a unique alias object.*

Stack and heap locations are both context-sensitive (i.e., multiple invocations of a function with different contexts will have different alias objects). Furthermore, because of our context propagation, heap locations are call-site sensitive (i.e., for a given context, one object will be created for each call site of an allocation function).

Definition 4.2.8 *Our points-to map, ϕ, is the map between a value and all of the possible locations that it can point to, represented as a set of tuples containing alias objects and*

offsets into those objects.

$$\phi : v \rightarrow \{(n_1, \hat{a_1}), (n_1, \hat{a_2}), (n_2, \hat{a_3}), ...\}$$

For example, consider the instruction `val1 = &info->dirmap`*, where* `info` *represents a structure on the stack and member* `dirmap` *is at offset 8. This instruction would result in the value (*`val1`*) pointing to the offset 8 within the alias object* `info` *(i.e.,* $\phi($`val1`$) =$ $\{(8,$ `info`$)\}$*).*

Definition 4.2.9 *The Global State, S, of our analysis contains all of the information computed for every function, at every context. We define it as*

$$S = \{\phi_c, \tau_c\},$$

where $\phi_c : \Delta \rightarrow \phi$ *is the map between a context and the corresponding points-to map, and* $\tau_c : \Delta \rightarrow \tau$ *is the map between a context and corresponding taint trace map.*

4.2.2 Soundy Driver Traversal (SDT)

While most of the existing static analysis techniques [155, 61] run their abstract analysis until it reaches a fixed-point before performing bug detection, this can be problematic when running multiple analyses, as the different analyses may not have the same precision. Thus, by performing analysis on the post-completion results, these tools are fundamentally limiting the precision of all of their analyses to the precision of the *least* precise analysis. To avoid this, and ensure the highest precision for all of our analysis modules, we perform a *flow-sensitive* and *context-sensitive* traversal of the driver starting from an entry point. Our specific analysis modules (i.e., taint and points-to) are implemented as *clients* in this framework, and are invoked with the corresponding context and current global state as the code is being traversed. This also allows all of the analyses,

or clients, to consume each other's results whenever the results are needed, and without loss of precision. Moreover, this allows us to perform a single traversal of the program for all of the underlying clients.

It is important to note that some of the client analyses may actually need more traversals through the CFG than others to reach a fixed point. For example, a points-to analysis might need more traversals through a loop to reach a fixed point than a taint analysis. However, our code exploration is analysis-agnostic, which means we must ensure that we always perform the maximum number of traversals required by all of our analyses. To ensure this property, we use reach-def analysis [156] as a baseline (i.e., we traverse the basic blocks such that a reaching definition analysis will reach a fixed point). This ensures that all of the writes that can reach an instruction directly will be reached. This means that our points-to analysis may not converge, as it would likely require far more iterations. However, in the worst case, points-to analysis could potentially grow unconstrained, resulting in everything pointing to everything. Thus, we make this necessary sacrifice to soundness to ensure convergence and a practical implementation.

Loops. When handling loops, we must ensure that we iterate over the loop enough times to ensure that every possible assignment of every variable has been exercised. Thus, we must compute the number of iterations needed for a reach-def analysis to reach a fix-point on the loop and then perform the corresponding number of iterations on all the basic blocks in the loop. Note that, the number of iterations to converge on a loop for a standard reach-def analysis is upper-bounded by the longest use-def chain in the loop (i.e., the longest number of instructions between the assignment and usage of a variable). The intuition behind this is that, in the worst case, every instruction could potentially depend on the variable in the use-def chain, such that their potential values could update in each loop. However, this can only happen as many times as their are instructions,

since an assignment can only happen once per instruction.

Function calls. If a function call is a direct invocation and the target function is within the code that we are analyzing (i.e., it is part of the driver), it will be traversed with a new context (Δ_{new}), and the state will be both updated with a new points-to map (ρ_{new}) and a new taint trace map (τ_{new}), which contains information about both the function arguments and the global variables. For indirect function calls (i.e., functions that are invoked via a pointer), we use type-based target resolution. That is, given a function pointer of type a = `(rettype)(arg1Type, arg2Type,..)`, we find all of the matching functions in the same driver that are referenced in a non-call instruction (e.g., `void *ptr = &fn`). This is implemented as the function *resolve_call* in Algorithm 1. Each call site or call instruction will be analyzed only once per context. We do not employ any special handlers for recursive functions, as recursion is rarely used in kernel drivers.

The complete algorithm, `SDTraversal`, is depicted in Algorithm 1. We start by topologically sorting the CFG of the function to get an ordered list of SCCs. Then, each SCC is handled differently, depending on whether it is a loop or not. Every SCC is traversed at the basic-block level, where every instruction in the basic block is provided to all of the possible clients (i.e., taint and points-to), along with the context and global state. The client analyses can collect and maintain any required information in the global state, making the information immediately available to each other.

To analyze a driver entry point ϵ, we first create an initial state: $S_{start} = \{\phi_{start}, \emptyset\}$, where ϕ_{start} contains the points-to map for all of the global variables. We then traverse all of the `.init` functions of the driver (i.e., the functions responsible for driver initialization [157]), which is where drivers will initialize most of their global objects. The resulting initialized state (S_{init}) is then appended with the taint map for any tainted arguments ($S_{init} = S_{init} \cup \tau_{init}$). We describe how we determine these tainted arguments

in Chapter 4.4.3. Finally, we invoke our traversal on this function, `SDTraversal(`S_{init}`,` Δ_{init}`, ` ϵ`)`, where the context $\Delta_{init} = \{e\}$.

We use the low-level virtual machine (LLVM) intermediate representation (IR), Bitcode [9], as our IR for analysis. Bitcode is a typed, static single assignment (SSA) IR, and well-suited for low-level languages like C. The analysis clients interact with our soundy driver traversal (SDT) analysis by implementing visitors, or transfer functions, for specific LLVM IR instructions, which enables them to both use and update the information in the global state of the analysis. The instructions that we define transfer functions for in the IR are:

1. *Alloca* (`v = alloca typename`) allocates a stack variable with the size of the type `typename` and assigns the location to v (e.g., `%1 = alloca i32`). SDT uses the instruction location to reference the newly allocated instruction. Since SDT is context-sensitive, the instruction location is a combination of the current context and the instruction offset within the function bitcode.

2. *BinOp* (`v = op op1, op2`) applies `op` to `op1` and `op2` and assigns the result to `v` (e.g., `%1 = add val, 4`). We also consider, the flow-merging instruction in SSA, usually called `phi` [158], to be the same as a binary operation. Since SDT is not path-sensitive, this does not affect the soundness.

3. *Load* (`v = load typename op`) is the standard load instruction, which loads the contents of type `typename` from the address represented by the operand `op` into the variable `v` (e.g., `%tmp1 = load i32* %tmp`).

4. *Store* (`store typename v, op`) is the standard store instruction, which stores the contents of type `typename` represented by the value `v` into the address represented by `op` (e.g., `store i8 %frombool1, %y.addr`).

5. *GetElementPtr (GEP)* is the instruction used by the IR to represent structure and array-based accesses and has fairly complex semantics [159]. A simplified way to represent this is `v = getelementptr typename ob, off`, which will get the address of the field at index `off` from the object `ob` of type `typename`, and store the referenced value in `v` (e.g., `%val = getelementptr %struct.point %my_point, 0`).

Both our points-to and taint analysis implement transfer functions based on these five instructions.

4.2.3 Points-to Analysis

The result of our points-to analysis is a list of values and the set of all of the possible objects, and offsets, that they can point to. Because of the way in which we constructed our *alias location objects* and transfer functions, we are able to ensure that our points-to results are field-sensitive. That is, we can distinguish between objects that are pointed to by different fields of the same object (e.g., different elements in a `struct`). Thus, as implemented in SDT, we are able to obtain points-to results that are flow-, context-, and field-sensitive.

Dynamic allocation. To handle dynamic allocation in our points-to analysis, we maintain a list of kernel functions that are used to allocate memory on the heap (e.g., `__kmalloc`, `kmem_cache_alloc`, `get_free_page`). For each call-site to these functions, we create a unique *alias object*. Thus, for a given context of a function, each allocation instruction has a single `alias location`, regardless of the number of times that it is visited. For example, if there is a call to `kmalloc` in the basic block of a loop, we will only create one alias location for it.

Internal kernel functions. Except for few kernel API functions, whose effects can be easily handled (e.g., `memcpy`, `strcpy`, `memset`), we ignore all of the other kernel APIs and core kernel functions. For example, if the target of a call instruction is the function `i2c_master_send`, which is part of the core kernel, we do not follow the call. Contrary to the other works, which check for valid usage of kernel API functions [64, 63], we assume that all usages of these functions are valid, as we are only concerned with analyzing the more error-prone driver code. Thus, we do no follow any function calls into the core kernel code. While, we may miss some points-to information because of this, again sacrificing soundness, this assumption allows us to be more precise within the driver and scale our analysis.

The *update points-to* transfer functions (`updatePto*`) for the various instructions are as shown in Algorithm 2.

4.2.4 Taint Analysis

Taint analysis is a critical component of our system, as almost all of our bug detectors use its results. Similar to our points-to analysis, the results of our taint analysis are flow-, context-, and field-sensitive.

The *taint sources* in our analysis are the arguments of the entry functions. Chapter 4.4.3 explains the different types of entry functions and their correspondingly tainted arguments. We also consider special kernel functions that copy data from user space (e.g., `copy_from_user`, `simple_write_to_buffer`) as taint sources and taint all of the fields in the alias locations of the points-to map for the destination operands of these functions. Our *taint propagators* are implemented as various transformation functions (`updateTaint*` in Algorithm 3). Similar to our points-to analysis, we do not propagate taint for any core kernel function calls, aside from a few exceptions (e.g., `memcpy`). The

taint sinks in our analysis are dependent on the vulnerability detectors, as every detector has its own taint policy. These detectors will raise warnings if any tainted data violates a specified policy (e.g., if a tainted value is used as the length in a `memcpy`).

4.3 Vulnerability Detectors

This chapter describes the various vulnerability detectors that were used in our analysis. These detectors are highly configurable and are able to act on the results from both our points-to and taint analysis. They are implemented as plugins that are run continuously as the code is being analyzed, and operate on the results from our analysis clients (i.e., taint and points-to analysis). Our architecture enables us to very quickly implement new analyses to explore new classes of vulnerabilities. In fact, in the process of analyzing our results for this paper, we were able to create the Global Variable Race Detector (GVRD) detector and deploy it in less than 30 minutes.

Almost all of the detectors use taint analysis results to verify a vulnerable condition and produce a taint trace with all of their emitted warnings. The warnings also provide the line numbers associated with the trace for ease of triaging. The various bug detectors used by Dr. Checker in our analysis are explained below:

Improper Tainted-Data Use Detector (ITDUD) checks for tainted data that is used in risky functions (i.e., `strc*`, `strt*`, `sscanf`, `kstrto`, and `simple_strto` family functions). An example of a previously unknown buffer overflow, detected via ITDUD, is shown in Listing 4.1.

Tainted Arithmetic Detector (TAD) checks for tainted data that is used in operations that could cause an overflow or underflow (e.g., `add`, `sub`, or `mul`). An example of a zero-day detected by TAD is shown in Listing 4.2.

Invalid Cast Detector (ICD) keeps tracks of allocation sizes of objects and checks for

any casts into an object of a different size.

Tainted Loop Bound Detector (TLBD) checks for tainted data that is used as a loop bound (i.e., a loop guard in which at least one of the values is tainted). These bugs could lead to a denial of service or even an arbitrary memory write. The example in Listing 4.2 shows this in a real-world bug, which also triggered on TAD.

Tainted Pointer Dereference Detector (TPDD) detects pointers that are tainted and directly dereferenced. This bug arises when a user-specified index into a kernel structure is used without checking.

Tainted Size Detector (TSD) checks for tainted data that is used as a size argument in any of the `copy_to_` or `copy_from_` functions. These types of bugs can result in information leaks or buffer overflows since the tainted size is used to control the number of copied bytes.

Uninit Leak Detector (ULD) keeps tracks of which objects are initialized, and will raise a warning if any `src` pointer for a userspace copy function (e.g., `copy_to_user`) can point to any uninitialized objects. It also detects structures with padding [160] and will raise a warning if `memset` or `kzalloc` has not been called on the corresponding objects, as this can lead to an information leak. An example of a previously unknown bug detected by this detector is as shown in Listing 4.3

Global Variable Race Detector (GVRD) checks for global variables that are accessed without a mutex. Since the kernel is reentrant, accessing globals without syncronization can result in race conditions that could lead to time of check to time of use (TOCTOU) bugs.

4.4 Implementation

Dr. Checker is built on top of LLVM 3.8 [9]. LLVM was chosen because of its flexibility in writing analyses, applicability to different architectures, and excellent community support. We used integer range analysis as implemented by Rodrigues *et al.* [161]. This analysis is used by our vulnerability detectors to verify certain properties (e.g., checking for an invalid cast).

We implemented Dr. Checker as an LLVM module pass, which consumes: a bitcode file, an `entry function name`, and an `entry function type`. It then runs our SDT analysis, employing the various analysis engines and vulnerability detectors. Depending on the `entry function type`, certain arguments to the entry functions are tainted before invoking the SDT (See Chapter 4.4.3).

Because our analysis operates on LLVM bitcode, we must first identify and build all of the driver's bitcode files for a given kernel (Chapter 4.4.1). Similarly, we must identify all of the entry points in these drivers (Chapter 4.4.2) in order to pass them to our SDT analysis.

4.4.1 Identifying Vendor Drivers

To analyze the drivers independently, we must first differentiate driver source code files from that of the core kernel code. Unfortunately, there is no standard location in the various kernel source trees for driver code. Making the problem even harder, a number of the driver source files omit vendor copyright information, and some vendors even modify the existing sources directly to implement their own functionality. Thus, we employ a combination of techniques to identify the locations of the vendor drivers in the source tree. First, we perform a `diff` against the mainline sources, and compare those files with a referenced vendor's configuration options to search for file names containing

the vendor's name. Luckily, each vendor has a code-name that is used in all of their options and most of their files (e.g., Qualcomm configuration options contain the string `MSM`, Mediatek is `MTK`, and Huawei is either `HISI` or `HUAWEI`), which helps us identify the various vendor options and file names. We do this for all of the vendors, and save the locations of the drivers relative to the source tree.

Once the driver files are identified, we compile them using clang [162] into both ARM 32 bit and 64 bit bitcode files. This necessitated a few non-trivial modifications to clang, as there are numerous GNU C Compiler (GCC) compiler options used by the Linux kernel that are not supported by clang (e.g., the `-fno-var-tracking-assignments` and `-Wno-unused-but-set-variable` options used by various Android vendors). We also added additional compiler options to clang (e.g., `-target`) to aid our analysis. In fact, building the Linux kernel using LLVM is an ongoing project [163], suggesting that considerable effort is still needed.

Finally, for each driver, we link all of the dependent vendor files into a single bitcode file using `llvm-link`, resulting in a self-contained bitcode file for each driver.

4.4.2 Driver Entry Points

Linux kernel drivers have various ways to interact with the userspace programs, categorized by 3 operations: file [164], attribute [165], and socket [166].

File operations are the most common way of interacting with userspace. In this case, the driver exposes a file under a known directory (e.g., `/dev`, `/sys`, or `/proc`) that is used for communication. During initialization, the driver specifies the functions to be invoked for various operations by populating function pointers in a structure, which will be used to handle specific operations (e.g., `read`, `write`, or `ioctl`). The structure used for initialization can be different for each driver type. In fact, there are at least

86 different types of structures in Android kernels (e.g., `struct snd_pcm_ops`, `struct file_operations`, or `struct watchdog_ops` [167]). Even worse, the entry functions can be at different offset in each of these structures. For example, the `ioctl` function pointer is at field 2 in `struct snd_pcm_ops`, and at field 8 in `struct file_operations`. Even for the same structure, different kernels may implement the fields differently, which results in the location of the entry function being different for each kernel. For example, `struct file_operations` on Mediatek's mt8163 kernel has its `ioctl` function at field 11, whereas on Huawei, it appears at field 9 in the structure.

To handle these eccentricities in an automated way, we used `c2xml` [168] to parse the header files of each kernel and find the offsets for possible entry function fields (e.g., `read` or `write`) in these structures. Later, given a bitcode file for a driver, we locate the different file operation structures being initialized, and identify the functions used to initialize the different entry functions. These serve as our entry points for the corresponding operations. For example, given the initialization as shown in Listing 4.4, and the knowledge that read entry function is at offset 2 (zero indexed), we mark the function `mlog_read` as a read entry function.

Attribute operations are operations usually exposed by a driver to read or write certain attributes of that driver. The maximum size of data read or written is limited to a single page in memory.

Sockets operations are exposed by drivers as a socket file, typically a UNIX socket, which is used to communicate with userspace via various socket operations (e.g., `send`, `recv`, or `ioctl`).

There are also other drivers in which the kernel implements a main wrapper function, which performs initial verification of the user parameters and *partially* sanitizes them before calling the corresponding driver function(s). An example of this can be seen

in the `V4L2 Framework` [169], which is used for video drivers. For our implementation we consider only `struct v4l2_ioctl_ops`, which can be invoked by userspace via the wrapper function `video_ioctl2`.

4.4.3 Tainting Entry Point Arguments

An entry point argument can contain either *directly* tainted data (i.e., the argument is passed directly by userspace and never checked) or *indirectly* tainted data (i.e., the argument points to a kernel location, which contains the tainted data). All of the tainted entry point functions can be categorized in six categories, which are shown in Table 4.1, along with the type of taint data that their arguments represent.

An explicit example of directly tainted data is shown in Listing 4.5. In this snippet, `tc_client_ioctl` is an `ioctl` entry function, so argument 2 (`arg`) is directly tainted. Thus, the statement `char c=(char*)arg` would be dereferencing tainted data and is flagged as a warning. Alternatively, argument 2 (`ctrl`) in `iris_s_ext_ctrls` is a `V4Ioctl` and is indirectly tainted. As such, the dereference

(`data = (ctrl>controls[0]).string`) is safe, but it would taint `data`.

4.5 Limitations

Because of the Dr. Checker's soundy nature, it cannot find all the vulnerabilities in all drivers. Specifically, it will miss following types of vulnerabilities:

- *State dependent bugs:* Since Dr. Checker is a stateless system, it treats each entry point independently (i.e., taint does not propagate between multiple entry points). As a result, we will miss any bugs that occur because of the interaction between multiple entry points (e.g., CVE-2016-2068 [170]).

- *Improper API usage:* Dr. Checker assumes that all the kernel API functions are *safe and correctly used* (Assumption 1 in Chapter 4.2). Bugs that occur because of improper kernel API usage will be missed by Dr. Checker. However, other tools (e.g., APISan [63]) have been developed for finding these specific types of bugs and could be used to complement Dr. Checker.

- *Non-input-validation bugs:* Dr. Checker specifically targets input validation vulnerabilities. As such, non-input validation vulnerabilities (e..g, side channels or access control bugs) cannot be detected.

4.6 Evaluation

To evaluate the efficacy of Dr. Checker, we performed a large-scale analysis of the following nine popular mobile device kernels and their associated drivers (437 in total). The kernel drivers in these devices range from very small components (31 LOC), to much more complex pieces of code (240,000 LOC), with an average of 7,000 LOC per driver. In total, these drivers contained over 3.1 million lines of code. However, many of these kernels re-use the same code, which could result in analyzing the same entry point twice, and inflate our results. Thus, we have grouped the various kernels based on their underlying chipset, and only report our results based on these groupings:

Mediatek:

- Amazon Echo (5.5.0.3)
- Amazon Fire HD8 (6th Generation, 5.3.2.1)
- HTC One Hima (3.10.61-g5f0fe7e)
- Sony Xperia XA (33.2.A.3.123)

Qualcomm

- HTC Desire A56 (a56uhl-3.4.0)
- LG K8 ACG (AS375)
- ASUS Zenfone 2 Laser (ZE550KL / MR5-21.40.1220.1794)

Huawei

- Huawei Venus P9 Lite (2016-03-29)

Samsung

- Samsung Galaxy S7 Edge (SM-G935F_NN)

To ensure that we had a baseline comparison for Dr. Checker, we also analyzed these drivers using 4 popular open-source, and stable, static analysis tools (flawfinder [78], RATs [76], cppcheck [171], and Sparse [172]). We briefly describe our interactions with each below, and a summary of the number of warnings raised by each is shown in Table 4.2.

Flawfinder & RATs Both Flawfinder and RATs are pattern-matching-based tool used to identify potentially dangerous portions of C code. In our experience, the installation and usage of each was quite easy; they both installed without any configuration and used a simple command-line interface. However, the criteria that they used for their warnings tended to be very simplistic, missed complex bugs, and where overly general, which resulted in an extremely high number of warnings (64,823 from Flawfinder and 13,117 from RATs). For example, Flawfinder flagged a line of code with the warning, *High: fixed size local buffer*. However, after manual investigation it was clear this code was unreachable, as it was inside of an `#if 0` definition.

We also found numerous cases where the string-matching algorithm was overly general. For example, Flawfinder raised a critical warning (*[4] (shell) system*), incorrectly reporting that `system` was being invoked for the following define:

```
#define system_cluster(system, clusterid).
```

Ultimately, the tools seemed reasonable for basic code review passes, and perhaps for

less-security minded programs, as they do offer informational warning messages:

> *Flawfinder:* Statically-sized arrays can be improperly restricted, leading to potential overflows or other issues (CWE-119:CWE-120). Perform bounds checking, use functions that limit length, or ensure that the size is larger than the maximum possible length.
>
> *RATs:* Check buffer boundaries if calling this function in a loop and make sure you are not in danger of writing past the allocated space

Sparse Sparse was developed by Linus Torvalds and is specifically targeted to analyze kernel code. It is implemented as a compiler front end (enabled by the flag `C=2` during compilation) that raises warnings about known problems, and even allows developers to provide static type annotations (e.g., `__user` and `__kernel`). The tool was also relatively easily to use. Although, Sparse is good at finding annotation mis-matches like unsafe user pointer dereferences [147]. Its main drawback was the sheer number of warnings (64,823 in total) it generated, where most of the warnings generated were regarding non-compliance to good kernel code practices. For example, warnings like, "*warning: Using plain integer as NULL pointer*" and "*warning: symbol 'htc_smem_ram_addr' was not declared. Should it be static?*," were extremely common.

cppcheck Cppcheck was the most complicated to use of the tools that we evaluated, as it required manual identification of all of the includes, configurations, etc. in the source code. However, this knowledge of the source code structure did result in much more concise results. While the project is open-source, their analysis techniques are not well-documented. Nevertheless, it is clear that the tool can handle more complex interactions (e.g., macros, globals, and loops) than the other three. For example, in one of the raised warnings it reported an out-of-bounds index in an array lookup. Unfortunately, after

manual investigation there was a guard condition protecting the array access, but this was still a much more valuable warning that those returned by other tools. It was also able to identify an interesting use of `snprintf` on overlapped objects, which exhibits undefined behavior, and appeared generally useful. It also has a configurable engine, which allows users to specify additional types of vulnerability patterns to identify. Despite this functionality, it still failed to detect any of the complex bugs that Dr. Checker was able to help us discover.

To summarize our experience, we provide a side-by-side feature comparison of the evaluated tools and Dr. Checker in Table 4.3. Note that cppcheck and Dr. Checker where the only two with an extensible framework that can be used to add vulnerability detectors. Similarly, every tool aside from Sparse, which needs manual annotations, was more-or-less completely automated. As previously mentioned, Sparse's annotations are used to find unsafe user pointer dereferences, and while these annotations are used rigorously in the mainline kernel code, they are not always used in the vendor drivers. Moreover, typecasting is frequently used in Linux kernel making Sparse less effective. Pattern-based tools like flawfinder and RATS do not require compilable source code, which results in spurious warnings because of pre-processor directives making them unusable. Of the evaluated features, traceability of the warnings is potentially the most important for kernel bug-finding tools [173], as these warnings will ultimately be analyzed by a human. We consider a warning to be traceable if it includes all of the information required to understand how a user input can result in the warning. In Dr. Checker, we use the debug information embedded in the LLVM bitcode to provide traceable warnings.

Algorithm 1: Soundy driver traversal analysis

```
function SDTraversal((S, Δ, F))
    sccs ← topo_sort(CFG(F))
    forall scc ∈ sccs do
        if is_loop(scc) then
            HandleLoop(S, Δ, scc)
        else
            VisitSCC(S, Δ, scc)
        end
    end

function VisitSCC((S, Δ, scc))
    forall bb ∈ scc do
        forall I ∈ bb do
            if is_call(I) then
                HandleCall(S, Δ, I)
            else
                if is_ret(I) then
                    S ← S ∪ {φ_Δ(ret_val), τ_Δ(ret_val)}
                else
                    DispatchClients(S, Δ, I)
                end
            end
        end
    end

function HandleLoop((S, Δ, scc))
    num_runs ← LongestUseDefChain(scc)
    while num_runs ≠ 0 do
        VisitSCC(S, Δ, scc)
        num_runs ← num_runs − 1
    end

function HandleCall((S, Δ, I))
    if ¬is_visited(S, Δ, I) then
        targets ← resolve_call(I)
        forall f ∈ targets do
            Δ_new ← Δ||I
            φ_new ← (Δ_new → (φ_c(Δ)(args), φ_c(Δ)(globals)))
            τ_new ← (Δ_new → (τ_c(Δ)(args), τ_c(Δ)(globals)))
            S_new ← {φ_new, τ_new}
            SDTraversal(S_new, Δ_new, f)
        end
        mark_visited(S, Δ, I)
    end
```

Algorithm 2: Points-to analysis transfer functions

function `updatePtoAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $loc_x \leftarrow (x, \emptyset, \emptyset)$
  $map_{pt}(v) \leftarrow (0, loc_x)$
function `updatePtoBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_1 \leftarrow map_{pt}(op_1)$
  $pto_2 \leftarrow map_{pt}(op_2)$
  $set_1 \leftarrow \{(0, ob) \mid \forall(\_, ob) \in pto_1\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall(\_, ob) \in pto_2\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_1 \cup set_2$
function `updatePtoLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_1 \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_2 \leftarrow \{(0, ob) \mid \forall ob \in set_1\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_2$
function `updatePtoStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $pto_v \leftarrow map_{pt}(v)$
  $set_v \leftarrow \{ob \mid \forall(\_, ob) \in pto_v\}$
  $\forall(n, ob) \in pto_{op}$ **do** $ob(n) \leftarrow ob(n) \cup set_v$
function `updatePtoGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob(n) \mid \forall(n, ob) \in pto_{op}\}$
  $set_v \leftarrow \{(off, ob) \mid \forall ob \in set_{op}\}$
  $map_{pt}(v) \leftarrow map_{pt}(v) \cup set_v$

Algorithm 3: Taint analysis transfer functions

function `updateTaintAlloca` $(\phi_c, \tau_c, \delta, I, v, loc_x)$
  Nothing to do

function `updateTaintBinOp` $(\phi_c, \tau_c, \delta, I, v, op_1, op_2)$
  $map_t \leftarrow \tau_c(\delta)$
  $set_v \leftarrow map_t(op_1) \cup map_t(op_2)$
  $map_t(v) \leftarrow set_v || I$

function `updateTaintLoad` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $set_{op} \leftarrow \{ob_t(n)||I \mid \forall(n, ob) \in pto_{op}\}$
  $map_t \leftarrow \tau_c(\delta)$
  $map_t(v) \leftarrow map_t(v) \cup set_{op}$

function `updateTaintStore` $(\phi_c, \tau_c, \delta, I, v, op)$
  $map_{pt} \leftarrow \phi_c(\delta)$
  $pto_{op} \leftarrow map_{pt}(op)$
  $map_t \leftarrow \tau_c(\delta)$
  $tr_v \leftarrow map_t(v)$
  $\forall(n, ob) \in pto_{op}$ **do** $ob_t(n) \leftarrow ob_t(n) \cup (tr_v||I)$

function `updateTaintGEP` $(\phi_c, \tau_c, \delta, I, v, op, off)$
  updateTaintBinOp$(\phi_c, \tau_c, \delta, I, v, op, off)$

Listing 4.1: A buffer overflow bug detected in Mediatek's Accdet driver by Improper Tainted-Data Use Detector (ITDUD) where `buf` is assumed to be a single character but the use of "%s" will continue reading the buffer until a null-byte is found.

```
static char call_status;
...
static ssize_t
  accdet_store_call_state
  (struct device_driver *ddri,
  const char *buf, size_t count)
{
  // ** Improper use of tainted data **
  // buf can contain more than one char!
  int ret = sscanf(buf, "%s", &call_status);

  // The return value is checked, but it's too late
  if (ret != 1) {
    ACCDET_DEBUG("accdet: Invalid values\n");
    return -EINVAL;
  }

  switch (call_status) {
  case CALL_IDLE:
  ...
}
```

Listing 4.2: A zero-day vulnerability discovered by Dr. Checker in Mediatek's mlog driver using our Tainted Arithmetic Detector (TAD) and Tainted Loop Bound Detector (TLBD) analysis. First TAD identified an integer overflow bug (`len - MLOG_STR_LEN`). TLBD then identified that this tainted length was being used as a bound condition for the while loop where data is being copied into kernel space.

```
#define MLOG_STR_LEN          16
...
int mlog_doread(char __user *buf, size_t len)
{
  unsigned i;
  int error = -EINVAL;
  char mlog_str[MLOG_STR_LEN];
  ...
  // len is unsigned
  if (!buf || len < 0)
    goto out;
  error = 0;
  // len not checked against MLOG_STR_LEN
  if (!len)
    goto out;
  // buf of len confirmed to be in user space
  if (!access_ok(VERIFY_WRITE, buf, len)) {
    error = -EFAULT;
    goto out;
  }
  ...
  i = 0;
  ...
  // ** Integer underflow bug **
  // len - MLOG_STR_LEN (16) can be negative
  // and is compared with unsigned i
  while (!error && (mlog_start != mlog_end)
         && i < len - MLOG_STR_LEN) {
    int size;
    ...
    size = snprintf(mlog_str, MLOG_STR_LEN,
             strfmt_list[strfmt_idx++], v);
    ...
    // this function is an unsafe copy
    // this results in writing past buf
    // potentially into kernel address space
    if (__copy_to_user(buf, mlog_str, size))
      error = -EFAULT;
    else {
      buf += size;
      i += size;
    }
  }
}
```

Listing 4.3: An information leak bug via padded fields detected by our Uninit Leak Detector (ULD) in Mediatek's FM driver where a struct's memory is not sanitized before being copied back to user space leaking kernel stack data.

```
fm_s32 fm_get_aud_info(fm_audio_info_t *data)
{

  if (fm_low_ops.bi.get_aud_info) {
    return fm_low_ops.bi.get_aud_info(data);
  } else {
    data->aud_path = FM_AUD_ERR;
    data->i2s_info.mode = FM_I2S_MODE_ERR;
    data->i2s_info.status = FM_I2S_STATE_ERR;
    data->i2s_info.rate = FM_I2S_SR_ERR;
    return 0;
  }
}
...
  case FM_IOCTL_GET_AUDIO_INFO:
    fm_audio_info_t aud_data;
    // ** no memset of aud_data **
    // Not all fields of aud_data are initialized
    ret = fm_get_aud_info(&aud_data);
    if (ret) {
      WCN_DBG(FM_ERR|MAIN, "fm_get_aud_info err\n");
    }
    // Copying the struct results in data-leakage
    // from padding and uninitialized fields
    if (copy_to_user((void *)arg, &aud_data,
        sizeof(fm_audio_info_t))) {
      WCN_DBG(FM_ERR|MAIN, "copy_to_user error\n");
      ret = -EFAULT;
      goto out;
    }
    ...
```

Listing 4.4: An initialization of a file operations structure in the mlog driver of Mediatek

```
static const struct file_operations
proc_mlog_operations = {
  .owner = NULL,
  .llseek = NULL,
  .read = mlog_read,
  .poll = mlog_poll,
  .open = mlog_open,
  .release = mlog_release,
  .llseek = generic_file_llseek,
};
```

Listing 4.5: Example of tainting different arguments where `tc_client_ioctl` has a directly tainted argument and `iris_s_ext_ctrls`'s argument is indirectly tainted.

```
static long tc_client_ioctl(struct file *file,
        unsigned cmd, unsigned long arg) {
        ...
        char c=(char*)arg
        ...
}
static int iris_s_ext_ctrls(struct file *file,
  void *priv, struct v4l2_ext_controls *ctrl) {
      ...
  char *data = (ctrl->controls[0]).string;
      ...
  char curr_ch = data[0];
}
```

Table 4.1: Tainted arguments for each driver entry function type wether they are directly and indirectly tainted.

Entry Type	Argument(s)	Taint Type
Read (*File*)	`char *`*buf*, `size_t` *len*	Direct
Write (*File*)	`char *`*buf*, `size_t` *len*	Direct
Ioctl (*File*)	`long` *arg*	Direct
DevStore (*Attribute*)	`const char *`*buf*	Indirect
NetDevIoctl (*Socket*)	`struct *`*ifreq*	Indirect
V4Ioctl	`struct v4l2_format *`*f*	Indirect

Table 4.2: Summary of warnings produced by popular bug-finding tools on the various kernels that we analyzed.

Kernel	Number of Warnings			
	cppcheck	**flawfinder**	**RATS**	**Sparse**
Qualcomm	18	4,365	693	5,202
Samsung	22	8,173	2,244	1,726
Hauwei	34	18,132	2,301	11,230
Mediatek	168	14,230	3,730	13,771
	242	**44,900**	**8,968**	**31,929**

Table 4.3: Comparison of the features provided by popular bug-finding tools and Dr. Checker, where √ indicates availability of the feature.

Feature	cppcheck	flawfinder	RATS	Sparse	Dr. Checker
Extensible	√	-	-	-	√
Inter-procedural	-	-	-	-	√
Handles pointers	-	-	-	-	√
Kernel Specific	-	-	-	√	√
No Manual Annotations	√	√	√	-	√
Requires compilable sources	√	-	-	√	√
Sound	-	-	-	-	-
Traceable Warnings	-	-	-	√	√

Table 4.4: Summary of the bugs identified by DR. CHECKER in various mobile kernel drivers. We list the total number of warnings raised, number correct warnings, and number of bugs identified as a result.

	Warnings per Kernel (Count / Confirmed / Bug)				
Detector	**Huawei**	**Qualcomm**	**Mediatek**	**Samsung**	**Total**
TaintedSizeDetector	62 / 62 / 5	33 / 33 / 2	155 / 153 / 6	20 / 20 / 1	270 / 268 / 14
TaintedPointerDereferenceChecker	552 / 155 / 12	264 / 264 / 3	465 / 459 / 6	479 / 423 / 4	1760 / 1301 / 25
TaintedLoopBoundDetector	75 / 56 / 4	52 / 52 / 0	73 / 73 / 1	78 / 78 / 0	278 / 259 / 5
GlobalVariableRaceDetector	324 / 184 / 38	188 / 108 / 8	548 / 420 / 5	100 / 62 / 12	1160 / 774 / 63
ImproperTaintedDataUseDetector	81 / 74 / 5	92 / 91 / 3	243 / 241 / 9	135 / 134 / 4	551 / 540 / 21
IntegerOverflowDetector	250 / 177 / 6	196 / 196 / 2	247 / 247 / 6	99 / 87 / 2	792 / 707 / 16
KernelUninitMemoryLeakDetector	9 / 7 / 5	1 / 1 / 0	8 / 5 / 5	6 / 2 / 1	24 / 15 / 11
InvalidCastDetector	96 / 13 / 2	75 / 74 / 1	9 / 9 / 0	56 / 13 / 0	236 / 109 / 3
Total	1,449 / 728 / 78	901 / 819 / 19	1,748 / 1,607 / 44	973 / 819 / 24	**5,071 / 3,973 / 158**

4.6.1 Dr. Checker

The summarized results of all of the warnings that were reported by Dr. Checker are presented in Table 4.4. In this table, we consider a warning as *correct* if the report and trace were in fact true (e.g., a tainted variable was being used by a dangerous function). All of these warnings were manually verified by the authors, and those that are marked as a *bug* were confirmed to be critical zero-day bugs, which we are currently in the process of disclosing to the appropriate vendors. In fact, 7 of the 158 identified zero-days have already been issued Common Vulnerabilities and Exposures (CVE) identifiers [149, 150, 151, 152, 153]. Of these, Sparse correctly identified 1, flawfinder correctly identified 3, RATs identified 1 of the same ones as flawfinder, and cppcheck failed to identify any of them. These bugs ranged from simple data leakages to arbitrary code execution within the kernel. We find these results very promising, as 3,973 out of the 5,071 were confirmed, giving us a precision of 78%, which is easily within the acceptable 30% range [145].

While the overall detection rate of Dr. Checker is quite good (e.g., KernelUninitMemoryLeakDetector raised 24 warnings, which resulted in 11 zero-day bugs), there a few notable *lessons learned*. First, because our vulnerability detectors are stateless, they raise a warning for every occurrence of the vulnerable condition, which results in a lot of correlated warnings. For example, the code `i = tainted+2; j = i+1;` will raise two IntegerOverflowDetector warnings, once for each vulnerable condition. This was the main contributor to the huge gap between our confirmed warnings and the actual bugs as each bug was the result of multiple warnings. The over-reporting problem was amplified by our context-sensitive analysis. For example, if a function with a vulnerable condition is called multiple times from different contexts, Dr. Checker will raise one warning for each context.

GlobalVariableRaceDetector suffered from numerous false positives because of gran-

ularity of the LLVM instructions. As a result, the detector would raise a warning for any access to a global variable outside of a critical section. However, there are cases where the mutex object is stored in a structure field (e.g., `mutex_lock(&global->obj)`). This results in a false positive because our detector will raise a warning on the access to the global structure, despite the fact that it is completely safe, because the field inside of it is actually a mutex.

TaintedPointerDerefenceDetectors similarly struggled with the precision of its warnings. For example, on Huawei drivers (row 2, column 1), it raised 552 warnings, yet only 155 were true positives. This was due to the over-approximation of our points-to analysis. In fact, 327 of these are attributed to only two entry points `rpmsg_hisi_write` and `hifi_misc_ioctl`, where our analysis over-approximated a single field that was then repeatedly used in the function. A similar case happened for entry point `sc_v4l2_s_crop` in Samsung, which resulted in 21 false warnings. The same over-approximation of points-to affected InvalidCastDetector, with 2 entry points (`picolcd_debug_flash_read` and `picolcd_debug_flash_write`) resulting in 66 (80%) false positives in Huawei and a single entry point (`touchkey_fw_update.419`) accounting for a majority of the false positives in Samsung. IntegerOverflowDetector also suffered from over-approximation at times, with 30 false warnings in a single entry point `hifi_misc_ioctl` for Hauwei.

One notable takeaway from our evaluation was that while we expected to find numerous integer overflow bugs, we found them to be far more prevalent in 32 bit architectures than 64 bites, which is contrary to previously held beliefs[174]. Additionally, Dr. Checker was able to correctly identify the critical class of Boomerang [6] bugs that were recently discovered.

Table 4.5: Runtime comparison of 100 randomly selected entry points with our analysis implemented a "sound" analysis (*Sound*), a soundy analysis, without analyzing kernel functions (*No API*), and a soundy analysis without kernel functions or fixed-point loop analysis (Dr. Checker).

	Runtime (seconds)			
Analysis	**Avg.**	**Min.**	**Max.**	**St. Dev.**
Sound*	175.823	0.012	2261.468	527.244
No API	110.409	0.016	2996.036	455.325
Dr. Checker	35.320	0.008	978.300	146.238

* Only 18/100 sound analyses completed successfully.

4.6.2 Soundy Assumptions

Dr. Checker in total analyzed 1207 entry points and 90% of the entry points took less than 100 seconds to complete. Dr. Checker's practicality and scalability are made possible by our *soundy* assumptions. Specifically, not analyzing core kernel functions and not waiting for loops to converge to a fixed-point. We evaluate how these assumptions affected both our precision (i.e., practicality) and runtime (i.e., scalability). This analysis was done by randomly selecting 25 entry points from each of our codebases (i.e., Huawei, Qualcomm, Mediatek, and Samsung), resulting in 100 randomly selected driver entry points. We then removed our two soundy assumptions, resulting in a "sound" analysis, and ran our analysis again.

Kernel Functions Our assumption that all kernel functions are bug free and correctly implemented is critical for the efficacy of Dr. Checker for two reasons. First, the state explosion that results from analyzing all of the core kernel code makes much of our analysis computationally infeasible. Second, as previously mentioned, compiling the Linux kernel for ARM with LLVM is still an ongoing project, and thus would require a significant engineering effort [163]. In fact, in our evaluation we compiled the 100

randomly chosen entry with best-effort compilation using LLVM, where we created a consolidated bitcode file for each entry point with all the required kernel API functions, caveat those that LLVM failed to compile. We ran our "sound" analysis with these compiled API functions and evaluated all loops until both our points-to and taint analysis reached a fixed point, and increased our timeout window to *four hours* per entry point. Even with the potentially missing kernel API function definitions, only 18 of these 100 entry points finished within the 4 hours. The first row (*Sound*) in Table 4.5 shows the distribution of time over these 18 entry points. Moreover, these 18 entry points produced 63 warnings and took a total of 52 minutes to evaluate, compared to 9 warnings and less than 1 minute of evaluation time using our soundy analysis.

Fixed-point Loop Analysis Since we were unable to truly evaluate a *sound* analysis, we also evaluated our second assumption (i.e., using a reach-def loop analysis instead of a fixed-point analysis) in isolation to examine its impact on Dr. Checker. In this experiment, we ignored the kernel API functions (i.e., assume correct implementation), but evaluated all loops until they reached a fixed point on the same 100 entry points. In this case, all of the entry points were successfully analyzed within our four hour timeout window. The second row (*No API*) in Table 4.5 shows the distribution of evaluation times across these entry points. Note that this approach takes 3× more time than the Dr. Checker approach to analyze an entry point on average. Similarly, our soundy analysis returned significantly fewer warnings, 210 compared to the 474 warnings that were raised by this approach.

A summary of the execution times (i.e., sound, fixed-point loops, and Dr. Checker) can be found in Table 4.5, which shows that ignoring kernel API functions is the main contributor of the Dr. Checker's scalability. This is not surprising because almost all the kernel drivers themselves are written as kernel modules [175], which are small (7.3K

lines of code on average in the analyzed kernels) and self-contained.

4.7 Discussion

Although Dr. Checker is designed for Linux kernel drivers, the underlying techniques are generic enough to be applied to other code bases. Specifically, as shown in Chapter 4.6.1, ignoring external API functions (i.e., kernel functions) is the major contributor to the feasibility of Dr. Checker on the kernel drivers. Dr. Checker in principle can be applied to any code base, which is modular and has well-defined entry points (e.g., ImageMagick [176]). While our techniques are portable, some engineering effort is likely needed to change the detectors and setup the LLVM build environment. Specifically, to apply Dr. Checker, one needs to:

1. Identify the source files of the module, and compile them in to a consolidated bitcode file.

2. Identify the function names, which will serve as entry points.

3. Identify how the arguments to these functions are tainted.

We provided more in-depth documentation of how this would be done in practice on our website.

4.8 Just Finding Vulnerabilities is Not Enough!

As mentioned in Chapter 1, vendors frequently customize the open-source versions of the system software to suit their devices. The customizations are usually done on codebases that are maintained in different repositories (e.g., `forks`) separate from the main open-source repository. In order to ensure that vulnerability is fixed, the patch

for the vulnerability should propagate to all the codebases (or repositories) as soon as possible. Vulnerability databases such as the Common Vulnerabilities and Exposures (CVE) database were born to facilitate this process: project maintainers can take them as a reference to know which security-related patches they need to apply, without having to find them manually. Despite the existence of these databases, security patches still take a substantial amount of time to propagate to all the project forks [10, 11, 12, 13]. In the year 2016, the Android maintainers patched 76 publicly known vulnerabilities (i.e., CVEs) from the year 2014, two from 2013, and two from 2012, which means that 80 disclosed vulnerabilities remained unpatched in the Android code base for more than one year [14]. This shows that patch propagation is another important problem that needs to be addressed to ensure the security of smart devices. In the next chapter, we will should an automated technique that can help in automatically propagating patches to the related repositories.

Chapter 5

The problem of patch propagation

Given the existence of various vendors repositories for open source software, it is important that a security applied to the main repository is quickly propagated to all the vendors repositories. Common Vulnerabilities and Exposures (CVE) database were born to facilitate this process: project maintainers can take them as a reference to know which security-related patches they need to apply, without having to find them manually. When a security patch is available, maintainers have to "cherry-pick" the patch: That is, they have to understand the patch and its behavior, adapt it to their own code base, and finally ensure that the whole system, after applying the patch, still works as expected. Not surprisingly, this is a manual and resource-intensive process [177, 178]. As a result, changes in the main code base of a project are usually applied to the code of dependent software with a significant delay [179]: Android 10, for example, is based on Linux kernel 4.19, while the latest release of the Linux kernel is version 5.3.8 [180].

Recent work [181] shows that attackers who monitor source repositories often get a head start of weeks (and sometimes months) on targeting vulnerabilities prior to any public disclosure. Furthermore, as we will show in this study (Chapter 5.6.4), it is possible that the maintainers of a project underestimate the severity of a patched bug, and fail to

request a corresponding entry in a vulnerability database (a CVE ID) [182]. When this happens, maintainers of related projects are not aware that a patch actually addresses a security problem. This is a growing problem, as exemplified by the recent VLC security issue [183], which is caused because developers of `libebml` failed to associate the corresponding security fix with a CVE ID [184], and the vulnerability existed for nearly two years after the fix was available. Unfortunately, hackers are known to scan source repository commits for fixes that might address vulnerabilities, and then check for the presence of these vulnerabilities in related repositories [185]. Therefore, the security fixes lacking a CVE ID provide a potential source of unfixed vulnerabilities as they are most likely not ported to related repositories.

Existing approaches that ease the process of cherry-picking relevant patches rely on commit-related information, such as code *diff* or commit messages [15, 16, 17], or they look for specific patterns [18]. These tools have the advantage of being fast, lightweight, scalable, and suitable to be used on large codebases. However, either they only match simple patches, or analyze commit messages, which are often not expressive enough to convey the scope and effect of a change [19, 20, 21]. Other techniques attempt to go a step further and analyze the semantic differences introduced by a patch using static analysis [22, 23, 24, 25, 26] and symbolic execution [27, 28, 29, 30]. Unfortunately, these techniques suffer from scalability issues.

An ideal solution, which would help maintainers in selecting and applying important changes, would be a system that is capable of identifying those patches that do not affect the intended functionality of the software. If the intended functionality of the software is not changed by a patch, this patch can be applied without the need for testing: we call these changes *safe patches*. In this paper, we argue that a significant portion of all security-related fixes falls under the category of safe patches [**?**]. Thus, a tool that can identify safe patches could be used to monitor the main repository and automatically

alert or apply this kind of patches on a target forked repository.
To be effective and usable on large codebases, a system to identify safe patches should at least satisfy the following requirements:

- **R1**: Only rely on the original and patched versions of the modified source code file, without any other additional information (e.g., commit message, build environment, etc.)
- **R2**: Be fast, lightweight and scalable.

We present the design and implementation of a static analysis approach that aims to identify safe patches and that satisfies both the requirements above. Our approach is designed specifically to target source code changes and to identify patches that could be applied with minimal testing, as they do not modify the program's functionality. Specifically, we make the following contributions:

- We provide the first formal definition of safe patches, and design a general technique to identify them.
- We implement SPIDER, a system based on this technique, that takes as input only the source code of the original and patched file.
- We evaluate SPIDER on 341,767 commits taken from 32 source code repositories (Linux kernel repositories, Android kernel repositories, interpreters, firmware, utilities and various other repositories), as well as on 809 CVE patches.
- We identify 67,408 safe patches and show that SPIDER could help developers in the process of selecting and testing changes, resulting in a speed-up in the propagation of security fixes.

- We also provide the Security Patch mode of SPIDER that can precisely identify security patches. It identified 2,278 patches that most likely fix security vulnerabilities, despite the fact that they were not associated with any CVE entry. 229 of these issues are still unpatched in several kernel forks. As such, they can be considered unfixed vulnerabilities.

Unlike previous work, our approach is the first that focuses on determining those patches that can be propagated to related projects with minimal effort, and without defining *a priori* specific types of changes or semantic characteristics that should be detected (i.e., we do not just target patches that fix a specific type of vulnerability). We envision our system to be part of the recently introduced Github security alerts [186], or it could be used to build a variant of the *git rebase* feature that suggests patches that are most likely safe and should be prioritized.

5.1 What are Safe Patches?

Our goal is to identify patches that can be applied without subsequent testing. We call such patches *safe patches* (*sp*s). Intuitively, for a patch to be considered an *sp*, it should satisfy the following two conditions:

- **Non-increasing input space (C1)**: The patch *should not* increase the valid input space of the program. That is, the patched version should be more restrictive in the inputs that it accepts. The assumption is that some of inputs that the original program accepted resulted in security violations, and the patched version "removes" these inputs as invalid.

- **Output equivalence (C2)**: For all the valid inputs that the patched program accepts, the output of the patched program must be the same as that of the original

program.

The condition $C1$ ensures that there is *no need to add new test cases*, as there are no new inputs that are accepted by the patched program [1]. Furthermore, the condition $C2$ ensures that there is *no need to run the existing test cases* as the output will be the same as that of the original program (for all the valid inputs). Consequently, if a patch satisfies the above two conditions then it can be applied without any effect on the existing test cases. Of course, the purpose of testing is to ensure that the program behaves as expected, so it is always a recommended step after applying a patch. In Chapter 5.1.2, we define more formally the two conditions above.

5.1.1 Running Example

Listing 5.1 shows our running example, a C language example of a safe patch in the unified *diff* format (i.e., where + and − indicate inserted and deleted lines, respectively). In this example, the programmer decided that it was necessary to add an extra length check (Lines 3-5), presumably to protect against a buffer overflow later in the program. In addition, the patch also includes the length of the header (HDR) as part of a size check in Line 10.

This patch is safe. The inserted modifications to the variables `len` and `tlen` do not change the output of the function. Moreover, the extra conditional statement in Line 3 adds a missing length check, thereby restricting the input space. That is, all inputs where `t->len` is larger than `MAX_LEN` now lead to the function returning an error, while those inputs were accepted by the original function.

Figure 5.1 shows the control flow graph (CFG) after the application of this example

[1] However, for regression testing purposes, one may want to add a test case that checks that the inputs are indeed invalid and the corresponding security flaw is patched.

```
1  long get_read_size(struct dring *t) {
2      long len, tlen;
3  +   if(t->len > MAX_LEN) {
4  +       return -1;
5  +   }
6      ...
7  -   len = t->len;
8  +   len = t->len + 4;
9      ...
10     if(len % 2) {
11         len += DEF_SIZE;
12     }
13     ...
14 -   tlen = len;
15 +   tlen = len - 4;
16     ...
17     t->total = tlen;
18     ...
19     return tlen;
20 }
```

Listing 5.1: Running Example of a safe patch.

patch: underlined text indicates the pieces of code inserted, while the left (blue) and right (red) children of each basic block are the true and false branches, respectively.

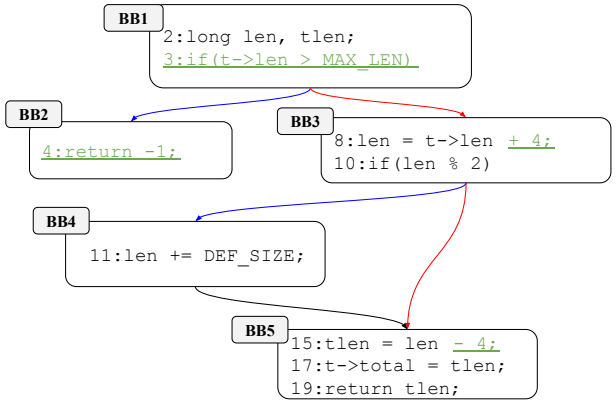

Figure 5.1: Control flow graph of the patched program from Listing 5.1

5.1.2 Formal Definition

We first define terminology used throughout the paper:

- *Input* i to a program: The input data with which the program is executed; I

indicates the set of all the possible inputs to the program.

- *Function of a program*: The symbol f denotes the original function, and any subscript to it identifies its patched version. For example: f_p indicates the function f after applying the patch p.

- *Error-handling basic blocks*: The symbol BB_{err} denotes the basic blocks of a function that are part of its error-handling functionality. In Figure 5.1, BB_2 is an error-handling basic block. We will explain later how error-handling basic blocks are identified.

- We use the notation $i \hookrightarrow f$ to indicate that input i *successfully executes* through function f. That is, starting from the entry basic block of f, and given input i, none of the error-handling basic blocks (BB_{err}s) of f will be reached. In other words, i represent a valid input to the function f.

- *Output of a function*: The *output* of a function f is its return value and all the externally visible changes to the program's data. Specifically, the output includes the return value, all writes to heap and global variables, and the arguments to all function calls. For instance, the output of the function `get_read_size` in Listing 5.1 is its return value (line 19), and the value written to the pointer variable `t->total` (line 17). Furthermore, $output(i, f)$ indicates the output of the function f when run with input i.

Now, we will use the definitions we introduced to formally define two conditions ($C1$ and $C2$) introduced at the beginning of Chapter 5.1.

Non-increasing input space (C1)

The non-increasing input space (C1) condition requires that the patched program does not accept any inputs as valid that are not also accepted as valid by the original program. This condition can be defined at the granularity of functions; that is, for C1 to hold, we require that all patched functions, individually, do not accept any additional valid inputs. In other words, any valid input to a patched function must also be a valid input to the corresponding original function. More formally:

$$\forall i \in I \mid (i \hookrightarrow f_p) \rightarrow (i \hookrightarrow f). \tag{5.1}$$

In the case of Listing 5.1, the patch restricts the original input space by adding an additional constraint (i.e., $t \rightarrow len > MAX\_LEN$ in Line 3). As a result, all valid inputs to the patched function are also valid inputs to the original function (but not vice versa). This satisfies Equation 5.1.

Output correspondence (C2)

The output correspondence (C2) condition requires that, for all valid inputs, the output of the patched program must be the same as the output of the original program. This condition, again, can be defined at the function granularity: For each patched function, for all corresponding valid inputs, the patched function must produce the same outputs as the original function. More formally:

$$\forall i \in I \mid (i \hookrightarrow f_p) \rightarrow (output(i, f_p) = output(i, f)). \tag{5.2}$$

In the case of Listing 5.1, although the patch inserts changes that modify the values of some variables (for example, `len`), the changes do not affect the externally visible data of the program, and thus, they do not change the output of the function, thereby satisfying Equation 5.2.

If all the patched functions satisfy both Equation 5.1 and Equation 5.2, then we can say that the patch satisfies the conditions $C1$ and $C2$. As a result, the patch can be considered as a *safe patch (sp)*. Note that, as a trivial case, an empty patch ($f_p = f$) satisfies Equations 1 and 2, making it an *sp*. Furthermore, there exist patches that do not satisfy the above conditions but still could be applied without testing, making our conditions sufficient but not necessary. There are also other shortcomings of our formal definition.

5.1.3 Shortcomings of the *sp* formalism

First, a patch that removes all the functionality as shown in Listing 5.2 is an *sp*. This is because none of the inputs *execute* through the program or in other words all the inputs will end up in an error basic block. Equation 5.1 and 5.2 trivially hold as for all the inputs ($i \hookrightarrow f_p$) evaluates to `false`.

```
int main(int argc, char **argv) {
+    if(argc > 0) {
+     return -1;
+    }
...
}
```

Listing 5.2: a patch that removes all the functionalities

```
int decrypt(..) {
     char secret_buff[4096];
...
...
-  memset(secret_buff, 0, sizeof(secret_buff));
}
```

Listing 5.3: an optimizing patch that may induce a security vulnerability.

Second, we ignore all the updates to local variables as they are not part of the function output. Consider the patch as shown in Listing 5.3, which removes a seemingly useless `memset`. This is valid and commonly known as dead-store elimination [187]. However, on closer inspection, one can recognize that the `memset` may be required as it would potentially clean up some secret data to avoid information leaks. Our current definition

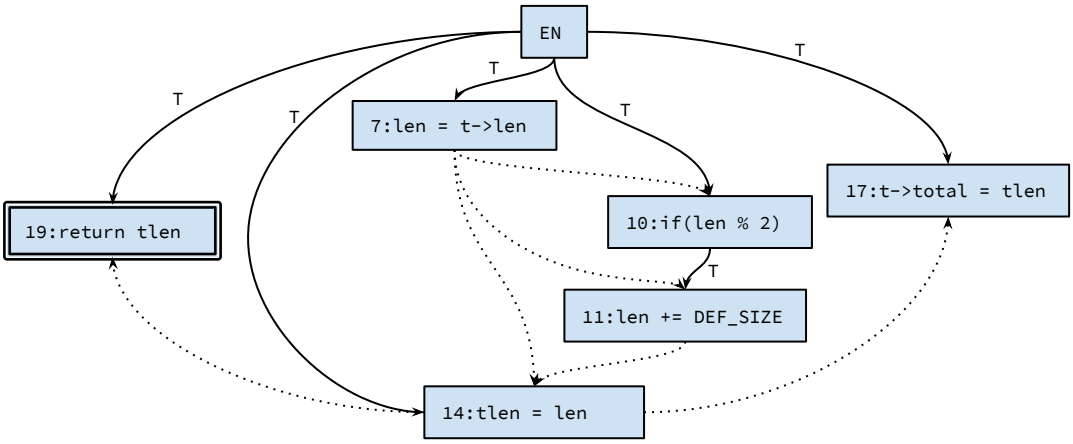

Figure 5.2: *PDG* of the *original* function in Listing 5.1

of *sp* does not handle these cases. Nonetheless, we believe that our definition provides a reasonable approach to identify safe patches.

5.2 Identifying Safe Patches

Lets first look at a possible way to determine whether a given patch is an *sp* based on our formal definition.

5.2.1 Program Dependency Graph (*PDG*)

Our technique leverages the concept of a program dependency graph (PDG). A program dependency graph [188] captures both data and control dependencies in a single graph. Formally, the *PDG* of a function f, denoted as $PDG(f) = (V, C, D)$, is a directed graph where

- $V = \{v_0, v_1, ..., v_n, E_n\}$ is a set of nodes, one for each *instruction* ($v_\bullet$) of the function. The additional node, E_n represents the function entry.

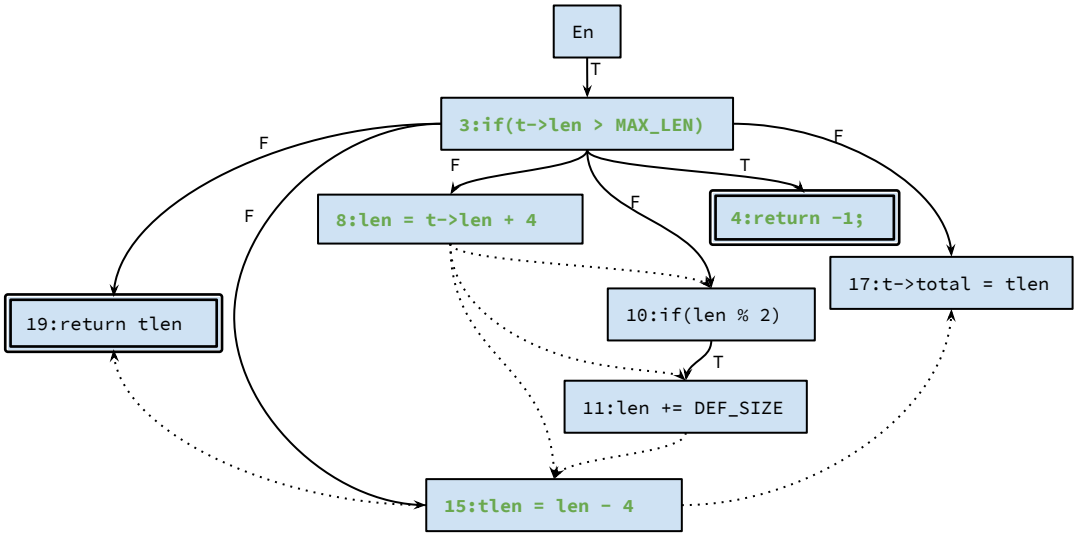

Figure 5.3: *PDG* of the *patched* function in Listing 5.1

- C is a set of directed, labeled edges, where each edge $(v_i, v_j, T|F)$ represents the (direct) control dependency of v_j on v_i. An instruction v_j is control-dependent on v_i, and the edge is labeled as *true* (T) [or *false* (F)], when v_j is executed if and only if v_i evaluates to true [or *false*]. To complete the PDG, if an instruction v is not control-dependent on any other instruction in the function (in other words, it does not have any incoming control flow edges), we connect it to the function entry node (E_n). That is, we add the edge (E_n, v, T) to C. Note that all source nodes of control-flow edges are either conditional statements (`if`, `while`, etc.) or the function entry node E_n. In addition, all conditional statements will have at least one outgoing control-dependency edge.

- D is a set of directed edges, where each edge (v_i, v_j) represents a data dependency. That is, instruction v_i defines a variable that can reach the corresponding use in instruction v_j.

For our running example in Listing 5.1, Figures 5.2 and 5.3 show the program depen-

dency graphs for the original and the patched function, respectively. The labels on the control dependency edges [$true(T)$ or $false(F)$] indicate whether the destination node is reachable from the source node via the *true* or *false* branch.

Control dependency versus control flow: The concept of control dependency is different from the more commonly-used concept of control flow. Control flow captures *possible flows* of execution, while control dependency captures the *necessary conditions* that must hold for the execution to reach a particular statement. Consider the *PDG* of the patched function in Figure 5.3. We can see a control dependency edge from the node (that corresponds to the instruction) at Line 3 to Line 15 with label F. This means that the condition at Line 3 must evaluate to `false` for the execution to reach Line 15. This is correct because if the condition at Line 3 evaluates to *true*, then the execution will immediately return from the function (Line 4). On the other hand, consider the control-flow graph of the patched function in Figure 5.1. There is no direct edge from $BB1$ (that contains Line 3) to $BB5$ (that contains Line 15). This is because the execution does not flow directly from $BB1$ to $BB5$ as there are other instructions in between (in $BB3$ and $BB4$).

Control-Dependency Path: Given a $PDG(f) = (V, D, C)$ of a function f, we say that a control dependency path exists from instruction $x \in V$ to instruction $y \in V$, denoted as $x \mapsto_c y$, if there exists a path in the *PDG* from x to y that only follows control-dependency edges. Formally,

$$x \mapsto_c y = \{< x, v_1, v_2, ..., v_n, y > |\ v_\bullet \in V \wedge (x, v_1, \bullet) \in C \wedge$$
$$(v_n, y, \bullet) \in C \wedge \forall_{1 \leq i < n} (v_i, v_{i+1}, \bullet) \in C\}.$$

In the *PDG* shown in Figure 5.3, there exists a control-dependency path (a path along solid edges) from the instruction at Line 3 to the instruction at Line 11: $\{3, 10, 11\}$.

Path Constraint (PC): For any instruction v, the condition derived from the sequence

of nodes and edges (with their labels) along the control-dependency path from the function entry E_n to v is called its *path constraint*. For example, consider the instruction at Line 11 in the *PDG* in Figure 5.3. The control-dependency path from E_n to Line 11 is $\{E_n, 3, 10, 11\}$. The corresponding path constraint is $PC(E_n \mapsto_c 11) = ((E_n == T) \wedge (\texttt{(t->len > MAX\_LEN)} == F) \wedge ((\texttt{(len \% 2)} \neq 0) == T))$. That is, Line 11 is only executed if `(t->len <= MAX_LEN)` and `(len % 2)` $\neq$ `0`.

Data-Dependency Path: Given a $PDG(f) = (V, D, C)$ of a function f, we say that a data-dependency path exists from instruction $x \in V$ to instruction $y \in V$, denoted as $x \mapsto_d y$, if there exists a path in the *PDG* from x to y that only follows data-dependency edges. More formally:

$$x \mapsto_d y = \{< x, v_1, v_2, ..., v_n, y > |\ v_\bullet \in V \wedge (x, v_1) \in D \wedge$$
$$(v_n, y) \in D \wedge \forall_{1 \leq i < n} (v_i, v_{i+1}) \in D\}.$$

In the *PDG* shown in Figure 5.3, there exists a data-dependency path (a path along dotted edges) from instruction at Line 8 to the instruction at Line 19: $\{8, 15, 19\}$. We say that a given data-dependency path $x \mapsto_d y =< x, v_1, v_2, ..., v_n, y >$ is *complete* if there is no data dependency path to x. Formally, $(\bullet, x) \not\in D$. The data dependency path example from Line 8 to 19 is complete as there is no data dependency path to Line 8.

Also, note that although a data dependency path exists from the instruction at Line 8 to instruction at Line 19, there is no control-dependency path between these instructions. This is because the execution of the instruction at Line 19 is not controlled by the instruction at Line 8.

5.2.2 The SPIDER Approach

Our system is given as input a patch p, with f and f_p being a function before and after applying the patch, respectively. The technique to detect whether p is a safe patch

works in four steps, as outlined below.

Checking modified instructions

We first need to identify what statements are affected by a patch, and determine whether these modifications can be soundly analyzed given our requirement $R1$. Recall that $R1$ requires that the analysis operates directly on the original and patched versions of the modified source code file, without any other additional information (e.g., commit message, build environment, etc.).

Affected Statements: A statement can be affected either *directly* or *indirectly* by the patch. We call a statement *directly affected* if it is modified, inserted, deleted, or moved by the patch. A statement is *indirectly affected* if it is either *control-* or *data-* dependent on any of the directly affected statements. Given the set of directly affected statements A_d and the *PDG* of the corresponding function, all the instructions *reachable* from the statements in A_d, either through control flow or data flow edges, are indirectly affected.

Consider the patch for our running example in Listing 5.1. Here, the directly affected statements are at Lines 3, 4, 8, and 15. However, looking at the corresponding *PDG* in Figure 5.3, we can see that all instructions are reachable from the node that corresponds to the instruction at Line 3. Consequently, all statements are affected by the patch.

Locally analyzable statement: We call a directly affected statement *locally analyzable* if all the writes made by the statement can be captured *without* any interprocedural and pointer analysis. Specifically, the modifications made by the patch should not involve any new function calls or pointer manipulation. Consider the patch represented by Listing 5.4: The inserted statement at Line 4 is locally analyzable. However, the inserted statement at Line 5 is not locally analyzable, because it involves a new function call.

If a patch has any directly affected statements that are not locally analyzable, we

do not consider it an *sp*. This is because we cannot soundly analyze the affected statements without analyzing the effects on the whole program. Moreover, performing whole-program analysis requires a static analysis tool (like LLVM), which in turn, requires access to the sources of the entire program, violating our requirement $R1$.

```
1  int kthread_init() {
2      ...
3  -   total_size = file->size;
4  +   total_size = header + file->size;
5  +   init_cleanup();
6      ...
7      if(total_size > MAX_SIZE) {
8          ...
9      }
10     ...
11 }
```

Listing 5.4: Patch illustrating locally analyzable statements.

Error-handling basic blocks

In the next step, we need to identify all the error-handling basic blocks (BB_{err}s) in f and f_p, so that all the changes to the statements within BB_{err}s are discarded and not considered in the next steps. This decision is based on the assumption that any changes to error basic blocks do not disrupt the original functionality (i.e., they just result in better or adjusted error-handling). The remaining statements affected by p are then analyzed to check if Equations 1 and 2 can be proved. We leverage previous work [189, 190] to identify error-handling basic blocks, as discussed in more detail in Chapter 5.3.4.

Non-increasing input space (C1)

To verify the non-increasing input space condition (C1), we need to ensure that the patch does not accept more inputs than the original function. In other words, the patch must not increase the valid input space for the modified function.

Intuitively, if a patch does not affect any control-flow statements (such as `if`, `while`, `for`, etc.), then it cannot change the input space of the function. However, if a patch

affects one or more control-flow statements, we must verify that no additional inputs can successfully execute through the function.

This can be done by first identifying the valid exit points (*VEP*) of a function. The valid exit points of a function are those instructions that, if reached during the execution of an input, imply that the input successfully executed through the function. For instance, in the case of our running example in Listing 5.1, the `return tlen` instruction at Line 19 is a valid exit point.

We consider all `return` statements as possible valid exit points. However, a function might exit because of an error (for instance, Line 4 in Listing 5.1), and the corresponding return statement does not represent a valid exit point. Hence, to identify the *VEP* set, we need to filter out all the return statements that are part of error basic blocks (BB_{err}).

In summary, to identify the *VEP* set of a function f with $PDG(f) = (V, D, C)$, we need to find all the exit points of f, i.e., $E_x(f)$, and filter out all the return instructions that belong to error basic blocks. More formally:

$$VEP(f) = \{r \mid ((r \in E_x(f)) \wedge (BB(r) \not\in BB_{errs}(f)))\}.$$

where $BB(r)$ indicates the basic block of instruction r.

To ensure that a patch satisfies condition $C1$, we need to verify that all inputs that go through the valid exit points, i.e., *VEP* of the patched function f_p, also go through the valid exit points of the original function f.

We observe that, in order for an input i to be successfully executed by a function, the input must satisfy the path constraint (PC) of a valid exit point. Thus, all the inputs that are accepted as valid by a function, which we denote as $vinputs(f)$, are the union of all the inputs that satisfy the path constraints for *at least* one valid exit point. More formally, the constraints on the inputs that are successfully executed by the function f

are captured by the following disjunction:

$$vinputs(f) = \bigvee_{i \in VEP(f)} (PC(i)). \tag{5.3}$$

If we have $vinputs(f_p) \rightarrow vinputs(f)$, which shows that all the inputs that can be successfully executed by the patched function f_p are also successfully executed by the original function f, we have succeeded in proving condition $C1$.

For our running example in Listing 5.1, with the PDG of the patched function in Figure 5.3, the valid exit point is at Line 19 (`return tlen`). By following the solid edges backwards and computing the path constraints for the patched function, we obtain $vinputs(f_p) = ((E_n == T) \wedge (\texttt{t->len > MAX\_LEN} == F))$. For the original function, whose PDG is in Figure 5.2, we obtain $vinputs(f) = (E_n == T)$. We can easily see that $vinputs(f_p) \rightarrow vinputs(f)$, thus satisfying $C1$.

To perform this step, we use symbolic interpretation to convert the C language statements into symbolic expressions (as discussed in more detail below). Then, we prove the implications between the two symbolic expressions using a SAT solver [191] (more details are provided in Chapter 5.3.5).

Output equivalence (C2)

To verify the output equivalence condition (C2), we need to verify that all externally visible changes (as described in Chapter 5.1.2) in the patched function are the same as that of the original one. Specifically, we want to ensure that for any input that successfully executes through the patched *and* original function, the output of the two functions will be identical.

We first look at all the affected (non-control-flow) statements. First, we discard all the statements that modify local variables. While local variables can have an indirect

effect on a function's output (which we take into account, as explained below), the local variables themselves are not externally visible. Thus, we do not need to consider them in this step. In the next step, we need to verify that all the updates (writes) to non-local (global and pointer) variables, function call arguments, and return values in the patched function are the same as that of the original function. In other words, we aim to prove that all global and pointer variables have the same values after the patched function has executed (compared to the original function), the patched function returns the same value, and it calls the same functions with the same arguments (and in the same order). When we are able to prove this, we are sure that, for every valid input, the patch does not change the externally visible effect of executing this function.

Given a statement t, we need to show that for all (valid) inputs that reach t in the patched and the original function, their outputs will be the same. More formally:

$$\forall i \in I \mid (i \hookrightarrow t_p) \rightarrow (output(t_p) = output(t))$$

The output value of a statement depends on the values of the inputs (input variables). Consider, for example, the statement `c = a + b`. Here, the output is assigned to the variable `c`, and the value depends on the inputs `a` and `b`. We can determine where these inputs come from by looking at the data-dependency graph for the statement. Of course, the inputs for a statement could come from multiple data-dependency paths. Consider again the *PDG* in Figure 5.3. For the statement at Line 15, there are two complete data dependency paths: $< 8, 15 >$ and $< 8, 11, 15 >$. The execution can take two different paths to reach this line, based on whether the function input satisfies the constraint on Line 10 or not.

For a given statement t, and for each data-dependency path to this statement, we compute a symbolic expression for the possible output values (along these paths). The idea is that the union of the symbolic expressions (overall data-dependency paths) for

Current Statement	Symbolic State	
	Input	**Output**
For path: $< 8, 15, 17 >$ starting with initial state		
`8: len = t->len + 4`	`len = sym1, t->len = sym2, MAX_LEN = sym3` `tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4`
`15: tlen = len - 4`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3` `tlen = sym5, DEF_SIZE = sym5, t->total = sym6`	`tlen = sym2`
`17: t->total = tlen`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3` `tlen = sym2, DEF_SIZE = sym5, t->total = sym6`	`t->total = sym2`
For path: $< 8, 11, 15, 17 >$ starting with initial state		
`8: len = t->len + 4`	`len = sym1, t->len = sym2, MAX_LEN = sym3` `tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4`
`11: len += DEF_SIZE`	`len = sym2 + 4, t->len = sym2, MAX_LEN = sym3` `tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`len = sym2 + 4 + sym5`
`15: tlen = len - 4`	`len = sym2 + 4 + sym5, t->len = sym2, MAX_LEN = sym3` `tlen = sym4, DEF_SIZE = sym5, t->total = sym6`	`tlen = sym2 + sym5`
`17: t->total = tlen`	`len = sym2 + 4 + sym5, t->len = sym2, MAX_LEN = sym3` `tlen = sym2 + sym5, DEF_SIZE = sym5, t->total = sym6`	`t->total = sym2 + sym5`

Table 5.1: Symbolic interpretation of the data-dependency path $< 8, 15, 17 >$ and $< 8, 11, 15, 17 >$ of the *PDG* in Figure 5.3.

t are the same for the patched function as for the original one. While this intuitively makes sense, there is one additional consideration. It is not enough to ensure that just the symbolic expressions are the same; they need to be the same under the same path constraints. Thus, we need to extend the symbolic expressions with their corresponding path constraints. We refer to these extended symbolic expressions as *symbolic output-constraint pairs*, which are computed as described hereinafter:

For a given statement t in a function f and the corresponding $PDG(f) = (V, D, C)$, we can compute the output-constraint pairs from *all* the complete data dependency paths to t. For each such path, we compute an output-constraint pair as:

$$\Psi_s = (interpret(< x_1, x_2, ..., x_n, t >), \bigwedge_{1 \leq i \leq n} PC(x_i)).$$

where *interpret* represents the symbolic expression that is computed by interpreting each of the instructions in sequence, and $PC(\bullet)$ is the path constraint of the corresponding instruction in the PDG.

Let $\Psi_p(t)$ and $\Psi(t)$ be the symbolic output-constraint pairs for the statement t in the patched and original function, respectively. We say that the output of statement t is equivalent in the original and the patched function, denoted as $\Psi_p(v) \equiv \Psi(v)$, if the following equation holds:

$$\forall (o_x, c_x) \in \Psi_p(v) \cdot \exists (o_y, c_y) \in \Psi(v) \vdash (o_x == o_y) \wedge (c_x \rightarrow c_y). \tag{5.4}$$

Note that $o_\bullet$ are not concrete values but rather symbolic values.

It is possible that there is an infinite number of data dependency paths that lead to a statement. This happens when there are loops or cyclic dependencies in the data dependency graph (for example, when a value is updated inside the body of a loop and later used by an affected statement). We will show in Chapter 5.3.5 how we resolve cycles in the data dependency graph. We will further argue that our approach is safe for a subset of instances, and we only consider these cases as safe patches.

Consider how we verify that condition C2 holds for our running example in Listing 5.1: The affected statements are at Lines 3, 8, 10, 11, 15, 17, and 19. Recall that we only consider non-control-flow statements. Thus, we can remove Line 3 and 10 from further consideration. Next, we can discard all statements that write to local variables, which removes Lines 8, 11, and 15. We end up with the statements at Lines 17 and 19, which write to a non-local variable through a pointer and return a value, respectively.

Looking at the *PDG* for the patched function (in Figure 5.3), we see that there exist two complete data dependency paths for Line 17: $< 8, 15, 17 >$ and $< 8, 11, 15, 17 >$. The symbolic interpretation steps for both paths is shown in Table 5.1. For every path, we first initialize each of the variables with a unique symbol, and then start interpreting each instruction according to its semantics. The symbolic output with corresponding path constraints along the path $< 8, 15, 17 >$ is $(o_p^1, c_p^1) = (\texttt{t->total} = sym2, ((En == T) \wedge ((sym2 > sym3) == F) \wedge ((((sym2+4)\%2) \neq 0) == F)))$. For the path $< 8, 11, 15, 17 >$, the result is $(o_p^2, c_p^2) = (\texttt{t->total} = sym2 + sym5, ((En == T) \wedge ((sym2 > sym3) == F) \wedge ((((sym2 + 4)\%2) \neq 0) == T)))$.

For interpreting the original function, we start with the same initial symbols for the same variables that were used in the patched function. From the original function's *PDG* in Figure 5.2, for Line 17, there are also two data dependency paths: $< 7, 14, 17 >$ and $< 7, 11, 14, 17 >$. The symbolic output along with the corresponding path constraints are $(o_c^1, c_c^1) = (\texttt{t->total} = sym2, ((En == T) \wedge (((sym2\%2) \neq 0) == F)))$ and $(o_c^2, c_c^2) = (\texttt{t->total} = sym2 + sym5, ((En == T) \wedge (((sym2\%2) \neq 0) == T)))$.

We can see that $o_p^1 == o_c^1 \wedge c_p^1 \rightarrow c_c^1$ and $o_p^2 == o_c^2 \wedge c_p^2 \rightarrow c_c^2$. Hence, Equation 5.4 holds.

Similarly, we can show that the output at Line 19 is equivalent in both the patched and the original function. As a result, our system has verified that the patch satisfies condition $C2$, and the patch is safe. For a patch that affects multiple functions, the steps

described above are performed for each function.

5.3 SPIDER: Design and Implementation

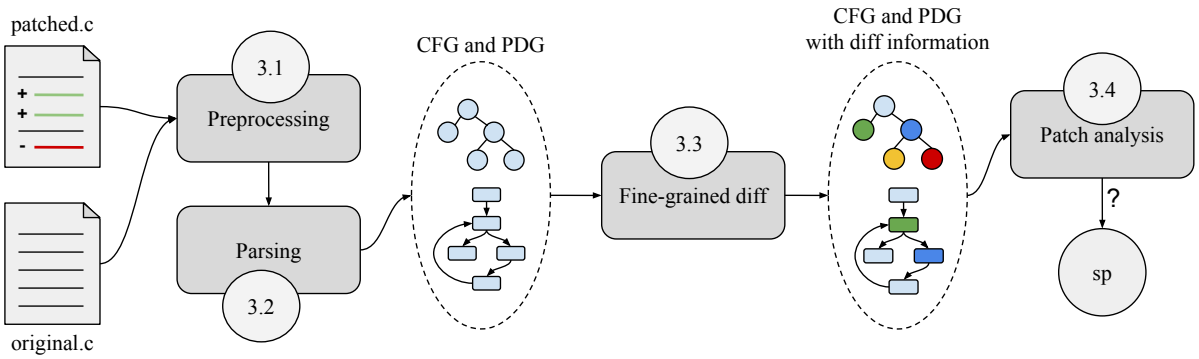

Figure 5.4: Flow of the actions performed by SPIDER.

In this chapter we show the details of SPIDER, a tool, that satisfies our requirements $R1$ and $R2$, uses the approach described in Chapter 5.2 to analyze a given C source code patch and determine if it is an *sp*. The Figure 6.1 shows the various steps of SPIDER when analyzing a patch.

5.3.1 Preprocessing

SPIDER starts by handling the C preprocessor directives. File inclusions (i.e., `#include`) are ignored, since as a requirement we do not want to collect information outside of the two input source code files. Macro definitions are ignored as well: macro calls will be treated as regular function calls, as explained later. The system then uses the *unifdef*[1] tool to handle conditional code inclusion directives (e.g., `#ifdef`, `#ifndef`, etc.): the output of *unifdef* is a valid C source file, without any of these constructs. Note that this

step could exclude certain code segments. Chapter 5.4 explains this in detail. This first step outputs two C source files ready to be parsed.

5.3.2 Parsing

The preprocessed source files are parsed using the Joern [75] fuzzy parser, which provides an Abstract Syntax Tree (AST) for all the functions in the file. Although Joern also provides a Control Flow Graph (CFG), with nodes linked to the ones in the AST, we had to modify it to suite our needs. Specifically, we had to implement the reaching definitions analysis [156], simple type inference [192], control dependency analysis [193], and, finally, program dependency graph [188]. At the end of this phase, SPIDER has access to the AST, CFG, and PDG for each of the functions affected by the patch.

5.3.3 Fine-grained diff

SPIDER uses function names to pair the functions in the original file with the corresponding ones in the new files, assuming patches that insert, delete, or rename one or more functions not to be *sp*s. SPIDER then identifies the functions affected by the patch using *java-diff-utils*[2], a common text *diff* tool. Our system then applies a state-of-the-art AST diffing technique, Gumtree [194], between the original and patched ASTs of the affected functions. Gumtree maps the nodes in the old AST with the corresponding nodes in the new one and identifies nodes that have been moved, inserted, deleted, or updated. A moved node is a node that the patch moved in another position in the AST, but whose content was unchanged, while an updated node is a non-moved node whose content was changed. The differences in the ASTs are also associated to the corresponding nodes in the CFG.

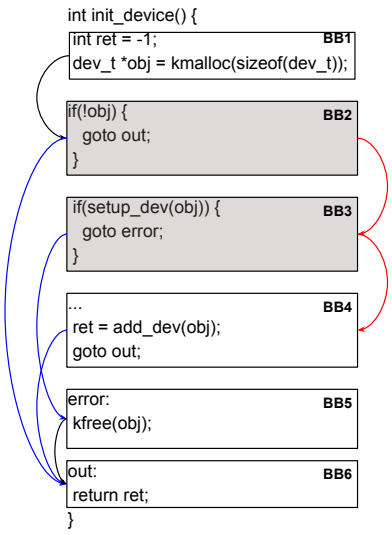

Figure 5.5: Control flow annotated listing where the greyed out blocks, i.e., $BB2$ and $BB3$, represent the error-handling basic blocks identified by our approach.

5.3.4 Identification of error-handling basic blocks

We use a technique similar to the ones proposed in the works by Kang et al. [189] and Tian et al. [190] in order to identify error-handling basic blocks.

Figure 5.5 illustrates our approach, where the identified error-handling basic blocks are greyed out. Specifically, we consider a basic block BB to be an error-handling basic block if it satisfies any of the following conditions:

- If BB *forces* the function to `return` a constant negative value or a C standard error code (i.e., one of the constant symbols defined in `errno.h`, e.g., `EINVAL`) prepended by a minus sign or `NULL`. For this, we do a basic reaching definition analysis and check that all paths through the basic block reach a function exit that returns a constant negative value or a C standard error code. This is based on the observation that functions use negative integers or values in `errno.h` or `NULL` to indicate error conditions. For the CFG of our running example in Figure 5.1, we detect $BB2$ as an error-handling basic block as it causes the function to return a negative integer

(`return -1`). Similarly, in Figure 5.5, $BB2$ causes the function to return the value of the variable `ret`, which is a negative integer (`-1`) set in $BB1$. Hence, $BB2$ will be considered as a BB_{err}.

- If BB ends in a direct jump (a `goto`) to a label that might indicate an error condition. We maintain a set of 15 error-related labels (e.g., `panic, error, fatal, err`), and we check if the BB ends with a `goto error-related-label;` statement. We derived our labels from an existing survey [195] and our experience in working with system code. This is based on the observation that most of the system code, especially operating system kernels [196], use `goto` to handle error conditions [197, 195]. In Figure 5.5, $BB3$ has the `goto error;` statement, and since `error` is one of our labels, $BB3$ will be considered as a BB_{err}. Note that, $BB3$ also satisfies the first condition, similar to $BB2$, as it can also cause the function to return a negative integer.

Unlike the work by Tian et al. [190], we do *not* consider the post-dominators of a BB_{err} to be BB_{err}s, thus, in Figure 5.5, the post-dominators of the error-handling basic blocks $BB2$ and $BB3$ ($BB5$ and $BB6$, respectively) will not be considered as BB_{err}s. This conservative approach improves precision by avoiding certain basic blocks to be wrongly identified as BB_{err}s (such as $BB6$). However, we may miss certain error-handling basic blocks ($BB5$). Note that, our approach for improving the precision by missing potential error-handling basic blocks is safe.

To check that our error-block detection approach is accurate, we randomly sampled 100 patches, and we verified that all the error basic blocks that we identified are indeed valid BB_{err}s.

As explained in Chapter 5.2, SPIDER discards all changes that happen within the identified BB_{err}s.

```
1  - max_len = strlen(buf);
2  + max_len = strlen(buf) + msg->len;
3  total_mem = max_len;
4  if(max_len < MIN) {
5    total_mem = MIN;
6  }
7  if(total_mem >= MAXMEM) {
8    return -EINVAL;
9  }
10 return send_msg(msg, buf);
```

Listing 5.5: Patch affecting the control-flow of a function.

5.3.5 Patch Analysis

In the remaining part of this chapter, we explain how SPIDER identifies *sp*s based on the general technique described in Chapter 5.2.

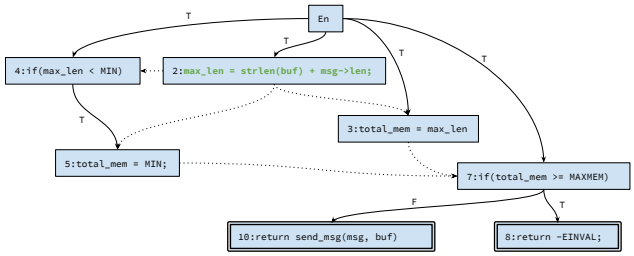

Figure 5.6: Program-Dependency Graph of the *patched* function in Listing 5.5.

Given the PDG, we remove all the data-dependency and control-dependency cycles in the PDG by removing all the back edges [198]. Given a statement t, we consider an edge to be a back edge if it is originated from a statement that is dominated by t in the PDG.

Note that, removing back-edges is safe when a patch does not directly modify a statement within a loop. In principle, removing back-edges in the PDG unrolls [199] the corresponding loop once. The symbolic expression of the values computed inside the loop will be as if the loop is executed once.

If the output of the function does *not* depend on the number of iterations of a loop then unrolling the loop once or multiple times does not affect our output equivalence

checking, and hence it is safe. As explained in Chapter 5.2.2, we use symbolic expressions (Table 5.1) to check the output equivalence of the functions. Now, consider the case where the output of the function depends on the number of iterations of a loop, and the symbolic expressions of the output are same in the original and the patched function. This means that the number of iterations of the loop will be the same in the original and patched function, and consequently, the output should be the same.

Hence, our approach of removing the back-edges and using symbolic expressions for output equivalence checking is safe when a patch does *not directly* modify a statement within a loop. However, if the patch directly modifies a statement within a loop, the removal of the back-edges prevents the back-propagation of this information resulting in computation of potentially wrong symbolic output-constraint pairs, thus is not safe. To be safe, a patch that directly modifies a statement within a loop will not be considered as an *sp*.

Using the diff-annotated CFG of the patched function, first we find all the directly affected statements. As explained in Chapter 5.2, these are the statements that are directly modified by the patch.

Second, given the PDG, we follow the edges from the nodes corresponding to the directly affected statements to identify all the statements that are reachable, which represent the indirectly affected statements. The union of the directly and indirectly affected statements is our total affected statements. As mentioned in Chapter 5.2, we ignore the affected statements that belong to the error-handling basic blocks ($BB_{err}s$).

Verifying non-increasing input space (C1): To verify condition $C1$, we first check if any of the affected statements is a conditional statement. By the definition of PDG (Chapter 5.2.1), these are the nodes that have an outgoing control dependency (solid) edge.

If there are affected conditional statements, then we find all the valid exit points, i.e.,

the valid `return` statements (or *VEP*). For each statement in the *VEP*, we identify the conditional statements that are part of the path constraint by following the solid edges backward until E_n.

Given a path constraint, we convert each of the conditional in the path constraint into a symbolic expression. As explained in Chapter 5.2.2, we start by initializing each of the variables with unique symbolic values in the original and patched function. Therefore, if a variable is *not* modified by the patch, it will have the *same* symbolic value in both the original and patched functions.

Conversion to symbolic expression: For a statement to be converted into symbolic expression, its data dependencies need to be first converted to symbolic expressions as well.

Therefore, given a statement s, we first check if it has any incoming data dependency edges, if this is the case, we go to the parent and try to repeat this process backward in a breadth-first manner until we find all the nodes with no incoming data dependency edges, i.e., the nodes from which all the data-dependency paths are complete (Chapter 5.2.1).

We call the nodes with no incoming data dependency edges as *free nodes*. We first convert each of the free nodes into symbolic values by following the corresponding instruction semantics (as shown in Table 5.1).

We then forward-propagate the values from the free nodes to the statements along the data dependency edges until we reach s.

To interpret function calls, we create a new symbolic value based on the hash of the function name *and* the symbolic values of its arguments. For instance, for the call `strlen(buf)`, we create a symbolic value with name equal to `hash(strlen, sym(buf))`, where `sym(buf)` is the symbolic value of the variable `buf`.

When multiple definitions of a variable reach an instruction, we use conditional symbolic variables based on the path constraint of the stricter path. For a variable x, if two

definitions d_1 and d_2 from statements v_1 and v_2, respectively, reach a statement v_3. Then the symbolic value of x at v_3 would be:

$$v_3(x) = \begin{cases} Ite(PC(v_1), d_1, d_2) & if\ PC(v_1) \rightarrow PC(v_2) \\ Ite(PC(v_2), d_2, d_1) & otherwise \end{cases} \tag{5.5}$$

Where $Ite(c, a, b)$ represents an `if-then-else` symbolic value, which dictates to use the value a if c is satisfiable else b, PC is the path constraint, and, from the rules of implication, $PC(c_1) \rightarrow PC(c_2)$ indicates that $PC(c_1)$ is a stricter condition than $PC(c_2)$. The Equation 5.5 correctly handles multiple definitions.

Consider the statement at line 7 in the PDG of Figure 5.6. Here, multiple definitions of the variable `total_mem` reach line 7. i.e., from line 3 and 5. By using the initial symbolic values, for `strlen(buf)` = $sym1$, `msg->len` = $sym2$, `MIN` = $sym3$, and `MAXMEM` = $sym4$. The definitions of `total_mem` at line 3 and 5 are $sym1 + sym2$ and $sym3$, respectively. The path constraint for line 3 and 5 are $PC(3)$ = (`En==T`) and $PC(5)$ = (`(En==T)` $\wedge$ $((sym1 + sym2) < sym3)$), respectively. We can see that $PC(5) \rightarrow PC(3)$, as $PC(5)$ is a stricter condition, consequently the symbolic value of `total_mem` at line 7 will be: $Ite(PC(5), sym3, (sym1 + sym2))$.

The symbolic expression for the path constraint of the valid `return` (line 10) in the patched function from the PDG of Figure 5.6 would be $(E_n == T \wedge (Ite(PC(5), sym3, (sym1 + sym2)) >= sym4))$, for brevity we did not expand $PC(5)$, but the actual symbolic expression would be only in terms of initial symbolic values.

Following the steps described above, we convert the path constraints of each of the valid `return`s in the patched function to symbolic expressions. Then we obtain Equation 5.3 by the disjunction of the symbolic expressions. Finally, we convert the disjuncted symbolic expression into a Z3 [191] expression, i.e., $vinputs(f_p)$ (see Chapter 5.2.2).

We follow the same steps in the original function to compute $vinputs(f)$, then, using the Z3 tool once again, we verify the implication $vinputs(f_p) \rightarrow vinputs(f)$, thus proving that the patch satisfies condition $C1$.

Verifying output equivalence (C2): Given the list of affected non-control-flow statements, as explained in Chapter 5.2.2, we only consider the statements that update the non-local state of the function, i.e., the function output.

Consider the patch in Listing 5.1, where, although all the statements are affected by the patch, the only statements of interest are at line 17 and 19, as they update the heap and return value.

As explained in Chapter 5.2.2 and shown in Table 5.1, we compute the symbolic expressions along each complete data dependency path along with the corresponding path constraints.

Finally, we convert the symbolic expressions into Z3 expression and verify Equation 5.4 using Z3. This verifies that the function affected by the patch satisfies condition $C2$. Note that the patch showed in Listing 5.5 changes only local variables and thus the output of the function remains the same as that of the original function for all valid inputs, thus satisfying condition $C2$.

We follow the above steps for each of the functions modified by the patch. We consider a patch to be a *safe patch*, only when $C1$ and $C2$ can be proved by following the steps described above.

Handling library functions: As explained in Chapter 5.2, we consider patches that have only locally analyzable statements, i.e., patches that do not directly affect function calls and pointers. However, we noticed that there are certain library functions, whose effects can be easily summarized. Such as, `memset`. There are other print and logging library functions, like `printf` and `printk`, that do not affect the output of the patched function.

To handle this, we maintain a few categories of commonly used, well-known library functions, whose effects can be either summarized or ignored.

5.4 Assumptions

Our implementation as specified in Chapter 5.3.5 tries to guarantee that a patch is a safe patch. However, a careful reader might have noticed that there are certain assumptions made by our implementation. In this chapter, we explicitly describe the assumptions in our implementation:

Non-alias dependencies: As explained in Chapter 5.3.5, we use a *PDG* based on variables to compute all the affected statements. However, this ignores the data dependencies that could happen through pointers [200]. Handling this requires precise pointer analysis, which in turn require access to the whole program violating our requirement $R1$.

Pure functions: We consider all functions to be pure functions [201], i.e., the output of a function only depends on the input arguments. In other words, multiple calls to a function with the same arguments results in the same output. Furthermore, reordering function calls without any change to the arguments will also be treated as equivalent. That is, `f1(arg1); f2(arg2);` is equivalent to `f2(arg2); f1(arg1);`. However, there could exist impure functions, whose output could also depend on the global state of the program. Soundly detecting whether a function is impure requires analyzing the function and its callees, which is not scalable and requires resolving function pointers.

Conditional compilation: The preprocessor conditional code directives (e.g., `#ifdef`, `#ifndef`, `#else`, etc.) allow different pieces of code to be be compiled depending on the values of certain preprocessor variables. We use the *unifdef* tool to handle these conditional compilation directives. *unifdef* attempts to obtain maximal code by enabling all preprocessor variables. However, for `#ifdef-#else` constructs, to be consistent, it has

to select the code either under the `if` or the `else` directive. This could result in certain statements in the patch (which are controlled by preprocessor variables) to be invisible to SPIDER, and, in turn, this could lead to false positives. Handling conditional code compilation precisely requires analyzing the patch under all possible values of preprocessor variables and their combinations. This is not scalable for large codebases like the Linux kernel. To handle this, we allow users to enable the *no preprocessor mode* (*NoPP*). In *NoPP* mode, any patch that affects statements controlled by preprocessor variables will *not* be considered as an *sp*.

We consider the limitations above to be fundamental implications of our requirements $R1$ and $R2$. Nonetheless, we believe that our system provides a reasonable approach to identify safe patches. Moreover, if these assumptions are considered too strong, it is always possible to fall back to the more conservative Security Patch (SeP) mode (see Chapter 5.5 for details). Finally, it is also possible to use our system to rank patches and prioritize those identified as safe for manually vetting (and testing).

5.5 Security Patch mode

As explained in Chapter 5, there could exist security patches without a corresponding CVE entry. To verify this, we have a configuration of SPIDER called *Security Patch (SeP) mode* that identifies security patches with *no false positives*, i.e., all the patches identified by this configuration are indeed security patches. SeP is based on the intuition that most of the security patches add additional input validation checks. Therefore, in SeP mode, we restrict ourselves to safe patches that affect only control-flow statements. Furthermore, when the commit message is available, we use the technique proposed by Zhou et al. [202] to filter out non-security related fixes. However, there can be false negatives, that is, potential security patches not detected as such.

Note that while SeP mode is more limited in the patches that it considers safe, it does not rely on any of the assumptions discussed in Chapter 5.4. We believe that SeP mode of SPIDER is the first step towards a practical solution of automatically identifying security patches that could be easily integrated into any source-control system. We plan to integrate SeP mode of SPIDER into GitHub security alerts [186], which helps both the developers and maintainers to handle security patches. Note that our running example (shown in Listing 5.1), although being a security patch, is not detected by the SeP mode because it also affects non-control flow instructions.

5.6 Evaluation

We evaluate the effectiveness of SPIDER in three different ways. First, we run it on a large dataset of 341,767 changes (i.e., commits) spanning over 32 repositories, collected from the year 2016 for a total of 32 months, in order to understand if it actually detects *sp*s, according to our definition (see Chapter 5.6.1). Second, we run SPIDER on a set of security patches (i.e., CVE patching commits) to evaluate the usefulness of this tool in speeding the propagation of these critical fixes. Third, in Chapter 5.6.4, we show a way to use the SeP mode of SPIDER as a vulnerability finding tool by identifying non-CVE security patches that are missing in various active forks of the analyzed projects. Finally, we show in Chapter 5.6.5, that there are several non-CVE security patches in the Linux kernel and many of these are still unpatched, at the time of writing, on some of its Android-related forks: this provides real examples where SPIDER can be useful in fixing potential *n-day* vulnerabilities.

The analysis that SPIDER performs, described in Chapter 5.3, is an intra-procedural static analysis that does not consider the interaction between different modified functions. For this reason, to isolate the effect of these interactions that represent a possible

confounding factor, we evaluate SPIDER only on patches that affect a single C source file (i.e., .c format only). All the patches studied in our evaluation are real changes extracted from repositories of widely used open-source projects (see Chapter 5.6.1 for more details).

Performance: On average SPIDER took 3.4 seconds to analyze a patch on a machine equipped with a two-core 2.40 GHz CPU, and 8GB RAM, demonstrating its speed and scalability.

Active forks: We noticed that most of the forks of repositories are inactive or dead, i.e., there are no new commits made to the repository since they are forked. Considering such inactive forks could exaggerate our results, and, therefore, we considered only active forks. We consider a fork to be active if it has at least ten new commits in the last six months, and using this filter, we were able to eliminate a number of forks. For instance, in the case of Linux kernel (ID 1), we consider only 269 active forks out of 23,854 forks.

5.6.1 Large-scale evaluation

We ran SPIDER on a large set of patches: we selected 32 open-source projects widely used by desktop, mobile, and embedded operating systems, and we collected from each of them all the single-C-file commits for the past 32 months from the time of writing (considering merges as single commits). All the details of the projects are shown in Table 5.2.

5.6.2 Effectiveness of patch analysis

Table 5.2 also shows the number of *sp*s identified by SPIDER in the dataset. Over the total 341,767 commits studied, SPIDER identified 67,408 (19.72%) safe patches (Column 6). Furthermore, 58.72% of these patches are missing in at least one of the active forks (MIAFs).

Checking for patch applicability: We use the following syntactic approach to identify

whether a patch of a project is applicable to (or missing from) a fork or other projects. Given a patch, we extract the affected file's source code before the patch (i.e., original file) and compare it to the latest version of the corresponding file in the fork. If the file is present in the fork *and* all the functions affected by the patch do not differ between the original file of the patch and the corresponding latest file in the fork, then it means that the patch can be applied to the fork. To perform the comparison, we use the *git diff* tool, and check that there are no modifications in the targeted functions.

It is interesting to note that, across all repositories, the percentage of *sp*s mostly stays around 20%-25%, without much variation. There are certain projects where the percentage of detected *sp*s is low, such as IDs 15 and 16. After manual investigation of the subset of these patches, we found the following reasons:

Complex code: There are certain projects that mostly contains complex functions with data-dependencies inside nested loops. Specifically, the Python (ID 15) and PHP (ID 16) interpreters, and cURL (ID 24). Here, although the patches themselves are simple, the data dependencies increase the complexity of constraints, resulting in SPIDER failing to prove implication for the condition $C1$ (Chapter 5.2.2) resulting in a smaller *sp* detection rate.

Complex patches: In projects such as libpng (ID 30) and, OpenVPN (ID 28), the commits tend to be complex as they deal with media file formats and cryptographic protocols. Consequently, SPIDER fails to prove the equivalence for the condition $C2$.

Listing 5.9 shows a patch identified as an *sp*, where the patch modifies error basic blocks, which are ignored. Also, the patch moves certain function calls `Py_INCREF` and `Py_DECREF`. However, as the arguments to these calls (i.e., `dll` and `ftuple`) are not modified by the patch, the symbolic expressions of the arguments are proved to be equivalent by Z3, resulting in the patch being considered an *sp*.

Looking at these results, we argue that SPIDER would be helpful for project maintainers and could be directly used to port the fixes or to prioritize the changes that must be ported.

5.6.3 Evaluation on CVEs

We wanted to determine how many security patches are indeed *sp*s, as claimed in Chapter 5. To this end, we collected all the patching commits linked as reference fixes for kernels CVEs from the Android security bulletins [14], and, similar to the large-scale evaluation, we studied only the CVEs that patch a single C file. We also collected all the CVEs for the remaining repositories over the same amount of time. This resulted in the analysis of 809 CVE patches.

Table 5.3 shows the results obtained after running SPIDER on these patches, which show that 55.37% of the CVE-patching commits are non-disruptive, while on generic patches (i.e.,
Table 5.2) the percentage was 19.72%. This finding shows that SPIDER could be useful not only to speed-up the process of selecting and applying a significant number of changes (as shown in Chapter 5.6.1) but also to apply more than half of the security patches in a faster way.

Listing 5.7 shows an example of CVE patching commit from Android security bulletin identified as a *sp* by SPIDER. Listing 5.7 is also one of the CVEs that we mentioned in Chapter 5, which was patched in Android more than a year after the appearance of the corresponding entry in the database.

Looking at Table 5.3, it is interesting to see that SPIDER performed relatively well with more than 50% success rate in all but OpenSSL and VLC CVEs.Most of the CVEs in OpenSSL fix security issues related to cryptographic operations that affect the control

flow in complex ways. A few OpenSSL CVEs fix cryptographic implementations against time side-channel attacks, which SPIDER is unable to reason about. For instance, the commit hash ae50d8270026edf5b3c7f8aaa0 c6677462b33d97 [203] for CVE-2016-0703 of the OpenSSL repository fixes SSLv2 implementation against the Bleichenbacher [204] attack. We fail to identify this as an *sp* because the changes does not satisfy our definition of *sp* (refer Chapter 5.1.2).

5.6.4 Security patches missing a CVE number

We used SPIDER in SeP mode on all the commits to identify security patches. We then checked if these patches have an associated CVE number. Listing 5.9 and 5.6 show examples of security patches missing CVE entries, which are detected by the SeP mode of SPIDER.

ID	Project	git branch	Commits	Active forks	*sps* (% over commits)		Non-CVE security patches	
					Total (%)	MIAFs (%)	Total	MIAFs (%)
Linux Kernels								
1	Linux kernel mainline[3]	master	102,607	269	20,171 (19.66%)	9,427 (46.74%)	635	297 (46.77%)
2	Linaro ARM Linux kernel[4]	optee	96,990	7	19,172 (19.77%)	6,846 (35.71%)	587	211 (35.95%)
3	Rasberry Pi Linux kernel[5]	rpi-4.14.y	54,585	168	11,250 (20.61%)	10,511 (93.43%)	394	362 (91.88%)
Android Kernels								
4	Qualcomm Msm Android kernel[6]	msm-oreo [205]	472	N/A	128 (27.12%)	N/A	8	0 (0.0%)
5	NVIDIA Tegra Android kernel[7]	tegra-oreo [?]	297	N/A	88 (29.63%)	N/A	9	0 (0.0%)
6	Xiaomi Android kernel[8]	sagit-o-oss	9,718	85	2,332 (24.0%)	2,332 (100.0%)	94	94 (100.0%)
7	Android x86_64 kernel[9]	android-8.0.0_r0.23	211	N/A	56 (26.54%)	N/A	4	0 (0.0%)
8	Xperia Android kernel[10]	aosp/LA.UM.6.4.r1	10,932	69	2,559 (23.41%)	2,554 (99.8%)	99	99 (100.0%)
9	Base Android Kernel[11]	android-4.9-o	21,927	39	4,967 (22.65%)	4,330 (87.18%)	201	173 (86.07%)
Bootloader and Firmware								
10	LK Embedded kernel[12]	master	30	26	7 (23.33%)	7 (100.0%)	0	N/A
11	Qualcomm LK Kernel[13]	lk.lnx.1.0.r21-rel	219	N/A	42 (19.18%)	N/A	5	0 (0.0%)
Trusted Operating Systems								
12	OP-TEE Trusted OS[14]	master	499	45	96 (19.24%)	71 (73.96%)	4	3 (75.0%)
OpenBSD								
13	OpenBSD[15]	master	8,651	14	1,424 (16.46%)	843 (59.2%)	42	30 (71.43%)
Windows Compatible OS								
14	reactos-Open Source Windows Compatible OS[16]	master	2,936	35	510 (17.37%)	123 (24.12%)	11	1 (9.09%)
Interpreters								
15	Python Interpreter[17]	master	862	207	108 (12.53%)	38 (35.19%)	12	2 (16.67%)
16	PHP Interpreter[18]	master	3,096	289	344 (11.11%)	274 (79.65%)	7	3 (42.86%)
Graphical Subsystems								
17	nautilus-Ubuntu default graphical subsystem[19]	master	591	N/A	129 (21.83%)	N/A	2	0 (0.0%)
18	winfile-Windows File Manager[20]	master	28	23	2 (7.14%)	1 (50.0%)	0	N/A
19	X Windows Subsystem[21]	master	644	1	119 (18.48%)	60 (50.42%)	5	2 (40.0%)
Multimedia								
20	VLC Player[22]	master	6,815	98	1,025 (15.04%)	778 (75.9%)	34	28 (82.35%)
21	FFmpeg-multimedia processing tools[23]	master	6,538	449	697 (10.66%)	573 (82.21%)	42	37 (88.1%)
Distributed Databases								
22	redis-In memory Database[24]	unstable	1,385	659	113 (8.16%)	69 (61.06%)	11	7 (63.64%)
Utilities								
23	Tmux-Terminal Multiplexer[25]	master	543	77	110 (20.26%)	86 (78.18%)	5	4 (80.0%)
24	curl-transfer a URL[26]	master	984	239	100 (10.16%)	70 (70.0%)	1	0 (0.0%)
25	evince-GNOME document viewer[27]	debian/master	202	N/A	51 (25.25%)	N/A	4	0 (0.0%)
26	Git-version control system[28]	master	2,834	526	360 (12.7%)	286 (79.44%)	15	13 (86.67%)
27	GDB-GNU Debugger[29]	master	55	25	13 (23.64%)	9 (69.23%)	1	1 (100.0%)
28	OpenVPN-Open source VPN daemon[30]	master	201	70	18 (8.96%)	11 (61.11%)	0	N/A
29	Systemd System[31]	master	4,815	N/A	1,067 (22.16%)	N/A	29	0 (0.0%)
Libraries								
30	libpng-PNG reference library[32]	libpng16	82	42	1 (1.22%)	1 (100.0%)	0	N/A
31	OpenSSL-Open Source TLS toolkit[33]	master	1,998	290	347 (17.37%)	283 (81.56%)	17	16 (94.12%)
32	glibc-GNU libc[34]	master	20	7	2 (10.0%)	2 (100.0%)	0	N/A
Total			341,767	3,759	67,408 (19.72%)	39,585 (58.72%)	2,278	1,383 (60.71%)

Table 5.2: Overall results of our large-scale evaluation. Active forks are computed for only GitHub hosted projects. The percentage for Total *sps* is over commits for the corresponding projects. MIAFs are percentage over total *sps* and non-CVE commits that are missing in at least one active fork.

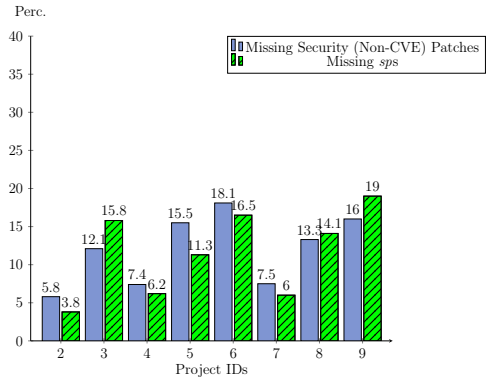

Figure 5.7: Distribution of the security (non-CVE) patches (identified by the SeP Mode of SPIDER) and *sp*s in Mainline Linux kernel (Project ID 1) that are missing in other related kernel projects.

CVE patches source	***sp*s / CVE**
Linux	333 / 611 (54.5%)
Android bulletin	98 / 164 (59.75%)
OpenBSD	5 / 6 (83.33%)
OpenSSL	7 / 21 (33.33%)
Systemd	4 / 4 (100%)
VLC	1 / 3 (33.33%)
Total	448 / 809 (55.37%)

Table 5.3: Results of SPIDER on CVE patches.

```
 int check_aboot_addr_range_overlap(uint32_t start, uint32_t size)
 {
         /* Check for boundary conditions. */
-        if ((start + size) < start)
+        if ((UINT_MAX - start) < size)
                 return -1;
```

Listing 5.7: Real integer overflow patch identified as *sp* by SPIDER (CVE-2014-9795 from July 2016 Android security bulletin).

The last two columns of Table 5.2 shows the complete results: Overall SPIDER identified 2,278 security patches across all the repositories. After manual verification, we found these results to be correct. This shows that CVE IDs are not always used for security patches, and that relying on them is not an effective way to secure related

```
                error = -EINVAL;
                goto out_put_tmp_file;
        }

+       if (f.file->f_op != &xfs_file_operations ||
+           tmp.file->f_op != &xfs_file_operations) {
+               error = -EINVAL;
+               goto out_put_tmp_file;
+       }
+
        ip = XFS_I(file_inode(f.file));
        tip = XFS_I(file_inode(tmp.file));
```

Listing 5.6: a security patch identified by SPIDER on the main Linux kernel repository (commit 3e0a3965464505). *which does not have a corresponding CVE ID.*

repositories. The number of patches identified by the SeP mode is smaller compared to the total number of *sp*s (i.e., 2,278 $\ll$ 67,408). This is because the SeP mode, as explained in Chapter 5.5, imposes strict requirements. Nonetheless, the SeP mode identified 2,278 security patches missing a CVE number.

Furthermore, 60.71% of these patches are missing in at least one of the active forks denoted as MIAFs. This is alarming, as these cases reveal unpatched security vulnerabilities in the forks, which could be exploited by a motivated attacker monitoring the patches. We observed a considerable number of patches (for example see Listing 5.8), where the commit message contains the vulnerability-triggering input, further reducing the effort for the attacker.

```
      avpriv_report_missing_feature(s->avctx, "Lowres for weird subsampling
          ");
      return AVERROR_PATCHWELCOME;
    }
+   if ((AV_RB32(s->upscale_h) || AV_RB32(s->upscale_v)) && s->progressive
    && s->avctx->pix_fmt == AV_PIX_FMT_GBRP) {
+   avpriv_report_missing_feature(s->avctx, "progressive for weird
    subsampling");
+   return AVERROR_PATCHWELCOME;
+   }

    if (s->ls) {
      memset(s->upscale_h, 0, sizeof(s->upscale_h));
      memset(s->upscale_v, 0, sizeof(s->upscale_v));
```

Listing 5.8: A non-CVE security patch (commit ee1e3ca5eb1) in FFmpeg (ID 21) that has triggering input in the commit message.

```
-Py_INCREF(dll); /* for KeepRef */
-Py_DECREF(ftuple);
```

```
-if (!_validate_paramflags(type, paramflags))
+if (!_validate_paramflags(type, paramflags)) {
+   Py_DECREF(ftuple);
    return NULL;
+}
self = (PyCFuncPtrObject *)GenericPyCData_new(type, args, kwds);
-if (!self)
+if (!self) {
+   Py_DECREF(ftuple);
    return NULL;
+}
...
*(void **)self->b_ptr = address;
+Py_INCREF(dll);
+Py_DECREF(ftuple);
```

Listing 5.9: A non-CVE security patch (commit d77d97c9a1f) fixing a reference counting vulnerability in the Python interpreter identified by SPIDER. *This patch does not have a corresponding CVE ID.*

5.6.5 Missing patches in vendor kernels

```
    parse_exthdrs(struct sk_buff *skb, const struct sadb_msg *hdr, void *
        ...
    uint16_t ext_type;
    int ext_len;

+   if (len < sizeof(*ehdr))
+       return -EINVAL;

    ext_len  = ehdr->sadb_ext_len;
```

Listing 5.10: A non-CVE security patch (commit 4e7REDACTED) in Main kernel (ID 1) that is missing in Qualcomm (ID 4) kernel.

To identify missing patches in vendor kernels, we check how many of the Linux Kernel mainline commits identified as *sp*s still have to be applied to one or more of the eight vendor kernel repositories that we studied (i.e., projects 2 - 9 in Table 5.2), at the time of writing. To do that, given a commit identified as an *sp*, we extract the affected file's source code before the change, and we compare it to the same file, if present, in all the listed kernel repositories (Table 5.2 show the git branch or tag that we studied) using the `git diff` technique described in Chapter 5.6.2.

The stripe bars in Figure 5.7 shows the percentage of missing *sp*s in different vendor kernels. We found that 9,427 of the 20,171 Linux kernel identified *sp*s (i.e., 46.74%) are still not applied in at least one of the considered vendor kernels. A significant portion

of these changes not considered useful by the maintainers (e.g., removals of unused code, small refactoring, etc.), and therefore, not imported. However, we found out that 297 of them are CVE patching commits (i.e., the ones that we linked to the corresponding CVEs, as shown in Chapter 5.6.3) that still have to be imported by the maintainers of some repositories: this supports the findings of previous studies [10, 11, 12, 13] that report that vulnerability databases are not always effective in speeding the propagation of security fixes.

Unfixed vulnerabilities in vendor kernels: We also checked the security patches (which do not have a CVE number) identified by the SeP mode in the Linux Kernel mainline that still have to be applied to one or more of the eight Linux Kernel repositories that we studied (i.e., projects 2 - 9 in Table 5.2). The plain bars in Figure 5.7 show the percentage of missing non-CVE security patches in different vendor kernels. There are in total **229** security patches that do not have a corresponding CVE number and are missing on different kernel repositories, including the ARM Linux kernel main repository (i.e., project 2 in Table 5.2): these can be seen as potential unfixed or *n-day* vulnerabilities. Given their potential severity, we manually verified them to assess their impact. For a few of these vulnerabilities, the impact is less severe because of the variation in kernel configurations. However, we found several missing patches in critical components like `netfilter`, which applies to all kernel configurations. The snippet of a non-CVE security patch that is missing in the msm kernel (ID 4) is shown in Listing 5.10, this patch, as mentioned before also contains the triggering input.

We are in the process of reporting all of these patches to the corresponding project maintainers and vendors, and submit all the necessary requests for CVEs.

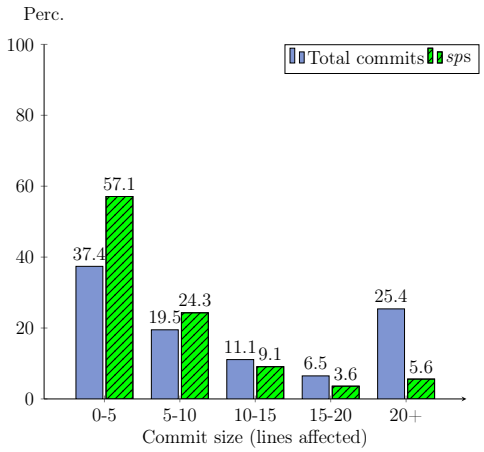

Figure 5.8: The distribution of the size of all the commits studied and *sp* identified by SPIDER.

5.7 Limitations

Along with the assumptions described in Chapter 5.4, SPIDER comes with several limitations. Specifically,

Small patches: As we can see from Figure 5.8, the majority (57.1%) of the patches detected as *sp*s are small (0-5 lines). Furthermore, SPIDER cannot verify patches that modify statements within a loop. These limitations are mainly because SPIDER tries to verify a patch to be safe in a sound way. We believe it is important to have a system with no false positives, that provides stronger guarantees, and that can be used by the maintainers safely.

Syntactic approach for patch applicability check: We use a syntactic approach to check for patch applicability in the related repositories. However, a patch although syntactically applicable to a file in a project may not be semantically applicable because the condition fixed by the patch could be impossible to occur in the project [206]. This limitation is induced by our requirement R1, as checking for semantic applicability of patches require sound static analysis techniques which require build environment and

access to all source files, thus violating our requirement R1.

Heuristic approach for error-handling basic blocks detection: As explained in Chapter 5.3.4, we use a heuristic approach to identify error-handling basic blocks. However, these heuristics may not hold for other projects resulting in cases where a basic block matching our heuristics is not a true error-handling basic block. Consequently, we could have unsafe patches being identified as safe. To handle this, we provide the *NoEB* mode of SPIDER where we do not ignore the changes in the error-handling basic blocks. This mode provides a safer version of SPIDER, albeit with a slight decrease in detection rate.

Susceptible to adversarial evasion: As a consequence of our assumptions (Chapter 5.4), SPIDER is susceptible to adversarial evasion. For instance, as we treat macros as function calls, an adversarial developer or contributor could use macro calls to make SPIDER consider otherwise safe patches as unsafe. However, as we explained in Chapter 5, the main use case of SPIDER is for developers and maintainers. Furthermore, we assume developers to be non-malicious users who want to ensure that their applications are as secure as possible.

Tool dependencies: The current implementation of SPIDER works only on C source code; however, the parser that we use should be easily extensible to other languages. The fine-grained diff step is language agnostic, thus, to extend the tool to other languages, we would only need to add language-specific heuristics and preprocessing. A good solution would be to have a configurable front end for different languages, similar to LLVM [207]. As our implementation is based on Joern and Gumtree, we also share the same limitations that these tools have.

5.8 Can we prevent memory corruption?

It is the 21st century, with over three decades of research on finding memory safety issues, but memory corruption is still the most prevalent class of vulnerability in modern applications. Specifically, spatial memory issues remain the most common vulnerability category [208].

Although memory-safe systems programming languages exist, they are not well-suited to seamlessly interact with legacy C code [35]. On the other hand, retrofitting techniques [36, 37, 38] that add memory safety to legacy code have high-performance overhead and are not backward compatible.

What we need is a memory-safe language that should not have a steep learning curve, i.e., *should be very similar to C*, backward-compatible, i.e., *allows safe and unsafe code to co-exist*. So that the developers can write the new code in the safe dialect, which could interact with unsafe legacy code, and finally *should have very low-performance overhead (both memory and runtime)*.

The Checked C [40] is an extension to the C programming language that satisfies all the above requirements. It extends C with type annotations using which it tries to prove spatial memory safety statically. It adds dynamic checks for the cases where safety cannot be proven statically. However, an issue here is that these type-annotations needs to be added to the existing C code. Can we solve this problem? Specifically, can we have a technique that can automatically add type annotations to already existing C code? In the next chapter, I will present our on-going work on a technique that can automatically convert C to Checked C.

Chapter 6

Interactively converting C to Checked C

Suppose a developer wishes to port an existing C program to take advantage of the guarantees of Checked C. Ideally, the developer will aim to completely port the program, i.e., to place all of the code inside of checked regions. What sorts of changes will a developer have to make?

1. *Best case*: Add annotations (mostly to types) but otherwise not change the code as shown in Listing 6.1.

2. *Second-best case*: Add minor/obvious bits of code, e.g., to initialize newly-annotated variables as shown in Listing 6.2.

3. *So-so case*: Rewrite bits of code and annotate them, so they are checkable (Listing 6.3).

4. *Worst case*: Not able to port the code at all—either it's fundamentally incompatible with Checked C, or else the port would impose a too-high performance overhead (Listing 6.4).

Listing 6.1: The best case where we just need to add annotations without changing any code. The code in comments is the original C code.

```
int foo(void) {
  //int *p = NULL;
  _Ptr<int> p = NULL;
  //static char datebuf[64];
  static char datebuf _Checked[64];
  ....
}
```

Listing 6.2: The second best case where we need to add annotations and split the declaration in to multiple lines and add initializer.

```
int foo1(void) {
  //int p,*q;
  int p;
  _Ptr<int> q = NULL;
  ....
}
```

Listing 6.3: The so-so case that involves adding explicit Checked C casts.

```

// const char*
// vsf_sysutil_group_getname(const struct vsf_sysutil_group* p_group)
// {
//  const struct group* p_grp = (const struct group*) p_group;
//  return p_grp->gr_name;
// }
const _Ptr<char>
vsf_sysutil_group_getname(const _Ptr<struct vsf_sysutil_group> p_group)
{
  const _Ptr<struct group> p_grp = _Assume_bounds_cast<const _Ptr<struct group>>(
      p_group);
  return p_grp->gr_name;
}
```

Listing 6.4: Function that has inline assembly that represents a worst case for conversion and is impossible to safely convert to Checked C.

```
static inline void cpuid(int code, uint32_t* a, uint32_t* d)
{
    asm volatile ( "cpuid" : "=a"(*a), "=d"(*d) : "0"(code) : "ebx", "ecx" );
}
```

Ideally, the developer performs all of steps (1)-(3) and minimizes resorting to step (4). We call this a *full port*. Carrying out this process entirely manually would be incredibly time consuming, and thus unreasonable [209]. Therefore, in this paper, we present a tool we call `checked-c-convert` (CC-CONV for short) aimed to assist a developer to perform a full port.

The core of CC-CONV is a novel type-inference based approach to determine the Checked C types of all pointers in a given program.

However, there are cases where the type-inference fails to determine the Checked C types e.g., In the presence of unsafe pointer casting. Furthermore, automatically converting arbitrary C to Checked C code is hard or rather impossible.

We had been tacitly assuming that the CC-CONV would be run once to annotate as many pointers as possible, and the developer would take over to finish the job from there. No conversion tool can be perfect (the conversion problem is undecidable), so there will always be work left for the human to do. This follow-on, human work is often neglected in, but should not be: After all, our goal is to minimize the *total* work required to port a program.

To minimize total porting work, we should make the conversion algorithm *interactive*: Do some work, let the human do some, do some more work automatically, and so on until the job is complete. Such a workflow allows us to let the developers move things forward at the places where their knowledge is most useful. The tool can handle all of the grunt work.

There are two times where interaction is particularly helpful.

1. There are times when a series of constraints leads to a proliferation of non-checked (or `WILD`) pointers. We can ask the user to "break" the constraints at a particular place (via an unsafe cast) to avoid this proliferation. After asking for his/her input, we solve. This is illustrated by the code in Listing 6.5, which can be converted with human input (`(const char*)`) by introducing an explicit cast as shown in Listing 6.6.

2. In general, we want the user to be able to make a small change to the program, e.g., by manually annotating a variable, and then be able to re-run the porting tool to propagate the effect of that change, e.g., to other variables to which (or from which) the annotated variable is assigned. This should be as fast as possible.

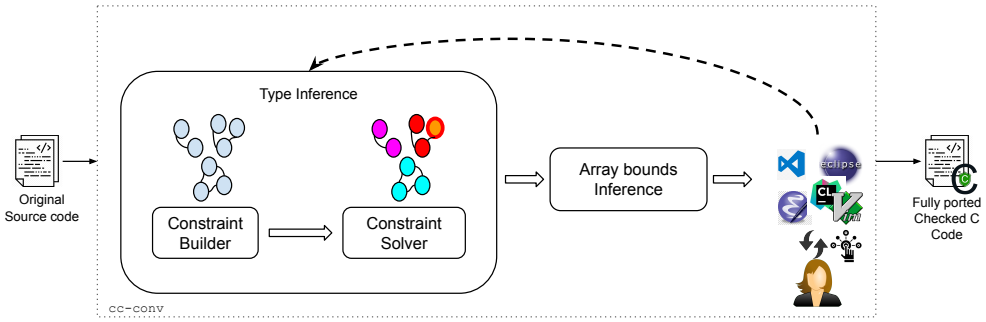

Figure 6.1: Dataflow diagram of CC-CONV showing interaction between various components. The user can use any IDE that implements language server to interact with our base constraint solving mechanism.

Listing 6.5: The case of proliferation where passing a pointer to a library function will make it WILD and consequently this information is propagated to all the callers of the function.

```
int vsf_sysutil_mkdir(const char* p_dirname, const unsigned int mode)
{
  // Here, passing p_dirname as a parameter to external function will make it WILD
  // and this propagates to all the callers of vsf_sysutil_mkdir
  return mkdir(p_dirname, mode);
}
void caller1() {
  char *buff = NULL;
  ...
  initialization of buff
  ...
  vsf_sysutil_mkdir(buff, RW);
}
```

The first case is, in some sense, an instance of the second case, but it uses the constraint graph to guide the programmer to variables that are particularly high-value to update.

Based on the above observations, we designed CC-CONV to be interactive, wherein CC-CONV directs the developer to the places where the inference failed. This enables the developer to provide directions to CC-CONV, on how the failure should be resolved, much like the way developers interact with a code refactoring tool.

Listing 6.6: The conversion of code in Listing 6.5 by using explicit casts.

```
int vsf_sysutil_mkdir(_Nt_array_ptr<const char> p_dirname, const unsigned int mode)
{
  // Adding cast at the following call-site will break the propagation of
  // constraints and thus make p_dirname a checked pointer.
  return mkdir((const char*)p_dirname, mode);
}
void caller1() {
  _Nt_array_ptr<char> buff = NULL;
  ...
  vsf_sysutil_mkdir(buff, RW);
}
```

6.1 Design and Implementation

The high level architecture of CC-CONV is as shown in the Figure 6.1. First, given a program with C sources, we automatically parse the source files to get the AST and create a constraint graph where each node represents a pointer declaration. We then solve the constraints to get Checked C types. Second, for each of the identified array pointers, we infer length associations using various heuristics. Third, for the unconverted pointers (Listing 6.5), we identify the root cause and provide the user suggestions on how to fix it. Finally, the user can provide fixes which cause the constraint solver to re-run, and this process continues until the user is satisfied with the currently converted code, at which point we rewrite the code with Checked C annotations.

We use `clang` [162] tooling for our implementation. This is currently work in progress and maintained open-source at `https://github.com/plum-umd/checkedc-clang`, with various collaborators from the University of Maryland, College Park and Microsoft Research.

6.2 Preliminary Evaluation

We used `vsftpd` (a full-blown FTP server) and various benchmark programs from ptrdist [210] dataset to evaluate CC-CONV. The overall results are shown in the Table 6.1.

6.2.1 Inference Effectiveness

As shown under column **WILD**, on average CC-CONV is unable to infer Checked C types for 41.15% of the total pointers. As mentioned before, this is because either the pointer is improperly used (i.e., `p_dirname` in Listing 6.5), or the pointer depends on other pointer that is improperly used (i.e., `buff` (that depends on `p_dirname`) in Listing 6.5).

However, the actual pointers that are improperly used are only 8.33%, as shown by the last column of Table 6.1. This confirms our intuition that it is only a small set of pointers that pollute various other pointers through constraints. Further investigating the improper usages of these pointers, we observed that most of the improper usages are either because of casting or being passed as arguments to external library functions, whose parameters we assume to be unsafe.

Furthermore, the small percentage (8.33%) of improperly used pointers shows that the amount of effort a human has to invest in achieving the full conversion is less and can be further eased with an interactive system.

6.2.2 Array bounds Inference

The column **Inferred Bounds** shows our techniques were able to infer bounds for 27% of variable-length array pointers. This again could be further improved by an interactive system, where the user provides bounds that can be then propagated to other dependent array pointers.

Program	Category	Size (SLoc)	Total Pointers (Tot)	Regular Pointers (PTR)	Array Pointers					Unconverted Pointers (WILD) (% over Tot)	Directly Unconverted (% over Tot)
					Regular (ARR)	Null Terminated (NTARR)	Constant Size	Need Bounds (nb)	Inferred Bounds (% over nb)		
vsftpd	FTP Server	17K	2,020	1,114	23	35	39	19	7 (36.84%)	848 (41.98%)	189 (9.35%)
anagram	Pointer Benchmarks	352	72	23	21	2	15	7	1 (14.29%)	26 (36.11%)	8 (11.11%)
ft		1,175	209	115	8	0	6	2	0 (0%)	86 (41.15%)	5 (2.39%)
ks		609	76	15	8	5	11	4	0 (0%)	48 (63.16%)	1 (1.32%)
yacr2		2,915	109	12	82	0	77	5	2 (40%)	15 (13.76%)	4 (3.67%)
Total	-	22K	2,486	1,279	142	42	148	37	10 (27.03%)	1,023 (41.15%)	**207 (8.33%)**

Table 6.1: Results of running CC-CONV on various programs. Here, columns **PTR**, **ARR**, and **NTARR** shows the number of regular pointers, array pointers and null-terminated array pointers respectively that are successfully inferred by CC-CONV. The column **WILD** shows the total number of pointers we fail to infer and are considered as unchecked pointers. The last column shows the number of pointers that are improperly used.

The results of our preliminary evaluation are encouraging. This is an on-going work where steady progress is made and will be released as an open-source tool that can be used arbitrary C to Checked C.

Chapter 7

Conclusion

In this work, we looked into the smart device ecosystem and various issues that arise with new processor security features and extreme customizations. The ARM architecture, which is commonly used in smart devices, has an advanced security architecture feature called ARM TrustZone, which provides a secure, isolated, and trusted execution environment (TEE). In Chapter 3, we identified a previously unknown class of vulnerabilities, BOOMERANG, that affects systems where the secure world (i.e., the TEEs) and the non-secure world (i.e., the traditional OS) share resources. The vulnerability arises from the critical semantic gap when passing data between the two worlds, specifically memory pointers, and flaws in sanitizing these pointers. We identified BOOMERANG vulnerabilities in four of the most popular commercial TEE platforms (affecting hundreds of millions of devices world-wide). In order to explore the generality and severity of BOOMERANG, we developed a static-analysis tool to automatically identify BOOMERANG bugs in real-world TEE applications. These findings have resulted in major efforts from the respective parties (e.g., Google and Qualcomm) to fix their implementations, as the identified vulnerabilities could be leveraged to completely compromise the untrusted OS (e.g., Android) of the affected devices. We similarly analyzed three potential BOOMERANG de-

fenses, comparing the trade-offs and design considerations of each. Due to the limitations of the existing defenses (i.e., shared memory and page table introspection), we devised a novel solution, *Cooperative Semantic Reconstruction*, which addresses the shortcomings of the previous proposals, while still offering an efficient and easy-to-use interface.

Smart devices have extensible software support in the form of open-source system software. The availability of well-supported open-source system software enables vendors to perform quick customizations, e.g., by adding device drivers to the operating systems. Unfortunately, these customizations are poorly developed, which results in a lot of critical security issues. In Chapter 4, we have presented Dr. Checker, a fully-automated static analysis bug-finding tool for Linux kernels that is capable of general context-, path-, and flow-sensitive points-to and taint analysis. Dr. Checker is based on well-known static analysis techniques and employs a *soundy* analysis, which enables it to return precise results, without completely sacrificing soundness. We have implemented Dr. Checker in a modular way, which enables both analyses and bug detectors to be easily adapted for real-world bug finding. In fact, during the writing of this paper, we identified a new class of bugs and were able to quickly augment Dr. Checker to identify them, which resulted in the discovery 63 zero-day bugs. In total, Dr. Checker discovered 158 previously *undiscovered* zero-day bugs in nine popular mobile Linux kernels. While these results are promising, Dr. Checker still suffers from over-approximation as a result of being *soundy*, and we have identified areas for future work. Nevertheless, we feel that Dr. Checker exhibits the importance of analyzing Linux kernel drivers and provides a useful framework for adequately handling this complex code.

The extreme customization also results in the problem of patch propagation. The customizations are usually done on codebases that are maintained in different repositories (e.g., `forks`) separate from the main open-source repository. In order to ensure that vulnerability is fixed, the patch for the vulnerability should propagate to all the

codebases (or repositories) as soon as possible. We noticed that security patches still take a substantial amount of time to propagate to all the project forks. To solve this, we need an automated technique that can help in automatically propagating patches to the related repositories. In Chapter 5, we designed, implemented, and evaluated SPIDER, a fast and lightweight tool (**R2**) based on our *sp* identification approach that can determine if a patch is safe using only the original and the patched source code of the affected file (**R1**), without the need for external information (e.g., build environment, commit message, etc.). Our large-scale evaluation on 341,767 commits extracted from 32 different open-source repositories, and on 809 CVE patches, demonstrates the effectiveness of SPIDER, and shows that a significant amount of security patches could have been automatically identified (i.e., 55.37%). Furthermore, we show how the SeP mode of SPIDER can be used to find unpatched security issues.

Finally, I believe that Checked C provides a promising alternative to prevent spatial memory issues in C code. In Chapter 6, we briefly present our on-going work on automatically converting C to Checked C code.

Chapter 8

What's next?

In this work, I had the opportunity to apply various program analysis techniques to systems codebases. System codebases being inherently complex, large, and not well-specified present the worst case for program analysis techniques. However, as presented in Chapters 45, we can apply precise program analysis techniques on system codebases by scoping to only certain components, e.g., Drivers instead of entire Linux kernel. There are other techniques one can apply to scale precise program analysis techniques to system codebases.

Gradual precise analysis: For vulnerability detection, instead of directly applying very precise analysis, we can gradually increase the precision of the analysis such that increasing the precision will gradually eliminate false positives.

In the past, this technique has been explored in pointer analysis [211] and recent work [212] shows that being gradually precise might help in finding very complex bugs with very low false positives. Can we have a system where this can be done in a principled manner? and What are the guarantees of such a system?

Interactive analysis: Our on-going work with Checked C has shown that it is feasible to seamlessly interact with humans to improve the precision of the underlying analysis.

There are many interesting directions in this area: Can we have interactive analysis designs? Specifically, can we create an analysis engine where human input is treated as a first-class entity? Can we integrate human input into dynamic analyses? For example, Fuzzing.

Finally, I believe collaborations result in high-quality research and are necessary for the field of security, which is, by nature, interdisciplinary. If you want to work on any of the above problems, please do not hesitate to contact me.

Bibliography

[1] D. Kumar, K. Shen, B. Case, D. Garg, G. Alperovich, D. Kuznetsov, R. Gupta, and Z. Durumeric, *All things considered: An analysis of iot devices on home networks*, in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pp. 1169–1185, 2019.

[2] A. Nordrum, *The internet of fewer things [news]*, *IEEE Spectrum* **53** (2016), no. 10 12–13.

[3] Direct Industry, "ARM Processor manufacturers." `https://www.directindustry.com/industrial-manufacturer/arm-processor-77989.html`, 2016.

[4] ARM, "ARM TrustZone." `http://www.arm.com/products/processors/technologies/trustzone/index.php`, 2015.

[5] P. Stewin and I. Bystrov, *Understanding dma malware*, in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 21–41, Springer, 2012.

[6] A. Machiry, E. Gustafson, C. Spensky, C. Salls, N. Stephens, R. Wang, A. Bianchi, Y. R. Choe, C. Kruegel, and G. Vigna, *Boomerang: Exploiting the semantic gap in trusted execution environments.*, in *NDSS*, 2017.

[7] Skybox Security, *2019 vulnerability and threat trends*, 2019. `https://lp.skyboxsecurity.com/rs/440-MPQ-510/images/Skybox_Report_Vulnerability_and_Threat_Trends_2019.pdf`.

[8] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, *{DR}.{CHECKER}: A soundy analysis for linux kernel drivers*, in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 1007–1024, 2017.

[9] C. Lattner and V. Adve, *Llvm: A compilation framework for lifelong program analysis & transformation*, in *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[10] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, *The attack of the clones: A study of the impact of shared code on vulnerability patching*, in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 692–708, IEEE, 2015.

[11] M. A. McQueen, T. A. McQueen, W. F. Boyer, and M. R. Chaffin, *Empirical estimates and observations of 0day vulnerabilities*, in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pp. 1–12, IEEE, 2009.

[12] A. Arora, R. Krishnan, R. Telang, and Y. Yang, *An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure*, *Information Systems Research* **21** (2010), no. 1 115–132.

[13] J. Jang, A. Agrawal, and D. Brumley, *Redebug: finding unpatched code clones in entire os distributions*, in *Security and Privacy (SP), 2012 IEEE Symposium on*, pp. 48–62, IEEE, 2012.

[14] "2016 android security bulletins." `source.android.com/security/bulletin/2016.html`. Accessed: 2017-02-11.

[15] S. Rastkar and G. C. Murphy, *Why did this code change?*, in *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, (Piscataway, NJ, USA), pp. 1193–1196, IEEE Press, 2013.

[16] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, *Relink: Recovering links between bugs and changes*, in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), pp. 15–25, ACM, 2011.

[17] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, *Is it a bug or an enhancement?: A text-based approach to classify change requests*, in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, (New York, NY, USA), pp. 23:304–23:318, ACM, 2008.

[18] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, *A machine learning approach for text categorization of fixing-issue commits on cvs*, in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, p. 6, ACM, 2010.

[19] G. Bavota, *Mining unstructured data in software repositories: Current and future trends*, in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 5, pp. 1–12, IEEE, 2016.

[20] E. A. Santos and A. Hindle, *Judging a commit by its cover: correlating commit message entropy with build status on travis-ci*, in *Proceedings of the 13th*

International Conference on Mining Software Repositories, pp. 504–507, ACM, 2016.

[21] "The biggest and weirdest commits in linux kernel git history." `www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history`. Accessed: 2017-02-15.

[22] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, *Symdiff: A language-agnostic semantic diff tool for imperative programs*, in *International Conference on Computer Aided Verification*, pp. 712–717, Springer, 2012.

[23] S. K. Lahiri, K. Vaswani, and C. A. R. Hoare, *Differential static analysis: Opportunities, applications, and challenges*, in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, (New York, NY, USA), pp. 201–204, ACM, 2010.

[24] D. Binkley, *Using semantic differencing to reduce the cost of regression testing*, in *Software Maintenance, 1992. Proceerdings., Conference on*, pp. 41–50, IEEE, 1992.

[25] D. Binkley, R. Capellini, L. R. Raszewski, and C. Smith, *An implementation of and experiment with semantic differencing*, in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pp. 82–91, IEEE, 2001.

[26] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, *Dex: A semantic-graph differencing tool for studying changes in large code bases*, in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pp. 188–197, IEEE, 2004.

[27] P. D. Marinescu and C. Cadar, *Katch: high-coverage testing of software patches*, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 235–245, ACM, 2013.

[28] R. P. Buse and W. R. Weimer, *Automatically documenting program changes*, in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), pp. 33–42, ACM, 2010.

[29] S. Person, M. B. Dwyer, S. Elbaum, and C. S. PÇŐsÇŐreanu, *Differential symbolic execution*, in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pp. 226–237, ACM, 2008.

[30] D. Gao, M. K. Reiter, and D. Song, *Binhunt: Automatically finding semantic differences in binary programs*, in *International Conference on Information and Communications Security*, pp. 238–255, Springer, 2008.

[31] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, *Symbolic execution for software testing in practice: preliminary assessment*, in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 1066–1071, ACM, 2011.

[32] "Linux kernel configuration." `http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html`. Accessed: 2017-02-13.

[33] A. Machiry, N. Redini, E. Camelini, C. Kruegel, and G. Vigna, *SPIDER: Enabling Fast Patch Propagation in Related Software Repositories (conditionally accepted)*, in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[34] N. D. Matsakis and F. S. Klock II, *The rust language*, in *ACM SIGAda Ada Letters*, vol. 34, pp. 103–104, ACM, 2014.

[35] A. Zeng and W. Crichton, *Identifying barriers to adoption for rust through online discourse*, *arXiv preprint arXiv:1901.01001* (2019).

[36] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, *Addresssanitizer: A fast address sanity checker*, in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 309–318, 2012.

[37] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, *Softbound: Highly compatible and complete spatial memory safety for c*, *ACM Sigplan Notices* **44** (2009), no. 6 245–258.

[38] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, *Cets: compiler enforced temporal safety for c*, *ACM Sigplan Notices* **45** (2010), no. 8 31–40.

[39] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, *Cyclone: A safe dialect of c.*, in *USENIX Annual Technical Conference, General Track*, pp. 275–288, 2002.

[40] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, *Checked c: Making c safe by extension*, in *2018 IEEE Cybersecurity Development (SecDev)*, pp. 53–60, IEEE, 2018.

[41] A. Ruef, L. Lampropoulos, I. Sweet, D. Tarditi, and M. Hicks, *Achieving safety incrementally with checked c*, in *International Conference on Principles of Security and Trust*, pp. 76–98, Springer, 2019.

[42] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, *Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World*, in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[43] X. Ge and T. Jaeger, *Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture*, in *Proceedings of the Mobile Security Technologies 2014 Workshop (MoST)*, 2014.

[44] A. M. Azab, K. Swidowski, J. M. Bhutkar, W. Shen, R. Wang, and P. Ning, *Skee: A lightweight secure kernel-level execution environment for arm*, 2016.

[45] Samsung Knox News, "Real-time Kernel Protection (RKP)." `https://www2.samsungknox.com/en/blog/real-time-kernel-protection-rkp`, 2016.

[46] H. Sun, K. Sun, Y. Wang, J. Jing, and S. Jajodia, *TrustDump: Reliable Memory Acquisition on Smartphones*, in *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS)*, 2014.

[47] J. Williams, *Inspecting data from the safety of your trusted execution environment*, in *BlackHat USA*, 2015.

[48] H. Sun, K. Sun, Y. Wang, and J. Jing, *Trustotp: Transforming smartphones into secure one-time password tokens*, in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 976–988, ACM, 2015.

[49] D. Rosenberg, *Reflections on trusting trustzone*, *BlackHat USA* (2014).

[50] Qualcomm, *Qualcomm mobile security*, 2018. `https://www.qualcomm.com/solutions/mobile-computing/features/security`.

[51] laginimaineb, *Exploring qualcomms secure execution environment*, 2016. `http://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html`.

[52] D. Shen, *Attacking your "Trusted Core," Exploiting TrustZone on Android*, in *BlackHat USA*, 2015.

[53] N. Keltner, "Here Be Dragons: Vulnerabilities in TrustZone." `https://atredispartners.blogspot.com/2014/08/here-be-dragons-vulnerabilities-in.html`, 2014.

[54] D. Rosenberg, *Reflections on Trusting TrustZone*, in *BlackHat USA*, 2014.

[55] D. Shen, *Exploiting trustzone on android*, *Black Hat USA* (2015).

[56] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, *Truspy: Cache side-channel information leakage from the secure world on arm devices.*, *IACR Cryptology ePrint Archive* **2016** (2016) 980.

[57] S. K. Bukasa, R. Lashermes, H. Le Bouder, J.-L. Lanet, and A. Legay, *How trustzone could be bypassed: Side-channel attacks on a modern system-on-chip*, in *IFIP International Conference on Information Security Theory and Practice*, pp. 93–109, Springer, 2017.

[58] A. Tang, S. Sethumadhavan, and S. Stolfo, *Clkscrew: exposing the perils of security-oblivious energy management*, in *26th USENIX Security Symposium*, pp. 1057–1074, 2017.

[59] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, *Quire: Lightweight Provenance for Smart Phone Operating Systems*, in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[60] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, *Towards Taming Privilege-Escalation Attacks on Android*, in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, 2012.

[61] R. Johnson and D. Wagner, *Finding user/kernel pointer bugs with type inference*, in *Proceedings of the 2004 USENIX Conference on Security*, SEC'04, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2004.

[62] J. S. Foster, T. Terauchi, and A. Aiken, *Flow-sensitive type qualifiers*, in *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), pp. 1–12, ACM, 2002.

[63] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, *Apisan: Sanitizing api usages through semantic cross-checking*, in *Proceedings of the 2016 USENIX Conference on Security*, SEC'16, pp. 363–378, USENIX Association.

[64] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, *Thorough static analysis of device drivers*, *ACM SIGOPS Operating Systems Review* **40** (2006), no. 4 73–85.

[65] M. J. Renzelmann, A. Kadav, and M. M. Swift, *Symdrive: Testing drivers without devices*, in *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 279–292, USENIX Association, 2012.

[66] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, *Improving integer security for systems with kint*, in *Proceedings of the 2012 USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (Berkeley, CA, USA), pp. 163–177, USENIX Association, 2012.

[67] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, *Precise and scalable detection of double-fetch bugs in os kernels*, in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 661–678, IEEE, 2018.

[68] I. S. Zakharov, M. U. Mandrykin, V. S. Mutilin, E. M. Novikov, A. K. Petrenko, and A. V. Khoroshilov, *Configurable toolset for static verification of operating systems kernel modules*, *Program. Comput. Softw.* **41** (Jan., 2015) 49–64.

[69] V. Mutilin, E. Novikov, K. A. Strakh AV, and P. Shved, *Linux driver verification [linux driver verification architecture]*, *Trudy ISP RĐŘN [The Proceedings of ISP RAS]* **20** (2011) 163–187.

[70] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, *Software verification with blast*, in *Proceedings of the 2003 International Conference on Model Checking Software*, SPIN'03, (Berlin, Heidelberg), pp. 235–239, Springer-Verlag, 2003.

[71] H. Chen and D. Wagner, *Mops: An infrastructure for examining security properties of software*, in *Proceedings of the 2002 ACM Conference on Computer and Communications Security*, CCS '02, (New York, NY, USA), pp. 235–244, ACM, 2002.

[72] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, *Automatic inference of search patterns for taint-style vulnerabilities*, in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, (Washington, DC, USA), pp. 797–812, IEEE Computer Society, 2015.

[73] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, *Chucky: Exposing missing checks in source code for vulnerability discovery*, in *Proceedings of the 2013 ACM Conference on Computer and Communications Security*, CCS '13, (New York, NY, USA), pp. 499–510, ACM, 2013.

[74] F. Yamaguchi, M. Lottmann, and K. Rieck, *Generalized vulnerability extrapolation using abstract syntax trees*, in *Proceedings of the 2012 Annual Computer Security Applications Conference*, ACSAC '12, (New York, NY, USA), pp. 359–368, ACM, 2012.

[75] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, *Modeling and discovering vulnerabilities with code property graphs*, in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, (Washington, DC, USA), pp. 590–604, IEEE Computer Society, 2014.

[76] I. Secure Software, "Rats - rough auditing tool for security." `https://github.com/andrew-d/rough-auditing-tool-for-security`, December, 2013.

[77] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, *Its4: A static vulnerability scanner for c and c++ code*, in *Proceedings of the 2000 Annual Computer Security Applications Conference*, ACSAC '00, (Washington, DC, USA), pp. 257–, IEEE Computer Society, 2000.

[78] D. A. Wheeler, *Flawfinder*, 2011.

[79] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, *Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits*, in *Proceedings of the 2015 ACM Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 426–437, ACM, 2015.

[80] J. Yang, T. Kremenek, Y. Xie, and D. Engler, *Meca: An extensible, expressive system and language for statically checking security properties*, in *Proceedings of the 2003 ACM Conference on Computer and Communications Security*, CCS '03, (New York, NY, USA), pp. 321–334, ACM, 2003.

[81] M. Das, S. Lerner, and M. Seigle, *Esp: Path-sensitive program verification in polynomial time*, in *Proceedings of the 2002 ACM Conference on Programming Language Design and Implementation*, PLDI '02, (New York, NY, USA), pp. 57–68, ACM, 2002.

[82] S. Boyd-Wickizer and N. Zeldovich, *Tolerating malicious device drivers in linux*, in *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2010.

[83] M. M. Swift, B. N. Bershad, and H. M. Levy, *Improving the reliability of commodity operating systems*, in *Proceedings of the 2003 ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 207–222, ACM, 2003.

[84] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, *Vulpecker: an automated vulnerability detection system based on code similarity analysis*, in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pp. 201–213, ACM, 2016.

[85] H. Li, H. Kwon, J. Kwon, and H. Lee, *A scalable approach for vulnerability discovery based on security patches*, in *International Conference on Applications and Techniques in Information Security*, pp. 109–122, Springer, 2014.

[86] D. Brumley, P. Poosankam, D. Song, and J. Zheng, *Automatic patch-based exploit generation is possible: Techniques and implications*, in *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 143–157, IEEE, 2008.

[87] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, *C aramel: detecting and fixing performance problems that have non-intrusive fixes*, in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 902–912, IEEE Press, 2015.

[88] S. Son, K. S. McKinley, and V. Shmatikov, *Fix me up: Repairing access-control bugs in web applications.*, in *NDSS*, 2013.

[89] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, *Semantic patch inference*, in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*, pp. 382–385, IEEE, 2012.

[90] M. Monperrus, *Automatic software repair: a bibliography*, *University of Lille, Tech. Rep. hal-01206501* (2015).

[91] T. Zhang, M. Song, J. Pinedo, and M. Kim, *Interactive code review for systematic changes*, in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pp. 111–122, IEEE Press, 2015.

[92] N. Meng, M. Kim, and K. S. McKinley, *Systematic editing: generating program transformations from an example*, *ACM SIGPLAN Notices* **46** (2011), no. 6 329–342.

[93] F. Long, P. Amidon, and M. Rinard, *Automatic inference of code transforms and search spaces for automatic patch generation systems*, .

[94] P. Kreutzer, G. Dotzler, M. Ring, B. M. Eskofier, and M. Philippsen, *Automatic clustering of code changes*, in *Proceedings of the 13th International Conference on Mining Software Repositories*, pp. 61–72, ACM, 2016.

[95] A. E. Hassan, *The road ahead for mining software repositories*, in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pp. 48–57, IEEE, 2008.

[96] E. Giger, M. Pinzger, and H. C. Gall, *Comparing fine-grained source code changes and code churn for bug prediction*, in *Proceedings of the 8th Working Conference on Mining Software Repositories*, pp. 83–92, ACM, 2011.

[97] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, *Bootstomp: on the security of bootloaders in mobile devices*, in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pp. 781–798, 2017.

[98] M. Lu, *TrustZone, TEE and Trusted Video Path Implementation Considerations*, 2013.

[99] ARM, "Securing the Future of Authentication with ARM TrustZone-based Trusted Execution Environment and Fast Identity Online (FIDO)." `https://www.arm.com/files/pdf/TrustZone-and-FIDO-white-paper.pdf`, 2015.

[100] K. Kostiainen, J.-E. Ekberg, N. Asokan, and A. Rantala, *On-board Credentials with Open Provisioning*, in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security (Asia CCS)*, 2009.

[101] GlobalPlatform, *TEE System Architecture*, 2011.

[102] Qualcomm, "Qualcomm Secure Execution Environment Communicator (QSEECOM) driver." https://android.googlesource.com/kernel/msm.git/+/77cac325253126dd9e6c480d885aa51f1abf3c40/drivers/misc/qseecom.c.

[103] Trustonic, "Trustonic." https://www.trustonic.com/.

[104] H. Nahari, "TLK: A FOSS Stack for Secure Hardware Tokens." http://www.w3.org/2012/webcrypto/webcrypto-next-workshop/papers/webcrypto2014_submission_25.pdf, 2012.

[105] STMicroelectronics and Linaro Security Working Group, "Open Source TEE." https://github.com/OP-TEE/optee_os.

[106] J. Bangert, S. Bratus, R. Shapiro, and S. W. Smith, *The Page-Fault Weird Machine: Lessons in Instruction-less Computation*, in *Proceedings of the 7th USENIX Workshop on Offensive Technologies (WOOT)*, 2013.

[107] K. Lady, "Sixty Percent of Enterprise Android Phones Affected by Critical QSEE Vulnerability." https://duo.com/blog/sixty-percent-of-enterprise-android-phones-affected-by-critical-qsee-vulnerability, 2016.

[108] laginimaineb, "Bits, Please!." https://bits-please.blogspot.com/, 2016.

[109] AOSP, "/dev/ion driver!." https://lwn.net/Articles/480055/, 2006.

[110] Google, "QSEEComAPI.h." https://android.googlesource.com/platform/hardware/qcom/keymaster/+/master/QSEEComAPI.h, 2012.

[111] Qualcomm, "Msm scm communicator." https://android.googlesource.com/kernel/msm/+/android-5.1.0_r0.6/arch/arm/mach-msm/scm.c.

[112] J. Bennett, "Devices with Trustonic TEE." https://www.trustonic.com/news-events/blog/devices-trustonic-tee, 2015.

[113] Samsung, "Knox Technology." https://www.samsungknox.com/en/knox-technology, 2015.

[114] N. Golde and D. Komaromy, *Breaking Band: reverse engineering and exploiting the shannon baseband*, in *REcon*, 2016.

[115] Trustonic, "tee-mobicore-driver.daemon." https://github.com/TrustonicNwd/tee-mobicore-driver.daemon, 2016.

[116] Trustonic, "tee-mobicore-driver.kernel." https://github.com/TrustonicNwd/tee-mobicore-driver.kernel/blob/MC12/drivers/gud/MobiCoreDriver/fastcall.h, 2015.

[117] Trustonic, "trustonic-tee-user-space." https://github.com/Trustonic/trustonic-tee-user-space/blob/e3b0b06025605b06fc1e19588098e5011f6afc83/MobiCoreDriverLib/Daemon/MobiCoreDriverDaemon.cpp, 2015.

[118] B. McGillion, T. Dettenborn, T. Nyman, and N. Asokan, *Open-TEE–An Open Virtual Trusted Execution Environment*, in *Proceedings of the 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2015.

[119] GlobalPlatform, *TEE Internal Core API Specification v.1.1.1*, 2016.

[120] Linaro Security Working Group, "Linux Kernel, OP-TEE driver." https://github.com/linaro-swg/linux/blob/optee/drivers/tee/optee/core.c, 2016.

[121] OP-TEE, "optee_os." https://github.com/OP-TEE/optee_os/blob/master/core/arch/arm/kernel/tee_ta_manager.c, May, 2016.

[122] STMicroelectronics and Linaro Security Working Group, "OP-TEE non-secure world kernel driver." https://github.com/linaro-swg/linux/tree/optee/drivers/tee.

[123] STMicroelectronics and Linaro Security Working Group, "OP-TEE normal world client library." https://github.com/OP-TEE/optee_client.

[124] STMicroelectronics and Linaro Security Working Group, "OP-TEE non-secure world/secure world SMC call." https://github.com/linaro-swg/linux/blob/optee/drivers/tee/optee/call.c#L117.

[125] A. Machiry, "Shared memory allocated by tee linux kernel driver is not zeroed out." https://github.com/linaro-swg/linux/issues/13/, 2016.

[126] A. Machiry, "Shared Memory IDs are stored globally." https://github.com/linaro-swg/linux/issues/14/, 2016.

[127] A. Machiry, "Potential Heap Buffer overflow in tee_supp_com.c." https://github.com/OP-TEE/optee_linuxdriver/issues/52/, 2016.

[128] A. Machiry, "Potential invalid MEMREF translation, this could be used for bad." https://github.com/OP-TEE/optee_linuxdriver/issues/53/, 2016.

[129] Sierraware, "SierraWare Trusted Execution Environment." http://www.sierraware.com/, 2016.

[130] Sierraware, "Open Virtualizations SierraVisor and SierraTEE." http://openvirtualization.org, 2016.

[131] K. Rubinov, L. Rosculete, T. Mitra, and A. Roychoudhury, *Automated Partitioning of Android Applications for Trusted Execution Environments*, in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016.

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

[133] M. Egele, M. Woo, P. Chapman, and D. Brumley, *Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components*, in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[134] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, *ret2dir: Rethinking Kernel Isolation*, in *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[135] Huawei, "Security Advisory - Multiple Security Vulnerabilities in Huawei Smart Phone Products." http://www.huawei.com/en/psirt/security-advisories/huawei-sa-20161123-01-smartphone-en, 2016.

[136] Y. Jang, S. Lee, and T. Kim, *DrK: BreakingKernel Address Space Layout Randomization with Intel TSX*, in *BlackHat USA*, 2016.

[137] Lenovator, "HiKey (LeMaker version) 2GB RAM." http://www.lenovator.com/product/90.html.

[138] STMicroelectronics and Linaro Security Working Group, "OP-TEE Test Suite." https://github.com/OP-TEE/optee_test.

[139] C. Spensky, J. Stewart, A. Yerukhimovich, R. Shay, A. Trachtenberg, R. Housley, and R. K. Cunningham, *SoK: Privacy on Mobile Devices–It's Complicated*, *Proceedings on Privacy Enhancing Technologies* **2016** (2016), no. 3 96–116.

[140] J. V. Stoep, *Android: protecting the kernel*, *Linux Securit Summit* (August, 2016).

[141] "CVE-2016-5195.." Available from MITRE, CVE-ID CVE-2016-5195., May, 2016.

[142] D. Kirat, G. Vigna, and C. Kruegel, *Barecloud: Bare-metal analysis-based evasive malware detection*, in *Proceedings of the 2014 USENIX Conference on Security*, SEC'14, (Berkeley, CA, USA), pp. 287–301, USENIX Association, 2014.

[143] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, *Ether: Malware analysis via hardware virtualization extensions*, in *Proceedings of the 2008 ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), pp. 51–62, ACM, 2008.

[144] C. Spensky, H. Hu, and K. Leach, *Lo-phi: Low-observable physical host instrumentation for malware analysis*, in *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS)*, Internet Society, 2016.

[145] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, *A few billion lines of code later: Using static analysis to find bugs in the real world*, *Commun. ACM* **53** (Feb., 2010) 66–75.

[146] K. Lu, C. Song, T. Kim, and W. Lee, *Unisan: Proactive kernel memory initialization to eliminate data leakages*, in *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 920–932, ACM, 2016.

[147] S. Bugrara and A. Aiken, *Verifying the safety of user pointer dereferences*, in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, (Washington, DC, USA), pp. 325–338, IEEE Computer Society, 2008.

[148] B. Livshitz, *Soundness is not even necessary for most modern analysis applications, however, as many*, *Communications of the ACM* **58** (2015), no. 2.

[149] "CVE-2016-8470.." Available from MITRE, CVE-ID CVE-2016-8470., May, 2016.

[150] "CVE-2016-8471.." Available from MITRE, CVE-ID CVE-2016-8471., May, 2016.

[151] "CVE-2016-8472.." Available from MITRE, CVE-ID CVE-2016-8472., May, 2016.

[152] "CVE-2016-8433.." Available from MITRE, CVE-ID CVE-2016-8433., May, 2016.

[153] "CVE-2016-8448.." Available from MITRE, CVE-ID CVE-2016-8448., May, 2016.

[154] R. Tarjan, *Depth-first search and linear graph algorithms*, *SIAM journal on computing* **1** (1972), no. 2 146–160.

[155] T. Ball and S. K. Rajamani, *The slam project: Debugging system software via static analysis*, in *Proceedings of the 2002 ACM Symposium on Principles of Programming Languages*, POPL '02, (New York, NY, USA), pp. 1–3, ACM, 2002.

[156] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer, 2015.

[157] P. J. Salzman, M. Burian, and O. Pomerantz, "Hello World (part 3): The __init and __exit Macros." http://www.tldp.org/LDP/lkmpg/2.6/html/lkmpg.html#AEN245, May, 2007.

[158] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *An efficient method of computing static single assignment form*, in *Proceedings of the 1989 ACM Symposium on Principles of Programming Languages*, POPL '89, (New York, NY, USA), pp. 25–35, ACM, 1989.

[159] The LLVM Project, "The Often Misunderstood GEP Instruction." http://llvm.org/docs/GetElementPtr.html.

[160] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo, *Detecting stack based kernel information leaks*, in *Proceedings of the 2014 International Joint Conference SOCO'14-CISIS'14-ICEUTE'14*, pp. 321–331, Springer, 2014.

[161] F. M. Quintao Pereira, R. E. Rodrigues, and V. H. Sperle Campos, *A fast and low-overhead technique to secure programs against integer overflows*, in *Proceedings of the 2013 International Symposium on Code Generation and Optimization*, CGO '13, (Washington, DC, USA), pp. 1–11, IEEE Computer Society, 2013.

[162] The Clang Project, "clang: a C language family frontend for LLVM." http://clang.llvm.org/.

[163] The Linux Foundation, "LLVMLinux Project Overview." http://llvm.linuxfoundation.org/index.php/Main_Page.

[164] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers: Where the Kernel Meets the Hardware*. " O'Reilly Media, Inc.", 2005.

[165] P. Mochel and M. Murphy, "sysfs - _The_ filesystem for exporting kernel objects." https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[166] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre, *Communicating between the kernel and user-space in linux using netlink sockets*, *Software: Practice and Experience* **40** (2010), no. 9 797–810.

[167] "The linux watchdog timer driver core kernel api.." https://www.kernel.org/doc/Documentation/watchdog/watchdog-kernel-api.txt. Accessed: 2017-02-14.

[168] J. Aubert and D. Tuset, "c2xml." http://c2xml.sourceforge.net/.

[169] H. ZHANG, X.-h. LI, B. LIU, and X. QIAN, *The video device driver programming and profiting based on v4l2 [j]*, *Computer Knowledge and Technology* **15** (2010) 062.

[170] "CVE-2016-2068.." Available from MITRE, CVE-ID CVE-2016-2068., 2016.

[171] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis." http://cppcheck.sourceforge.net/, December, 2016.

[172] L. Torvalds, J. Triplett, and C. Li, *Sparse–a semantic parser for c*, *see http://sparse. wiki. kernel. org* (2007).

[173] P. J. Guo and D. Engler, *Linux kernel developer responses to static analysis bug reports*, in *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIXATC'09, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2009.

[174] C. Wressnegger, F. Yamaguchi, A. Maier, and K. Rieck, *Twice the bits, twice the trouble: Vulnerabilities induced by migrating to 64-bit platforms*, in *Proceedings of the 2016 ACM Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), pp. 541–552, ACM, 2016.

[175] "Kernel modules." http://tldp.org/LDP/lkmpg/2.6/html/x427.html. Accessed: 2017-05-26.

[176] "Imagemagick: Convert different image formats.." https://github.com/ImageMagick/ImageMagick. Accessed: 2017-05-26.

[177] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, *How do software engineers understand code changes?: An exploratory study in industry*, in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, (New York, NY, USA), pp. 51:1–51:11, ACM, 2012.

[178] J. Admanski and S. Howard, *Autotest-testing the untestable*, in *Proceedings of the Linux Symposium*, Citeseer, 2009.

[179] E. Rescorla, *Security holes... who cares?*, in *USENIX Security*, Washington, DC, 2003.

[180] "Android msm kernel." https://android.googlesource.com/kernel/msm.git/+/android-msm-angler-3.10-marshmallow-mr1. Accessed: 2017-02-13.

[181] F. Li and V. Paxson, *A large-scale empirical study of security patches*, in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215, ACM, 2017.

[182] "libebml fixed a vulnerability but no cve was assigned." https://twitter.com/wdormann/status/1154138404910768134. Accessed: 2017-07-25.

[183] "Vlc media player affected by a major vulnerability in a 3rd library, libebml." https://hub.packtpub.com/vlc-media-player-affected-by-a-major-vulnerability-in-a-3rd-library-libebml-updating-to-the-latest-version-may-help/. Accessed: 2017-07-25.

[184] "libembl security fix without a cve id." https://github.com/Matroska-Org/libebml/commit/05beb69ba60acce09f73ed491bb76f332849c3a0. Accessed: 2017-07-25.

[185] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, *Hackers vs. testers: A comparison of software vulnerability discovery processes*, in *Security and Privacy (SP), 2018 IEEE Symposium on*, IEEE, 2018.

[186] "Introducing security alerts on github." https://github.com/blog/2470-introducing-security-alerts-on-github. Accessed: 2017-02-13.

[187] F. Chow, S. Chan, R. Kennedy, S.-M. Liu, R. Lo, and P. Tu, *A new algorithm for partial redundancy elimination based on ssa form*, in *ACM Sigplan Notices*, vol. 32, pp. 273–286, ACM, 1997.

[188] J. Ferrante, K. J. Ottenstein, and J. D. Warren, *The program dependence graph and its use in optimization*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **9** (1987), no. 3 319–349.

[189] Y. Kang, B. Ray, and S. Jana, *Apex: Automated inference of error specifications for c apis*, in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 472–482, ACM, 2016.

[190] Y. Tian and B. Ray, *Automatically diagnosing and repairing error handling bugs in c*, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 752–762, ACM, 2017.

[191] L. De Moura and N. Bjørner, *Z3: An efficient smt solver*, in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.

[192] B. C. Pierce and D. N. Turner, *Local type inference*, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **22** (2000), no. 1 1–44.

[193] F. E. Allen, *Control flow analysis*, in *ACM Sigplan Notices*, vol. 5, pp. 1–19, ACM, 1970.

[194] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, *Fine-grained and accurate source code differencing*, in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pp. 313–324, ACM, 2014.

[195] "Use of goto in systems code." `https://blog.regehr.org/archives/894`. Accessed: 2019-07-13.

[196] "Using goto in linux kernel code." `https://koblents.com/Ches/Links/Month-Mar-2013/20-Using-Goto-in-Linux-Kernel-Code/`. Accessed: 2019-07-13.

[197] "Error handling via goto in c." `https://ayende.com/blog/183521-C/error-handling-via-goto-in-c`. Accessed: 2019-07-13.

[198] M. Barnett and K. R. M. Leino, *Weakest-precondition of unstructured programs*, in *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, (New York, NY, USA), pp. 82–87, ACM, 2005.

[199] J. M. Cardoso and P. C. Diniz, *Modeling loop unrolling: Approaches and open issues*, in *International Workshop on Embedded Computer Systems*, pp. 224–233, Springer, 2004.

[200] W. Amme and E. Zehendner, *Data dependence analysis in programs with pointers*, *Parallel Computing* **24** (1998), no. 3-4 505–525.

[201] P. Wadler, *The essence of functional programming*, in *POPL*, vol. 92, pp. 1–14, Citeseer, 1992.

[202] Y. Zhou and A. Sharma, *Automated identification of security issues from commit messages and bug reports*, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 914–919, ACM, 2017.

[203] "Openssl bug fix for cve-2016-0703." `https://git.openssl.org/?p=openssl.git;a=commit;h=ae50d8270026edf5b3c7f8aaa0c6677462b33d97`. Accessed: 2017-02-13.

[204] D. Bleichenbacher, *Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1*, in *Advances in CryptologyâĂŤCRYPTO'98*, pp. 1–12, Springer, 1998.

[205] "Git branch full name." android-msm-angler-3.10-oreo-r6. Accessed: 2017-02-11.

[206] Z. Huang, D. Lie, G. Tan, and T. Jaeger, *Using safety properties to generate vulnerability patches*, in *2019 2019 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 1174–1189, IEEE Computer Society, may, 2019.

[207] C. Lattner, *Llvm and clang: Next generation compiler technology*, in *The BSD Conference*, pp. 1–2, 2008.

[208] "Memory corruption is still the most prevalent security vulnerability." `https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/`. Accessed: 2020-02-11.

[209] J. Duan, Y. Yang, J. Zhou, and J. Criswell, *Refactoring the freebsd kernel with checked c*, .

[210] "Pointer-intensive benchmark suite." http://pages.cs.wisc.edu/ austin/ptr-dist.html. Accessed: 2017-02-11.

[211] B. Hardekopf and C. Lin, *Flow-sensitive pointer analysis for millions of lines of code*, in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 289–298, IEEE, 2011.

[212] F. Brown, D. Stefan, and D. Engler, *Sys: a static/symbolic tool for finding good bugs in good (browser) code*, in *29th USENIX Security Symposium*, 2020.