UNIVERSITY OF CALIFORNIA
Santa Barbara

Analyzing and Securing Firmware for IoT Devices

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

Doctor of Philosophy

in

Computer Science

by

Nilo Redini

Committee in Charge:

Professor Giovanni Vigna, Co-Chair

Professor Christopher Kruegel, Co-chair

Professor Ben Hardekopf

December 2020

The Dissertation of
Nilo Redini is approved:

Professor Ben Hardekopf

Professor Christopher Kruegel, Co-chair

Professor Giovanni Vigna, Co-chair

November 2020

Analyzing and Securing Firmware

for IoT Devices

by

Nilo Redini

Acknowledgements

I want to thank my family, my friends, and all the people who have been close to me during this incredible journey. In particular, I want to thank my parents, who have never stopped believing in me and supporting me in all of my decisions. My brother Matteo and my sister-in-law Antonella, for all the good times and laughs during all these years, and all the stolen hair straighteners. My friends Alessandro, Serena, and Arjola for being "virtually" close to me during the hard times. I want to thank Yanick, Machiry, Britt, Francesco, Margery, Romain, and Mikel for all the talks and nights spent together. A special thanks go to Antonio Bianchi, who took me aside on my first day at UCSB and explained to me how to be a successful Ph.D. student. I want to thank my high school teacher professor Domenico Iracá for making me fall in love with computer science, and Tommaso Cucinotta for introducing me to the world of research. I want to thank my advisors, Giovanni Vigna and Christopher Kruegel, for being the best advisors I could ask for and for teaching me how to be an independent researcher and critical thinker. Finally, I want to thank my grandma Alda, to whom, if I could, I would just say "nonna, alla fine in California ci son finito per davvero".

Curriculum Vitæ

Nilo Redini

Education

2014 – 2020	PhD in Computer Science
	University of California, Santa Barbara
2010 – 2013	Master's Degree in Computer Engineering
	Università di Pisa, Pisa, Italy
2005 – 2010	Bachelor's Degree in Computer Engineering
	Università di Pisa, Pisa, Italy

Experience

Mar 2020 – Jun 2020	Interim Engineering Intern, Qualcomm, CA
	Advisor: Murali Somanchy
Apr 2019 – Jun 2019	Visiting PhD Student, University of Pennsylvania, PA
	Advisor: Prof. Mayur Naik
Mar 2013 – Aug 2013	Research Scholarship, Università di Pisa, Italy
	Advisor: Prof. Luciano Lenzini
Jul 2012 – Nov 2012	Interim Engineering Intern, Alcatel-Lucent Bell-Labs, Ireland
	Advisor: Prof. Tommaso Cucinotta

Publications

1. Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, Giovanni Vigna, *"DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices"*. In Proceedings of the IEEE Symposium on Security & Privacy (S&P), May, 2021.

2. Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, *"KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware"*. In Proceedings of the IEEE Symposium on Security & Privacy (S&P), May, 2020.

3. Aravind Machiry, Nilo Redini, Eric Cammellini, Christopher Kruegel, Giovanni Vigna, *"SPIDER: Enabling Fast Patch Propagation in Related Software Repositories"*. In Proceedings of the IEEE Symposium on Security & Privacy (S&P), May, 2020.

4. Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna, *"Toward the Analysis of Embedded Firmware through Automated Re-hosting"*. Symposium on Research in Attacks, Intrusion, and Defenses (RAID), Beijing, Sep, 2019.

5. Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, Christopher Kruegel, *"Bintrimmer: Towards Static Binary Debloating Through Abstract Inter-*

pretation". In Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), Aug, 2019.

6. Aravind Machiry, Nilo Redini, Eric Gustafson, Hojjat Aghakhani, Christopher Kruegel, Giovanni Vigna, *"Towards Automatically Generating a Sound and Complete Dataset for Evaluating Static Analysis Tools"*. Workshop on Binary Analysis Research (BAR), San Diego, Feb, 2019.

7. Aravind Machiry, Nilo Redini, Eric Gustafson, Yanick Fratantonio, Yung Ryn Choe, Christopher Kruegel, Giovanni Vigna, *"Using Loops For Malware Classification Resilient to Feature-unaware Perturbations"*. Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC), San Juan, Dec, 2018.

8. Nilo Redini, Aravind Machiry, Dipanjan Das, Yanick Fratantonio, Antonio Bianchi, Eric Gustafson, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, *"BootStomp: On the Security of Bootloaders in Mobile Devices"*. 26th USENIX Security Symposium (USENIX), Vancouver, Aug, 2017.

9. Francesco Disperati, Dario Grassini, Enrico Gregori, Alessandro Improta, Luciano Lenzini, Davide Pellegrino, Nilo Redini, *"SmartProbe: a Bottleneck Capacity Estimation Tool for Smartphones"*. IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing (GreenCom-iThings-CPSCom), Beijing, 2013.

10. Tommaso Cucinotta, Gianluca Dini, Nilo Redini, *"Access Control for the Pepys Internet-wide File-System"*. In Proceedings of the 7th International Workshop on Plan 9 (IWP9 2012), Dublin, Nov, 2012.

Awards

2019	First place in CSAW Embedded Security Challenge (ESC)
2013	Winner of research scholarship at Università di Pisa
2012	Winner of the "Erasmus Placement" for a paid internship in a company of the European Union

Abstract

Analyzing and Securing Firmware for IoT Devices

Nilo Redini

Internet of Things (IoT) devices have rooted themselves in the everyday life of billions of people. While they automate and simplify many aspects of the users' lives, the widespread usage of IoT devices constitutes a security concern for our modern society. Aside from the privacy and safety implications of having a smart door lock that could succumb to an Internet-based attack, or a smoke detector that an assailant could disable by connecting to it from a compromised light bulb, vulnerabilities in these devices have wider implications. Recent large-scale attacks have shown that the sheer number of Internet-connected IoT devices poses a severe threat to the Internet infrastructure. The most prominent example is represented by the Mirai botnet that, in 2016, compromised millions of devices and leveraged them in denial-of-service attacks to disrupt core Internet services and shut down websites.

For these reasons, it is of crucial importance to assess the security of IoT devices. Analyzing and securing IoT devices present different and specific challenges than analyzing and securing traditional desktop computers. The main reason is that IoT devices are manufactured by a plethora of different vendors, which often use vendor-specific

hardware and software (or *firmware*) for their products. Given the heterogeneity and widespread usage of IoT devices, we need novel, automated, and scalable solutions able to improve the security of these devices.

During my Ph.D., I approached the problem of securing IoT devices from different angles and using different strategies, which I present in detail in this dissertation. First, I introduce the IoT landscape, with particular attention to the peculiarities that characterize embedded firmware. Then, I present in detail my work that advances the state of the art of firmware security. In particular, I present (i) BOOTSTOMP, a novel tool to find bugs in bootloaders for embedded devices, (ii) KARONTE, a novel static analysis approach to track data flows across the different components of a firmware sample to precisely uncover security vulnerabilities, (iii) BINTRIMMER, a tool that relies on a novel abstract domain (called *Signedness-Agnostic Strided Interval*) to perform code debloating on binaries, thus decreasing the attack surface that could be used by an attacker to harm end-users, and, finally, (iv) DIANE, a novel approach to fuzz IoT devices that leverages the logic of the device's companion app (i.e., the application commonly used to interact with IoT devices). I evaluate the performance of the proposed approaches and show that the developed tools are effective in improving the security of firmware for IoT devices.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The term *Internet of Things* (*IoT*) was first introduced in 1999 by Kevin Ashton, as a way to describe a system where the physical world is connected to the Internet through ubiquitous sensors [6]. In the literature, the term Internet of Things was later generalized and defined as the pervasive presence around us of a variety of Internet-connected devices (or *things*) – such as Radio-Frequency Identification (RFID) tags, sensors, mobile phones, etc. – which interact with each other to reach common goals [42].

These devices (called *IoT devices*) act to automate people's daily routines and mundane tasks to increase individual efficiency and productivity [33,106,113,144]. As market research suggests, the adoption of IoT devices is continuously growing, and their widespread usage will enable industries, and more in general the world, with opportunities that, until now, were deemed impossible. For instance, an increasing number of American urban areas are already moving toward the *Smart City* [146] model, where

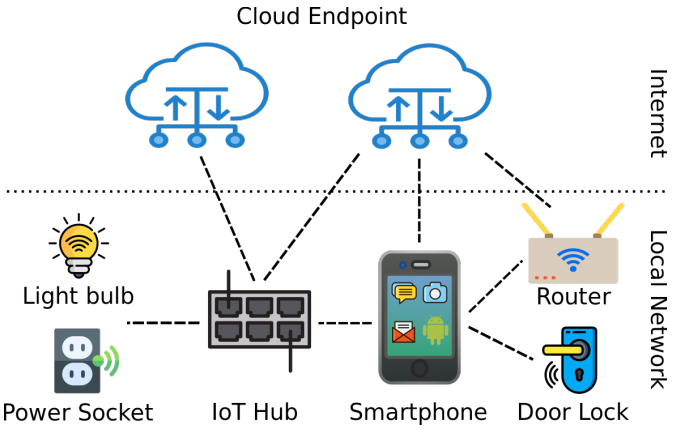


Figure 1.1: Typical IoT deployment.

public administrations use the data generated by IoT devices to provide citizens with new services (e.g., car-sharing services [69]).

IoT Ecosystem. A typical IoT setup is depicted in Figure 1.1. Generally, an IoT device is accompanied by its *companion app*, which is the mobile application used to configure and interact with the device. The companion app can communicate directly with the device, indirectly through a hub, or indirectly through the vendor's cloud. The first approach is by far the most employed: a recent survey [3] showed that out of 45 analyzed IoT devices, 43 (95.56%) use a local connection (either wired, WiFi, or Bluetooth) to communicate with the companion app, at least during their pairing phase.

Architecturally, IoT devices are generally composed of a main printed circuit board (or PCB) and one or more external peripherals (e.g., the motor of a smart lock or a camera). In turn, the PCB is composed of several different chips – such as low-power microcontrollers, BLE SoC (low-energy Bluetooth integrated circuit), external EPROM

memories, and peripherals' controllers – that exchange data with the physical world and manage the device's peripherals accordingly. IoT devices usually use RISC processors, predominantly ARM, as their principal microcontrollers. Unlike traditional computers where Intel and AMD are the leading processor manufacturers, ARM does not produce processors, rather it provides specifications (and support) for companies to manufacture chips that are compliant with the ARM instruction set architecture. However, as microcontroller manufacturers do not need to follow ARM guidelines closely, the actual hardware mounted on IoT devices is very diverse, and presents different characteristics.

Similarly, IoT manufacturers usually rely on proprietary software (or *firmware)* to manage the device's specific hardware. This software is either a single small program that manages directly the device's peripherals without relying on any abstraction layers (these systems are usually referred to as "bare-metal" systems), or a more complex system made up of different *components* that interact to accomplish various tasks. The latter type of firmware can either be a Linux distribution packaged with a collection of different (vendor-specific) binaries, or a large single-binary embedded OS (commonly referred to as "blob firmware") composed of a set of different modules – a classic example of blob firmware is a set of applications packaged together with a real-time operating system, such as VxWorks [126]. In general, Linux-based firmware is, by far, the most ubiquitous: a large-scale experiment analyzed tens of thousands of firmware samples and found that 86% of them were Linux-based [27]. Similar to other Linux-

based systems, Linux-based firmware images include a large number of interdependent binaries.

Overall, the hardware and software that govern IoT devices are very diverse, which, in recent years, led the security research community to focus on the IoT ecosystem.

1.1 Security of IoT devices

Given their large adoption and diversity, the security of IoT devices is of crucial importance. Unfortunately, very much like their popularity, the number of reported security and privacy violations concerning these devices is increasing too. According to the 2020 IBM X-Force Threat Intelligence Index [60], attacks against these systems have increased over 2000% since 2018.

Cyber-attacks on IoT devices might be successful mainly because of three reasons: (1) users' bad practices, (2) vendors' bad practices, and (3) bugs in the firmware running on these IoT devices.

A study conducted by Cui and Stolfo [32] on an Internet-scale active probing of IoT devices, showed that around 540,000 embedded devices (13% of all discovered embedded devices) use default credentials and that these devices belong to various realms such as enterprises, government organizations, Internet Service Providers (ISPs), educational institutions, and private networks [90]. Unfortunately, vendors are prone to

adopt unsafe practices too. Costin et al. [27] performed a study over 32,000 firmware samples and recovered the login credentials for 681 distinct firmware images, which belonged to 27 different vendors. Furthermore, they successfully extracted 109 private RSA keys from 428 firmware samples, and 56 self-signed certificates out of 344 firmware samples. In total, they were able to obtain 41 self-signed SSL certificates together with their corresponding private RSA keys.

Although it is hard to estimate the number of IoT devices whose firmware is affected by security-related flaws (e.g., buffer overflows), the numbers are not encouraging. In June 2020, the Israeli security firm JSOF revealed a collection of vulnerabilities called Ripple20, which could be exploited to compromise more than 100,000 Internet-connected devices [139].

These vulnerabilities undermine the security of our modern society. For instance, in 2016, the Mirai botnet [73] infected hundreds of thousands of vulnerable IoT devices to disrupt core Internet services and shut down high-profile websites such as Twitter and Netflix. In order to guarantee the safety and privacy of our *smart* world, we need to secure IoT devices. Though the literature contains many approaches (based on both static and dynamic analyses) to find flaws in computer programs, given the diversity of the IoT ecosystem, these approaches rarely can be adapted to the firmware running on IoT devices.

Dynamic Analysis. Gray-box fuzzing is the most commonly employed dynamic analysis technique to find bugs in software [52, 71, 82, 103, 104, 120, 133]. However, this fuzzing technique usually requires access to the runtime state of the target program, thus making approaches based on gray-box fuzzing unsuitable for IoT devices, as most devices are shipped with disabled hardware debug capabilities [80, 86]. For this reason, black-box approaches, which do not require access to a device's firmware, are usually employed in the IoT domain. However, the existing black-box approaches [66, 68, 141] require knowledge about the data format accepted by the device under analysis. Consequently, given the heterogeneity and lack of documentation of the protocols adopted by IoT devices, these approaches are not readily applicable. Other approaches have proposed to fuzz IoT devices by emulating the corresponding firmware [20, 50, 119, 150]. Unfortunately, a faithful emulation of a firmware image is a hard problem, and these approaches have scalability issues. In fact, during the booting process of an IoT device, the firmware usually checks the status of the device's peripherals, and, if a peripheral is not detected, the firmware enters into a fault state and does not proceed any further.

Static Analysis. The diversity of IoT devices and their software poses a limitation to the scalability and efficiency of traditional vulnerability-finding approaches. As a result, researchers usually focus their work on specific hardware architectures [34, 50], or specific vulnerability types [35, 39, 77, 92, 115]. In recent years, researchers have proposed techniques to automatically identify vulnerabilities in firmware distributions, generally

by unpacking them into components, which are then analyzed in isolation [20, 115]. Nonetheless, despite these advances in vulnerability discovery techniques, state-of-the-art approaches are insufficient, and vulnerabilities persist. A key reason behind the insufficiency of current techniques is that embedded devices are made up of interconnected components. For example, embedded devices often expose web-based interfaces comprised of a web server and various back-end applications. In this architecture, any given piece of functionality often relies on the execution of multiple programs [28], e.g., the web server that accepts an HTTP request, a local binary that is summoned by the web server (e.g., using sockets), and an external command that is executed by the local binary to accomplish the request. Each interacting firmware component (the web server, the back-end applications, and other helper programs) can make different assumptions about the data being shared, and inconsistencies can manifest as security vulnerabilities. Precisely detecting these insecure multi-binary interactions among the different components of a firmware sample is challenging. Program analysis approaches that consider each component in isolation, without accounting for the internal flow of data, yield suboptimal results, as they (i) ignore meaningful constraints imposed by components in the course of inter-binary communication, (ii) cannot effectively differentiate between attacker-controlled and non-attacker-controlled sources of input, and (iii) might uncover only superficial bugs. *Thus, an effective firmware analysis must take into account multiple binaries, and reason about the data they share.*

For all these reasons, we need novel solutions able to assess and improve the security of IoT devices.

Contributions. In this dissertation, I present my work to advance the state of the art of firmware security. We approach the problem of securing firmware from different angles (e.g., static versus dynamic analyses) and application layers. In particular, in my work, I studied the security of firmware for IoT devices by considering the different software layers present in these devices: from bootloaders to user applications (i.e., programs that elaborate the data received from users). Each class of software presents different challenges that we discuss in this dissertation. In this dissertation, I will demonstrate that by leveraging the structure of firmware samples and the program practices adopted by firmware developers, it is possible to improve the effectiveness of vulnerability discovery based on static analyses. In summary, I make the following contributions:

- We perform a study of popular bootloaders present on mobile devices, and compare the security properties they implement with those suggested by ARM and Google. To do this, we develop a novel combination of program analysis techniques, including static analysis as well as symbolic execution, to detect vulnerabilities in bootloader implementations that can be triggered from the high-level OS. Then, we implement this technique in a tool, called BOOTSTOMP [1], to evaluate modern, real-world bootloaders, and find *six* previously-unknown critical

[1] `https://github.com/ucsb-seclab/bootstomp`

vulnerabilities (which could lead to persistent compromise of the device) as well as two unlock-bypass vulnerabilities. Finally, we propose mitigations against such attacks, which are trivial to retrofit into existing implementations.

- We design novel combinations of static analysis techniques to perform multi-binary taint analysis. Then, we propose KARONTE [2], a novel static analysis approach to identify insecure interactions between binaries. KARONTE radically reduces the number of false positives, making real-world firmware analysis practical. We implement and evaluate our prototype of KARONTE on 53 real-world firmware samples, showing that our tool can successfully propagate taint information across multiple binaries, resulting in the discovery of 46 unknown (zero-day) bugs, and producing few false positives. Finally, we leverage a bigger dataset of 899 firmware samples to assess the performance of our tool.

- I propose the first sound, test-case agnostic program debloating approach for binaries. We design and formalize a novel signedness-agnostic abstract domain, which outclasses the related work in terms of both soundness and precision, and implement it in two different frameworks: LLVM [3] (for source code analysis) and angr [4] (for binary analysis). We implemented our approach in a prototype, called BINTRIMMER, that using iterative value-flow refinement, recovers a com-

[2] `https://github.com/ucsb-seclab/karonte`
[3] `https://github.com/ucsb-seclab/sasi`
[4] `https://github.com/angr/claripy/blob/master/claripy`

plete and precise CFG from a binary, identifies unreachable code, and removes it. We perform a preliminary evaluation of BINTRIMMER on real-world applications and show that our approach is effective at program debloating. We extensively evaluate our abstract domain, SASI, against domains proposed in related work on both source code and binary files.

- I propose a novel approach to fuzz IoT devices to produce valid yet under-constrained fuzzing inputs that penetrate deeper into the device's firmware code. To do this, we design a novel combination of static and dynamic analyses to find and fuzz specific functions, within the device's companion app, that are located between the app-side validation logic and the data-encoding functions. These functions, which we call *fuzzing triggers*, when invoked generate inputs that are not constrained by app-side validation, and, at the same time, are well-structured, so that they are not immediately discarded by the fuzzed IoT device. We leverage our approach to implement DIANE [5], an automated black-box fuzzer for IoT devices. We evaluate our tool against 11 popular, real-world IoT devices. In our experiments, we show that by identifying *fuzzing triggers* and using them to generate inputs for the analyzed devices, we can effectively discover vulnerabilities. Specifically, we found 11 vulnerabilities in 5 different devices, 9 of which were previously unknown. Finally, we show that, for a majority of IoT devices

[5] `https://github.com/ucsb-seclab/diane`

and companion apps, identifying and leveraging *fuzzing triggers* is essential to generate bug-triggering inputs.

The remaining of this dissertation is structured as follows: In Chapter 2, I present BOOTSTOMP: a novel tool to find bugs in bootloaders for embedded devices. BOOTSTOMP uses a novel combination of static analysis techniques and under-constrained symbolic execution to build a multi-tag taint analysis capable of identifying security issues in bootloaders for embedded systems. We ran BOOTSTOMP against five bootloaders for Android OS, and we were able to identify six previously unknown vulnerabilities, as well as rediscover one that had been previously reported (CVE-2014-9798).

In Chapter 3, I present KARONTE, a novel static analysis approach to track data flows across the different components of a firmware sample to precisely uncover security vulnerabilities. KARONTE is based on the intuition that *binaries communicate using a finite set of common Inter-Process Communication (IPC) paradigms*, and it leverages commonalities in these paradigms to detect where user input is introduced into the firmware sample and to identify interactions between the various components. In our experiments, we show that our approach successfully identifies data flows across different firmware components, which allowed us to discover 46 *zero-day* software bugs, and the rediscovery of another five *n-days* bugs.

In Chapter 4, I present a novel abstract domain, which we call the *Signedness-Agnostic Strided Interval* (or *SASI*) domain, specifically designed to achieve sound

program debloating on binaries (such as firmware samples). We used this novel abstract domain to build a tool, called BINTRIMMER, that soundly identifies and removes unused code within binaries. In our evaluation, we show that our tool soundly removed almost 36% of the code, which contained around the 25% of ROP gadgets contained in the original program.

In Chapter 5, I present a novel approach to fuzz IoT devices by leveraging the companion app (i.e., the mobile application typically used to control an IoT device) to generate well-structured fuzzing inputs. Our approach aims at covering those cases where the firmware governing a certain device is not available. As we see in this Chapter, DIANE analyzed 11 popular IoT devices, and identified 11 bugs, 9 of which are *zero* days.

Chapter 6 contains the related work behind the work presented in this dissertation, and, finally, Chapter 7 contains the conclusions drawn from my work.

Chapter 2

BootStomp: A Bootloader Analyzer

In this Chapter, we investigate the security in both the design and implementation of embedded bootloaders. A bootloader is a particular software of a firmware image that is responsible for initializing the device's memory and peripherals when a device is switched on. To assure the integrity of embedded devices, vendors have implemented a string of inter-dependent mechanisms aimed at removing the possibility of persistent compromise from the device. Known as "Trusted Boot" [41] or "Verified Boot" [46], these mechanisms rely on the idea of a Chain of Trust (CoT) to validate each software component that the system loads as it begins executing code. This procedure (called booting process) is performed through one or more stages by the aforementioned dedicated software components called bootloaders. Bootloaders verify cryptographically that each stage, from a Hardware Root of Trust through the device's file system, is both unmodified and authorized by the hardware manufacturer. Any unverified modification of the various bootloader components, system kernel, or file system image should re-

sult in the device being rendered unusable. Ideally, this is an uncircumventable, rigid process, removing any possibility of compromise, even when attackers can achieve arbitrary code execution on the high-level operating system (e.g., Android). However, hardware vendors are given a great amount of discretion when implementing these bootloaders, leading to variations in both the security properties they enforce and the size of the attack surface available to an adversary. Unfortunately, analyzing the code of bootloaders to locate vulnerabilities represents a worst-case scenario for security analysts. Bootloaders are typically closed-source [95], proprietary programs, and tend to lack typical metadata (such as program headers or debugging symbols) found in normal programs. By their very nature, bootloaders are tightly coupled with hardware, making dynamic analysis outside of the often uncooperative target platform impractical. Manual reverse-engineering is also tedious, as bootloaders typically do not use system calls or well-known libraries, leaving few semantic hints for an analyst to follow.

We examine bootloaders from four popular manufacturers, and discuss the standards and design principles that they strive to achieve. Then, I present BOOTSTOMP: a multi-tag taint analysis resulting from a novel combination of static analyses and dynamic symbolic execution, designed to locate problematic areas where input from an attacker in control of the OS can compromise the bootloader's execution, or its security features. Using our tool, we found six previously-unknown vulnerabilities (of which five have been confirmed by the respective vendors), as well as rediscover one that had

been previously reported. Some of these vulnerabilities would allow an attacker to execute arbitrary code as part of the bootloader, or to perform permanent denial-of-service attacks. Our tool also identified two vulnerabilities that can be leveraged by an attacker with root privileges on the OS to unlock the device, thus allowing the attacker to load a custom OS without the user's consent. We conclude by proposing simple mitigation steps that can be implemented by manufacturers to safeguard the bootloader and OS from all of the discovered attacks, using already-deployed hardware features.

2.1 Bootloaders in Theory

Today's mobile devices incorporate a number of security features aimed at safeguarding the confidentiality, integrity, and availability of users' devices and data. In this Section, we discuss Trusted Execution Environments, which allow for isolated execution of privileged code, and Trusted Boot, aimed at ensuring the integrity and provenance of code, both inside and outside of TEEs.

2.1.1 TEEs and TrustZone

A Trusted Execution Environment (TEE) is the notion of separating the execution of security-critical ("trusted") code from that of the traditional operating system ("untrusted") code. Ideally, this isolation is enforced using hardware, such that even in

the event the un-trusted OS is completely compromised, the data and code in the TEE remain unaffected.

Modern ARM processors, found in almost all mobile phones sold today, implement TrustZone [5], which provides a TEE with hardware isolation enforced by the architecture. When booted, the primary CPU creates two "worlds"–known as the "secure" world and "non-secure" world, loads the un-trusted OS (such as Android) into the non-secure world, and a vendor-specific trusted OS into the secure world. The trusted OS provides various cryptographic services, guards access to privileged hardware, and, in recent implementations, can be used to verify the integrity of the un-trusted OS while it is running. The un-trusted kernel accesses these commands by issuing the Secure Monitor Call (SMC) instruction, which both triggers the world-switch operation, and submits a command the Trusted OS and its services should execute.

ARM Exception Levels (EL). In addition to being in either the secure or non-secure world, ARM processors support "Exception Levels," which define the amount of privilege to various registers and hardware features the executing code has. The 64-bit ARM architecture defines four such levels, EL0-EL3. EL0 and EL1 map directly to the traditional notion of "user-mode" and "kernel mode," and are used for running unprivileged user applications and standard OS kernels respectively. EL2 is used for implementing hypervisors and virtualization, and EL3 implements the Secure Monitor, the most privileged code used to facilitate the world-switch between secure and non-secure. During

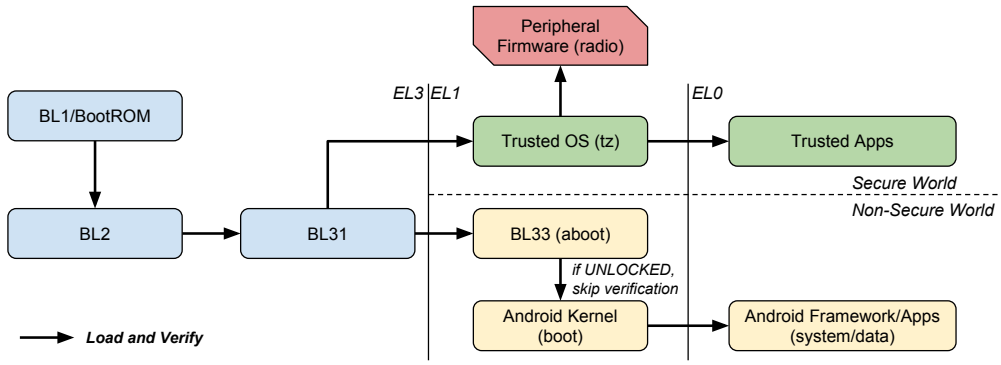


Figure 2.1: Overview of the Trusted/Verified Boot implementation according to the ARM and Google specifications. Between parentheses the name of the internal storage partition where the code is located in a typical implementation.

the boot process described below, the initial stages, until the non-secure world bootloader is created, runs at EL3.

2.1.2 The Trusted Boot Process

In a traditional PC environment, the bootloader's job is to facilitate the location and loading of code, across various media and in various formats, by any means necessary. However, in modern devices, particularly mobile devices, this focus has shifted from merely loading code to a primary role in the security and integrity of the device. To help limit the impact of malicious code, its job is to verify both the integrity and provenance of the software that it directly executes.

As with the traditional PC boot process, where a BIOS loaded from a ROM chip would load a secondary bootloader from the hard disk, mobile bootloaders also contain

a chain of such loaders. Each one must, in turn, verify the integrity of the next one, creating a Chain of Trust (CoT).

On ARM-based systems, this secured boot process is known as *Trusted Boot* and is detailed in the ARM Trusted Board Boot Requirements (TBBR) specification. While this document is only available to ARM's hardware partners, an open-source reference implementation that conforms to the standard is available [41].

While this standard, and even the reference implementation, does leave significant room for platform-specific operations, such as initialization of hardware peripherals, implementations tend to follow the same basic structure. One important aspect is the Root of Trust (RoT), which constitutes the assumptions about secure code and data that the device makes. In ARM, this is defined to be 1) the presence of a "burned-in," tamper-proof public-key from the hardware manufacturer that is used to verify subsequent stages, and 2) the very first bootloader stage being located in read-only storage.

While manufacturers are free to customize the Trusted Boot process when creating their implementations, ARM's reference implementation serves as an example of how the process should proceed. The boot process for the ARM Trusted Firmware occurs in the following steps, as illustrated in Figure 2.1.

1. The CPU powers on, and loads the first stage bootloader from read-only storage.

2. This first stage, known as BL1, Primary Boot Loader (PBL), or BootROM, performs any necessary initialization to locate the next stage from its storage, loads it into memory, verifies its integrity using the Root of Trust Public Key (ROTPK), and if this is successful, executes it. Since it is on space-restricted read-only media, its functionality is extremely limited.

3. BL2, also known as the Secondary Boot Loader (SBL) is responsible for creating the secure and non-secure worlds and defining the memory permissions that enforce this isolation. It then locates and loads into memory up to three third-stage bootloaders, depending on the manufacturer's configuration. These run at each of the EL3, EL2, and EL1 levels, and are responsible for setting up the Secure Monitor, a hypervisor (if present), and the final-stage OS bootloader.

4. BL2 then executes BL31, the loader running at EL3, which is responsible for configuring various hardware services for the trusted and un-trusted OSes, and establishing the mechanism used to send commands between the two worlds. It then executes the BL32 loader, if present, which will eventually execute BL33.

5. BL33 is responsible for locating and verifying the non-secure OS kernel. Exactly how this is done is OS-dependent. This loader runs with the same privilege as the OS itself, at EL1.

Next, we detail extensions to this process developed for the Android ecosystem.

2.1.3 Verified Boot on Android

ARM's Trusted Boot standard only specifies stages of the boot process up to the point at which the OS-specific boot loader is executed. For devices running Android, Google provides a set of guidelines for *Verified Boot* [46], which describes high-level functionality an Android bootloader should perform.

Unlike the previous stages, the Android bootloader provides more functionality than just ensuring integrity and loading code. It also allows for the user or OS to elect to boot into a special recovery partition, which deploys firmware updates and performs factory reset operations. Additionally, modern Android bootloaders also participate in enabling full-disk encryption and triggering the initialization of Android-specific TrustZone services.

Ideally, the verification of the final Android kernel to be booted would effectively extend the Chain of Trust all the way from the initial hardware-backed key to the kernel. However, users wishing to use their devices for development need to routinely run kernels not signed by the device manufacturer. Therefore, Google specifies two classes of bootloader implementations: Class A, which only run signed code, and Class B, which allow for the user to selectively break the Chain of Trust and run unsigned code, in a tamper-evident manner, referred to as *unlocking*. Devices will maintain a *security state* (either LOCKED or UNLOCKED) and properties of the transition between the

two states must be enforced. With regard to Class B implementations, Google requires that:

- The bootloader itself must be verified with a hardware-backed key.
- If verification of the Android kernel with the OEM key (a key hard-coded by the device's manufacturer in the bootloader code) fails for any reason, a warning will be displayed to the user for at least five seconds. Then, if the bootloader is in the `LOCKED` state, the device will not boot, otherwise, if the bootloader is in the `UNLOCKED` state the Android kernel will be loaded.
- The device will only transition from the `LOCKED` state to the `UNLOCKED` state if the user first selects the "allow OEM Unlock" option from the Developer Options menu in Android's settings application, and then issues the Fastboot command `oem unlock`, or an equivalent action for devices without Fastboot.
- When the device's lock state changes for any reason, user-specific data will be rendered unreadable.

Beyond the guidelines, Android bootloaders (typically those that fall into Class B) also provide some means of rewriting partitions on internal storage over USB. Google suggests the use of the Fastboot protocol, also utilized for the locking and unlocking process, for this functionality.

2.2 Bootloaders in Practice

While the standards and guidelines on bootloader design in Section 2.1 do cover many important security-related aspects, a significant amount of flexibility is given to OEMs to allow for functionality specific to their platforms. These involve both aspects of the hardware itself, but also logical issues with managing the security state of the device. Even though this flexibility makes it hard to reason about the actual security properties of bootloaders, it is difficult to envision a future for which these standards would be more precise. In fact, there are a number of technical reasons due to which the definition of these standards cannot be as comprehensive as we would hope.

One of these technical aspects is related to peripherals and additional custom hardware that is shipped with each device. While platform-specific code can be inserted at every stage in ARM's prototypical Trusted Boot implementation, no direction is given as to what code should be inserted at which points in the boot process. Additionally, initialization tasks cannot be too tightly coupled with the rest of the boot sequence, as peripheral hardware, such as modems, may incorporate code from different vendors and necessitate a modification of the initialization process. Furthermore, vendors of the final devices may not be able to alter earlier stages of the boot process to add necessary initialization code, as they may be locked to code supplied by the chip manufacturer. Finally, even aside from these issues, there are constraints on storage media. ROMs, such

as those mandated for the first bootloader stage, tend to be small, and are inherently a write-once medium, precluding their use for any code that may need to be updated.

As an example, consider a mobile device with an on-board GSM or LTE modem. Depending on the hardware used, this modem could exist either as part of the System-on-a-chip (SoC) package or externally on another chip. Because the initialization of these two layouts has different requirements (e.g., initializing memory busses and transferring code to an external modem vs. executing modem code on the same chip), this may need to happen at different phases in the boot process, where different levels of hardware access are available.

This also applies to various bootloader services, such as partition management and unlocking. Google's implementation provides the Fastboot protocol in the final-stage bootloader, but manufacturers are free to use alternative methods, as well as incorporate this functionality into other boot stages.

Where and how all of these features are implemented can have a significant security impact. If a stage in the bootloader is compromised, this could lead to the compromise of all following stages, along with any peripherals or secured storage that they manage. The impact of gaining control over a bootloader can be mitigated by using the lowest-possible Exception Level (discussed in the previous section), and performing tasks that involve taking potentially-untrusted input in later, less-privileged stages of the process. However, once again, other than the Trusted Firmware reference implementation, no

guidance is given on how to manage exception levels with respect to bootloader features.

One aspect that increases the attack surface of modern bootloaders is that the code used to bootstrap additional hardware, such as modems, needs to be *updateable*, and thus needs to be stored on writable partitions. These writeable partitions, in turn, could be modified by an attacker with privileged code execution. Thus, it is critical that the content of these partitions is verified, such as by checking the validity of a cryptographic signature. This should ideally be accomplished by a previous bootloader stage, which thus needs to load, parse, and verify these partitions. This usage of data from writeable (and, as discussed previously, potentially attacker-controlled) partitions is what makes common memory corruption vulnerabilities in bootloaders very dangerous.

2.2.1 Bootloader Implementations

In the remainder of this Section, we explore four bootloaders from popular device manufacturers. These implementations all serve the same functions for their respective hardware platforms and aim to comply with both ARM and Google's standards, but do so in vastly different ways.

A comparison of the implementations can be found in Table 2.1. If an attacker can compromise the final stage bootloader, they will likely be able to also affect any

Table 2.1: Bootloader features, and the Exception Level they occur in.

Vendor	EL	Fastboot	Modem Initialization	Peripherals Initialization
Qualcomm	EL1	✓	✗	✗
HiSilicon	EL3	✓	✓	✓
NVIDIA	EL1	✓	✗	✗
MediaTek	EL1	✓	✓	✗

functionality it contains, as well as any that it in turn loads, which in these cases, is the Android kernel and OS.

Qualcomm. The Qualcomm MSM chipset family is by far the most popular mobile chipset in devices today, representing over 60% of mobile devices [76]. While many manufacturers of MSM-based devices will customize the bootloader to fit their specific product's features, Qualcomm's "`aboot`" bootloader is still used with little modifications on many of them.

`aboot` is based on the Little Kernel (LK) open-source project, and provides the final stage non-secure OS loading functionality (equivalent to BL33 in ARM's reference implementation). In further similarity to BL33, it runs at EL1, giving it the same level of privilege as the kernel it aims to load. It conforms very closely to Google's Verified Boot guidelines, implementing the traditional set of Android-specific features, including Fastboot, recovery partition support, and unlocking. `aboot` can be used in

either a Class A or Class B Verified Boot implementation, as Fastboot, and therefore unlocking can be disabled by the OEM or mobile carrier.

HiSilicon and Huawei. HiSilicon Kirin-based devices, such as those from Huawei, implement a very different bootloader architecture to the others we examined. Instead of merely being responsible for the initialization required to load Android, this loader also combines functionality usually found elsewhere in the boot process, such as initializing the radio hardware, secure OS, secure monitor, among others, giving it the equivalent roles of BL31, BL33, and BL2 in the ARM reference implementation. In fact, this bootloader is loaded directly by the ROM-based first-stage bootloader (BL1). To have the privilege necessary to perform all these tasks, HiSi's bootloader runs at EL3, and executes the Linux kernel in the boot partition at EL1 when it is finished. Along with its hardware initialization tasks, it also includes Fastboot support, which allows for unlocking.

MediaTek. Devices based on MediaTek chipsets, such as the Sony Xperia XA and other similar handsets, implement a bootloader similar to Qualcomm's but using a very different codebase. The Android-specific loader runs at EL1, and is also responsible for partition management and unlocking via Fastboot. Unlike Qualcomm's, this loader is also responsible for bootstrapping the modem's baseband firmware, meaning that any compromise in the bootloader could impact this critical component as well.

NVIDIA. NVIDIA's Tegra-based devices ship with a bootloader known as `hboot`. This bootloader is very similar to Qualcomm's, in that it runs at EL1, and implements only the fastboot functionality at this stage.

2.3 Unlocking Bootloaders

While security-focused bootloaders do significantly raise the bar for attackers wishing to persistently compromise the device, there are many cases in which "unlocking," as detailed in Section 2.1, has legitimate benefits. Only permitting the execution of signed code makes the development of the Android OS itself problematic, as well as disallowing power-users from customizing and modifying the OS's code.

Of course, this is a very security-sensitive functionality; an attacker could unlock the bootloader and then modify the relevant partitions as a way of implementing a persistent rootkit. Google's Verified Boot standard covers the design of this important mechanism, discusses many high-level aspects of managing the device's security state (see Section 2.1), and even provides specifics about digital signatures to be used. However, as with the ARM specifications covering Trusted Boot, these specs must also allow for platform-specific variations in implementation, such as where or how these security mechanisms are integrated into the boot process.

Furthermore, there are many unspecified, implicit properties of Verified Boot that a valid implementation should enforce, to ensure that the device is protected from privileged code execution or unauthorized physical control. These properties include:

The device state should only transition from locked to unlocked with explicit user content. This is implicitly handled by requiring a command sent to Fastboot to unlock, as this usually requires physical access to activate, and causes a warning to be displayed to the user. Similarly, a malicious app — no matter how privileged it is — should not be able to silently unlock the bootloader.

Only the authorized owner of the device should be able to unlock the bootloader. This means that anyone in possession of a phone that is not theirs cannot simply access Fastboot or similar protocol (i.e., by rebooting the phone) and trigger an unlock. This is avoided on some devices through checking an additional flag called "OEM unlock," (or, more informally "allow unlock"). This flag is controlled by an option in the Android Settings menu, and it is only accessible if the device is booted and the user has authenticated (for instance, by inserting the correct "unlock pattern"). A proper implementation of Fastboot will honor the "OEM unlock" flag and it will refuse to unlock the bootloader if this flag is set to false.

Interestingly, there is no requirement for the storage of the device's security state. While the standard offers a suggestion about how to tie this state and its transitions to the security properties they wish to enforce, the exact storage of this information

is left out, likely to account for hardware variations with respect to secured storage. Unfortunately, as we discuss in Section 2.4, specifics of such implementation details can negatively impact the security properties of the bootloader.

2.3.1 Unlocking vs Anti-Theft

Another interesting factor related to bootloaders and bootloader locking is the overall usability of a device by an attacker after it has been stolen. As mandated by laws [123] and industry standards [49], phones should implement mechanisms to prevent their usage when stolen. Google refers to this protection as Factory Reset Protection (FRP) [45], and it has been enabled in Android since version 5.0. In Google's own implementations, this means that the Android OS can restrict the usage of a phone, even after a factory-reset, unless the legitimate user authenticates.

This presents an interesting contradiction in relation to bootloader unlocking capabilities. First, since this mechanism is governed from within the OS, it could be leveraged by a malicious process with sufficient privilege. Of course, the original owner should be able to authenticate and restore the device's functionality, but this could still be used as a form of denial-of-service. Second, some manufacturers offer low-level firmware upload functionality, such as in the BL1 or BL2 stages, designed to restore the device to a working state in the event it is corrupted. This feature is in direct opposition to anti-theft functionality, as if a user can recover from any kind of corruption, this

mechanism may be able to be bypassed. However, if this mechanism respects the anti-theft feature's restrictions on recovering partitions, this also means the device can be rendered useless by a sufficiently-privileged malicious process. In other words, there is an interesting tension between anti-theft and anti-bricking mechanisms: if the anti-theft is implemented correctly, an attacker could use this feature against the user to irremediably brick her device; vice versa, if an anti-bricking mechanism is available, a thief could use this mechanism to restore the device to a clean, usable state. In Section 2.7, we explore how this tension can be resolved.

2.4 Attacking Bootloaders

Regardless of implementation specifics, bootloaders have many common functions that can be leveraged by an attacker. While they may appear to be very isolated from possible exploitation, bootloaders still operate on input that can be injected by a sufficiently-privileged attacker. For example, the core task a bootloader must perform (that of booting the system) requires the bootloader to load data from non-volatile storage, figure out which system image on which partition to boot, and boot it. To enforce the Chain of Trust, this also involves parsing certificates and verifying the hash of the OS kernel, all of which involve further reading from the device's storage. In Class B implementations, the device's security state must also be consulted to determine how

much verification to perform, which could be potentially stored in any number of ways, including on the device's storage as well. While bootloader authors may assume that this input is *trusted*, it can, in fact, be controlled by an attacker with sufficient access to the device in question.

In this work, we assume an attacker can control any content of the non-volatile storage of the device. This can occur in the cases that an attacker attains root privileges on the primary OS (assumed to be Android for our implementation). While hardware-enforced write protection mechanisms could limit the attacker's ability to do this, these mechanisms are not known to be in wide use today, and cannot be used on any partition the OS itself needs to routinely write to.

Given this attacker model, our goal is to automatically identify weaknesses, in deployed, real-world bootloader firmware, that can be leveraged by an attacker conforming to our attacker model to achieve a number of goals:

Code execution. Bootloaders process input, read from attacker-controlled non-volatile storage, to find, validate, and execute the next step in the boot process. What if the meta-data involved in this process is *maliciously crafted*, and the code processing it is not securely implemented? If an attacker is able to craft specified meta-data to trigger memory corruption in the bootloader code, they may achieve code execution *during the boot process*. Depending on when in the boot process this happens, it might grant the attacker control at exception levels considerably higher than what they may achieve

with a root or even a kernel exploit on the device. In fact, if this is done early enough in the boot process, the attacker could gain control over Trusted Execution Environment initialization, granting them a myriad of security-critical capabilities that are unavailable otherwise.

Bricking. One aspect that is related to secure bootloaders is the possibility of "bricking" a device, i.e., the corruption of the device so that the user has no way to regain control of it. Bootloaders attempt to establish whether a piece of code is trusted or not: if such code is trusted, then the bootloader can proceed with their loading and execution. But what happens when the trust cannot be established? In the general case, the bootloader stops and issues a warning to the user. The user can, usually through the bootloader's recovery functionality (e.g., Fastboot) restore the device to a working state. However, if an attacker can write to the partition holding this recovery mechanism, the user has no chance to restore the device to an initial, *clean* state, and it may be rendered useless.

This aspect becomes quite important when considering that malware analysis systems are moving from using emulators to using real, physical devices. In this context, a malware sample has the capability of bricking a device, making it impossible to reuse it. This possibility constitutes a limitation for approaches that propose baremetal malware analysis, such as BareDroid [88].

One could think of having a mechanism that would offer the user the possibility of restoring a device to a clean state no matter how compromised the partitions are. However, if such a mechanism were available, any anti-theft mechanism (as discussed in Section 2.3), could be easily circumvented.

Unsafe unlock. As discussed in Section 2.3, the trusted boot standard does not mandate the implementation details of storing the secure state. Devices could use an eMMC flash device with RPMB, an eFuse, or a special partition on the flash, depending on what is available. If the security state is stored on the device's flash, and a sufficiently-privileged process within Android can write to this region, the attacker might be able to unlock the bootloader, bypassing the requirement to notify the user. Moreover, depending on the implementation, the bootloader could thus be unlocked without the user's data being wiped.

In Section 2.5, we propose a design for an automated analysis approach to detect vulnerabilities in bootloader implementations. Unfortunately, our experiments in Section 2.6 show that currently deployed bootloaders are vulnerable to combinations of these issues. But hope is not lost – in Section 2.7, we discuss a mechanism that addresses this problematic aspect.

2.5 BOOTSTOMP

The goal of BOOTSTOMP is to automatically identify security vulnerabilities that are related to the (mis)use of attacker-controlled non-volatile memory, trusted by the bootloader's code. In particular, we envision using our system as an automatic system that, given a bootloader as input, outputs a number of alerts that could signal the presence of security vulnerabilities. Then, human analysts can analyze these alerts and quickly determine whether the highlighted functionality indeed constitutes a security threat.

Bootloaders are quite different from regular programs, both regarding goals and execution environment, and they are particularly challenging to analyze with existing tools. In particular, these challenges include:

- Dynamic analysis is infeasible. Because a primary responsibility of bootloaders is to initialize the hardware, any concrete execution of bootloaders would require this hardware.

- Bootloaders often lack available source code, or even debugging symbols. Thus, essential tasks, including finding the entry point of the program, become much more difficult.

- Because bootloaders run before the OS, the use of syscalls and standard libraries that depend on this OS is avoided, resulting in all common functionality, includ-

ing even functions such as `memcpy`, being reimplemented from scratch, thus making standard signature-based function identification schemes ineffective.

To take the first step at overcoming these issues, we developed a tool, called BOOTSTOMP, combining different static analyses as well as a dynamic symbolic execution (DSE) engine, to implement a taint analysis engine. To the best of our knowledge, we are the first to propose a traceable offline (i.e., without requiring to run on real hardware) taint analysis completely based on dynamic symbolic execution. Other works as [105] [130] propose completely offline taint analyses on binaries. In contrast to our work, they implement static taint analyses and are hence not based on dynamic symbolic execution.

The main problem with these types of approaches is that, though sound, they might present a high rate of false positives, which a human analyst has to filter out by manually checking them. Note that, in the context of taint analysis, a false positive result is a path that is mistakenly considered tainted. Furthermore, producing a trace (i.e., a list of basic blocks) representing a tainted path using a static taint analysis approach is not as simple as with symbolic execution.

On the other hand, our approach based on DSE, though not sound (i.e., some tainted paths might not be detected as explained in Section 2.6.4), presents the perk of returning a traceable output with a low false positives rate, meaning that the paths we detected as tainted are indeed tainted, as long as the initial taint is applied and propagated correctly.

Note that there is a substantial difference between false positives when talking about taint analyses and when talking about vulnerability detection. Though our tool might return some false positives in terms of detected vulnerabilities, as seen in Section 2.6, false positives in tainted path detection are rare (we never found any in our experiments) as our tool is based on DSE. For a deeper discussion about the results obtained by BOOTSTOMP, please refer to Section 2.6.4.

With these considerations in mind, since the output of our analysis is supposed to be triaged by a human, we opted for a taint analysis based on DSE.

This Section discusses the goal, the design features, and the implementation details of BOOTSTOMP.

2.5.1 Design

Our system aims to find two specific types of vulnerabilities: uses of attacker-controlled storage that result in a memory-corruption vulnerability, and uses of attacker-controlled storage that result in the unlocking of the bootloader. While these two kinds of bugs are conceptually different, we are able to find both using the same underlying analysis technique.

The core of our system is a taint analysis engine, which tracks the flow of data within a program. It searches for paths within the program in which a *seed of taint* (such as the attacker-controlled storage) is able to influence a *sink of taint* (such as a sensitive

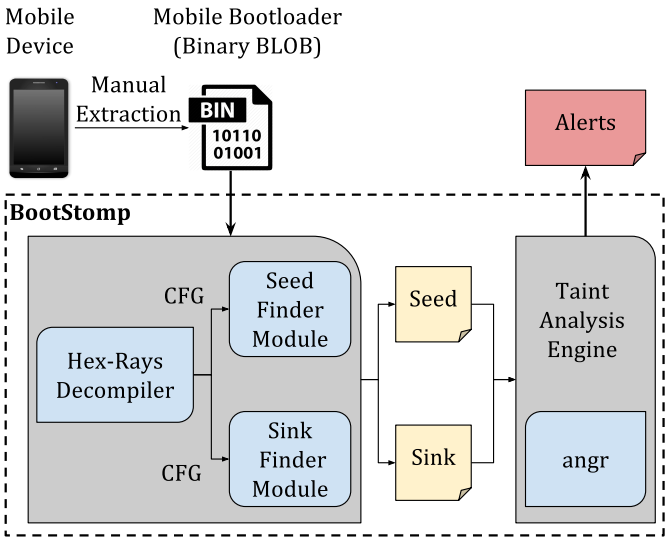


Figure 2.2: BOOTSTOMP overview.

memory operation). The tool raises an alert for each of these potentially vulnerable paths. The human analyst can then process these alerts and determine whether these data flows can be exploitable.

Our system proceeds in the following steps, as shown in Figure 2.2:

Seed Identification. The first phase of our system involves collecting the seeds of taint. We developed an automated analysis step to find all the functions within the program that read data from any non-volatile storage, which are used as the seeds when locating memory corruption vulnerabilities. However, if the seeds have semantics that cannot be automatically identified, such as the unlocking mechanism of the bootloader, BOOTSTOMP allows for the manual specification of seeds by the analyst. This feature

comes particularly in handy when source code is available, as the analyst can rely on it to manually provide seeds of taint.

Sink Identification. We then perform an automated analysis to locate the sinks of taint, which represent code patterns that an attacker can take advantage of, such as bulk memory operations. Moreover, writes to the device's storage are also considered sinks for locating potentially attacker-controlled unlocking mechanisms.

Taint Analysis. Once the seeds of taint have been collected, we consider those functions containing the seed of taint and, starting from their entry point, perform a multi-tag taint analysis based on under-constrained symbolic execution [102] to find paths where seeds reach sinks. This creates alerts, for an analyst to review, including detailed context information, which may be helpful in determining the presence and the exploitability of the vulnerability.

In the remainder of this Section, we explore the details about each of these steps.

2.5.2 Seed Identification

For finding memory corruption vulnerabilities, our system supports the automatic identification of seeds of taint. We use approaches similar to those in prior work (e.g., [115]). We rely on error logging because there are many different mechanisms that may read from non-volatile memory, or different types of memory (plain flash memory vs. eMMC), and these error log strings give us semantic clues to help finding

```
1  #define SEC_X_LEN 255
2
3  void get_conf_x() {
4      n = read_emmc("sec_x", a2, a3);
5      if (n < SEC_X_LEN) {
6          return;
7      }
8      //...
9  }
10
11 int get_user_data() {
12     if(!read_emmc(b1, b2, 0)) {
13         debug("EMMC_ERROR: no data read");
14         return -1;
15     }
16     // ...
17 }
```

Figure 2.3: BOOTSTOMP first determines the function reading from `emmc` by scanning error messages, then it infers the function arguments types.

them. Our system looks for error logging functions using keywords as *mmc*, *oeminfo*, *read*, and *fail*, and avoiding keywords like *memory* and *write*.

This approach is useful for identifying functions that somehow retrieve the content from a device's storage. However, since the signature of these functions is not known, it is challenging to identify which argument of this function stores the receiving buffer. To determine the argument to be tainted, we use an approach based on type inference.

Ideally, the taint should only be applied to the seed's argument pointing to the memory location where the read data will be stored. As distinguishing pointers from integers is an undecidable problem [128], our analysis might dereference an integer in the process of applying the taint, resulting in a possible huge rate of false positive alarms. Nonetheless, during this study, we observed that, surprisingly, strings might not always be passed by reference to a function, but rather by value. During our analysis, we

check every call site of the functions we retrieved using the above-mentioned method and check the entity of every passed argument. If an argument is composed of only ASCII printable characters, we assume it is a string, and we consider the same argument to be a string for every other call to the same function. When looking for the memory locations to apply the taint, we consider this information to filter out these arguments. We also do not taint arguments whose passed values are zeroes, as they might represent the NULL value.

As an example, consider Figure 2.3. First, BOOTSTOMP retrieves the function $read\_emmc$ as a possible seed function, by analyzing the error log at line 13. Then, it scans every call site of $read\_emmc$ and infers that the returned value is an integer (as it is compared against an integer variable), the first parameter is a string and the third parameter can assume the value zero. As $read\_emmc$ is a candidate seed function, it has to store the content read from non-volatile storage in a valid buffer, pointed by a non-null pointer. Therefore, BOOTSTOMP applies the taint only to the second parameter of $read\_emmc$ ($a2$ and $b2$). Note that, as the receiving buffer could be returned by a seed function, if the type of the returned value cannot be inferred, the variable it is assigned to is tainted as well. Note that, when a tainted pointer is dereferenced, we taint the entire memory page it points to.

In the case of locating unlocking-related vulnerabilities, there is no bootloader-independent way of locating the unlocking function, since the implementation details

significantly vary. Therefore, BOOTSTOMP also supports supplying the seeds manually: an analyst can thus perform reverse-engineering to locate which function implements the "unlock" functionality and manually indicate these to our analysis system. While this is not a straightforward process, there is a specific pattern a human analyst can rely on: Fastboot's main command handler often includes a basic command line parser that determines which functionality to execute, and the strings involved are often already enough to quickly pinpoint which function actually implements the "unlock" functionality.

2.5.3 Sink Identification

Our automatic sink identification strategy is designed to locate four different types of sinks:

memcpy-like functions. BOOTSTOMP locates memcpy-like functions (e.g., memcpy, strcpy) by looking for semantics that involve moving memory, unchanged, from a source to a destination. As mentioned above, there are no debugging symbols, and standard function signature-based approaches would not be effective. For this reason, we rely on a heuristic that considers the basic blocks contained within each function to locate the desired behavior. In particular, a function is considered memcpy-like if it contains a basic block that meets the following conditions: 1) Loads data from memory; 2) stores this same data into memory; 3) increments a value by one unit (one word, one

byte, etc). Moreover, since it is common for bootloaders to rely on wrapper functions, we also flag functions that directly invoke one (and only one) function that contains a block satisfying the above conditions.

We note that there may be several other functions that, although satisfy these conditions as well, do not implement a memcpy-like behavior. Thus, we rely on an additional observation that memcpy and strcpy are among the most-referenced functions in a bootloader, since much of their functionality involves the manipulation of chunks of memory. We therefore sort the list of all functions in the program by their reference count, and consider the first 50 as possible candidates. We note that, empirically, we found that memcpy functions often fall within the top five most-referenced functions.

Attacker-controlled dereferences. BOOTSTOMP considers memory dereferences controlled by the attacker as sinks. In fact, if attacker-controlled data reaches a dereference, this is highly indicative of an attacker-controlled arbitrary memory operation.

Attacker-controlled loops. We consider as a sink any expression used in the guard of a loop. Naturally, any attacker able to control the number of iterations of a loop, could be able to mount a denial-of-service attack.

Writes to the device's storage. When considering unlocking vulnerabilities, we only use as sinks any write operation to the device's storage. This encodes the notion that an unlocking mechanism that stores its security state on the device's storage may be controllable by an attacker. To identify such sinks, we adopt the same keyword-based

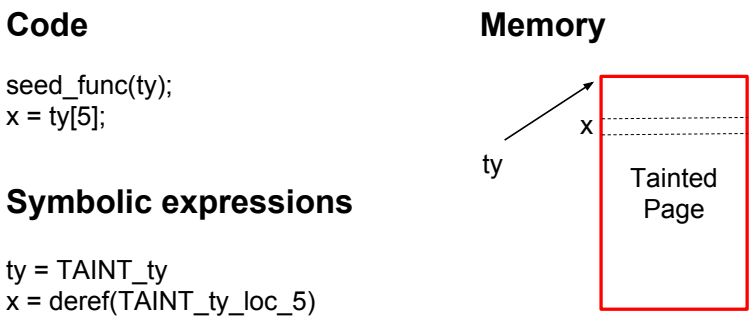


Figure 2.4: Taint propagation example.

approach that we employed to identify the seeds of taint (i.e., by using relevant keywords in error logging messages).

2.5.4 Taint Tracking

While we cannot execute the bootloaders concretely, as we discussed above, we can execute them symbolically. Our interest is in the path the data takes in moving from a seed to a sink, and path-based symbolic execution lets us reason about this, while implicitly handling taint-propagation. Given a bootloader, along with the seeds and sinks identified in the previous stages, the analysis proceeds as follows:

- Locate a set of *entry points*, defined as any function that directly calls one of the identified seeds.

- Begin symbolic execution at the beginning of each entry point. Note that, before starting to symbolically execute an entry point, BOOTSTOMP tries to infer, looking for known header as ELF, where the global data is located. If it does find it,

it unconstrains each and every byte in it, so to break any assumptions about the memory content before starting to analyze the entry point.

- When a path encounters a function, either step over it, or step into it, considering the code traversal rules below.

- When a path reaches a seed, the appropriate taint is applied, per the taint policy described below.

- Taint is propagated implicitly, due to the nature of symbolic execution. This includes the return values of functions handling tainted data.

- If a path reaches a sink affected by tainted data, an alert is raised.

Code traversal. To avoid state explosion, we constrain the functions that a path will traverse, using an *adaptive inter-function level*. Normally, the inter-function level specifies how many functions deep a path would traverse. However, the handling of tainted data in our analysis means that we implicitly care more about those functions which consume tainted data. Therefore, we only step into functions that consume tainted data, up to the inter-function level. For our experiments, we fixed the inter-function level at 1. More in detail, our analysis traverses the code according to the following rules:

- When no data is tainted, functions are not followed, such as at the beginning of an entry point, before the seed has been reached. Particularly, this path selection criteria allows us to have a fast yet accurate taint analysis, at the expense of possible false negative results, as some tainted paths might not be discovered due to some missed data aliases.

- Functions are not followed if their arguments are not tainted.

- Analysis terminates when all the possible paths between the entry point and its end are analyzed, or a timeout is triggered. Note that we set a timeout of ten minutes for each entry point. As we show in Section 2.6.2 our results indicate that this is a very reasonable time limit.

- Unless any of the above conditions are met, we follow functions with an inter-function level of 1. In other words, the analysis will explore at least one function away from the entry point.

- We explore the body of a loop (unroll the loop) exactly once, and then assume the path exits the loop.

(Under-Constrained) Symbolic Execution. Our approach requires, by design, to start the analysis from arbitrary functions, and not necessarily from the bootloader's entrypoint, which we may not even be able to determine. This implies that the initial state may contain fewer constraints than it should have at that particular code

point. For this reason, we use under-constrained symbolic execution, first proposed by Ramos et al. [102], which has been proven to reach good precision in this context.

Multi-tag taint analysis. To reach a greater precision, our system implements a multi-tag tainting approach [84]. This means that, instead of having one concept of taint, each taint seed generates tainted data that can be uniquely traced to where it was generated from. Furthermore, we create unique taint tags for each invocation of a seed in the program. This means, for example, that if a taint seed is repeatedly called, it will produce many different taint tags. This improves precision when reasoning about taint flow.

Taint propagation and taint removal. Taint is implicitly propagated using symbolic execution, as no constraint is ever dropped. This means that if a variable x depends on a tainted variable ty, the latter will appear in the symbolic expression of the former. As an example consider Figure 2.4. Suppose that a location of an array pointed by ty is dereferenced and assigned to x, such as $x = ty[5]$. Assuming now that ty is tainted because pointing to data read from untrusted storage, the memory page it points to will be tainted, meaning that every memory location within that page will contain a symbolic variable in the form $TAINT\_ty\_loc\_i$. After the instruction $x = ty[5]$, the symbolic variable x will be in the form $deref(TAINT\_ty\_loc\_5)$.

On the other hand, taint is removed in two cases. Implicitly when a non-tainted variable or value is written in a tainted memory location, or when a tainted variable

is constrained within non tainted values. As an example and by referring to the above tainted variable x, if a check such as $if(x < N)$, where N is non-tainted value, is present, x would get untainted.

Concretization strategy. When dealing with memory writes in symbolic locations, the target address needs to be concretized. Unlike existing work [19], our analysis opts to concretize values with a bias toward smaller values in the possible range (instead of being biased toward higher values). This means that, when a symbolic variable could be concretized to more than one value, lower values are preferred. In previous work, higher values were chosen to help find cases where memory accesses off the end of an allocated memory region would result in vulnerabilities. However, these values may not satisfy conditional statements in the program that expect the value to be "reasonable," (such as in the case of values used to index items in a vector) and concretizing to lower values allows paths to proceed deeper into the program. In other words, we opt for this strategy to maximize the number of paths explored. Also, when BOOTSTOMP has to concretize some expressions, it tries to concretize different unconstrained variables to different (low) values. This strategy aims to keep the false positive rate as low as possible. For a deeper discussion about how false negatives and positive might arise, please refer to Section 2.6.4.

Finally, our analysis heavily relies on angr [116] (taint engine) and IDA Pro [55] (sink and seed finding).

2.6 Evaluation

This Section discusses the evaluation of BOOTSTOMP on bootloaders from commercial mobile devices. In particular, for each of them, we run the analysis tool to locate the two classes of vulnerabilities discussed in Section 2.5. As a first experiment, we use the tool to automatically discover potential paths from attacker-controllable data (i.e., the flash memory) to points in the code that could cause memory corruption vulnerabilities. As a second experiment, we use the tool to discover potential vulnerabilities in how the lock/unlock mechanism is implemented. We ran all of our experiments on a 12-Core Intel machine with 126GB RAM and running Ubuntu Linux 16.04.

We first discuss the dataset of bootloaders we used, an analysis of the results, and an in-depth discussion of several use cases.

2.6.1 Dataset

For this work, we considered five different bootloaders. These devices represent three different chipset families: Huawei P8 ALE-L23 (Huawei / HiSilicon chipset), Sony Xperia XA (MediaTek chipset), and Nexus 9 (NVIDIA Tegra chipset). We also considered two versions of the LK-based bootloader, developed by Qualcomm. In particular, we considered an old version of Qualcomm's LK bootloader (which is known

Table 2.2: Vulnerabilities Evaluation. Alerts raised by BOOTSTOMP on potential security vulnerabilities. In order: bootloader, number of seeds, number of sinks, number of entry points, number of generated alerts, number of bug-related alerts, number of unique bugs, number of triggered timeouts, analysis time, average analysis time per entry point, and memory consumption.

Bootloader	Seeds	Sinks	EP	Total Alerts			Bug-Related Alerts			Bugs	Timeout	Time	Time per EP	Memory
				loop	deref	memcpy	loops	deref	memcpy			[mm:ss]	[mm:ss]	[MB]
Qualcomm (Latest)	2	1	3	1	1	2	0	0	0	**0**	1	12:49	04:16	512
Qualcomm (Old)	3	1	5	3	0	5	0	0	4	**1**	0	10:14	02:03	478
NVIDIA	6	1	12	7	0	0	1	0	0	**1**	0	24:39	02:03	248
HiSilicon	20	4	27	8	4	5	8	4	3	**5**	1	21:28	00:48	275
MediaTek	2	2	2	-	-	-	-	-	-	**-**	-	00:08	00:04	272
Total	33	9	49	19	5	12	9	4	7	**7**	2	69:18	09:14	1785

to contain a security vulnerability, CVE-2014-9798 [85]) and its latest available version (according to the official git repository [101]).

2.6.2 Finding Memory Corruption

We used BOOTSTOMP to analyze the five bootloaders in our dataset to discover memory corruption vulnerabilities. These vulnerabilities could result in arbitrary code execution or denial-of-service attacks. Table 2.2 summarizes our findings. In particular, the table shows the number of seeds, sinks, and entry points identified in each bootloader. The table also shows the number of alerts raised for each bootloader. Out of a total of 36, for 12 of them, the tool identified a potential path from a source to memcpy-like sink, leading to the potential of a buffer overflow. The tool raised 5 alerts about the possibility of a tainted variable being dereferenced, which could in

turn constitute a memory corruption bug. Finally, for 19, the tool identified that tainted data could reach the conditional for a loop, potentially leading to denial-of-service attacks. We then manually investigated all the alerts to determine whether the tool uncovered security vulnerabilities. Our manual investigation revealed a total of seven security vulnerabilities, *six* of which previously-unknown (five are already confirmed by the respective vendors), while the remaining one being the previously-known CVE-2014-9798 affecting an old version of Qualcomm's LK-based bootloader. Note that, as BOOTSTOMP provides the set of basic blocks composing the tainted trace together with the involved seed of taint and sink, manual inspection becomes easy and fast even for not-so-experienced analysts. We also note that, due to bugs in angr related to the analysis of ARM's THUMB-mode instructions, the MediaTek bootloader was unable to be processed correctly.

These results illustrate some interesting points about the scalability and feasibility of BOOTSTOMP. First, we note that each entry point's run elapsed on average less than five minutes (Duration per EP column), discovering a total of seven bugs. We ran the same set of experiments using a time limit of 40 minutes. Nonetheless, we noticed that no additional alerts were generated. These two results led us to believe that a timeout of ten minutes (i.e., twice as the average analysis run) was reasonable. Second, we noted a peak in memory consumption while testing our tool against LK bootloaders. After investigating, we found out that LK was the only bootloader in the

dataset having a well-known header (ELF), which allowed us to unconstrain all the bytes belonging to the $.data$ and $.bss$ segments, as stated in Section 2.5. Third, we note that the overall number of alerts raised is very low, in the range that a human analyst, even operating without debugging symbols or other useful reverse-engineering information, could reasonably analyze them. Finally, as we show in the table, more than one alert triggered due to the same underlying vulnerability; the occurrence of multiple alerts for the same functionality was a strong indicator to the analyst of a problem. This can occur when more than one seed fall within the same path generating a unique bug, for instance, when more than one tainted argument is present in a memcpy-like function call.

With this in mind, and by looking at the table, one can see that around 38.3% of the tainted paths represent indeed real vulnerabilities. Note also that in the context of tainted paths, none of the reported alerts were false positives (i.e., not tainted paths), though false positives are theoretically possible, as explained in Section 2.6.4.

Our tool uncovered five new vulnerabilities in the Huawei Android bootloader. First, an arbitrary memory write or denial of service can occur when parsing Linux Kernel's device tree (DTB) stored in the boot partition. Second, a heap buffer overflow can occur when reading the root-writable `oem_info` partition, due to not checking the `num_records` field. Additionally, a user with root privileges can write to the `nve` and `oem_info` partitions, from which both configuration data and memory access per-

missions governing the phone's peripherals (e.g., modem) are read. The remaining two vulnerabilities will be described in detail below.

Unfortunately, due to the architecture of the Huawei bootloader, as detailed in Section 2.2.1, the impact of these vulnerabilities on the security of the entire device is quite severe. Because this bootloader runs at EL3, and is responsible for the initialization of virtually all device components, including the modem's baseband firmware and Trusted OS, this vulnerability would not only allow one to break the chain of trust, but it would also constitute a means to establish persistence within the device that is not easily detectable by the user, or available to any other kind of attack. Huawei confirmed these vulnerabilities.

BOOTSTOMP also discovered a vulnerability in NVIDIA's `hboot`. `hboot` operates at EL1, meaning that it has equivalent privilege on the hardware as the Linux kernel, although it exists earlier in the Chain of Trust, and therefore its compromise can lead to an attacker gaining persistence. We have reported the vulnerability to NVIDIA, and we are working with them on a fix.

Finally, we rediscovered a previous vulnerability reported against Qualcomm's aboot, CVE-2014-9798. These vulnerabilities allowed an attacker to perform denial-of-service attack. However, this vulnerability has been patched, and our analysis of the current version of aboot did not yield any alerts.

Case study: Huawei memory corruption vulnerability. BOOTSTOMP raised multiple alerts concerning a function, whose original name we believe to be `read_oem()`. In particular, the tool highlighted how this function reads content from the flash and writes the content to a buffer. A manual investigation revealed how this function is vulnerable to memory corruption. In particular, the function reads a monolithic record-based datastructure stored in a partition on the device storage known as `oem_info`. This partition contains a number of *records*, each of which can span across multiple *blocks*. Each block is 0x4000 bytes, of which the first 512 bytes constitute a header. This header contains, among others, the four following fields: `record_id`, which indicates the type of record; `record_len`, which indicates the total length of the record; `record_num`, which indicates the number of blocks that constitute this record; `record_index`, which is a 1-based index.

The vulnerability lies in the following: the function will first scan the partition for blocks with a matching `record_id`. Now, consider a block whose record_num is 2 and whose record_index is 1. The fact that record_num is 2 indicates that this record spans across two different blocks. At this point, the `read_oem` function assumes that the length of the current block is the maximum, i.e., 0x4000, and it will thus copy all these bytes into the destination array, *completely ignoring the `len` value passed as argument*. Thus, since the `oem_info` partition can be controlled by an attacker, an attacker can create a specially crafted record so that a buffer overflow is triggered.

Unfortunately, this bootloader uses this partition to store essential information that is accessed at the very beginning of every boot, such as the bootloader's logo. Thus, an attacker would be able to fully compromise the bootloader, fastboot, and the chain of trust. As a result, it would thus be possible for an attacker to install a persistent rootkit.

Case study: Huawei arbitrary memory write. The second case study we present is related to an arbitrary memory write vulnerability that our tool identified in Huawei's bootloader.

In particular, the tool raised a warning related to the `read_from_partition` function. BOOTSTOMP pinpointed the following function invocation `read_from_partition("boot", hdr->kernel_addr)`, and, more precisely, the tool highlighted that the structure `hdr` can be attacker-controllable. Manual investigation revealed that not only `hdr` (and its field, including `kernel_addr`) are fully controllable by an attacker, but that the function actually reads the content from a partition specified as input ("boot", in this case), and it copies its content to the address specified by `hdr->kernel_addr`. Since this destination address is attacker-controllable, an attacker could rely on this function to write arbitrary memory (by modifying the content of the "boot" partition) to an arbitrary address, which the attacker can point to the bootloader itself. We note that this vulnerability is only exploitable when the bootloader is unlocked, but, nonetheless, it is a vulnerability that allows an attacker to run arbitrary code as the bootloader itself (and not just as part of non-secure OS). Moreover, the next

Table 2.3: Unlocking Functionality Evaluation. Alerts raised by BOOTSTOMP on potentially vulnerable `write` operation inside unlock routines. In order: bootloader, number of identified sinks, whether or not the unlocking routine is potentially vulnerable, whether or not the timeout triggered, analysis time, and remarks.

Bootloader	Sinks	Potentially vulnerable?	Timeout	Time [mm:ss]	Remarks
Qualcomm (Latest)	6	✓	✗	01:00	Detected write on flash and mmc
Qualcomm (Old)	4	✓	✗	00:40	Detected write on flash and mmc
NVIDIA	9	✗	✗	02:21	Memory mapped IO
HiSilicon	17	✓	✓	10:00	Write `oeminfo`
MediaTek	1	✗	✓	10:00	Memory mapped IO

Section provides evidence that, at least for this specific case, it is easy for an attacker to unlock the bootloader.

2.6.3 Analyzing (In)Secure State Storage

As a second use case for our tool, we use it to analyze the same five bootloaders we previously consider to determine how their security state (i.e., their lock/unlock state) is stored. In particular, as we discussed in Section 2.3, if the bootloader merely stores the security state on one of the flash partitions, then an attacker may be able to change the content of this partition, unlock the phone without the user's consent, and thus violate one of Google's core Verified Boot principles.

To run this experiment, we begin with the manually-identified unlocking functionality, as described in Section 2.5.2, and locate paths that reach automatically-identified writes to the device's storage. This means that each bootloader has one entry point. Table 2.3 shows the overall results of this experiment, including the number of possible write operations to the device's storage that occurred within the unlocking functionality. Our system was easily able to locate paths in Qualcomm's bootloader (both the old and the newest version) and Huawei's bootloader where the security state was written to the device's non-volatile storage. Upon manual investigation, we discovered that Qualcomm's simply stores the bit '1' or '0' for whether the device is locked. Huawei's stores a static hash, but can still be recovered and replayed (see case study at the end of this Section). In both cases, writing the needed value to the flash will unlock the bootloader, potentially bypassing the mandatory factory reset, if additional steps are not taken to enforce it, such as those mentioned in Section 2.7. Our tool did not identify any path to non-volatile storage for NVIDIA's or MediaTek's bootloaders. Upon manual investigation, we discovered that these two bootloaders both make use of memory-mapped I/O to write the value, which could map to anything from the flash to special tamper-resistant hardware. Thus, we cannot exclude the presence of vulnerabilities.

Case Study: Huawei bootloader unlock. Our tool identified a path from a function, which we believe to be called `oem_unlock`, to a "write" sink. Upon manual investigation, we were able to determine the presence of a vulnerability in the implementation

```
x = md5sum(unlock_code);
if (x == ``<target_value>'') {
  unlock_state = custom_hash(x);
  write(oem_info,unlock_state);
}
```

Figure 2.5: Implementation of the (vulnerable) unlock functionality in Huawei's bootloader.

of this functionality, as shown in Figure 2.5. In a normal scenario, the user needs to provide to the bootloader a device-specific `unlock_code`. Such code can be obtained by a user through Huawei's website, by providing the hardware identifiers of the device. The problem lies in the fact that the "correct" MD5 of the `unlock_code`, `<target_value>`, is stored in a partition of the device's storage. Thus, even if it not possible to determine the correct `unlock_code` starting from its hash, an attacker could just *reuse* the *correct* MD5, compute the expected `unlock_state`, and store it to the `oem_info` partition, thus entirely bypassing the user's involvement.

2.6.4 Discussion

As stated in Section 2.5, and as demonstrated by the results in this Section, our tool might present some false negatives as well as false positives. In this Section we consider the results achieved by our taint analysis engine, and we discuss how false positive and false negatives might arise.

As symbolic execution suffers from the path explosion problem, generally speaking, not all the possible paths between two program points can be explored in a finite amount of time. This might cause some tainted paths to be missed, causing some vul-

nerabilities to be missed. False negatives might be present also because BOOTSTOMP does not follow function calls when no taint is applied. This approach is very useful, since it makes our tool faster as less code has to be analyzed, but it might miss some correlation between pointers. In fact, if a future tainted variable is aliased, within a skipped function to a variable whose scope falls within the current function, and this variable later happens to reach a sink, it will not be reported.

Furthermore, since BOOTSTOMP relies on a maximum fixed inter-function level, it might not follow all the function calls it encounters, possibly resulting in some tainted variables not to be untainted as well as some pointer aliases not being tainted. This problem might create false positives and false negatives.

Additionally, false positives could possibly arise from the fact that not all the reported tainted paths lead to actual vulnerabilities. In fact, when the initial taint is applied, our tool tries to understand which parameter represents the variable(s) that will point to the read data, as explained in Section 2.5. If the taint is not applied correctly, this will result in false positive results. Note, however, that our tool would taint every parameter that our type inference heuristic does not exclude. Therefore, false negatives are not possible in this case.

Our concretization strategy could possibly introduce both false positives and false negatives. Given two unconstrained pointers, intuitively it is unlikely that they will point to the same memory location. Therefore, the most natural choice is to concretize

them (if necessary) to two different values. Assuming that these two pointers are indeed aliases, if one of them is tainted and the other reaches a sink, no alarm will be raised causing then a false negative. On the other hand, if both of them are tainted, but the former becomes untainted and the latter reaches a sink, an alarm would be raised causing then a false positive. According to our observations, these cases are very rare though, as we never encountered two unconstrained pointers that happened to be aliases.

Finally, it is worth noting that while we found some tainted paths that were not leading to actual vulnerabilities, our tool never detected a tainted path that was supposed to be untainted.

2.7 Mitigations

In this Section, we explore ways of mitigating the vulnerabilities discovered in the previous Section. With the increasing complexity of today's devices, it may be difficult to completely ensure the correctness of bootloaders, but taking some simple steps can dramatically decrease the attack surface.

As we have discussed throughout the previous Sections, the goal of Trusted Boot and Verified Boot is to prevent malicious software from persistently compromising the integrity of the operating system and firmware. The attacks we discovered all rely on the attacker's ability to write to a partition on the non-volatile memory, which the boot-

loader must also read. We can use hardware features present in most modern devices to remove this ability.

Binding the Security State. Google's implementations of Verified Boot bind the security state of the device (including the lock/unlock bit) to the generation of keys used to encrypt and decrypt user data, as described in Section 2.1.3. While not specifically requiring any particular storage of the security state, this does ensure that if the security state is changed, the user's data is not usable by the attacker, and the system will not boot without first performing a factory reset. This, along with the cryptographic verification mandated by Verified Boot, achieves the goals Google sets, but does not completely shield the bootloader from arbitrary attacker-controlled input while verifying partitions or checking the security state.

Protect all partitions the bootloader accesses. Most modern mobile devices utilize non-volatile storage meeting the eMMC specification. This specifies the set of commands the OS uses to read and write data, manage partitions, and also includes hardware-enforced security features. Since version 4.4, released in 2009 (a non-public standard, summarized in [83]), eMMC has supported *Power-on Write-Lock*, which allows individual partitions to be selectively write-protected, and can only be disabled when the device is rebooted. The standard goes as far as to specify that this must also be coupled with binding the reset pin for the eMMC device to the main CPU's reset pin, so that intrusive hardware attacks cannot be performed on the eMMC storage alone.

While we are not able to verify directly whether any handsets on the market today makes use of this feature, we note that none of the devices whose bootloaders we examined currently protect the partitions involved in our attacks in this manner. Furthermore, we note that many devices today make use of other features from the same standard, including Replay-protected Memory Blocks (RPMB) [83] to provide secure storage accessible from Secure-World code.

eMMC Power-on Write-protect can be used to prevent any partition the bootloader must read from being in control of an attacker with root privileges. Before executing the kernel contained in the boot partition, the final stage bootloader should enable write protection for every partition which the bootloader must use to boot the device. In Android, the `system` and `boot` partitions contain entirely read-only data (excluding during OS updates), which the bootloader must read for verification, and therefore can be trivially protected in this way. To close any loopholes regarding unlocking the bootloader, the partition holding device's security state should also be write-protected. The `misc` partition used by Qualcomm devices, for example, is also used to store data written by the OS, so the creation of an additional partition to hold the security state can alleviate this problem.

This does not impede any functionality of the device, or to our knowledge, cause any impact to the user whatsoever. Of course, this cannot be used to protect partitions the OS must write to. While the OS does need to write to `system` and `boot` to

perform routine software updates, this too can be handled, with only small changes. If an update is available, the bootloader should simply not enable write-protection when booting, and perform the update. This increases only marginally the attack surface, adding only the update-handling code in the bootloader.

It should be noted that this method cannot protect the status of the "Allow OEM Unlock" option in the Android Settings menu, which by its very design must be writable by the OS. This means that a privileged process can change this setting, but unlocking the bootloader still requires physical control of the device as well.

Alternative: Security State in RPMB. eMMC Power-on Write Lock can be used to protect any partition which is not written to by the OS. If, for whatever reason, this is not possible, this could also be stored in the Replay-protected Memory Block (RPMB) portion of the eMMC module.

We can enforce the property that the OS cannot tamper with the security state by having the Trusted OS, residing in the secure world, track whether the OS has booted, and only allow a change in the security state if the bootloader is running. Using RPMB allows us to enforce that only TrustZone can alter this state, as it holds the key needed to write the data successfully.

When the device boots to the final stage bootloader, it will signal to TrustZone, allowing modifications to the security state via an additional command. Once the boot-

loader is ready to boot the Android OS, it signals again to TrustZone, which disallows all writes to the device until it reboots.

While this requires minor modifications to the Trusted OS and final-stage bootloader, it does not require a change in the write-protection status or partition layout.

Chapter 3

KARONTE: Detecting Insecure Multi-binary Interactions in Embedded Firmware

In the previous Chapter, we studied the security of the software that is executed when a device is switched on: the bootloaders. In this Chapter, we study the security of the software in a firmware image that is responsible for handling the device's peripherals and managing user requests. In the literature, this software is usually just referred to as the *firmware* (or *firmware sample*) of an IoT device, and, therefore, it will be referred to as such in this dissertation too. As introduced in Chapter 1, firmware samples are often made of different components that interact with each other to fulfill users' requests. In the following, we study the details of such interconnected environments, and discuss why existing firmware analysis techniques are ineffective to precisely uncover bugs in firmware for IoT devices. Then, I present KARONTE: a novel static analysis tool capable of analyzing embedded-device firmware by modeling and tracking multi-binary

interactions. Our tool propagates taint information between binaries to detect insecure, attacker-controlled interactions, and effectively identify vulnerabilities.

First, I explain the attacker model that we assumed in this work, then I describe how the different components of a firmware sample communicate, and, finally, in the rest of this Chapter I discuss KARONTE in depth. We tested KARONTE on 53 firmware samples from various vendors, showing that our prototype tool can successfully track and constrain multi-binary interactions. In doing so, we discovered 46 zero-day bugs, which we disclosed to the responsible entities. We performed a large-scale experiment on 899 different samples, showing that KARONTE scales well with firmware samples of different size and complexity, and can effectively and efficiently analyze real-world firmware in a generic and fully automated fashion.

3.1 IoT Attacker Model

IoT devices exchange data over the network. This data can come directly from the user (e.g., through a web interface), or indirectly from a trusted remote service (e.g., cloud backends). Many devices, especially routers, smart meters, and a host of low-power devices, such as smart light bulbs and locks, use the former paradigm. Moreover, recent attacks have shown that such devices can be exploited by clever remote attackers, even when their communication is restricted to a closed local network [62]. In this

work, we consider network-based attackers who communicate directly with the device, either through a local network or the Internet. However, as shown in Section 3.19, KARONTE can be easily extended to other scenarios.

3.2 Firmware Complexity

The firmware of modern IoT devices is complex and made of multiple components. These components can take the form of either different binaries, packaged in an embedded Linux distribution, or different modules, compiled into a large, single-binary embedded OS ("blob firmware"). The former type of firmware is, by far, the most ubiquitous: a large-scale experiment analyzed tens of thousands of firmware samples, and found that 86% of them were Linux-based [27]. Similar to other Linux-based systems, Linux-based firmware includes a large number of interdependent binaries.

The different binaries (or components) of the firmware on embedded devices share data to carry out the device's tasks. Under our attacker model, this interaction is critical, as we focus on bugs that can be triggered by attacker input from "outside" of the device (i.e., over the network), but may affect binaries other than those directly facing the network. Any analysis that focuses only on these *network-facing* binaries would miss bugs contained in other components [21]. On the other hand, an analysis that focuses on all the binaries in isolation would produce an unacceptable amount of false alerts.

```
1   char* parse_URI(Req* req) {
2   char* p  = req[1];
3   if (!strncmp(p, "<soap:AddRule", 13))
4     return p; // unconstrained data
5   // ...
6   if (strlen(p) > 127)
7     p[128] = 0;
8   return p; //   constrained data
9 }
10 int serve_request(Req *req) {
11   char *data = parse_URI(req);
12   setenv("QUERY_STRING", data, 1);
13   execve(get_handler(req));
14 }
```

Figure 3.1: Decompiled code of a network-facing program of a real firmware sample.

We demonstrate this in the following example service, based on a real-world firmware sample. This service is composed of a network-facing web server (Figure 3.1) that executes a CGI handler binary (Figure 3.2). When the web server receives a user request, it invokes the function `serve_request`. Then, after parsing the request (`parse_URI`), the web server executes the handler program, passing data via the `QUERY_STRING` environment variable. The handler binary retrieves the data and passes it to `process_request`. This function contains a bug: if the value of the field `op` in the user request is longer than 128 bytes, a buffer overflow occurs. This overflow is attacker-controlled and represents a significant vulnerability.

While this specific overflow would be detected by an analysis that only focuses on the handler binary, any single-binary analysis would detect *two* vulnerabilities in this program. The second one is the overflow of the `log_dir` buffer caused by the `LOG_PATH` environment variable. Though this is a legitimate bug, its classification as a vulnerability depends on the provenance of the data in `LOG_PATH`. If an attacker

cannot control this data, the bug is not a vulnerability, and the real vulnerability should be prioritized. Ideally, every alert would be examined, and every bug fixed. Unfortunately, this goal is not feasible in practice. While this simple example has two alerts that reveal one vulnerability, our evaluation shows that static analysis on individual binaries in real-world firmware can produce *thousands* of alerts per device, requiring months of analyst time to process.

For static analyses to be feasible on binaries, an approach to filter out bugs that cannot be triggered by an attacker is critical. KARONTE is such an approach. It identifies data dependencies across binaries, such as the one in this example, by using static analyses to connect functions that produce (or *set*) data to functions in other binaries that consume (or *get*) it.

Throughout this Chapter, we refer to the program interactions shown in the above example as *multi-binary* interactions. Similarly, we refer to vulnerabilities that involve data flows across multiple binaries as *multi-binary vulnerabilities*. Finally, we refer to the binary producing data (e.g., the web server in Figure 3.1) as a *setter* binary, and the binary consuming data (e.g., the handler binary in Figure 3.2) as a *getter* binary.

```
1  int process_request(char *query, char *log_path) {
2    char *q, arg[128];
3    char log_dir[128];
4    if (!(q=strchr(query, "op=")))
5      return;
6    strcpy(arg, q); // query string argument
7    strcpy(log_dir, dirname(log_path));
8    // ...
9    return 0;
10 }
11 int main(int argc, char *argv[], char *envp[]) {
12   char *query = getenv("QUERY_STRING");
13   char *log_path = getenv("LOG_PATH");
14   process_request(query, log_path);
15 }
```

Figure 3.2: Decompiled code of a handler binary that contains two bugs. However, only one bug is reachable by an attacker.

3.3 IPC in IoT Firmware

Automatically determining how user input is introduced into and propagates through an embedded device is an open problem [93, 143, 152], and prone to a discouraging rate of false positives [57]. However, we observed that, in practice, processes communicate through a finite set of *communication paradigms*, known as Inter-Process Communication (or IPC) paradigms.

An instance of an IPC is identified through a unique *key* (which we term a *data key*) that is known by every process involved in the communication. As this information has to be available to all the involved programs before their execution, it is usually hard-coded in the binaries themselves. For example, two binaries exchanging data through a file have to know the filename (i.e., the data key) prior to transferring the data.

Data keys associated with common IPC paradigms can be used to *statically* track the flow of attacker-controlled information between binaries. Below, we describe the most common IPC paradigms employed in firmware[1].

Files. Processes can share data using files. A process writes data on a given file, and another process reads and consumes such data. The data key is the name of the file itself.

Shared Memory. Processes can share memory regions. Shared memory can be either backed by a file on the filesystem, or be anonymous (if two processes are in a parent-child relationship). In the former case, the data key is represented by the backing file name, whereas in the latter case by the virtual address of the shared memory page.[2]

Environment Variables. Processes can share data via environment variables. In this case, the data key is the environment variable name (e.g., `QUERY_STRING`).

Sockets. Processes can use sockets to share data with processes that reside on the same host (Unix domain sockets with a file path) or on a different host (network sock-

[1] We focus on IPC mechanisms that enable rich data exchange. IPCs that do not transport data (e.g., signals) are not included, as they are out of our scope. Additionally, we reference UNIX-based concepts for user-space IPC. Other systems (e.g., iOS) have analogous concepts.

[2] Note that, components in a "blob" can use a statically mapped region to exchange data. By using the addresses of these regions as data keys, we can reason about data flows without analyzing the prohibitively large amount of control flow that separates the components themselves in a real-world firmware.

ets). The socket's endpoint (e.g., IP address and port, or file path of a Unix domain socket) represents the data key.

Command Line Arguments. A process can spawn another process and pass data through command line arguments. The data key is the name of the invoked program.

We represent shared data as a tuple $(data\_key,\ data)$.

3.4 KARONTE

KARONTE is an approach that performs *inter-binary* data-flow tracking to automatically detect insecure interactions among binaries of a firmware sample, ultimately discovering security vulnerabilities. Although our system focuses on detecting memory-corruption and DoS vulnerabilities, it can be easily extended, as discussed in Section 3.18. KARONTE analyzes firmware samples through the following five steps (Figure 3.3):

Firmware Pre-processing. KARONTE's input is comprised of a firmware sample (i.e., the entire firmware image). As a first step, KARONTE unpacks the firmware image using the off-the-shelf firmware unpacking utility *binwalk* [54].

Border Binaries Discovery. The Border Binaries Discovery module analyzes the unpacked firmware sample, and automatically retrieves the set of binaries that export the device functionality to the outside world. These *border* binaries incorporate the logic

necessary to accept user requests received from external sources (e.g., the network). As such, they represent the point where attacker-controlled data is introduced within the firmware itself. For each border binary, this module identifies the program points that reference attacker-controlled data (Section 3.5).

Binary Dependency Graph (BDG) Recovery. Given a set of border binaries, KARONTE builds a *Binary Dependency Graph* (*BDG*), which is a directed graph [137] that models communications among those binaries processing attacker-controlled data. The BDG is iteratively recovered by leveraging a collection of *Communication Paradigm Finder* (*CPF*) modules, which are able to reason about the different inter-process communication paradigms (Section 3.6).

Multi-binary Data-flow Analysis. Given a binary b in the BDG, we leverage our static taint engine (see Section 3.7) to track how the data is propagated through the binary and collect the constraints that are applied to such data. We then propagate the data with its constraints to the other binaries in the BDG that have inbound edges from b (Section 3.8).

Insecure Interactions Detection. Finally, KARONTE identifies security issues caused by insecure attacker-controlled data flows, which are reported for further inspection (Section 3.9).

KARONTE's novelty lies in the creation of its Binary Dependency Graph and its ability to accurately propagate taint information *across* binary boundaries, enabling

the detection of complex, multi-binary vulnerabilities in an efficient manner, and drastically decreasing the number of false positives that would be otherwise generated. While KARONTE focuses on inter-binary software bugs, it also performs single-binary analysis.

Furthermore, though KARONTE detects data-flows across binaries of a firmware sample, its generic design allows KARONTE to also reason about interactions of different modules of a monolithic embedded OS, as long as a separation among these modules is present (e.g., they represent different processes at runtime), as shown in Section 3.19. Finally, given our attacker model (Section 3.1), we assume that border binaries are represented by network-facing binaries (i.e., binaries implementing network services). For this reason, we interchangeably use the terms border binaries and network-facing binaries.

3.5 Border Binaries Discovery

KARONTE is designed to detect vulnerabilities that may be exploited by attackers over the network. To do so, KARONTE first identifies the set of binaries that export network services (i.e., network-facing binaries) in a firmware sample. We leverage the observation that network-facing binaries are the components of a firmware sample that

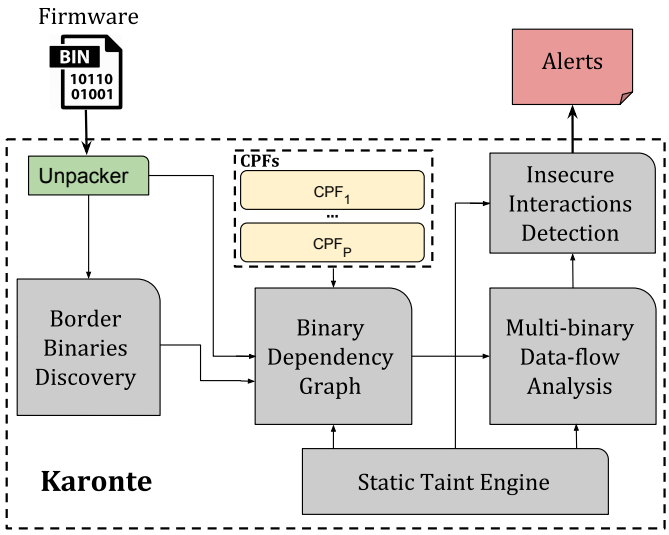


Figure 3.3: KARONTE Overview. After unpacking a firmware sample, KARONTE extracts the binaries handling user requests, identifies their data dependencies to build the Binary Dependency Graph (BDG), and uses its inter-binary taint analysis engine to find insecure data flows.

receive and *parse* user-provided data. Therefore, we identify those binaries within a firmware sample that parse data read from a network socket.

Following Cojocar et al. [24] work, we utilize three features to identify functions in embedded systems that implement parsers: (i) the number of basic blocks ($\#bb$), (ii) the number of branches (e.g., if-then-else, loops) ($\#br$), and (iii) the number of conditional statements used in conjunction with memory comparisons ($\#cmp$). Since we want to specifically identify input-affected network parsers, we consider two additional features: (iv) a metric we call *network mark* ($\#net$), and (v) a flag we call *connection mark* ($\#conn$).

The network mark feature encodes the probability that a parsing function handles network messages, and it is calculated by identifying every memory comparison in

the code of the function, and comparing the referenced memory locations against a preset list of *network-encoding* strings (e.g., `soap` or `HTTP`). We initialize $\#net$ to 0 and increment it for every comparison against network-encoding strings present in the code.

The connection mark flag, instead, indicates if any data read from a network socket is used in a memory comparison. We initialize $\#conn$ to 0 and set it to 1 if there exists a data-flow between a socket read and a memory comparison operation.

We combine the aforementioned five features to compute the *parsing score* ps_b of a binary b as follows:

$$\begin{aligned} ps_b &= max(\{ps_j \mid \forall j \in get\_functions(b)\}), \\ ps_j &= (\sum_{i \in \{bb,br,cmp\}} k_i * \#i_j) * (1 + k_n * \#net_j) * (1 + k_c * \#conn_j) \end{aligned} \tag{3.1}$$

where each constant k_i is set to maximize the parsing detection capabilities ($k_{bb} = 0.5$, $k_{br} = 0.4$, $k_{cmp} = 0.7$ [24]), whereas k_n and k_c promote functions that refer to network-encoding keywords and binaries that parse network data, respectively. The optimal values for the last two constants are found empirically in Section 3.19.2. Finally, ps_j is the parsing score of the *j*-th function of b. Note that, we introduce our two features as multipliers in order to highlight input-affected network parsers.

Since all binaries are likely to have a score greater than zero, we need to distinguish and separate the "most significant" scores. To this end, we leverage the DBSCAN density-based clustering algorithm [36], which groups binaries whose scores are closely

packed together. Then, we select the cluster that contains the binary having the highest parsing score in the firmware sample, and consider all the binaries belonging to the cluster as the initial set of network-facing binaries.

Finally, the algorithm implemented by this module returns the unpacked firmware sample, the set of identified network-facing binaries, and the program locations containing memory comparisons against network-encoding keywords. These memory comparisons represent the program locations where attacker-controlled data is *more likely* to be referenced.

3.6 Binary Dependency Graph

The Binary Dependency Graph module detects data dependencies among a set of binaries or components belonging to a firmware sample. Furthermore, it establishes how data is propagated from a setter binary to a getter binary. Data propagation across different processes differs from data transfer during subroutine calls/returns and program-library dependency analyses, as both of these are guided by control flow information. For inter-process interactions, there is no control flow transfer to rely on, because after making the data available (e.g., through environment variables), processes proceed with their execution. Since processes do not normally access other processes' memory regions, traditional points-to analyses are also futile.

KARONTE tackles these problems by modeling the various inter-process communication paradigms through the use of a set of modules that we call *Communication Paradigm Finder*s (or *CPF*es). KARONTE uses them to build a graph, called *Binary Dependency Graph* (or *BDG*), which encodes the data flow information among binaries within a firmware sample.

3.6.1 Communication Paradigm Finders

A CPF provides the necessary logic to detect and describe instances of a communication paradigm (e.g., socket-based communication) used by a binary to share data. To achieve this goal, a CPF considers a binary and a program path (i.e., a sequence of basic blocks), and checks whether the path contains the necessary code to share data through the communication paradigm that the CPF represents. If so, it gathers the details of the communication paradigm through the following paradigm-specific functionality:

Data Key Recovery. The CPF recovers data keys that reference data being set or retrieved by the binary under the associated communication paradigm.

Flow Direction Determination. The CPF identifies all the program points where data represented by the collected data keys is accessed. If such program points exist, it determines the *role* of each program point in the communication flow (i.e., setter or getter).

Binary Set Magnification. The CPF identifies other binaries in the firmware sample that refer to any of the data keys previously identified. These binaries are likely to share data with the binary currently under consideration, and are thus scheduled for further analysis.

We then combine the information gathered by the different CPFes to create edges in the Binary Dependency Graph, recovering the data flow across different binaries.

The specifics of each CPF depend on the OS that the firmware sample runs on (e.g., Linux). Therefore, to maintain OS-independence and to reason about inter-process communication paradigms when some information is missing (e.g., a firmware blob), KARONTE uses a generic OS-independent CPF, which we call the *Semantic CPF*. This CPF leverages the intuition that any communication among processes must rely on data keys, which are often hard-coded in binaries (e.g., hard-coded addresses). To this end, the Semantic CPF detects if a hard-coded value is used to *index* a memory location to access some data of interest (e.g., attacker-controlled data). Our prototype of KARONTE implements the Environment, File, Socket and Semantic CPFes (details in Section 3.10).

3.6.2 Building the BDG

KARONTE models data dependencies among binaries through a disconnected cyclic digraph [137], called the *Binary Dependency Graph* (or *BDG*). A BDG, G, of the

Algorithm 1 Binary Dependency Graph Algorithm

```
function BDG(int_locs, B, fw)
    comm_info ← {}
    E ← {}
    for each b ∈ B do
        locs ← get_locs(int_locs, b)
        for each loc ∈ locs do
            f_addr ← get_faddr(loc)
            for each block ∈ explore_paths(f_addr) do
                if (address(block) == loc) then
                    buf ← get_buf(loc)
                    apply_taint(buf)
                end if
                if matches_CPF(block) then
                    CPF_p = get_CPF(block)
                    k ← find_data_key_and_role(block, CPF_p)
                    B_new, int_locs_new ← get_new_binaries(fw, k, CPF_p)
                    update_binaries(B, int_locs, B_new, int_locs_new)
                    comm_info ← comm_info ∪ {b, block, CPF_p, k}
                end if
            end for
        end for
    end for
    for each {b, block, CPF_p, k} ∈ comm_info do
        if is_setter(block, k) then
            getters ← get_getters(comm_info, k, CPF_p)
            E ← E ∪ create_edges(b, getters)
        end if
    end for
    return (B, E)
end function
```

set of binaries B is denoted as $G = (B, E)$, where, E is the set of directed edges. Each directed edge $e \in E$ from $b_1 \in B$ to $b_2 \in B$ is represented by a triplet $e = ([b_1, loc_1, cp_1], [b_2, loc_2, cp_2], k)$, which indicates that the information associated with the data key k (e.g., an environment variable name) can flow from binary b_1 at location loc_1 (e.g., a program point containing a call to the `setenv` function) via the communication paradigm cp_1 (e.g., the OS environment), to the binary b_2 at location loc_2 (e.g., a call to the `getenv` function) via the communication paradigm cp_2.

The algorithm to recover the Binary Dependency Graph (Algorithm 1) begins by considering the information gathered by the Border Binaries Discovery module: (i)

the unpacked firmware sample in analysis (fw), (ii) the border binaries (B), and (iii) a set of program locations (int_locs) performing memory comparisons. Then, for each binary b in B, we consider each location loc in int_locs belonging to b (function get_locs), and we leverage our taint analysis engine (Section 3.7) to bootstrap a symbolic path exploration starting from the beginning of the function containing loc (function $explore\_paths$). When the analysis reaches loc, we taint the memory location buf being referenced, i.e., the memory location being compared against the network-encoding keyword (functions get_buf and $apply\_taint$).

In each step of the path exploration (i.e., for each visited basic block), we invoke each of our CPF modules, which analyze the current path and use the taint information (propagated by the taint engine during the path exploration) to detect if the binary b is sharing some tainted data d. If a CPF_p *matches*, i.e., it detects that the analyzed binary relies on the communication paradigm p to share some data, we leverage CPF_p to recover all of the *details* of the communication paradigm instance in use. More precisely, the CPF_p recovers the data key k used to share data through p and infer the role (i.e., setter or getter) of the binary for k (function $find\_data\_key\_and\_role$) and finds other binaries within the firmware sample that might communicate through this channel (function $get\_new\_binaries$). Newly discovered binaries are then added to the overall set of binaries to analyze. Note that, when any of these new binaries B_{new} is scheduled to be analyzed, the analysis has to know where to apply the taint initially.

In other words, we have to detect where the shared data is initially *introduced* in these new binaries. Therefore, for each newly added binary b_a, the CPF_p also retrieves the program points int_locs_{new} where the data key k is referenced, and add them to int_locs. These last two operations are performed by the function $update\_binaries$. Finally, for each analyzed binary b, we consider each CPF (cp) that matched for b over some key k, and use cp to retrieve the role of b for k (e.g., setter). Then, we create an edge between b and any other binaries that have the opposite role of b for k (e.g., getter).

To demonstrate the BDG algorithm, we again refer to Listing 3.1. The BDG algorithm starts by considering the memory comparison against a network-encoding keyword (Line 3). After inferring that the variable `p` is used in the memory comparison, we taint the memory location it points to, and bootstrap the intra-procedural taint analysis exploration, starting from the function `parse_URI` (Line 1), and propagating the taint by following the control flow of the program. When the taint exploration reaches the `execve` function call (Line 13), the Environment CPF detects that another binary is being executed, and that the `setenv` function is used to set the data key `QUERY_STRING`. Therefore, the Environment CPF establishes that the binary in analysis is a setter for `QUERY_STRING`. Then, the Environment CPF scans the firmware sample and finds other binaries relying upon the same data key, and adds them to the set of binaries to analyze. Finally, for each newly added binary, the En-

vironment CPF retrieves the code locations where the data key `QUERY_STRING` is referenced (e.g., a call to the function `getenv("QUERY_STRING")`).

3.7 Static Taint Analysis

KARONTE uses taint propagation to detect multi-binary vulnerabilities. This Section describes the operation of the underlying taint engine, and the next Section discusses how KARONTE combines the taint engine with the BDG, described previously, to achieve such detection.

KARONTE's taint engine is based on BOOTSTOMP (see Chapter 2). Given a source of taint s (e.g., a function returning untrusted data) and a program point p, our taint engine performs a symbolic path exploration starting from p, and, every time s is encountered, the taint engine assigns a new taint ID (or tag) to the memory location receiving data from s. KARONTE's taint engine propagates taint information following the program data flow, and it untaints a memory location (i.e., by removing its taint tag) when the memory location gets overwritten by untainted data, or when its possible values are constrained (e.g., due to semantically equivalent `strlen` and `memcmp` functions). Our taint engine presents two improvements compared to related work: (i) it includes a path prioritization strategy, and (ii) it introduces the concept of *taint tag dependencies*.

```
1  char* parse(char *start) {
2    char* end = start + strlen(start) - 1;
3    while ( start < end )
4      switch ( *start[0] ) {
5        case '=':
6          return start + 1;
7        case ';':
8          return NULL;
9        default:
10         start ++;
11     }
12 }
13 void foo() {
14   char dst[512], *user_input = get_user_input();
15   char *cmd = parse(user_input);
16   size_t n = strlen(cmd);
17   if (n >= 512)
18     return -1;
19   strcpy(dst, cmd);
20 }
```

Figure 3.4: Path prioritization and taint dependencies use case.

The path prioritization strategy tackles the undertaint problem, which affects taint engines based on path exploration when dealing with implicit control flows [51], by prioritizing more *interesting* paths. In the scope of a taint analysis, a path p_1 is considered to be more interesting than a path p_2 if a variable of interest is tainted in p_1, and untainted in p_2.

Consider the example in Listing 3.4, and assume that the variable `user_input` (Line 14) points to tainted data. When the function `parse` is invoked, the variable `start` (Line 1) aliases `user_input` (i.e., they point to the same memory location), and, therefore, it points to tainted data. The function `parse` contains, potentially, an infinite number of paths: If the variable `start` is represented by an unconstrained symbolic expression, there is always a possible path passing through the `default` statement (Line 9) to the head of the while loop (Line 3). Among these paths, only

those passing through the first case statement (Line 5) would propagate the taint outside the function. Therefore, an analysis that does not explore these paths would mistakenly establish that `user_input` cannot affect the variable `cmd` (Line 15).

Our path prioritization strategy aims to valorize those paths within a function that *potentially* propagate the taint also outside the function (as the paths passing through the first `case` statement in Listing 3.4). As expected, we noticed that network-facing binaries contain various sanitization functions that can cause the issue just discussed. In Section 3.10, we describe the implementation details of our path prioritization feature.

Finally, in our taint engine, an analyst can create dependencies among tainted variables having different tags (*taint tag dependencies*). Tracking these dependencies plays an important role in having an effective untaint policy in a multi-tag taint tracking system, thus alleviating the overtainting problem [111].

To demonstrate this, consider again the example in Listing 3.4, and assume that there exists an untaint policy to remove a taint tag when a variable is explicitly constrained within a range of values. First, as `get_user_input` generates untrusted data (Line 14), a new taint tag t_1 is created and assigned to `user_input`. If the function `strlen` is not analyzed (e.g., its code is not available or the call is not followed to keep the overall analysis tractable), following the semantics of a multi-tag taint tracking, the variable `n` gets tainted using a different tag t_2. When the taint execution engine reaches the `if` statement (Line 17), following the untaint policy in use, the

variable n is automatically untainted by removing the tag t_2. Given that the taint tag of `user_input` (t_1) is different than `n`'s tag (t_2), `user_input` is not untainted, and the call to the unsafe `strcpy` (Line 19) could cause a false positive to be generated. This behavior emerges because some functions that semantically constrains tainted data might not be analyzed (due to lack of code, or limits of the employed analysis). The solution we propose is to maintain the information that the taint tag of `user_input` (i.e., t_1) *depends* on the taint tag of `n` (i.e., t_2), and, to untaint `user_input` when `n` is untainted. We say that a taint tag t_1 *depends* on a taint tag t_2, if removing t_2 (i.e., untainting the variable with taint tag t_2) provokes t_1 to be removed. Of course, the taint tag $t1$ might depend on multiple taint tags. In this case, if all the tags that t_1 depends on are removed, t_1 is removed too. Our prototype automatically finds semantically equivalent `memcmp` and `strlen` functions, and applies taint tag dependencies (see Section 3.10).

3.8 Multi-binary Data-flow Analysis

To discover insecure interactions among binaries and find vulnerabilities, we need to recover the data-flow details of the binaries in a BDG. Enumerating all the possible *inter-binary* paths in a BDG leads, in general, to the path explosion problem [18].

Our key insight is that the inter-binary paths *more likely* to lead to bugs are those that apply *less strict* constraints on the user-provided data d (i.e., the set of values that d can assume has a higher cardinality). To retrieve such paths, we collect the sets of constraints that a binary applies to d across different program paths, and propagate to other binaries only the least restrictive set of constraints.

To do so, we create a graph that we called the *Binary Flow Graph* (or *BFG*), which extends the BDG with the least strict set of constraints applied to the data shared among multiple binaries. In the BFG, an edge ([b_1, loc_1, cp_1, c_1], [b_2, loc_2, cp_2, c_2], k) indicates that the data associated with the data key k can flow from the binary b_1 at location loc_1 via the communication paradigm cp_1 with the set of constraints c_1 to the binary b_2 at location loc_2 via the communication paradigm cp_2 with the set of constraints c_2. The BFG building algorithm is based on the notion of chaotic iteration [4], and is composed of two phases.

Initialization. We consider every edge in the BDG and create a new edge setting $c_1 = c_2 = \perp$ ($\perp$ means "uninitialized"). Next, we consider every edge e whose setter (i.e., b_1) is a border binary, and retrieve the variable var_1 that contains the data being shared at location loc_1. Then, we use our taint engine to explore the paths between the entry point of the function containing loc_1 and loc_1 itself, and collect, for each path, the set of constraints applied to var_1. For instance, if var_1 maximum length is checked (e.g., through a `strlen`) against a constant value, we collect such constraint. Then,

we select the least strict set of constraints l_1, and set $c_1 = l_1$. Finally, we add e to a set $wset$, which is used during the second phase.

Constraint Propagation. We consider every edge $e_w \in wset$, and set $c_2 = c_1$, thus propagating the constraints from the setter binary to the getter binary. We then retrieve the variable var_2 used by b_2 to receive the data at loc_2 and find the least restrictive set of constraints l_2 that the binary applies to var_2 (relying on the same approach used to find l_1), and set $c_2 = c_2 \cup l_2$.

As b_2 might further share the data, we also determine the additional constraints that b_2 applies to such data before re-sharing it. To do this, we collect every edge e_r where the binary b_2 is the setter. Then, we run our taint engine to find a path between the program point where the binary previously received the data (i.e., loc_2 of edge e_w) and the location where it shares it further (i.e., loc_1 of edge e_r) and find the least strict set of constraints l_r applied to var_2 along these paths. If we cannot find a path between these two program points (e.g., due to limits of the underlying analyses), we determine l_r using the same approach used to find l_1 (i.e., starting from the entry point of the function containing loc_1 of e_r). Finally, we consider the constraints $c^* = l_r \cup c_2$ and the constraints for the setter of e_r. If the latter set is uninitialized (i.e., $c_1 = \perp$ for e_r) or more restrictive than c^*, we substitute it with c^* and add e_r to $wset$—thus keeping the least restrictive constraints. We iterate this phase until $wset$ is empty.

3.9 Insecure Interactions Detection

The Insecure Interactions Detection module leverages the BFG to find dangerous data flows and detect subsets of two classes of vulnerabilities: (i) memory-corruption bugs (e.g., buffer overflows) and (ii) denial of service (DoS) vulnerabilities (e.g., attacker-controlled loops). To detect the former class, we first find *memcpy-like* functions within a binary, that is, every function that is semantically equivalent to a `memcpy` (Section 3.10). Then, if attacker-controlled data unsafely reaches a memcpy-like function (e.g., without being sanitized), we raise an alert. To detect the latter class of vulnerabilities, we retrieve the conditions that control (guard) the iterations of a loop. Then, we check whether their truthfulness completely depends on attacker-controlled data, and, if so, we raise an alert. We refer to both *memcpy-like* functions and attacker-controlled loops with the general term *sinks*.

The Insecure Interactions Detection phase works as follows. First, we consider every edge e_f in a BFG, and for each node $(b, loc, cp, c) \in e_f$, we leverage the static taint engine to bootstrap a symbolic path exploration from the function f containing loc. Then, when we encounter the location loc, we rely on the provided CPF cp to retrieve the address of the buffer buf that references attacker-controlled data at location loc (e.g., the memory location returned by `getenv`), and apply the taint to it. Fur-

thermore, at each step of the path exploration, we collect any constraints on buf (in a similar way as explained in Section 3.8) and add them to c.

If a sink is encountered during the path exploration, we check whether it contains tainted data. If the sink is a loop, and one of its conditions completely relies on tainted variables, we raise an alert (for a possible DoS vulnerability). On the other hand, if the sink is a memcpy-like function, we retrieve the address of the destination buffer $bdst$. Then, we retrieve the allocation point of $bdst$ (e.g., its position in the function's stack) and estimate its boundaries (e.g., the offset of the surrounding variables in the stack) to recover its size. If the size of buf (given by its constraints c) is greater than the size of $bdst$, we raise an alert, as it means that the copy operation might produce a buffer overflow.

Finally, we consider every disconnected node in the BFG, and perform a single-binary static analysis.

3.10 KARONTE Implementaion Details

In this Section, we present the implementation details of our prototype of KARONTE, which is based on angr [116], and we show the details of a vulnerability discovered by our prototype.

3.11 Functions Identification

Our prototype of KARONTE scans all the functions in the binary under analysis to find three types of functions: (i) functions that are semantically equivalent to memory comparisons (e.g., `memcmp`), which we term *memcmp-like* functions, (ii) functions that copy the content of a memory location to another (e.g., `memcpy`), which we term *memcpy-like* functions, and (iii) functions that calculate the length of a buffer, which we term *strlen-like* functions.

Intuitively, a *memcmp-like* function f should contain a loop used to compare the memory locations pointed by different function parameters of f. To find these functions, we analyze each function f of a binary b that contains at least a loop. In particular, we linearly scan the instructions in the body of the loop, and retrieve each program point p containing a memory comparisons instruction (e.g., using an opcode from the x86 `cmp` instruction family). Then, we compute a static backward slice from p up to f's entry point, and inspect f's arguments to check whether they could affect the operands of the considered memory comparison at the program point p. If so, we consider f a candidate *memcmp-like* function. Finally, we calculate the size of f (in terms of the number of basic blocks), and adopt BootStomp's threshold 2 to filter out as many false positives as possible.

The approach to find *strlen-like* functions works in a very similar manner. The difference is that we also expect these functions to contain a counter that is incremented at each iteration of the loop.

Finally, to automatically identify *memcpy-like* functions, we adopt the same approach proposed in BOOTSTOMP 2.

If a function body is not available (i.e., the function is implemented in an external library not present in the firmware sample), we apply string matching heuristics on the name of the function to detect whether it belongs to one of the three function types just described.

Furthermore, as an optimization, we abstract the *memcpy-like*, *memcmp-like*, and *strlen-like* functions by providing function summaries, which we execute every time one of these functions is encountered during KARONTE's symbolic path exploration (e.g., during the BDG algorithm). With this optimization, we alleviate the path explosion problem and speed up the overall analysis, while maintaining unaltered its precision.

3.12 Border Binaries Discovery

As stated in Section 3.5, the connection mark (i.e., $\#conn$) is used as a flag, whereas the network mark (i.e., $\#net$) is used as a counter. We made this decision as we found

that, in practice, calculating the connection mark feature is computationally harder than calculating the network mark.

To calculate the network mark, we need to retrieve all the memory comparisons within a binary and consider those that might refer to hard-coded network-related strings. We found that finding memory comparisons that refer to these type of strings is computationally easy, as, in practice, the addresses of these strings are referred within the basic block containing the call to the memory comparison itself.

On the other hand, to set the connection mark, we have to determine whether any data read from a network socket (i.e., the source) is passed to a *memcmp-like* function (i.e., the sink). This would involve enumerating all the possible program paths between two arbitrary program points (i.e., a read from a socket and a call to a *memcmp-like* function), which is, in the general case, unfeasible [18]. Also, in principle, we do not know if a binary contains more sources than sinks, and, therefore, a classic forward taint analysis from a source to a sink might incur in scalability issues [81]. Therefore, to alleviate these problems and increase the chances to find a path between a source and a sink, we leverage our static taint engine and perform a combination of both forward and backward static taint analyses. In particular, we bootstrap a forward taint analysis from each program point containing a source (e.g., a `recv`), and a backward taint analysis from each program point containing a sink (i.e., a *memcmp-like* function). Also, to keep the analyses tractable, we constrain the number of functions traversed by each analysis

to a fixed value n_f (set to 5 in our experiments), and limit the symbolic exploration to a time limit of 10 minutes.

Nonetheless, we might fail to find a path between a source and a sink due to, for instance, an unresolved indirect control-flow transfer. Therefore, if we detect any imprecision while analyzing a function f of a binary b, we consider the analysis for f to be *incomplete*. If the number of functions not completely analyzed overcomes a fixed threshold (set to 50% in our experiments), we take the conservative decision to set the connection mark. Also, as the connection mark is operating system (OS) dependent (i.e., the analysis should know the syscall number used to read data from a socket), if the OS is unknown (e.g., in case of a firmware blob) we simply set the connection mark.

Finally, the feature cmp in our Parsing Score (see Equation 3.1) represents an adaptation for binaries of the feature br_fact presented by Cojocar et al. [24], and it is calculated by incrementing its value every time we find a memory comparison operation against any string.

3.13 Communication Paradigm Finders

As stated in Section 3.6.1, KARONTE provides a set of CPFes to recognize the IPC paradigms, whose specifics depend on the OS of the firmware sample under analysis. Furthermore, to maintain our prototype OS-independent, and to make it able to reason

about inter-process communication paradigms when some information is missing (e.g., embedded Linux distributions whose binaries are stripped by their symbols or firmware blobs), we provide our prototype with a generic CPF called the *Semantic CPF*, which abstracts from the underlying OS.

Since OS-dependent CPFes work in a similar fashion, we describe the Environment CPF, as an example of OS-dependent CPF, and the OS-independent Semantic CPF.

Environment CPF. This CPF detects whether user data is shared through the operating system environment. Given a program path (i.e., a sequence of basic blocks) between two program points p_1 and p_2, the Environment CPF checks whether there exists a block bb containing marks indicating that another binary is being executed (e.g., a call to `execve`). If so, this CPF scans each basic block in the program path prior to bb, and collects every program point p_c that contains a call to a function setting (or getting) environment variables (e.g., `setenv` or `getenv`). Finally, the Environment CPF considers each function f containing p_c, and performs a reach-def analysis from f's entry point to p_c itself to determine the values of the arguments of the function called at pc (e.g., the string `QUERY_STRING` in `setenv("QUERY_STRING"))`. Finally, the Environment CPF considers these values as data keys (e.g., `QUERY_STRING`).

The binary set magnification functionality (see Section 3.6.1) infers the possible names of the binaries that are invoked in bb. To do this, we perform a reach-def analysis starting from the entry point of the function containing bb to bb itself, and we collect

the strings used as arguments in the function call in bb. Finally, if we cannot resolve the names of the binaries being executed (e.g., because they are calculated at runtime), the Environment CPF finds all the binaries within the firmware sample that rely on the data keys previously recovered. We do this by retrieving all the strings in the binaries of the firmware sample, and selecting those that have at least one of the searched data keys.

Semantic CPF. Our key observation is that any communication among different processes must rely on the concept of data keys. That is, there must be some *known* information that is used as a *reference* to set, or get, some data d for another process to be accessed. Furthermore, as explained in Section 3.3, data keys are often hard-coded in the binary itself as constant values (e.g., hard-coded strings).

The Semantic CPF leverages this intuition, and given a program path, it checks whether a constant value k is used to *index* a memory location to set (or to get) some data of interest (e.g., attacker-controlled data). If so, k is considered as a candidate data key, and the binary under analysis as a potential *setter* (or *getter*) for k. A typical example of inter-process communication detected by the Semantic CPF is given by memory-mapped I/O in embedded devices. In this setting, peripherals' input and output channels are mapped to predefined addresses in memory, which are hardcoded in the firmware components that need to access them.

Given a function f_c to analyze, this CPF applies two different approaches to infer if a data key is used as a reference (base or index) to manage data.

First, we taint each argument of f_c that points to constant data (e.g., a string in a `.ro` section of the binary), using different taint tags. Then, we examine every load (or store) in f_c to check whether tainted variables are referenced to read (or write) from a memory location m. For example, if a tainted variable is used as an address to write at location m, we consider f_c as a setter for the data key.

Second, if the first step does not yield a positive result, we check the structure of the function f_c itself. In particular, we assume that any set or get oriented functions should look for an entry point into a data structure relying on a provided key, to set, or get, some value. To achieve this, we assume that such a function contains a simple loop with a memory comparison function (e.g., *memcmp-like* functions) that has a parameter that points to tainted data. If these conditions are met, the Semantic CPF considers the function to be a set or get oriented function. To distinguish between the two, we scan the basic blocks corresponding to the true branch of the memory comparison function call and checks whether any of the f_c's arguments are set to a new value. If a new value is set, we identify the function f_c as a setter. In the case where a value is returned, we label the function as a getter.

Consider the example in Listing 3.5, which represents a snippet of code of a setter function found in one of the firmware samples in our dataset. The stack variable at offset `-32` (`R11` represents the base pointer) points to a hard-coded string (i.e., a sequence of ASCII characters null-terminated), which is, therefore, tainted by the Se-

```
1  ; .text section
2  loc_A598:
3    LDR         R0, [R11, -28]   ; destination buffer
4    LDR         R1, [R11, -32]   ; data key pointer
5    LDR         R2, [R11, -24]   ; number of bytes
6    BL          0x9554           ; call to a memcpy-like
7                                 ; function
8    LDR         R2, [R11, -28]
9    LDR         R3, [R11,-24]
10   ADD         R3, R2, R3
11   MOV         R2, 61           ; append '='
12   STRB        R2, [R3]
13   LDR         R3, [R11, -24]
14   ADD         R3, R3, 1
15   LDR         R2, [R11, -28]
16   ADD         R3, R2, R3
17   MOV         R0, R3           ; destination
18   LDR         R1, [R11, -36]   ; source (data value)
19   LDR         R2, [R11, -16]   ; number of bytes
20   BL          0x9554           ; call to a memcpy-like
21                                ; function
22   LDR         R2, [R11, -24]
23   LDR         R3, [R11, -16]
24   ADD         R3, R2, R3
25   ADD         R3, R3, 1
26   LDR         R2, [R11, -28]
27   ADD         R3, R2, R3
28   MOV         R2, 0
```

Figure 3.5: Snippet of code that uses a data key to set a data value into a local structure.

mantic CPF. Due to a function semantically equivalent to `memcpy` (Line 6), the taint gets propagated to the destination buffer (stack variable at offset `-28`). Then, after considering its length, the character "=" is appended to the destination buffer (Line 11) and a value (stack variable at offset `-36`) is appended to it through the `memcpy-like` function call (Line 20). Finally, since a hard-coded value is used as the offset (through its length) to copy *arbitrary* data into memory, the Semantic CPF considers this function as a candidate setter function. After manual verification, we found that the above example was indeed setting data to be used by another process, and that the stack variable at offset `-36` (Line 18) was the value of the data.

As an optimization, we leverage debugging and loading symbols (when available) to drive our Semantic CPF to *interesting* functions. For example, if a function name contains the keyword 'send' we mark it as a candidate set function, and consider it for further analysis.

3.14 Binary Dependency Graph Algorithm

As explained in Section 3.6, KARONTE detects if a border binary shares user-provided data by: (i) considering the set of memory comparisons retrieved by the Border Binaries Discovery algorithm, (ii) using our taint engine to taint the involved memory locations, and, (iii) performing a taint analysis on the border binary to detect whether the binary shares some tainted data. This procedure might involve enumerating all the possible program paths in the border binary, and, therefore, it might lead to the path explosion problem. Therefore, to keep the analysis tractable, we run our taint engine up a certain time limit (set to 10 minutes in our experiments). However, as some paths might be left unexplored, our prototype might miss some valid data flows between binaries, and our BDG might not contain some valid edges. Therefore, in order to increase the path coverage within a prefixed time limit, we apply the taint to *each* function of a border binary that refers to a network-encoding string. This solution might involve more false positive edges within a BDG (thus affecting its soundness),

but it decreases the likelihood of false negative edges. This heuristic gave us noticeable improvements in practice, as the program points where data is read from sockets (e.g., `recv`) might be distant (in terms of the number of instructions in an execution trace) to those where such data is shared (e.g., `setenv`). However, as network-encoding strings might be used for other purposes within a binary (e.g., as data keys), we are able to alleviate this problem by considering as a source of taint every function that refers to network-encoding strings.

3.15 Static Taint Analysis

Our taint engine mainly introduces two contributions: (i) taint tag dependencies, and (ii) a path prioritization strategy.

To add taint tag dependencies, we enhanced the angr's symbolic state module with an additional data structure that maps each taint tag to its dependencies. When a symbolic expression e has to be untainted, we retrieve its taint tag t_e, and all the taint tags t_{dep} that depend on t_e. Then, we consider each taint tag t_d in t_{dep}, and check whether it depends on any other taint tag other than t_e. If not, we remove the taint tag t_d (thus untainting the tainted symbolic expressions represented by t_d). Finally, we remove t_e, thus effectively untainting e. Note that, taint tag dependencies based on memory comparisons (as explained in Section 3.7) are created automatically.

Our path prioritization strategy aims to prioritize those paths within a function that *potentially* return tainted variables. Given a function f to symbolically explore, we build its control flow graph (CFG), and we retrieve all the *exiting* basic blocks, that is, those containing a return statement r. For each of these basic blocks, we perform a static reach-def analysis from f's entry point up to r, and collect all the possible returning values. We then prioritize those paths that *do not* always return constant values.

3.16 Multi-binary Data-flow Analysis

The cornerstone of the multi-binary data-flow analysis module is to estimate the size of the buffers used to send (or receive) attacker-controlled data. Our prototype provides two sub-modules for this task: the *stack-size finder* and the *heap-size finder* to detect the size of buffers allocated on stack and heap, respectively.

Given a function fs and a buffer b allocated at offset bs on fs stack, the stack-size finder scans fs body, and collects the offsets of the variables allocated on fs stack. Then, this sub-module sorts the stack offsets in ascending order, and it picks the offset bz right after bs (remember that the stack grows downward). Finally, the stack-size finder considers the buffer b as big as $|bz - bs|$.

On the other hand, given the address bh of a heap-allocated buffer b, and a function fh allocating b, the heap-size finder leverages our static taint engine to taint bh, and bootstraps a symbolic path exploration from fh's entry point. For each basic block encountered during the symbolic path traversal, this sub-module detects whether the basic block contains a call to a heap allocation function fa (e.g., `malloc`). If any of fa's arguments is tainted, or fa's returning value gets assigned to a tainted memory location, the heap-size finder considers the call to fa for further inspection. In particular, it considers the symbolic expression of the fa's argument that represents the allocated size (e.g., the first argument in `malloc`), and leverages the z3 [3] theorem solver to concretize its value, thus retrieving the buffer b's allocated size. If the symbolic expression can be concretized to multiple values, we conservatively consider the greatest value.

3.17 Vulnerability Example

We provide the details of one of the vulnerabilities discovered by KARONTE [4] for the D-Link 880 firmware sample. This firmware is used on the D-Link Wireless AC1900 WiFi Gigabit routers, and it is composed of 129 different binaries executing on a Linux-based filesystem.

[3] https://github.com/Z3Prover/z3
[4] CVE-2017-14948

```
1  void add_data_key(e, key, data) {
2    int nk = strlen(key);
3    int nd = strlen(data);
4    char* tmp = (char*) malloc(nk + nd + 3)
5    e->vars = realloc(e->vars, e->size + nk + nd + 3);
6    memcpy(tmp, key, nk);
7    tmp[nk] = "=";
8    memcpy(tmp[nk + 1], data, nd);
9    e->vars[e->n_vars] = tmp;
10   e->n_vars ++;
11   // ...
12 }
13
14 int do_serve(r){
15   env_struct* e;
16   add_data_key(e, "CONTENT_TYPE", r->content_type);
17   // ...
18   exec_bin(e, "fileaccess.cgi");
19 }
20
21 void parse_req(char* raw_data, usr_req* r){
22   while (raw_data && *raw_data) {
23     char *s = get_next_field(raw_data);
24     // ...
25     if ( !strncmp(s, "Content-Type", 12) ) {
26       // set content type info in r
27     }
28     // ...
29   }
30 }
31
32 void serve_request() {
33   usr_req* r;
34   char* raw_data;
35   raw_data = get_req_socket();
36   parse_req(raw_data, r);
37   do_serve(r);
38 }
```

Figure 3.6: Decompiled snippet of code of `httpd`.

Two of the binaries involved in handling user's requests are the binary `httpd` and a binary called `fileaccess.cgi`. The former receives user's data from the network, whereas the latter uses such data to perform file operations.

A simplified code of `httpd` is shown in Listing 3.6.

First, `httpd` calls the function `get_req_socket` (Line 35) to receive user requests from the network, and stores them in the `raw_data` variable.

```
1  void get_content_type(char* dst) {
2    const char *haystack = getenv("CONTENT_TYPE");
3    char *haystacka;
4    haystacka = strstr(haystack, "boundary=");
5    // ...
6    strcpy(dest, haystacka + 9); // buffer overflow
7  }
8
9  int uploadfile_handler() {
10   char buff[256];
11   get_content_type(buf);
12   // ...
13 }
```

Figure 3.7: Decompiled snippet of code of `fileaccess.cgi`.

The content of the request is parsed by the function `parse_req` (Line 36), which also properly sets an internal data structure `r` (Line 26). Note that, the memory comparison contained in function `parse_req` (Line 25) refers to attacker-controlled data. This memory comparison is returned by our Border Binary Discover module (Section 3.5).

Then, `httpd` calls the function `do_serve` (Line 37), which prepares the execution environment for `fileaccess.cgi` and executes it. In particular, `do_serve` (Line 14) uses the function `add_data_key` (Line 16) to set the local variable `e` with attacker-controlled data. Note that, `add_data_key` (Line 1) does not impose any constraints on the size of the attacker-controlled data: it allocates a buffer `tmp` (Line 4) to accommodate arbitrarily long data. In our prototype, the function `add_data_key` was recognized by our Semantic CPF to be a setter for `httpd`.

Finally, the binary `fileaccess.cgi` is executed (trough `exec_bin`), and the variable `e` is used as its execution environment.

When `fileaccess.cgi` is executed (Listing 3.7), if the user's request involves uploading a file, the function `uploadfile_handler` is executed (Line 9). This function allocates a buffer of 256 bytes on the stack (Line 10), and then calls the function `get_content_type` to retrieve the content type of the user's request (at Line 1).

Unfortunately, this function contains a bug. In fact, if the variable `haystack` (which points to the environment variable identified by the data key `CONTENT_TYPE`) contains the string `"boundary="` followed by at least 257 characters, the `strcpy` function call (Line 6) will provoke a buffer overflow. KARONTE automatically identified this bug, and we reported it to D-Link, which promptly fixed the issue.

3.18 Discussion

In this Section, we discuss some key points of our system.

As with any other path-based exploration analyses, KARONTE suffers from the path explosion problem. In our prototype, we limit path explosion, while increasing precision, by: (i) providing precise taint propagation policies (e.g., function calls with no tainted arguments are not always followed, depending on call-stack depth), (ii) using timeouts (each symbolic path exploration is performed up to a certain time limit),

(iii) limiting loop iterations, and (iv) automatically creating function summaries (as explained in Section 3.10).

Our prototype may generate both false positives and false negatives. They are due to the fact that taint information might not be correctly propagated to unfollowed paths (e.g., due to time, call-stack depth, or loop constraints), or imprecisions of the underlying static analysis tool (i.e., angr), as shown in Section 3.19. This might result in incomplete BDGs, and, therefore, some security vulnerabilities might be left undiscovered. However, KARONTE alleviates this problem by generating taint tag dependencies (see Section 3.7).

Though by default, KARONTE finds buffer overflows and denial-of-service vulnerabilities, its design allows an analyst to support different types of vulnerabilities. The Insecure Interactions Detection algorithm (Section 3.9) relies on a set of detection modules designed to use taint information to recognize specific classes of vulnerability. For instance, an analyst can extend our system to find use-after-free bugs by providing a new detection module, such as [38].

3.19 Evaluation

In this Section, we first evaluate each phase of KARONTE's algorithm on several of the latest firmware samples available at the time of writing. Then, we evaluate

Table 3.1: Results on our dataset of current-version firmware samples. For each vendor we report the device series, the number of firmware samples, and those samples whose network services are handled by one and multiple binaries, respectively, the total number of binaries, the average number of border binaries, the number of alerts our prototype generated, the average execution time, the number of true positives, and the number of bugs retrieved by tracking the data-flow through one or more binaries.

Vendor	Device Series	# Firmware Samples	# Single Binary	# Multi Binaries	# Binaries	Avg # Border Binaries	#Alerts	Avg Time [hh:mm:ss]	# Bugs	# Single Binary Vulnerabilities	# Multi-binary Vulnerabilities
NETGEAR	R/XR/WNR	17	12	5	4,773	7	36	17:13:45	23	10	13
D-Link	DIR/DWR/DCS	9	4	5	1,290	5	24	14:09:12	15	0	15
TP-Link	TD/WA/WR/TX/KC	16	16	0	1,769	5	2	1:30:16	2	2	0
Tenda	AC/WH/FH	7	4	3	734	5	12	1:01:22	6	0	6
Huawei	ALE-L23	1	1	0	1	0	6	4:04:37	4	4	0
Nvidia	Nexus 9	1	1	0	1	0	0	0:25:01	0	0	0
Qualcomm	-	1	0	1†	1	0	0	2:28:27	0	0	0
Qualcomm*	-	1	0	1†	1	0	7	5:03:32	1	1	0
Total	-	**53**	**38**	**15**	**8,565**	**279**	**87**	**49:09**	**51**	**17**	**34**

†: The firmware sample was manually separated into distinct components.

KARONTE's performance using a dataset from related work [20]. We implemented a prototype of KARONTE on top of angr [116], and, in particular, our taint engine on top of BOOTSTOMP (see Chapter 2).

3.19.1 Datasets

We evaluated our prototype of KARONTE on both Linux-based firmware samples and firmware blobs.

Recent Linux-based Firmware. We selected four major IoT vendors that make the firmware of their devices available for download: NETGEAR, TP-Link, D-Link, and Tenda. Then, we scraped their official websites to collect the available firmware, for a total of 112 different products. Unfortunately, several firmware samples were not avail-

able for download or packaged with proprietary algorithms. We eventually successfully collected 49 different firmware samples.

Firmware Blobs. We used the BOOTSTOMP dataset, which provides us with the ground truth for our approach. BOOTSTOMP's dataset is composed of 5 firmware samples. In particular, it contains two versions of Qualcomm's Little Kernel (or LK): the most recent at the time of publication, and a version (not specified) that was released before 2016-07-05 that contains a known vulnerability. Throughout this work, we refer to the latter with a *. Also, as these firmware blobs receive data from persistent storage (rather than from the network), we modified our Border Binaries Discovery module to accommodate BOOTSTOMP's approach to identifying procedures that read from or write to the hard drive. Finally, we did not consider the Mediatek bootloader because angr fails to analyze it, as explained in Chapter 2.

Table 3.1 shows our dataset of 53 firmware images (the combination of the Linux-based and firmware blobs datasets).

Large-scale Dataset. To measure the scalability of KARONTE, we obtained Firmadyne's dataset [20], and considered the firmware samples whose architecture is supported by BOOTSTOMP (i.e., ARM, AARCH64, and PowerPC). We did not consider firmware samples for MIPS architectures, as angr only partially supports MIPS binaries, and some of its analyses might yield imprecise results in these cases (as explained in Section 3.19.3). This limitation is introduced by the employed tool, and not by

Table 3.2: Comparative Evaluation. Number of alerts generated for each step of KARONTE. For each vendor, we report the average values.

	ALL†			PARSERS			BDG			KARONTE		
Vendor	**Ana. Bin‡**	**No. Alerts**	**Time**	**No. Bin**	**No. Alerts**	**Time**	**No. Bin**	**No. Alerts**	**Time**	**No. Bin**	**No. Alerts**	**Time**
NETGEAR	71	729	7 days	7	312	13:53 h	8	443	25:31 h	8	2	17:13 h
D-Link	80	811	7 days	5	205	12:00 h	6	294	14:33 h	6	3	14:09 h
TP-Link	181	819	7 days	5	71	7:44 h	5	86	6:37 h	5	0	1:30 h
Tenda	41	474	7 days	5	154	10:41 h	6	175	11:07 h	6	2	1:01 h
Total	**2,424**	**20,931**	**28 days**	**279**	**9,363**	**44:18 h**	**312**	**12,778**	**48:57 h**	**312**	**74**	**33:57 h**

†: Experiment conducted up to 7 days.

our approach, which is architecture-independent. Overall, this dataset consists of 899 firmware samples from 21 different vendors (Table 3.3).

3.19.2 Border Binaries Discovery

First, we established the optimal values for k_n and k_c. We randomly selected one firmware sample and manually investigated its border binaries. We identified three binaries. Then, we ran the Binary Border Discovery module against the firmware sample using different values for k_n and k_c (ranging from 1 and 10). For $k_n \geq 5$ and $k_c \geq 1$ we correctly identified the three binaries as border binaries. Therefore, we set k_n and k_c to 5 and 1 respectively.

Next, we measured the effectiveness of the Border Binaries Discovery module to identify network parsers. We randomly picked 10 firmware samples, investigated their network-facing binaries and randomly selected 150 more binaries. Then, we ran the

Border Binaries Discovery module against all of these binaries three times: (i) considering only the features described in [24], (ii) considering also the $\#net$ feature, and, (iii) considering also the $\#conn$ feature. In the first case (i), this module identified 50 binaries containing parsers. However, after manual investigation, we concluded that only 16 of them handled data received from the network. In the second case (ii), our tool identified 51 binaries, and we found that 26 of them contained network parsers that are affected by user input. Finally, in the third experiment (iii), this module identified 50 binaries, and we verified that 26 of them contained network parsers affected by user input. One of the 51 binaries identified during experiment two (ii) was not detected as a network parser in experiment three (iii). We found that, indeed, it does not implement any network functionality. Finally, we found that our Border Binaries Discovery module's algorithm missed a real network parser. This false negative was due to the fact that angr failed to identify any strings, as the binary retrieved them by computing their addresses at runtime as offsets from the Global Offset Table (GOT), thus affecting the binary parsing score.

3.19.3 Binary Dependency Graph

We manually checked the soundness and completeness of the recovered BDGs. In all of the 53 cases, to the best of our knowledge, the BDGs were sound: every edge in the BDG corresponded to an existing data dependency between the involved bina-

ries. Then, we checked if any edge was missing. Out of 53 BDGs, we found that, for the three Tenda firmware samples, the BDG algorithm failed to connect an edge between two binaries, as a valid network-facing binary was missing (as explained in Section 3.19.2). However, our Semantic CPF correctly identified the binaries receiving data from the missing network-facing binary as getters. Furthermore, the BDG of 14 TP-Link firmware samples did not contain any edges, as angr failed to resolve several data attributes referenced within these firmware samples during the Border Binaries Discovery phase. We discovered that these firmware samples ran on a MIPS architecture, which is unfortunately poorly supported by angr.

We manually investigated all the matching CPFs, and we found that the Semantic and the Environment CPFes matched 11 and 32 times respectively, whereas the remaining CPFes did not identify any active IPC communication. After manual investigation, we concluded that these results were indeed correct.

3.19.4 Insecure Interactions Detection

Each alert produced by our prototype consists of an insecure data flow (e.g., a flow reaching an unsafe memcpy-like function), and we distinguish true positives from false positives according to the type of data reaching the sinks of the data flows. If the data is provided by the user (e.g., HTTP headers), we consider the alert a true positive bug (if the bug can be exploited, it is denoted as a security vulnerability). On the other hand,

if the data is not user-provided (e.g., the data is represented by filesystem file names), we consider the alert a false positive.

Our prototype produced 87 alerts, among which 51 were true positives (34 multi-binary bugs and 17 single-binary bugs), for a total of 8,565 considered binaries (Table 3.1). We manually verified each alert by reverse-engineering the involved binaries and inspecting the highlighted data flows. We reported all our findings to the appropriate manufacturers (responsible disclosure).

We also verified how many of these 51 bugs were security vulnerabilities. We acquired two of the devices and successfully crafted PoCs for three of the vulnerabilities, and obtained one CVE and one PSV[5]. Two other alerts were non-exploitable bugs: though user data reached a sensible program point, we were not able to achieve control-flow redirection. Five more vulnerabilities were confirmed by my work on bootloaders (see Chapter 2). For the remaining vulnerabilities, we relied on manufacturers' collaboration, since we could not obtain all of the devices for the firmware in our dataset without incurring in excessive expenses, and confirmed nine more vulnerabilities. Sadly, some of the manufacturers were uncooperative and refused to consider reports without a proof-of-crash (PoC) on the physical device. Therefore, we assessed the remaining vulnerabilities by reverse engineering the firmware. By using vendor-confirmed vulnerabilities and checking whether other firmware using the same codebase (infor-

[5]CVE-2017-14948, PSV-2017-3121

mation gathered from vendors) had similar bugs, we were able to confirm another 20. The remaining 12 were statically investigated for exploitability, and we believe that all of them are exploitable. Overall, we verified every alert, and 46 of the detected bugs, to the best of our knowledge, were not publicly known before KARONTE. The 12 confirmed vulnerabilities are being fixed as well as those bugs affecting samples sharing similar codebases (at least an additional 20).

To evaluate the false negative rate of our prototype, we searched for CVEs involving our dataset, and collected information for 30 different bugs. Since 21 of these bugs belonged to the binary that angr failed to analyze (Section 3.19.2), we manually added this binary to the BDG and annotate the functions referencing network-encoding keywords, and re-ran our analysis. KARONTE re-discovered all of these bugs. Overall, our prototype generated two false negatives belonging to the Nvidia and Huawei firmware, respectively. In these cases, we failed to introduce the initial taint, as angr failed to resolve two indirect control-flow transfers.

3.19.5 Comparative Evaluation

To evaluate the importance of every step of KARONTE, we compared the effort required by an analyst to verify the results generated by different approaches. To do this, we considered the 49 firmware samples containing multiple binaries, and selected those 29 samples whose architecture is fully supported by angr.

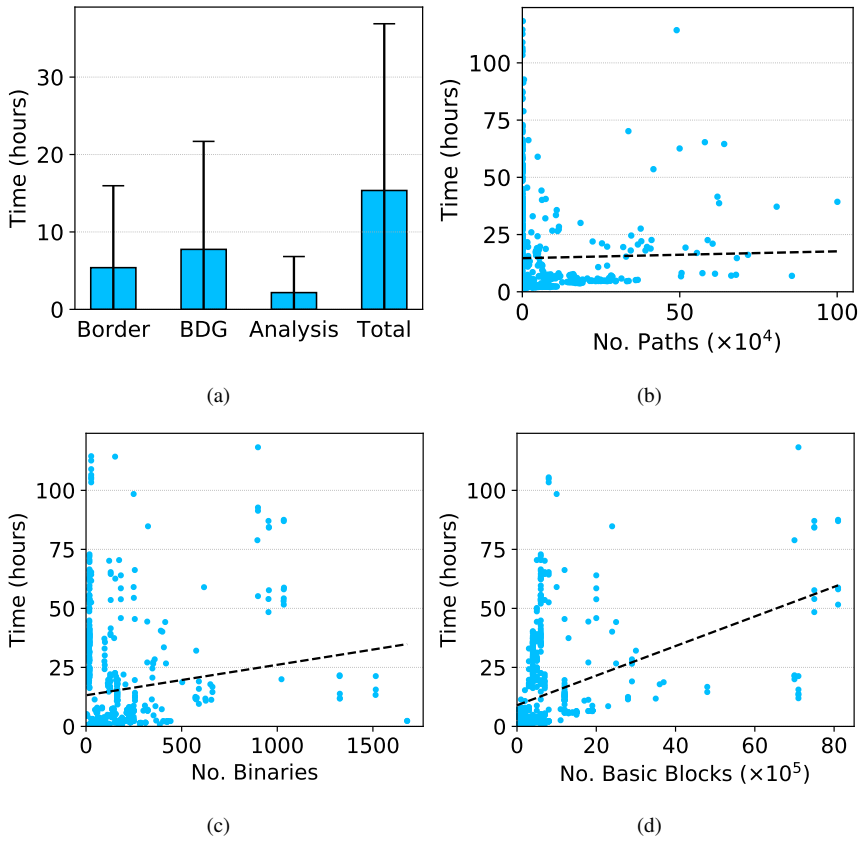


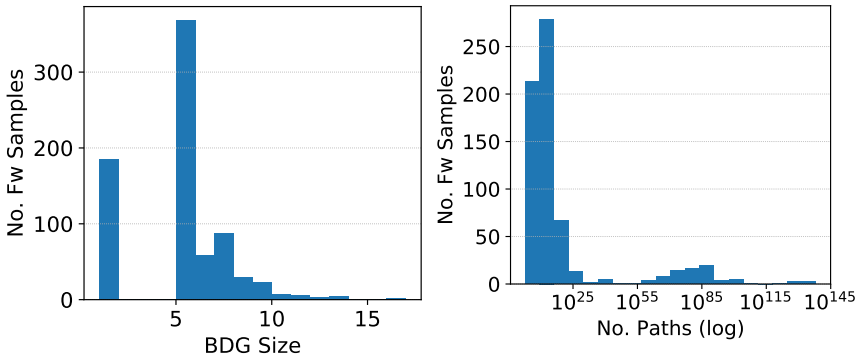
Figure 3.8: (a) Average and standard deviation of the execution time of each step of KARONTE. Analysis time includes BFG Recovery and Insecure Interaction Detection. (b) Dependency between execution time and the number of explored paths. (c) Dependency between execution time and the number of binaries in the firmware samples. (d) Dependency between execution time and the number of basic blocks in the firmware samples. The dashed lines represent the linear regressions.

We then compared four different approaches. First, we performed a static single-binary bug search using our static taint engine on every binary contained in each firmware sample (dataset **ALL**). Second, we ran our static taint engine on the border binaries of the firmware sample (dataset **PARSERS**). Third, we run the BDG algorithm on each firmware sample, and we applied our static taint engine to the binaries that handle user-provided data *without* propagating the data constraints (dataset **BDG**). Finally, we

considered our full approach (dataset **KARONTE**). During this evaluation, we made the realistic assumption that without propagating user input from network-facing binaries, the security analyst has no prior knowledge of where, or if, the user input is introduced in a given binary. Therefore, we considered every IPC channel as a possible source of input.

As clearly shown by our results depicted in Table 3.2, the number of generated alerts decreased to a manageable number (i.e., 20,931 to 74) only when applying the full KARONTE approach. We manually investigated 50 randomly picked alerts selected from those generated by the single-binary analysis experiments, which were effectively filtered out by the full KARONTE approach. All of them were false positives. In fact, in all of these cases, the binaries causing the alerts were spawned (e.g., through the `system` function call) only using hard-coded arguments and parameters, thus not being affected by the user input. On average, KARONTE uncovered 2 vulnerabilities per sample not discovered when only network-facing binaries were considered (**PARSERS**), which highlights the importance of considering all the binaries handling attacker-controlled data.

Throughout our experiment, our expert program analyst averaged 7 minutes per investigation of alert. Based on this, we estimate that the investigation of alerts stemming from a single-binary analysis of a NETGEAR firmware sample, for instance, would require approximately 138 hours. KARONTE decreases this time to 14 minutes.



(a) (b)

Figure 3.9: (a) Distribution of the sizes of the BDGs of our firmware samples. (b) Distribution (in log scale) of the estimated total number of paths in an average binary in the BDG. For graphical reasons, this figure shows 95% of our data.

3.19.6 Large-scale Scalability Assessment

We assessed KARONTE's performance and scalability by analyzing 899 firmware samples from Firmadyne dataset (all samples using architectures supported by KARONTE). We ran this evaluation on a cluster of machines equipped with Intel Xeon E5 CPU, 16 to 32 GB of RAM, and running Ubuntu 18.04.

Firmware Complexity. We investigated the complexity of the firmware samples in our dataset using three metrics: number of binaries, number of basic blocks, and number of paths present in the binaries handling user input (i.e., those in the BDG). In particular, we leveraged Bang et al.'s work [9] to calculate an upper bound on the number of paths of a program. To do this, Bang's approach requires us to retrieve the program's longest path, which is an NP-hard problem [121]. To overcome this issue, we approximated the longest path of a binary by performing a symbolic exploration for 10 minutes (while

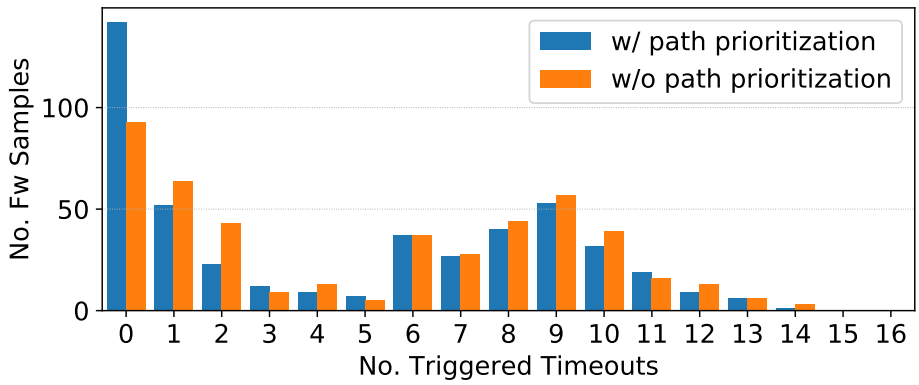


Figure 3.10: Distribution of the number of timeouts triggered during the symbolic exploration with and without our path prioritization.

limiting the maximum number of iterations of a loop to five), and recording the longest visited path.

Table 3.3 shows that, on average, a firmware sample contains around 157 binaries, for a total of $7.85 * 10^5$ basic blocks. Furthermore, 82% of the binaries in the BDGs contain less than 10^{25} paths, as shown in Figure 3.9.b. Interestingly, our dataset includes some far more complex firmware samples. Around 2% of them contain more than 1000 binaries (for a total of more than $7.15 * 10^6$ basic blocks), and those handling user input can reach a number of paths on the order of 10^{306}.

Overall, our dataset is composed of a collection of firmware samples with a wide range of complexity, thus making it suitable for studying the performance of our tool.

BDG. We investigated the BDGs of our dataset, and found that 38.7% of the firmware samples implement network-related services through the use of multiple binaries (#Multi-Binary column in Table 3.3). Their BDGs contain, on average, 5 binaries, among which

3 are border binaries. Most of BDGs are comprised of 5 or 6 binaries, though some samples have BDGs composed of more than 10 binaries, and one BDG contains 16 binaries (Figure 3.9.a). For 6 vendors our tool did not identify any firmware sample sharing user data among multiple binaries. We randomly picked 5 of these 18 firmware samples for manual investigation. In three cases, the network functionality was indeed performed by single binaries, not communicating with each other. In two cases, the Border Binary Discovery phase failed to find one border binary, as we could not statically resolve its strings (Section 3.19.2). However, the firmware samples were relying on a single program to implement the network functionality of the device.

On average, a BDG connected subgraph contains 4 nodes (i.e., four binaries communicating), and has a depth of 1 (i.e., a binary shares data with other 3 binaries). However, our dataset presented more complex cases. For instance, the BDG composed of 16 different binaries had 4 different connected subgraphs, and the biggest subgraph had a depth of 2 and contained 7 binaries. In this case, we found that a border binary exchanged data with 6 other binaries, and one of them modified the data and shared it further. Finally, there were a few cases where both the cardinality of a BDG connected subgraph and its depth were 1 (e.g., Belkin). In these cases, we found that a border binary was using IPC to exchange data with itself.

Overall, the results are in line with those discussed in Section 3.19.3, and show that firmware samples are made of highly interconnected components, whose interactions can be fairly complex, highlighting the importance of approaches like KARONTE.

Performance. We measured the time required by each phase of KARONTE, and the total analysis time. Our prototype fully analyzed 80% of the firmware samples within a day, and, on average, it completed each phase within 8 hours (Figure 3.8.a).

As we can see, the Border Binaries Discovery and BDG Recovery phases presented a great variance. We discovered that the time increase in the Border Binary Discovery phase was caused by the Z3 theorem solver, which sometimes required several minutes to solve a single symbolic expression and is heavily utilized by angr (some CFGs took 8 hours to be built). Time increases in the Binary Dependency Graph phase were also due to slow z3 solves, and, in a few cases, to an unusually high number of data keys. The time spent to build a BDG depends on the number of analyzed paths, which, in turn, depends on the number of data keys found in a binary. Some border binaries (around 7%) contain more than 50 data keys, which we analyzed to detect whether the binary is a setter or a getter. Since we perform each of these analyses up to a certain time limit (10 minutes in our experiments), the BDG phase might take several hours to analyze a single binary (around 8 hours for 50 data keys).

Figure 3.8.b depicts how the number of analyzed paths influences the total analysis time. Most samples that took longer to be fully analyzed are those for which we ex-

plored a small number of paths. These samples are those that caused angr to take a long time to generate the CFGs.

Finally, we found that the number of binaries and their size (in terms of the number of basic blocks) in a firmware sample do not significantly impact on the performance of our tool. In fact, 67% of the firmware samples that we analyzed for more than a day contained a number of binaries less than or equal to 27 (for a total number of basic blocks less than or equal to $7.64 * 10^5$), whereas far more complex firmware samples were analyzed faster, as shown in Figure 3.8.c and Figure 3.8.d.

Overall, KARONTE scales well with the firmware complexity, in terms of the number of binaries, basic blocks, and paths.

Symbolic Exploration. We studied the impact of our path prioritization strategy and untaint policies on our results. First, we ran our prototype on the KARONTE dataset with and without the path prioritization strategy and compared the number of times that a timeout (set to 10 minutes) triggered during the analysis (note that no timeout means all paths carrying tainted data have been exhausted). Figure 3.10 depicts the distribution of the number of timeouts triggered during the analysis of the samples in our dataset. Indeed, the number of firmware samples fully analyzed without any timeout is higher when the path prioritization is enabled. Specifically, considering the total number of times we ran our taint engine, we explored every tainted path 84% of the times when the path prioritization was enabled, and 75% of the times when it was

disabled. This corresponded to around $2 * 10^6$ paths being pruned away. On average, KARONTE explored around $15 * 10^3$ paths per firmware sample (Table 3.3). Though the average number of estimated paths is significantly higher, it is important to remind that KARONTE aims to find and analyze only those paths affected by user input.

Then, we ran our tool with and without untaint policies and compared the number of generated alerts. Overall, the number of alerts generated when the untaint policies were applied decreased by 2.5%. We manually inspected all of them and found them to be, indeed, false positives. In these cases, a buffer was safely copied using unsafe functions (e.g., using `strcpy` after checking their size through `strlen`).

Alerts & Vulnerabilities. On average, KARONTE generated 2 alerts per sample, for a total of 1,037 alerts. We sampled 100 alerts for inspection and found 44 to be true positive (i.e., user-provided data reached a sink), and 30 of them to be multi-binary vulnerabilities. This means that, in almost one case out of two, KARONTE is able to detect critical data flows that require immediate attention, and that often involve multiple binaries. We reported our findings to the respective vendors.

Firmadyne raised *zero* alerts for the large-scale dataset. Though we cannot be certain about *why* Firmadyne did not find bugs, we speculate that this emphasizes one of the advantages of a static approach over a dynamic one: though KARONTE makes certain trade-offs, it analyzes complex firmware without emulating it or tackling the dynamic coverage problem.

3.19.7 Verifiability

To promote reproducible research, we asked an independent researcher from Northeastern University to replicate our results shown in Table 3.1 (excluding the columns bugs and vulnerabilities, as they would have needed to contact the manufacturers, but including generated alerts). The large-scale evaluation and Table 3.2 were not replicated, due to the prohibitive cost of the required computational power.

Also, we created a Docker container with our tool and running environment (e.g., KARONTE's dataset). Along with this container, we provided the researcher with the source code of our tool, a copy of this paper, the necessary documentation explaining the purpose of each component in our tool, and our expected results. Finally, we instructed them on how to run our tool. The independent researcher was successfully able to obtain all of the results presented in Table 3.1.

Table 3.3: Dataset for large-scale evaluation. In order: vendor's name, number of firmware samples, number of firmware samples whose network services are handled by multiple binaries (percentage), number of binaries in the firmware samples, number of border binaries, number of binaries in the BDG, cardinality of a subgraph in the BDG, maximum depth of a subgraph in the BDG, number of basic blocks in the firmware sample, number of paths in binaries handling user input, execution time, and generated alerts.

Vendor	# Firmware Samples	# Multi Binary (%)	# Binaries†	# Border Binaries†	BDG Size†	Subgraph Cadinality‡	Subgraph Depth‡	# Basic Blocks†	# Paths†	Explored Paths	Time† [hh:mm:ss]†	# Alerts†
Airlink101	1	1 (100.0%)	94	5	8	4	1	9×10^{04}	1×10^{05}	68.58K	3:55:44	13
Belkin	6	1 (16.7%)	184	5	5	1	1	2×10^{05}	3×10^{81}	4.12K	0:49:46	1
Buffalo	3	0 (0.0%)	301	5	5	0	0	2×10^{06}	3×10^{14}	43.00	0:17:01	0
Cisco	21	6 (28.6%)	142	5	5	3	1	4×10^{05}	2×10^{22}	173.27K	5:36:15	4
D-Link	306	196 (64.1%)	103	3	3	1	1	7×10^{05}	3×10^{30}	41.64K	21:51:27	1
Foscam	5	5 (100.0%)	115	5	6	4	2	4×10^{05}	5×10^{15}	52.20K	18:01:00	7
Inmarsat	2	0 (0.0%)	640	5	5	0	0	2×10^{06}	9×10^{03}	3.10K	11:05:06	0
Linksys	12	1 (8.3%)	404	5	6	11	1	8×10^{05}	2×10^{305}	23.20K	3:32:36	1
NETGEAR	304	52 (17.1%)	115	5	5	3	1	5×10^{05}	4×10^{107}	82.83K	3:54:00	1
OpenWrt	12	1 (8.3%)	14	1	1	4	2	3×10^{04}	4×10^{15}	24.41K	1:06:16	0
Polycom	7	0 (0.0%)	130	4	4	0	0	1×10^{06}	2×10^{12}	1.01M	31:49:22	8
Supermicro	26	3 (11.5%)	209	5	5	2	1	4×10^{05}	2×10^{148}	12.16K	1:54:03	5
Synology	44	28 (63.6%)	679	3	3	1	1	5×10^{06}	1×10^{14}	4.55K	33:12:01	1
TP-Link	3	0 (0.0%)	200	5	5	0	0	7×10^{05}	1×10^{12}	2.00K	2:53:15	1
TRENDnet	55	26 (47.3%)	156	3	4	2	1	6×10^{05}	2×10^{118}	14.52K	22:59:12	1
Tenda	4	1 (25.0%)	332	5	5	1	1	6×10^{05}	2×10^{13}	13.04K	5:39:25	1
Tomato	51	11 (21.6%)	223	5	5	4	1	7×10^{05}	1×10^{26}	90.36K	9:40:55	6
Ubiquiti	15	7 (46.7%)	68	3	4	1	1	1×10^{05}	3×10^{08}	11.61K	3:06:21	2
Verizon	1	0 (0.0%)	10	5	5	0	0	1×10^{05}	5×10^{20}	2.49K	0:19:02	1
Zyxel	19	9 (47.4%)	153	5	6	3	1	3×10^{05}	4×10^{16}	260.87K	4:46:38	3
forceWare	2	0 (0.0%)	173	5	5	0	0	2×10^{05}	2×10^{03}	3.00	0:30:18	0
Total	**899**	**348 (38.7%)**	**140.82K**	**3.60K**	**-**	**-**	**-**	**16.43M**	**-**	**60.68M**	**11830:28:37**	**1.03K**

†: Averages considering all of the vendor's firmware samples.

‡: Averages considering the firmware samples whose network services are handled by multiple binaries (multi-binary samples).

Chapter 4

BINTRIMMER: Towards Static Binary Debloating Through Abstract Interpretation

In the previous Chapters, we studied how to find bugs in the different components of a firmware image. In this Chapter, we study how to improve the security of a program by safely reducing the attack surface that an attacker could use to harm users. This process is also known as *program debloating*. Note that, in this work, we assume that the source code of the target program is not available, as often happens with firmware for IoT devices.

Oftentimes, software developers rely on ready-to-use *third-party libraries* to implement complex software functionality (e.g., hardware abstraction layers to interact with a system-on-a-chip). However, these libraries might contain code that is not necessary (and therefore not referenced) by the main application, which makes the final program *bloated* with unused code. Other than making a program unnecessarily big, unused

code is potentially dangerous as it increases the *surface* (i.e., the amount of code) an attacker can use to exploit the application (e.g., using ROP chains), and perform, for instance, control-flow hijack attacks. The process of decreasing the attack surface of a program by inhibiting the execution of its dead code is called *program debloating*.

In this Chapter, I propose a novel abstract domain, called Signedness-Agnostic Strided Interval, which we use as the cornerstone to design a novel and sound static technique, based on abstract interpretation, to reliably perform program debloating. Throughout this Chapter, I detail the specifics of our approach and show its effectiveness and usefulness by implementing it in a tool, called BINTRIMMER, to perform static program debloating on binaries. Our evaluation shows that BINTRIMMER can remove up to 65.6% of a library's code and that our domain is, on average, 98% more precise than the related work.

4.1 Background and Motivation

Value range analysis [117] is a particular type of data-flow analysis that tracks the range of values that a numeric entity (e.g., a program variable) might assume at any point of a program's execution. These analyses are built on top of abstract domains [29, 30, 48], and can be utilized to guide the recovery of a program's CFG by: (i) helping

```
1  void main() {
2    uint8_t opt;
3    void (*f_ptr)(void) = [foo, bar, baz]; // foo, bar, and baz are
4                                           // defined in another module
5    scanf("%"SCNu8, &opt);
6    opt = (opt * 2) + 1;
7    // ...
8    if (opt == 0) {
9      f_ptr[0](); // call to foo
10    } else if(op == 100){
11     f_ptr[1](); // call to bar
12    } else if (opt > 127) {
13     f_ptr[2](); // cal to baz
14    }
15 }
```

Figure 4.1: Precisely determining variable values is crucial to recover the ideal CFG.

to determine control dependencies between programs statements, and (ii) resolving the targets of indirect control-flow transfers.

Consider for instance Code 4.1. A sound and precise value range analysis would determine that: (i) The variable `opt` can only assume odd values, and (ii) the function pointer f_ptr can point to the functions `foo`, `bar`, and `baz`. A CFG recovery algorithm employing this range analysis would leverage these two pieces of information to retrieve a complete and sound CFG. Precisely, the algorithm would determine that the `if` conditions at Line 8 and Line 10 are never satisfied, and, therefore, that the functions `foo` and `bar` are dead code and they should not appear in the program's CFG.

To recover a complete and (possibly) sound CFG, the CFG recovery algorithm should rely on a sound and precise range analysis. To produce sound results, range analyses must be able to reason about the signedness of program variables. Considering the example in Code 4.1, if a given range analysis a_s assumes incorrectly that the variable `opt` is signed, it would determine that `opt` cannot assume values higher

than 127, and, therefore, the `if` condition at Line 12 would be considered unsatisfied under any execution of the program. While the source code of programs written with strong-typed languages (e.g., C/C++) explicitly state a variable signedness, determining such information in binaries is a hard problem [17, 79]. In these cases, the solution is to consider each variable as *both* signed and unsigned, that is, to make the domain of each variable in a program *signedness-agnostic*. The first step in this direction has been taken by Navas et al. [89], who proposed an abstract domain called *Wrapped Intervals* (WI), which represents both signed and unsigned numeric values. Albeit sound, Wrapped Intervals produce too imprecise results to be applicable in practice. In fact, in this domain, a variable can assume *any* of the values within a range, whereas, in practice, only some of the values might be assumed during any execution of the program. This imprecision might impact the soundness of a CFG. Consider Code 4.1, and assume that the range analysis employed by the CFG recovery algorithm determines that the variable `opt` can assume every value between 1 and 255. In this case, the CFG recovery algorithm would mistakenly establish that the `if` condition at Line 10 can be satisfied, and, therefore, that the function `bar` should be included in the program's CFG.

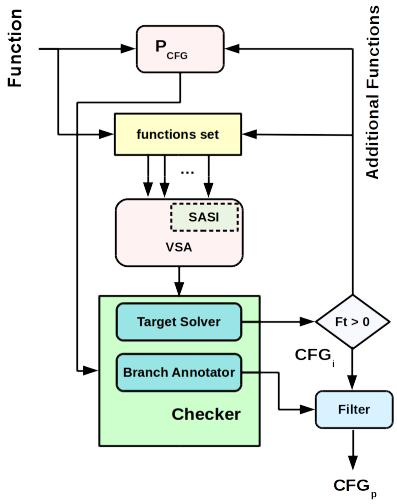
In this work, we restore this loss of precision, while maintaining signedness agnosticism, by designing a domain based on the fundamental concepts of Wrapped Intervals, but supporting a *stride*. We call this domain *Signedness-Agnostic Strided Interval*. The

use of a stride allows us to precisely determine the values that a program variable can assume (e.g., odd values for `opt` in Code 4.1), thus improving the precision of a program CFG.

Our domain is particularly suited for binary analysis. In fact, there are several high-level code constructs (e.g., `switch-case` statements) that are translated in binary code in a way (e.g., through jump tables) that Wrapped Intervals would not handle well. In these cases, the use of a stride would significantly improve the precision of the overall analysis (e.g., by precisely enumerating the destinations of a jump table).

4.2 Overview

Our approach to soundly perform code debloating of a program P is based on the recovery of a complete and precise CFG for P. Given a program P to debloat, if the CFG G for P is complete, every basic block not present in G can be safely removed from P without hindering its correctness. However, the more G is precise, the more basic blocks can be safely removed from P. In fact, if G is also sound *all* the useless basic blocks would be removed from P. To achieve this goal, we designed a new technique called *Iterative CFG Refinement*.



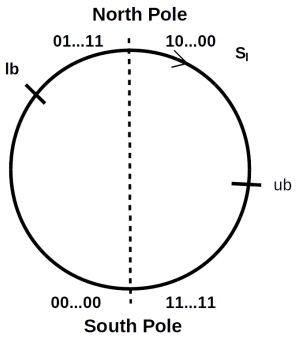


Figure 4.2: Iterative CFG Refinement Algorithm. **Figure 4.3:** Signed-Agnostic Strided Interval.

4.2.1 Iterative CFG Refinement

Given a function f (e.g., the address of a program's entry point), the Iterative CFG Refinement procedure iteratively builds f's CFG and leverages a sound algorithm based on value-range analysis to refine it.

The Iterative CFG Refinement algorithm relies on the availability of a procedure P_{CFG} to recover the CFG of the function f. We assume that P_{CFG} can recover all the basic blocks and code boundaries within f. We do not make any further assumptions about the precision of P_{CFG}. For example, P_{CFG} could be simply defined as a procedure that creates edges among all the possible basic blocks of f. The iterative CFG refinement algorithm is depicted in Figure 4.2, and can be divided into three main components, which we explain in the remaining of this Section.

CFG and VSA. First, we use P_{CFG} to compute f's CFG, and add f to a *function set* (initially empty). Then, we perform a *Value-Set Analysis* [8] (or *VSA*) on each function in the function set. The VSA is a static analysis based on abstract interpretation [29] that determines a conservative approximation of the set of numeric values and addresses that variables assume at each program point within a function f. The VSA utilizes our abstract domain SASI (detailed in Section 4.3) to analyze f and retrieve precise information about the binary variables (i.e., registers and memory locations).

Checker. The *Checker* module utilizes the VSA results to augment and refine the CFG through two different sub-modules: the *Branch Annotator*, and the *Target Solver*.

The *Branch Annotator* retrieves each CFG's conditional edge e_c (i.e., guarded by an *if-then-else* condition), and analyzes the logical expression of the condition that determines whether e_c would be taken or not at runtime. To this end, it relies on the abstract operations defined on SASI (shown in Section 4.5) to evaluate the theoretical satisfiability of the expression. If no solution exists, the Branch Annotator annotates e_c and marks it for removal.

On the other hand, the *Target Solver* considers f's basic blocks and collects those having indirect control-flow transfers (e.g., due to an indirect call). It then uses the VSA information to gather the set of function targets to which each indirect flow transfer can resolve, and add them to a set Ft. These functions are used to recover a new augmented CFG and to bootstrap a new round of VSA.

When a fixed-point is reached, that is when no new flow transitions are discovered (i.e., $Ft = 0$) and no new edge is annotated, the current CFG (i.e., CFG_i) is passed to the *Filter* module.

Filter. The Filter module scans every edge in CFG_i and removes each annotated edge. Then, for each basic block b in CFG_i, it checks whether it exists an inbound edge for b. If not, it retrieves all the nodes dominated (as defined in graph theory [137]) by b and removes them from CFG_i. Finally, it returns the filtered CFG_p.

4.2.2 Program Debloating

From a security point of view, the problem of program debloating is formulated as *decreasing the attack surface* of a program by removing its dead code. This goal can be achieved with two different techniques: (i) *deleting* the dead code from the binary, (ii) *rewriting* the dead code with useless instructions (e.g., `hlt`).

Though both approaches effectively remove the potentially dangerous dead code from a program, the former presents more challenges. In fact, if the code of a binary is modified, potentially all of its code and data pointers must be updated to reflect the new program layout. In literature, two main approaches are proposed to achieve this goal: *Binary Instrumentation* and *Binary Rewriting*. In the former approach, a binary file is usually augmented with pieces of trampoline code that fix the program pointers at runtime [15,53,91,147]. In the latter approach, Binary Rewriting techniques [127,128]

attempt to achieve perfect disassembling (i.e., by solving code and data pointers), thus being able to recompile a program. Unfortunately, any of the techniques mentioned above present several limitations and trade-offs (e.g., ignoring computed code pointers) that hinder their soundness.

For this reason, to preserve the soundness of our approach, we decided to eliminate the dead code of a program by rewriting it. This approach, though not decreasing the size of a program itself, presents mainly two advantages: (i) the program does not need any external support to be executed (e.g., a modified dynamic loader to perform runtime address resolution) and (ii) soundness is preserved. Note, however, that our approach can be easily extended to use one of the state-of-the-art solutions of binary rewriting, such as Ramblr [127], to effectively delete the dead code.

4.3 Signedness-Agnostic Strided Intervals

In this Section, we present a novel approach to the abstract modeling of numeric entities with a fixed width. We define a new *abstract domain* named *Signedness-Agnostic Strided Interval* to represent the set of values that a numeric entity (of a given bit-width) can possibly assume.

4.3.1 Definition

A Signed-Agnostic Strided Interval (or *SASI*) is indicated as $r = s_r[lb, ub]w$, where lb and ub are bit-vectors of w bits (called *lower bound* and *upper bound* respectively), whereas s_r (called *stride*), is a non-negative integer.

A SASI r represents the set of values: $\{lb,\ lb +_w s_r,\ lb +_w 2 * s_r,\ ...,\ ub\}$, where $+_w$ represents modular addition of bit-width w (i.e $x +_w y = (x + y) \bmod 2^w$). Formally,

$$r = \{(lb + k * s_r) \bmod 2^w \leq ub,\ k \in \mathbb{N}\} \tag{4.1}$$

For example, `2[1010, 0010]4` represents the set of values {`1010, 1100, 1110, 0000, 0010`}. Note that, the SASI $0[lb, lb]w$ represents the singleton lb.

A SASI variable can be graphically represented through a *number circle*, as depicted in Figure 4.3. The set of numerical values represented by a SASI are determined by traversing the number circle clockwise starting from the lower bound lb up to the upper bound ub with increments of the stride value s_r. SASI can represent *unsigned* and *signed* variables alike.

For example, consider the SASI $r = 1[0100, 1010]4$ representing a variable x (i.e., $x \in r$). r represents the values $4 \leq x \leq 10$ if x is interpreted as an unsigned variable, or the values $(4 \leq x \leq 7) \vee (-8 \leq x \leq -6)$ if interpreted as signed. In the case of signed values, the *South Pole* and the *North Pole* divide the positive and negative numbers: Positive numbers begin from the left of South Pole, proceeding clockwise up

to the left of North Pole. Similarly, negative values begin from the right of the North Pole, proceeding clockwise down to the right of South Pole. Note that, operations on SASI (Section 4.5) do not assume the signedness of variables, thus providing sound results for both signed and unsigned interpretations.

Throughout this work we use the following notation: B_w and W_w indicate the set of all the possible bit-vectors representable on *w* bits, and the set of all the possible SASIs representable on the same number of bits, respectively. A modular operation on w bits is indicated as op_w (e.g., $+_w$), where $x\ op_w\ y = (x\ op\ y) \bmod 2^w$. We use the sequence representation b^k to express a k-long sequence of the bit b ($b \in \{0, 1\}$), and the symbol $||$ to indicate sequence concatenation. Furthermore, the symbol $\leq$ represents the lexicographic ordering in B_w, whereas $\leq_x$ represents the relative ordering, with respect to the value x, on the number circle (Figure 4.3). That is to say:

$$a \leq_x b\ \ iff\ \ (a -_w x) \leq (b -_w x) \tag{4.2}$$

Informally, starting from x and proceeding clockwise on a number circle, a is encountered before b.

Using the above notations, we now define several functions as needed for any static analysis based on abstract interpretation.

Definition 1. ***Concretization Function.*** *Given a SASI $r = s_r[lb, ub]w$, the concretization function $\gamma : W_w \rightarrow P(B_w)$ is defined as follows:*

$$\begin{aligned} &\gamma(\bot) = \emptyset \\ &\gamma(r) = \{lb,\ lb +_w s_r,\ lb +_w 2 * s_r,\ ...,\ ub\} \\ &\gamma(\top) = B_w \end{aligned} \tag{4.3}$$

Where $P(B_w)$ is the power set of B_w, $\bot$ denotes an empty SASI (i.e., $0[,]w$) and $\top$ denotes the full SASI (i.e., $1[0^w, 1^w]w$).

Definition 2. ***Abstraction Function.*** *Given a set of values $V = \{v_1, v_2, ..., v_n\}$, the abstraction function $\alpha : P(B_w) \rightarrow W_w$ is defined as follows:*

$$\begin{aligned} &\alpha(\emptyset) = \bot \\ &\alpha(V) = s_r[a_1, a_n]w, (a_j)_{j=1}^{n} = sort(v_1, v_2, ..., v_n) \\ &\alpha(B_w) = \top \end{aligned} \tag{4.4}$$

where $s_r = gcd(d_1, d_2, ..., d_{n-1})$ and $d_j = a_{j+1} -_w a_j,\ for\ 1 \leq j \leq (n-1)$. gcd is the greatest common divisor function, and $sort$ is a function sorting values in ascending order.

Intuitively, given a set of bit-vectors, the abstraction function sorts its elements in ascending order, thus creating the sequence $(a_j)_{j=1}^{n}$. Then, it considers the first and last elements as the lower and upper bounds respectively, and, starting from the lower bound, it selects the greatest stride s_r that includes all the elements in $(a_j)_{j=1}^{n}$.

Definition 3. ***Membership Function.*** *Given a bit-vector v and a SASI $r = s_r[lb, ub]w$, the membership function $\in$ is defined as follows:*

$$v \in r = \begin{cases} true & if\ r = \top \\ false & if\ r = \bot \\ v \leq_{lb} ub \wedge (v -_w lb) mod\ s_r = 0 & if\ r = s_r[lb, ub]w \end{cases} \tag{4.5}$$

Definition 4. ***Cardinality Function.*** *Given a SASI $r = s_r[lb, ub]w$ the cardinality function $\#$ is defined as:*

$$\#(\bot) = 0$$

$$\#(\top) = 2^w$$

$$\#(r) = \left\lfloor \frac{ub - lb + 1}{s_r} \right\rfloor$$

Definition 5. ***Ordering Operator.*** *Given two SASIs $r = s_r[a, b]w$ and $t = s_t[c, d]w$, the ordering operator $\sqsubseteq$ is defined as follows:*

$$r \sqsubseteq t = \begin{cases} False & if\ r = \top \wedge t \neq \top \\ True & if\ r = \bot \vee\ t = \top \vee \\ & \quad ((a = c) \wedge (b = d) \wedge \\ & \quad (s_r\ mod\ s_t = 0)) \\ a \in t \wedge b \in t\ \wedge (c \notin r \vee d \notin r) & otherwise \\ \quad \wedge (a - c)\ mod\ s_t = 0 \wedge\ s_r\ mod\ s_t = 0 & \end{cases} \tag{4.6}$$

In other words, one SASI is considered to be *included* in another if every value in the former is contained in the latter, that is $\gamma(r) \subseteq \gamma(t)$.

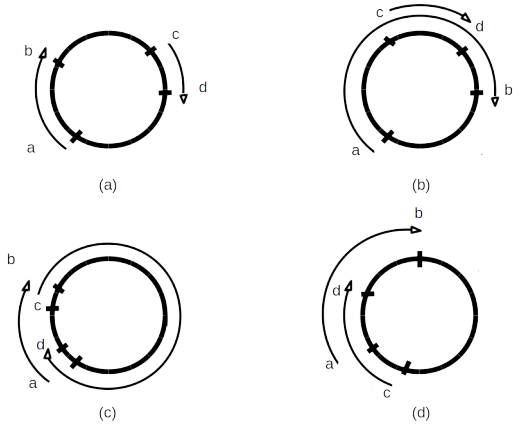


Figure 4.4: Possible relative positions of two SASIs $r = s_r[a, b]w$ and $t = s_t[c, d]w$.

Note that, the inclusion property holds if we replace the stride of a SASI by one of its divisors. i.e

$$r' = s_{r'}[a, b]w \wedge (s_r \bmod s_{r'} = 0)) \rightarrow r \sqsubseteq r' \tag{4.7}$$

Furthermore, while ($\sqsubseteq$, W_w) forms a partially ordered set (with least element $\perp$ and greatest element $\top$), it does not form a lattice as the ordering does not always provide a unique *least upper bound* (or *join*) and *greatest lower bound* (or *meet*). For example, consider the two SASIs $2[0010, 0100]4$ and $2[1000, 1110]4$. Two minimum upper-bounds (i.e., having the same cardinality) for these SASIs are $2[0010, 1110]4$ and $2[1000, 0100]4$. However, they are incomparable, thus violating the unique least upper bound requirement. Since a join and meet are not available, we must define a deterministic pseudo-join and a pseudo-meet.

Pseudo-Join and Generalized Join operators. The relative position of two SASIs (viz

$r = s_r[a, b]w$ and $t = s_t[c, d]w$) can be one of the four possibilities shown in Figure 4.4. With this in mind, we define the pseudo-join operator between two SASIs as follows:

Definition 6. ***Pseudo-Join Operator.*** *Given two SASI $r = s_r[a, b]w$ and $t = s_t[c, d]w$, the pseudo-join operator $\widetilde{\sqcup}$ is defined as follows:*

$$r \mathbin{\widetilde{\sqcup}} t = \begin{cases} t & if\ r \sqsubseteq t \\ r & if\ t \sqsubseteq r \\ \top & if\ a \in t \wedge b \in t \wedge c \in r \wedge d \in r \\ s_{ad}[a, d]w & if\ c \in r \wedge b \in t \wedge a \notin t \wedge d \notin r \\ s_{cb}[c, b]w & if\ a \in t \wedge d \in r \wedge c \notin r \wedge b \notin t \\ s_{ad}[a, d]w & if\ a \notin t \wedge d \notin r \wedge c \notin r \wedge b \notin t \wedge \\ & \quad \#(s_{ad}[a, d]w) \leq \#(s_{cb}[c, b]w) \\ s_{cb}[c, b]w & otherwise \end{cases} \tag{4.8}$$

Where $s_{xy} = gcd(s_r, s_t, y -_w x)$, with $xy \in \{(a, d), (c, b)\}$ and gcd is the great common divisor function.

The pseudo-join operator we defined assures that the SASI with the lowest cardinality, and, thus, most precise, is always picked. However, it is not associative, that is $((r \mathbin{\widetilde{\sqcup}} t) \mathbin{\widetilde{\sqcup}} z) \neq (r \mathbin{\widetilde{\sqcup}} (t \mathbin{\widetilde{\sqcup}} z))$. Therefore, we define a *generalized pseudo-join operator* ($\widetilde{\bigsqcup}$). Given a set of n SASIs, this operator has to produce the SASI z with the least cardinality possible, and such that the n SASIs are included in z. Theoretically, there are $n!$ possible join to consider to pick z. However, as SASIs are traversed clockwise on the number circle, only n of these should be considered. The results of the other

Algorithm 2 Generalized Join.

```
1: procedure ⨆̃(X)
2:     (y_j)_{j=1}^n ← sort_by_lowerbound(X)
3:     z ← ⊥
4:     for i in (1 ... n) do
5:         z_i ← reduce(lambda x, y: ⨆̃(x, y), (y_j)_{j=i}^n || (y_j)_{j=1}^{(i-1)})
6:         if z = ⊥ or (#(z_i) < #(z)) then
7:             z ← z_i
8:         end if
9:     end for
10:    return z
11: end procedure
```

$n! - n$ joins are included in one of these n joins. The generalized pseudo-join operator is defined in Algorithm 2, and works as follows: Given a set X of n SASIs, it sorts X elements according to the lexicographic ascending order of their lower bounds (Line 2), producing a new sequence (i.e., $(y_j)_{j=1}^{n}$). Then, referring to the circle number representation, it considers each SASI in $(y_j)_{j=1}^{n}$ and proceeding clockwise joins it with the other SASIs in lexicographical order, producing a final SASI z_i (function $reduce$ at Line 5). Finally, the SASI z_i with the least cardinality is returned. The generalized pseudo-join operator is sound by construction, but not monotone. Given three SASIs r, t and z such that $r \sqsubseteq t$, it is *not* always true that $\widetilde{\bigsqcup}(\{r, z\}) \sqsubseteq \widetilde{\bigsqcup}(\{t, z\})$. As an example, consider $r = 3[12, 15]4$, $t = 3[9, 15]4$ and $s = 2[2, 8]4$. We have $r \sqsubseteq t$, $\widetilde{\bigsqcup}(\{r, s\}) = 1[12, 8]4$ and $\widetilde{\bigsqcup}(\{t, s\}) = 1[2, 15]4$. However, The two final SASIs are not comparable, that it $1[12, 8]4 \not\sqsubseteq 1[2, 15]4$ and $1[2, 15]4 \not\sqsubseteq 1[12, 8]4$. This is a contradiction with what is believed in existing work, and in Section 4.4, we will prove that it

is not possible to define an *existential* monotonic join over abstract domains based on number circles.

The lack of the monotone property does not assure termination of the analysis [89], as a least fixed point might not exist. Unfortunately, this property holds for every domain based on number circles. To address this problem, we defined a *widening* operator to guarantee termination of the analysis. As our widening operator is similar to the one already defined in [89], it is not presented here.

Pseudo-Meet and Generalized Meet operators. The pseudo-meet operator ($\widetilde{\sqcap}$) is based on the intersection (Ω) between two SASI. By definition, the intersection between two SASIs should contain (at least) all the values represented by both of its operands.

Formally, given two SASIs r and t the following must hold:

$$\gamma(r) \cap \gamma(t) \subseteq \gamma(r \Omega t) \tag{4.9}$$

We define the intersection operation between two SASIs by considering all of their possible relative positions (Figure 4.4) as follows:

Definition 7. ***Intersection Operator.*** *Given two SASI $r = s_r[a, b]w$ and $t = s_t[c, d]w$, the intersection operator Ω is defined as follows:*

$$r\Omega t = \begin{cases} \{\} & if\ r = \bot \vee t = \bot \\ \{t\} & if\ r = t \vee r = \top \\ \{t\} & if\ t \sqsubseteq r \\ \{r\} & if\ t = \top \\ \{r\} & if\ r \sqsubseteq t \\ \{s_n[a_1, d_1]w, s_n[c_1, b_1]w\} & if\ a \in t \wedge b \in t \wedge c \in r \wedge d \in r \\ \{s_n[c_1, b_1]w\} & if\ c \in r \wedge b \in t \wedge a \notin t \wedge d \notin r \\ \{s_n[a_1, d_1]w\} & if\ d \in r \wedge a \in t \wedge c \notin r \wedge b \notin t \\ \{\} & otherwise \end{cases} \tag{4.10}$$

Where s_n is calculated as the least common multiple (lcm) of s_r and s_t. Note the two SASIs $\{s_n[a_1, d_1]w, s_n[c_1, b_1]w\}$ in Definition 4.10 (i.e., case c in Figure 4.4): a_1 is the first common value between the SASI *t* and the sub-interval $[a, d]$ of *r*, d_1 is the greatest value contained in the interval $[a, d]$ such that $(d_1 - a)\ mod\ s_n = 0$, c_1 is the first common value between the SASI *t* and the sub-interval $[c, b]$ of *r* and, finally, b_1 is the greatest value contained in the interval $[c, b]$ such that $(b_1 - c)\ mod\ s_n = 0$. A very similar reasoning applies to the remaining SASIs.

The first common value between two SASIs $r = s_r[a, b]w$ and $t = s_t[c, d]w$ is retrieved by finding the least non-negative integer values for k_r and k_t that satisfy the following equality:

$$k_r * s_r + a = k_t * s_t + c \tag{4.11}$$

The above equation is a linear diophantine equation that is solved for non-negative values to find k_r and k_t. These values are used in Equation 4.11 to get the first common value shared by two SASIs.

Finally, the pseudo-meet operator is defined as follows:

$$r \mathbin{\widetilde{\sqcap}} t = \widetilde{\bigsqcup}(\Omega(r,t)) \tag{4.12}$$

As it relies on the pseudo-join operation, the pseudo-meet operation is also not associative. For this reason, given a sequence S of SASIs, a generalized pseudo-meet has been defined as follows:

$$r \widetilde{\bigsqcap} t = \widetilde{\bigsqcup}(\{\Omega(s_i, s_{i+1}) \forall i,\ 0 \leq i \leq (\#(S) - 1)\}) \tag{4.13}$$

Where s_i is the i_{th} SASI in S.

Complement. Given a SASI r, the complement operator returns a SASI which represents the set of values not included by r:

$$\begin{aligned} \overline{\bot} &= \top \\ \overline{\top} &= \bot \\ \overline{s_r[a,b]w} &= s_n[b +_w 1, a -_w 1]w \end{aligned} \tag{4.14}$$

Where the new stride, s_n is computed as:

$$s_n = \begin{cases} 1 & if\ s_r = 0 \\ gcd(s_r, b -_w a -_w 2) & otherwise \end{cases} \tag{4.15}$$

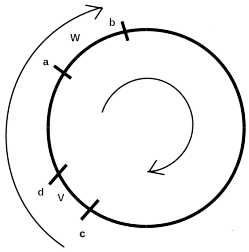


Figure 4.5: Join in number circle.

4.4 Termination

During our study, we discovered that no abstract domain which represents numeric entities as number circles can possibly form a lattice, as long as the definition of the (pseudo-)join is *existential*.

To generalize our discussion, in the following proof we will refer to number circle's based abstract domains using the w-interval's notation of [89]. This because [114] [47] and SASI can be reduced to be particular cases of w-intervals.

Given two arbitrary w-intervals w and v, we define a join operator as *existential* if:

(a) it returns v if $w \sqsubseteq v$, and

(b) it returns a w-interval $z = v \mathbin{\widetilde{\sqcup}} w \neq \top$ if $w, v \neq \top \wedge v \neq \overline{w}$.

Intuitively, the above means that an *existential* join operator returns a non-$\top$ interval, if there exists an interval z such that $w \sqsubseteq z$ and $v \sqsubseteq z$.

As an example, consider the intervals in Figure 4.5. The definition of an existential join operator should define an interval different than $\top$ as result of $v \widetilde{\sqcup} w$. Both intervals $[a, d]$ and $[c, b]$ are valid answers for an existential join operator.

We believe that this assumption is general and it should be used by any abstract domains employed in any static analyses. In fact, if a join between two arbitrary w-intervals would always return $\top$, an analysis based on w-intervals would not give any meaningful results.

In a lattice, by definition, there must exist a unique monotone join (as well as a unique monotone meet) between two arbitrary elements of the poset. In the following, we will show that if a join is existential, it cannot be monotone, and therefore the abstract domain considered cannot be a lattice.

Both [114] [47] define existential join operators and, even though do not claim their monotonicity, they wrongly claim their abstract domains to be lattices.

Proof. Considering w, v and x to be w-intervals, we have a monotone join if and only if:

$$w \sqsubseteq x \implies (w \widetilde{\sqcup} v) \sqsubseteq (x \widetilde{\sqcup} v) \tag{4.16}$$

As Definition 4.16 must hold for any join operators to be monotone, regardless of the relative positions of x, w and v, for simplicity, we assume that w and v do not intersect. In particular, we will refer to Figure 4.5 throughout the whole proof.

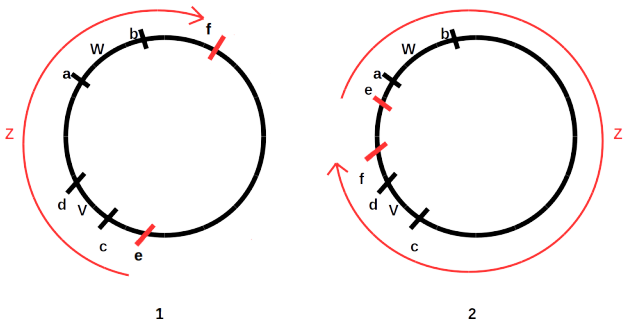


Figure 4.6: Possible joins for an existential join operator.

We show now that given any existential (pseudo-)join operators one can always choose a relative position for x such that $w \sqsubseteq x$ holds while Formula 4.16 does not.

Consider the interval z such that $z = (w \mathbin{\widetilde{\sqcup}} v) = [e, f]$. As $\mathbin{\widetilde{\sqcup}}$ is existential by hypothesis and v and w do not intersect and $z \neq \top$, z's relative position should be represented by either one of the two scenarios depicted in Figure 4.6. Note that, the cases where $z = [c, b]$ and $z = [a, d]$ are particular cases of scenario 1 and 2 respectively, in Figure 4.6.

Now, if x is equal to $[a, d]$ and the (pseudo-)join operator is defined such that z falls in scenario 1, we have:

$$w \sqsubseteq x \implies (w \mathbin{\widetilde{\sqcup}} v) \sqsubseteq (x \mathbin{\widetilde{\sqcup}} v) \tag{4.17}$$

As $(x \mathbin{\widetilde{\sqcup}} v) = [a, d]$ (by definition of existential join operator) and $(w \mathbin{\widetilde{\sqcup}} v)$ falls in scenario 1 (e.g., $[c, b]$), they are not two comparable elements of the poset.

Similarly, if the join operator is defined such that z falls in scenario 2 and we consider an x equal to $[c, b]$, the same reasoning applies.

Therefore, whatever w-interval (different from $\top$) is returned by the join of two w-intervals, one can always have a w-interval such that the following is true:

$$v \sqsubseteq x \implies v \mathbin{\widetilde{\sqcup}} w \sqsubseteq x \mathbin{\widetilde{\sqcup}} w \tag{4.18}$$

Proving that a monotone existential (pseudo-)join cannot exist in an abstract domain based on number circles. □

The major problem of the lack of monotonicity is that termination of the analysis is not guaranteed.

Since our poset contains finite ascending chains, some of them on the order of 2^w elements, a widening operator that accelerates towards a fixed point is needed. In this work, we use the same widening operator as defined in [89].

4.5 Signedness-Agnostic Strided Interval Operations

In this Section, we formally define various arithmetic and bitwise operations over the SASI abstract domain.

Note that, as in binary analysis the signedness of an operation is in some cases explicit (e.g., IMUL), SASI's implementation also includes signedness-aware operations.

4.5.1 Addition and Subtraction

Adding two SASIs results in a SASI whose lower bound is the sum of the operands' lower bound and whose upper bound is the sum of the operands' upper bound. This is similar to the addition operation defined in the well-known Interval arithmetic theory [56].

Formally, the addition operation on two generic SASIs $r = s_r[a, b]$ and $t = s_t[c, d]$ is defined as follows:

$$r +_w t = \begin{cases} \bot & if\ r = \bot \vee t = \bot \\ s_s[a +_w c,\ b +_w d]w & if\ \#r + \#t \leq \frac{2^w}{s_s} \\ \top & otherwise \end{cases} \tag{4.19}$$

Where $s_s = gcd(s_r,\ s_t)$.

In the SASI domain, the lower and upper bounds and are expressed on a finite number of bits, and the sum of bounds can wrap around, resulting in an *arithmetic overflow*. This possibility is detected by considering the cardinality of the SASIs r and t. In fact, if the sum of their cardinality is greater than the number of values representable on a number circle on w bits using a stride s_s, it means that starting from the lower bound of r and proceeding clock-wise we would end up crossing r's lower bound once again. In this case, $\top$ must be returned.

To be sound, the stride for the new interval has to be calculated as the greatest common divisor of the two operands strides.

Similarly, the subtraction operation can be defined as:

$$r -_w t = \begin{cases} \bot & if\ r = \bot \vee t = \bot \\ s_s[a -_w d,\ b -_w c]w & if\ r = s_r[a,b]w \wedge t = s_t[c,d]w \\ \top & otherwise \end{cases} \tag{4.20}$$

Where again $s_s = gcd(s_r,\ s_t)$.

4.5.2 Multiplication, Division and Modulus

Multiplication, Division and Modulus are *interpretation-dependent operations*, meaning that the result of these operations depends on the signed or unsigned interpretation of the values covered by the SASI operands [75].

A sound implementation of any interpretation-dependent operation should first compute the results of applying the operation on both signed and unsigned interpretations and then join these results to get the final result.

Each SASI is first separated at the North and South poles, so that the resulting SASIs does not cross any pole (resulting in an unsound result) and so that the ordering of both signed and unsigned interpretations can be reasoned about. To split a SASI on the poles we use the cut function as defined in [89]:

$$cut(r) = \cup\{ssplit(u)|u \in nsplit(r)\} \tag{4.21}$$

Where $\cup$ is the set union. Note that, both of the pole splittings are such that the join of the resulting SASIs is greater or equal (in the poset ordering) than the original SASI.

We then perform both signed and unsigned multiplication on these SASIs. Given two SASIs that do not straddle any pole (e.g., $r = s_r[a, b]w$ and $t = s_t[c, d]w$), we define signed multiplication ($\times_{ws}$) as follows:

$$r \times_{ws} t = \begin{cases} \bot & if\ r = \bot \vee t = \bot \\ s_n[a \times_w c, b \times_w d]w & if\ m(a) = m(b) = m(c) = m(d) \\ & \wedge\, b \times d - a \times c < 2^w \\ s_n[a \times_w d, b \times_w c]w & if\ m(a) = m(b) = 1 \\ & \wedge\, m(c) = m(d) = 0 \\ & \wedge\, b \times c - a \times d < 2^w \\ s_n[b \times_w c, a \times_w d]w & if\ m(a) = m(b) = 0 \\ & \wedge\, m(c) = m(d) = 1 \\ & \wedge\, a \times d - b \times c < 2^w \\ \top & otherwise \end{cases} \tag{4.22}$$

Where *m* is the function that extract the most significant bit of a bitvector and $s_n = lcm(s_r, s_t)$. As in the case of the addition operation, if, the multiplication overflows the stride is set to 1 ($\top$ case).

Unsigned multiplication ($\times_{wu}$) is defined as:

$$r \times_{wu} t = \begin{cases} \bot & if\ r = \bot \vee t = \bot \\ s_n[a \times_w c, b \times_w d]w & if\ b \times d - a \times c < 2^w \\ \top & otherwise \end{cases} \tag{4.23}$$

These two multiplications are the same as defined by Kulisch in [75], and are sound if the operand SASIs do not straddle the poles.

Now, given two non-straddling SASIs, a sound signedness-agnostic multiplication ($\times_{wus}$) can be defined as intersection of SASIs returned by $\times_{wu}$ and $\times_{ws}$.

$$r \times_{wus} t = (r \times_{wu} t) \Omega (r \times_{ws} t) \tag{4.24}$$

Finally, the general formula to multiply two SASIs can be defined using Generalized join as:

$$r \times_{w} t = \tilde{\bigsqcup} \{m | u \in cut(r), v \in cut(t), m \in u \times_{wus} v\} \tag{4.25}$$

Note that, if the signedness of a multiplication can be inferred during a given analysis (e.g., when explicitly specified by an operation), the corresponding signed-aware multiplication (as defined above) is used.

For division and modulus, we use the same implementation as in [89] and set the stride of the resulting SASI to 1. This is because it is not trivial to have a principled approach to pick a precise stride (i.e., stride > 1).

4.5.3 Bitwise operations

In this Section, we formally define the various bitwise operations.

Or. Defining a precise and sound Bitwise Or is not trivial. We can use the unsigned version of Warren's algorithm [134] to compute the bounds. To use this method, we need to first split the operand SASIs on the south pole (to make them unsigned), then perform Warren's algorithm, and then join the resulting SASIs.

Given two generic SASIs such as $r = s_r[a, b]w$ and $t = s_t[c, d]w$, the algorithm used to calculate the *or* operation between them is shown in code Listing 3.

Algorithm 3 Bitwise or.

```
1: procedure |_w(r, t)
2:     ret ← [ ]
3:     for u in ssplit(r) do
4:         for v in ssplit(t) do
5:             t ← min(ntz(u.stride), ntz(v.stride))
6:             s_n ← 2^t
7:             m ← (1 << t) − 1
8:             k ← (u.lb&m)|(v.lb&m)
9:             u_1 = [(u.lb& ∼ m), (u.ub& ∼ m)]
10:            v_1 = [(v.lb& ∼ m), (v.ub& ∼ m)]
11:            [lb, ub] ← u_1 |^{wr}_w v_1
12:            ret.append(s_n[((lb& ∼ m)|k), (ub& ∼ m)|k)])
13:        end for
14:    end for
15:    return ⨆̃(ret)
16: end procedure
```

First, we split each SASI on the south pole, so that we can employ the unsigned version of Warren's *or* algorithm (indicated with $\overset{wr}{|}_w$). Then, for each SASI u and v obtained from the splitting, we retrieve the number of trailing zeros (function ntz) in the bitvector representation of the considered SASI strides, and we get the minimum resulting value (line 5) to set a variable t. The stride of the new SASI is set to 2^t.

This is done because all the values represented by the SASI resulting from $u \mid v$ share the same t low-order bits. Therefore, the choice of a stride equal to 2^t is a sound choice (line 6) [8]. The value of these t bits is $k = (u.lb\&m)|(v.lb\&m)$ (where $m = (1 << t) - 1$ and $u.lb$ and $v.lb$ means lower bound of u and v respectively).

Since every value represented by the final stride share the first t bits, the $(w - t)$ high-order bits are handled by masking out the obtained t low-order bits, and then

applying unsigned version of Warren's *or* algorithm for finding bounds for the SASI resulting from $u|_w v$ (from line 9 to 11).

Finally, the SASI resulting from $r \mid_w t$ is obtained by applying the Generalized join on the list of SASIs collected by applying the algorithm just explained. Since the Warren's algorithms employed are sound, the *or* operation is sound [8].

Not. Given a SASI $r = s_r[a, b]w$, the *not* operation is formally defined as follows:

$$\sim r \; = \; s_r[\sim b, \sim a]w \tag{4.26}$$

This definition of the not operation is sound, as shown in [8].

And and Xor. Using De Morgan's laws we can soundly define *and* ($\&_w$) and *xor* ($\oplus_w$) operations using *Not* ($\sim$) and Or ($|_w$) as:

$$\begin{aligned} r \;\&_w\; t &= \sim (\sim r \mid_w \; \sim t) \\ r \oplus_W t &= \sim (\sim r|_w t) \mid_w \; \sim (r|_w \sim t) \end{aligned} \tag{4.27}$$

Shifts. We first define the logical right shift ($>>_{lw}$) of a SASI by a constant shift amount k. First, we will discuss about a SASI whose stride is 1 can be shifted, then we will generalize the discussion by including an arbitrary stride.

Also, in the following, the notation $>>_{lw}$ means logical right shift applied on operands of w-bits long.

If a SASI $x = 1[a, b]w$ does not cross the south pole (i.e., $b \geq_0 a$ or $b \geq a$) a logical right shift applied by shifting its bounds yields a sound result.

To prove this claim, it is sufficient to show that relative ordering of the values in such SASI are preserved after the shift. That is to say that for any two arbitrary values c and d contained in the interval x (i.e., $c \in x$ and $d \in x$) such that $d \geq c$, the following always holds: $(d >>_{lw} k) \geq (c >>_{lw} k)$.

Proof.

$$\begin{aligned} if\ d > c \implies \exists j \in \mathbb{N} : \forall i : w - j \leq i \leq w, \\ bit(d,i) = bit(c,i) \wedge bit(d,j) = 1 \wedge bit(c,j) = 0 \end{aligned} \tag{4.28}$$

where $bit(d,i)$ is the i_{th} bit of the bitvector d. The above definition existentially quantifies the most significant bit position where d and c differ i.e j.

Because of this, if k is lesser than j we have that $(d >>_{lw} k) > (d >>_{lw} k)$, if k is greater than j then $(d >>_{lw} k) = (c >>_{lw} k)$. Therefore, $(d >>_{lw} k) \geq (c >>_{lw} k)$ holds in any case. □

This means that the order of the elements in x is maintained even when its bounds are shifted. Now, since every value within x is greater (or equal) than its lower bound and smaller (or equal) than its upper bound, if shifted they will be contained in the interval $[a >>_{lw} k, b >>_{lw} k]$.

Let us now consider the case where x crosses the south pole (i.e., $b <_0 a$). In this case, after the shifting the relative order between the two bounds might be inverted (i.e., $a <_0 b$) as, regardless of the pole straddling, when a SASI is logically right shifted, both of its bounds are placed in the left hemisphere on the number circle.

This new SASI, though sound, can be unnecessarily large. The precision can be improved.

In fact, we can split the interval x on the south pole, perform the shifting on the splitted intervals and eventually pseudo-join them back. Because of the south split, no relative inversion might happen among the resulting intervals and Formula 4.28 holds resulting in a sound interval.

Let us now generalize the discussion by including the consider stride. The shifting of the stride can be handled if its least significant k bits are zeros. Given a SASI $r = s_r[a, b]w$, if the stride's least significant k-bits are zeros, then all the values in r will share k least significant bits: None of the last significant k bits of s_r can change any of the $(w-k)$ most significant bits for all values in r. This involves that if a SASI is logically right shifted of k bits, all its values are influenced only by the most significant $(w - k)$ bits of s_r. Therefore, in these cases a stride equal to $S = s_r >> k$ is a sound choice.

On the other hand, if the stride does not have all its least significant k bits as zeros, the only sound choice is to set the stride equal to 1.

Finally, given a SASI $r = s_r[a, b]w$ the logical right shift is defined as follows:

$$r >>_{lw} t = \begin{cases} \bot & if\ r = \bot \\ \widetilde{\bigsqcup}\{s_{ns}[l_s >> t,\ u_s >> t] | \\ \quad s_s[l_s, u_s] \in ssplit(r)\} & otherwise \end{cases} \tag{4.29}$$

Where s_{ns} is defined as follows:

$$s_{ns} = \begin{cases} max(s_s >> t,\ 1) & if\ lsb(s_s, k) = 0^k \\ 1 & otherwise \end{cases} \tag{4.30}$$

where $lsb(s_s, k)$ is a function that retrieves the least significant k bits from s_s.

Following a similar reasoning, the arithmetic right shift ($>>_{aw}$) is defined as:

$$r >>_{aw} t = \begin{cases} \bot & if\ r\ =\ \bot \\ \widetilde{\bigsqcup}\{s_{ns}[l_m,\ u_m] | s_s[l_s, u_s] \in nsplit(r)\} & otherwise \end{cases} \tag{4.31}$$

Where $nsplit$ splits a SASI on the North, $l_m = (l_s >> t) \mid m$ and $u_m = (u_s >> t) \mid m$. m is the mask used to propagate the bit sign (i.e., $m\ =\ ((2 * *t - 1) << (w - t))$), s_{ns} is defined as explained before for the logical shift case.

Note that, the reasoning explained in 4.28 applies also to the arithmetic right shift if the SASI does not cross the north pole.

The left shift is defined as follows:

$$r << t = \begin{cases} \bot & if\ r\ =\ \bot \\ \widetilde{\bigsqcup}\{s_{ns}[l_s << t,\ u_s << t]w | & otherwise \\ \qquad s_s[l_s, u_s]w \in ssplit(r)\} & \end{cases} \tag{4.32}$$

Where the new stride s_{ns} is calculated merely as $max(s_s\ <<\ t,\ 1)$. The argument to show the soundness of the right shifting is very similar to the one made for the left shifting case and it will not be discussed again.

Finally, if the shift amount is represented by a SASI s, we consider every values between the lower bound and the upper bound represented by s to compute the requested shift operation.

4.5.4 Truncate

The truncate operation $tr(r, k)$, truncates the given SASI r to a lower width k, where $r = s_r[a, b]w$ and $0 < k < w$.

The truncate operation is defined as follows:

$$tr(r,k) = \begin{cases} \bot & if\ r = \bot \\ lsb(s_r,k)[lsb(a,k),\ lsb(b,k)]k & if\ a >>_a k = b >>_a k \wedge \\ & lsb(a,k) \leq lsb(b,k) \\ lsb(s_r,k)[lsb(a,k),\ lsb(b,k)]k & if\ (a >>_a k) + 1 =_w \\ & (b >>_a k)\ \wedge lsb(a,k) \not\leq tr(b,k) \\ 0[a\&mask,\ a\&mask]k & if\ t \geq k \\ 2^t[0^{k-t}||lsb(a,t),\ 1^{k-t}||lsb(a,t)]k & otherwise \end{cases} \tag{4.33}$$

Where $>>_a$ is the arithmetic right shift, $mask = (2^k - 1)$ and t is the number of trailing zeros in the bit-vector representation of s_r. For example, if s_r = 101000, then t = 3. As already mentioned, $lsb(x, y)$ is a function which retrieves the least significant y bits from x.

In the above, definition the bounds in cases two, three, and four are retrieved using the approach of Wrapped Intervals [89], thus these are sound as long as the stride used is sound.

Given a generic SASI $r = s_r[a, b]w$, only the least significant k bits of s_r influence the least significant k bits of the values represented by r. Therefore, if we consider only the least significant k bits of all the values represented by r, s_r influences them in the

same way as $0^{w-k}lsb(s_r, k)$ does. Therefore, the stride $lsb(s_r, k)$ is a sound choice for cases two and three.

Observation: If a stride has t trailing zeros in its bit-vector representation, the least significant t bits of all the values in the SASI r are going to be identical.

Based on the above Observation, in case four (where $t \geq k$), the least significant k bits of all values in the operand SASI(r) will also be identical, resulting in a SASI with a single value.

In the last case, where $t < k$, we still have the same least significant k bits in all the values. However, the bits from position t to k might change. To be sound, we consider all the possible values for the bits from position t to k i.e 0^{k-t} to 1^{k-t}. The stride is computed so that the least significant k bits of all the values in the resulting SASI are identical (i.e., 2^t). Note that, this case results in $\top$, when $t = 0$.

4.5.5 Extension operations

We defined three types of extension operations: signdness-agnostic extension (or simply extension), signed extension and zero extension.

Extension. The extension operation $ext(r, k)$ extends the given SASI r to a higher bit width k, where $r = s_r[a, b]w$ and $k > w$.

Similar to multiplication, an extension is an interpretation-dependent operation, where the result of the operation depends on the signed or unsigned interpretation of the values covered by the SASI operand.

We implement the extension operation such that the resulting SASI is sound irrespective of the interpretation of the operand SASI. As an example, consider the SASI $r = 1[0010, 1010]4$. our implementation results in $ext(r, 5) = 1[00010, 11010]5$, which is sound irrespective of the signed or unsigned interpretation of r.

Formally, extension operation is defined as:

$$ext(r,k) = \begin{cases} s_r[0^{k-w}||a, 0^{k-w}||b]k & if\ m(a) = m(b) = 0 \\ s_r[0^{k-w}||a, 1^{k-w}||b]k & if\ m(a) = m(b) = 1 \wedge \\ & (10^{w-1} -_w a) \leq (10^{w-1} -_w b) \\ s_r[1^{k-w}||a, 1^{k-w}||b]k & if\ m(a) = m(b) = 1 \wedge \\ & (10^{w-1} -_w b) \leq (10^{w-1} -_w a) \\ s_r[0^{k-w}||a, 0^{k-w}||b]k & if\ m(a) = 1 \wedge m(b) = 0 \\ s_r[0^{k-w}||a, 1^{k-w}||b]k & if\ m(a) = 0 \wedge m(b) = 1 \\ 0[0^{k-w}||a, 1^{k-w}||b]k & if\ m(a) = 1 \wedge m(b) = 1 \wedge a = b \end{cases} \tag{4.34}$$

Where *m* is the function that extract the most significant bit of a bit-vector.

The inequality $(10^{w-1} -_w a) \leq (10^{w-1} -_w b)$ ensures that r does not straddle along the north pole. Similarly, $(10^{w-1} -_w b) \leq (10^{w-1} -_w a)$ ensures that r does straddle along the north pole.

We also implemented a signed extension operator as well as a zero extension operator. As these two operations are trivial they are not reported here.

Signed extension. The signed extension operation $sext(r, k)$ extends the SASI r to a higher bit width k, where $r = s_r[l, u]w$ and $k > w$. The extension depends on the values represented by r, and it is defined as follows:

$$sext(r, k) = \tilde{\bigsqcup}\{s_s[l_s|_k M(l_s),\ u_s|_k M(u_s)]w, s_s[l_s, u_s]w \in ssplit(r)\} \tag{4.35}$$

Where $M(x) = ((m(x) << (k - w)) - m(x)) << w)$, and $m(x)$ represent the most significant bit of the bit-vector x.

Note that, as a consequence of the above definition, if r falls completely in one hemisphere, the sign-extend operation merely prepends r's most significant bit to its bounds $(k - w)$ times.

Zero extension. The zero extension operation $zext(r, k)$, where $r = s_r[l, u]w$, preprend $(w - k)$ zeros to r and adjusts the number of bits to w. Note that, this operation means moving a SASI to the left hemisphere of the number circle.

4.6 Discussion

As stated in Section 4.2, our approach is based on the existence of a CFG recovery procedure P_{CFG} that guarantees that all the basic blocks of a function, and its boundary, are retrieved. We do not make any assumption about the capability of P_{CFG} to resolve

any indirect jumps, nor to resolve any path predicates. Given such a CFG recovery procedure, our approach can guarantee the soundness of the results.

Though our hypothesis might seem too restrictive in theory, we found it is not to be in practice. In fact, if a binary does not contain data within the boundary of a function f, state-of-the-art CFG recovery procedures, such as [116], can recover every basic blocks and boundary of f precisely. In our experience, most of the employed compilers (e.g., `gcc`/`g++`) insert data only in specific data sections (e.g., `rodata`). The only exception is represented by jump tables, which might be inserted within a function boundary, thus fooling (in principle) decompilers based on linear sweeping (e.g., objdump [1]).

However, most recent decompilers based on recursive approaches implement algorithms to precisely recover jump tables, and thus, providing in practice the guarantee our approach needs.

4.7 Evaluation

We run two different evaluations. First, we evaluate the precision of SASI against the related work on signedness-agnostic abstract domains. Then, we implement our static program bloating approach in a tool, called BINTRIMMER, and evaluate its efficiency.

[1] https://sourceware.org/binutils/docs/binutils/objdump.html

4.7.1 Signedness-Agnostic Strided Intervals

To compare SASI against Wrapped Interval (WI) [89] and quantify its precision, we performed two evaluations using range analyses on both source code and binary files. As shown in the following two sections, on average SASI is 98% more precise than the Wrapped Interval abstract domain.

Source code. For this evaluation, we implemented our Signedness-Agnostic Signed Interval analysis on LLVM and downloaded the publicly-available Wrapped Interval analysis. Then, we retrieved the same test suite utilized by Navas et al. in their work [89]: the SPEC CPU2000 [2]. This dataset is an industry-standardized CPU-intensive benchmark suite, developed from real user applications. As it contains an outstanding amount of mathematical and bitwise operations, it is particularly suited to evaluate abstract domains for numerical entities. Unfortunately, two benchmarks of SPEC CPU2000 (i.e., 300.twolf and 255.vortex) were unavailable at the time of the evaluation. Therefore, we used one more benchmark (462.libquantum) from the latest SPEC CPU (i.e., CPU2006 [3]). Note that, we did not use the whole SPEC CPU2006 suite, as it is only available for purchase. Then, we ran the LLVM range analysis [4] on each program in our dataset by using both the SASI and Wrapped Interval domains. For each one of these test, we collected four statistics: The number of variables recovered

[2] https://www.spec.org/cpu2000/
[3] https://www.spec.org/cpu2006/
[4] https://code.google.com/archive/p/range-analysis/

by using SASI and Wrapped Intervals, which were not $\top$ when the analyses reached a fix-point (indicated as R_{SASI} and R_{WI}, respectively). The number of recovered variables where SASIs provided a better over-approximation (i.e., lower cardinality) than the Wrapped Intervals (indicated as P_{SASI}), and finally the number of variables whose Wrapped Interval representation was more precise than the SASI's (indicated as P_{WI}). The results of our evaluation are represented in Figure 4.7. Δ *Variables Recovered* indicates the difference between R_{SASI} and R_{WI}, and the percentages above each bar quantify the variable recovery effectiveness of SASI (i.e., $\frac{R_{SASI}-R_{WI}}{R_{SASI}\cup R_{WI}}$). Δ *Precise Intervals* indicates the difference between P_{SASI} and P_{WI}), and, similarly, the percentages above each bar quantify the variable recovery *precision* of SASI (i.e., $\frac{P_{SASI}-P_{WI}}{P_{SASI}\cup P_{WI}}$).

As one can see, SASI always recovered more variables than Wrapped Intervals (the Δ of variables recovered is never a negative value), and, in most cases, the variables recovered by SASI were more precise than those recovered by Wrapped Interval. Nonetheless, there were a few cases where the variables recovered by Wrapped Intervals were more precise than SASI (e.g., 3.4% in $bzip2$). We investigated them and discovered that it was caused by the lack of associativity of the pseudo-join, as explained in Section 4.3. In fact, even though our domain's pseudo-join gives more precise results than the Wrapped Intervals' if taken individually, this is not strictly true if we chain them. However, our results clearly show that these cases are rare. For example, SASI recovered 1,170 (out of a total of 5,027) variables in gap whose intervals were

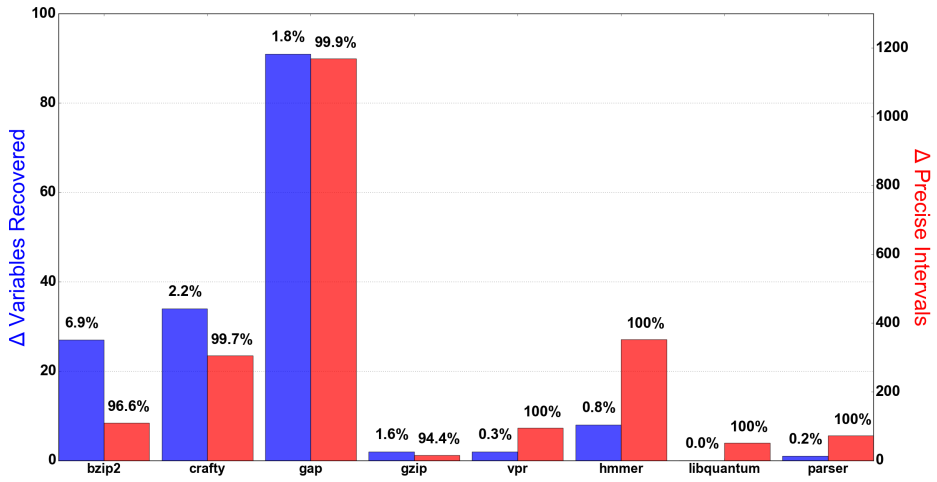


Figure 4.7: Source Code Evaluation. Δ *Variables Recovered* indicates the difference between the amount of variables recovered by SASI and Wrapped Intervals. Δ *Precise Intervals* indicates the difference between the number of instances SASI provided a better over-approximation and the number of instances Wrapped Intervals did.

more precise than those provided by Wrapped Intervals. On the other hand, Wrapped Intervals estimated only one variable in a more precise way than SASI. According to our tests on source codes, we can conclude that, on an average, SASI is 98% more precise than Wrapper Intervals (shown by Δ of precise intervals).

Binary files. To compare SASI's precision against Wrapped Interval's on binary files, we implemented Navas's abstract domain in angr [116]. In this evaluation, we collected all the binaries that DARPA released in the run-up to the CGC final event [5]. Then, we considered the functions of each binary and performed the angr's value-set analysis on them. Therefore, we collected the SASI and Wrapped Interval representations of each variable (i.e., memory location and CPU register) for each function at

[5] `http://archive.darpa.mil/cybergrandchallenge/`

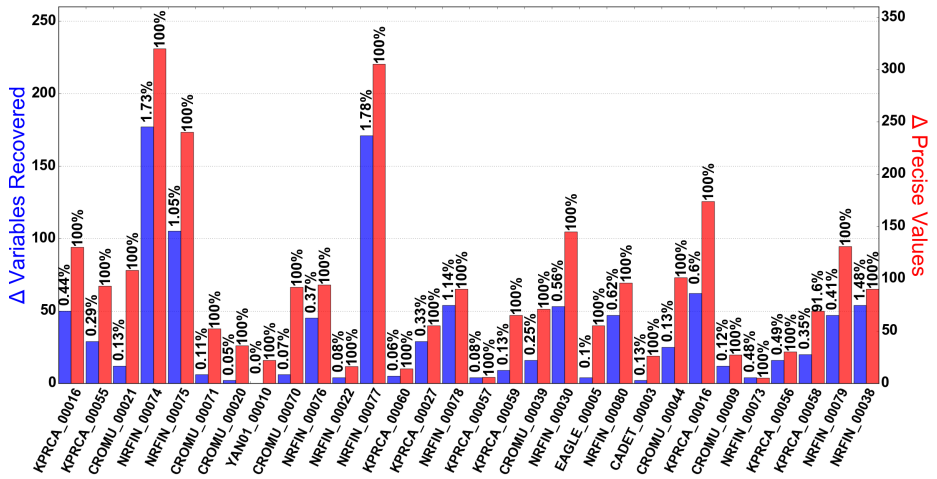


Figure 4.8: Binary Evaluation. Δ *Variables Recovered* indicates the difference between the amount of variables recovered by SASI and Wrapped Intervals. Δ *Precise Intervals* indicates the difference between the number of instances SASI provided a better over-approximation and the number of instances Wrapped Intervals did.

each program point, and collected the same four statistics (i.e., R_{SASI}, R_{WI}, P_{SASI} and P_{WI}) already introduced during the source code evaluation. The results collected are depicted in Figure 4.8.

As one can notice, even in this case SASI always outperformed Wrapped Intervals in terms of variables recovered (the Δ of variables recovered is never a negative value). Furthermore, we noticed that SASI excelled over Wrapped Intervals in terms of precision of recovered variables. In fact, in the case of binaries, SASI succeeded in recovering strictly more *precise* variables (100% values in Δ Precise Values), in every test but once (91.6% success in $KPRCA\_00058$). This result clearly shows the advantage of employing the SASI abstract domain when analyzing binary files.

4.7.2 BINTRIMMER

Our approach to program debloating was implemented in a tool, called BINTRIMMER. As introduced in Section 4.2, BINTRIMMER retrieves and patches those basic blocks in a binary that cannot be executed under any execution of a program. Also, in the following we use the term *partial trimming* when a function is partially removed, that is when some function's basic blocks were removed, but not all. BINTRIMMER was evaluated by using six binaries linked against two different C libraries: TinyExp [6] (containing 555 LOC) and b64 [7] (containing 192 LOC). We dynamically linked both of these libraries to the examples provided on their respective websites, for a total of six different programs.

After running BINTRIMMER and removing the identified dead code, we dynamically linked every binary to their patched library, and fuzzed them using *AFL* [8] for 48 hours. No crash was registered. Table 4.1 summarizes the results of this evaluation. *Total Trimmed* is the percentage of code patched, the *Min Partials*, *Max Partials* and *Avg Partials* values are calculated by considering only those functions that are not completely patched by BINTRIMMER. For each of these, we calculate their size (in bytes) and the number of patched bytes and report minimum, maximum, and average values respectively. The *Gadgets Removed* column represents the percentage of ROP gadgets

[6] `https://github.com/codeplea/tinyexpr`
[7] `https://github.com/littlstar/b64.c`
[8] `http://lcamtuf.coredump.cx/afl/`

(retrieved with *ROPGadget* [9]) removed by patching each binary's library. The columns *Tot ICF*, *ICF Resolved angr*, and *ICF Resolved* BINTRIMMER show the total number of indirect control-flow transfers, the percentage of those resolved by angr alone, and the percentage of indirect control-flow transfer resolved by BINTRIMMER, respectively. Finally, we report the *Time* in minutes employed to analyze each program.

Note that, failing to resolve even a single indirect control-flow transfer (i.e., ICF resolved less than 100%) might result in an incomplete CFG, and, therefore, an unsafe program debloating. We also manually checked for each of the six programs the completeness of the recovered CFG: while one CFG contained a super-set of all the possible control-flow transfers (completeness), the remaining five contained all and only the possible control-flow transfers (sound and complete). Note also that BINTRIMMER was able to patch code within functions (*Partials* columns). This is an important result as in these cases we outperform even a static linker: To the best of our knowledge, no linker can remove code within functions.

Finally, as we can see from the reported results, there are cases where our approach can remove a conspicuous portion of dead code: in TinyExpr3, we soundly removed 65.67% of the text section, with 40.33% represented by basic blocks within functions.

[9] `https://github.com/JonathanSalwan/ROPgadget`

Table 4.1: BINTRIMMER Results. *Total Trimmed* represents the total amount of code patched, *Min*, *max* and *Avg Partials* indicates the amount of code partially removed with functions. *Gadgets Removed* reports the amounts of ROP gadgets removed, *Tot ICF* is the total number of indirect control-flow transfers, and *ICF Resolved angr* and *ICF Resolved* BINTRIMMER indicates the percentage of ICF resolved by angr and BINTRIMMER respectively. *Time (min)* shows the time elapsed to analyze the binary.

Program	**Total trimmed**	**Min Partials**	**Max Partials**	**Avg Partials**	**Gadgets Removed**	**Tot ICF**	**ICF Resolved angr**	**ICF Resolved BINTRIMMER**	**Time (min)**
TinyExpr1	53.69%	3.62%	83.63%	29.12%	41.3%	2419	99.25%	100%	43
TinyExpr2	7.43%	0%	0%	0%	4.9%	2449	98.65%	100%	87
TinyExpr3	65.67%	3.7%	83.63%	40.33%	56.9%	2419	99.25%	100%	37
b641	1.17%	0%	0%	0%	3.0%	2389	99.6%	100%	24
b642	50.37%	0%	0%	0%	10.6%	2389	99.6%	100%	24
b643	34.43%	1.13%	0%	0%	36.4%	2389	99.6%	100%	22

Chapter 5

DIANE: Identifying Fuzzing Triggers in Apps to Generate Under-constrained Inputs for IoT Devices

In previous Chapters, we studied how to use static analyses to find bugs in firmware samples (see Chapter 2 and Chapter 3) and how to reduce the attack surface that an attacker could rely on to harm users (see Chapter 4). In this Chapter, we see how to use a combination of static and dynamic analyses to find bugs in firmware for IoT devices when the device itself is available but the firmware is not.

As introduced in Chapter 1, black-box approaches, which do not require access to a device's firmware, are usually employed in the IoT domain. However, black-box approaches usually require knowledge about the data format accepted by the device under analysis. Unfortunately, given the heterogeneity and lack of documentation of the protocols adopted by IoT devices, these approaches are hardly applicable.

However, most IoT devices have *companion apps* [136, 151] (i.e., mobile apps used to interact with the device), which contain the necessary mechanism to generate valid inputs for the corresponding device. Based on this observation, Chen et al. [21] proposed a tool, IoTFuzzer, which fuzzes IoT devices by leveraging their companion apps. IoTFuzzer analyzes the companion app and retrieves all the paths connecting the app's User Interface (UI) to either a network-related method or a data-encoding method. Then, IoTFuzzer fuzzes the parameters of the *first* function that handles user input along these paths, thus generating valid fuzzing inputs for the IoT device.

While this approach yields better results than randomly fuzzing the data directly sent to the IoT device over the network, in practice, it consists in mutating variables immediately after being fetched from the UI, before the app performs any input validation or data processing. Consequently, the effectiveness of IoTFuzzer suffers substantially when the app sanitizes the provided input—our experiments (Section 5.4.5) demonstrate that 51% of IoT companion apps perform app-side input validation. Indeed, recent research showed that mobile apps often perform input validation to trigger different behaviors [149]. For these reasons, IoTFuzzer's approach cannot produce *under-constrained* (i.e., not affected by app-side sanitization) yet *well-structured* (i.e., accepted by the IoT device) fuzzing inputs, which can reach deeper code locations, uncovering more vulnerabilities.

In this Chapter, I propose DIANE: a novel fuzzing tool for IoT devices that leverages companion apps to generate under-constrained yet well-structured fuzzing inputs. The key observation behind DIANE is that there exist functions inside the companion app that can be used to generate optimal (i.e., valid yet under-constrained) fuzzing inputs. As we will see, such functions, which we call *fuzzing triggers*, are executed before any data-transforming functions (e.g., network serialization), but *after* the input validation code. Consequently, they generate inputs that are not constrained by app-side sanitization code, and, at the same time, are not discarded by the analyzed IoT device due to their invalid format.

In the remaining, first I explain the motivation behind this work, then I discuss DIANE in detail and the results we obtained after using DIANE to analyze 11 popular IoT devices. DIANE identified 11 bugs, 9 of which are *zero* days. Our evaluation also shows that without using *fuzzing triggers*, it is not possible to generate bug-triggering inputs for many devices.

5.1 Motivation

To motivate our approach and exemplify the challenges that it addresses, consider the snippet of code in Figure 5.1. The app utilizes the method `PTZ` (Line 2) to send position commands (i.e., spatial coordinates) to an IoT camera. To do this, `PTZ` invokes

```
1  // Android Java Code
2  public int PTZ(String adminPwd, int x, int y, int z){
3    //..
4      byte data[] = MsgPtzControlReq(x, y, z);
5      if (!adminPwd.contains("&") && // Input
6          !adminPwd.contains("'")){ // validation
7          SendMsg(adminPwd, camId, data);
8      }
9  }
10
11 public static native int SendMsg(String adminPwd, String camId, byte[] data);
12
13 // Java Native Interface
14 int Java_SendMsg(char* pwd, char* cam_id, Msg* msg){
15     prepare_msg(pwd, cam_id, msg);
16     notify_msg(msg);
17 }
18 // JNI - Different thread
19 void sender() {
20     Msg* msg = get_message()
21     send_to_device(msg);
22 }
```

Figure 5.1: Snippet of code that implements a sanity check on the admin password, and uses the Java Native Interface to send messages to the device. The example is based on the Wansview app in our dataset.

the native function `SendMsg` (Line 7), which prepares the data to be sent (Line 15), and stores it into a shared buffer (Line 16). In parallel, another thread reads the data from the same buffer (Line 20), and sends commands to the device (Line 21). Notice that the IoT camera requires a password to authenticate commands, and the app performs a sanity check on the password string (Lines 5 and 6). This example shows two crucial challenges that have to be faced when generating IoT inputs from the companion apps.

First, apps communicate with IoT devices using *structured data*, encoded in either known protocols (e.g., HTTP), or custom protocols defined by the vendor. Messages that do not respect the expected format are immediately discarded by the device, and,

consequently, cannot trigger deep bugs in its code. In the example, the app uses the function `prepare_msg` (Line 15) to create a correctly structured message.

Second, while it is crucial to generate correctly structured inputs, an effective approach has to avoid generating inputs that are constrained by app-side validation code. In the example, the function `PTZ` (Line 2) forbids the password to contain the characters `&` and `'`. However, the presence of these characters may be crucial in generating crash-triggering fuzzing inputs.

The insight from the authors of IoTFuzzer is to leverage the companion app to transform fuzzing inputs into a format that the device can process. This means that the input values need to be mutated before the app "packages" and sends them to the device. While this is true, our crucial insight is that the mutation indeed has to occur *before* the app packages the inputs, but *after* the app performs any input validation, which is not done by IoTFuzzer. Note that, with the expression app-side validation we refer to all types of constraints that the app imposes on the data sent to an IoT device. These constraints might be imposed by typical sanitization checks (e.g., limiting the length of a string) or by parameters hard-coded in the generated request (e.g., hard-coded attributes in a JSON object).

Our work fills this gap: We identify strategic execution points that produce inputs that are not affected by the constraints that the app logic imposes. To achieve this goal, we analyze an IoT device companion app, and focus on identifying effective *fuzzing*

triggers: Functions that, when used as entry points for fuzzing, maximize the amount of unique code exercised on the device's firmware, thus potentially triggering security-relevant bugs. Consider, as an example, the app's execution as a sequence of functions that receives data from the UI and send it over the network. On the one hand, if the fuzzed function is too close to the UI, the fuzzing is ineffective due to app-side validation that might be present later in the execution. On the other hand, picking a function too close to the point where data is put onto the network might be ineffective. In fact, some protocol-specific data transformations applied early in the execution would be skipped, causing the generated inputs to be dropped by the IoT device. In Figure 5.1 the function `sendMsg` represents a *fuzzing trigger*.

Our approach identifies these *fuzzing triggers* automatically, relying on a combination of dynamic and static analyses, without the need for any *a priori* knowledge about neither the firmware nor the network protocol used by the analyzed IoT device. Additionally, previous work [21] relies on specific sources of inputs (e.g., text boxes in the app's UI) to bootstrap its analysis, and does not mutate data generated from unspecified sources (e.g., firmware updates through the companion app triggered by a timer). Our bottom-up approach (explained in Section 5.2) does not make any assumptions on input sources and is, therefore, more generic than the related work.

The example we discussed in this Section is the simplified version of the code implemented in the Wansview app. We also note that app-side validation is prevalent in

real-world apps, and that the challenges we described do not only apply to this example: Our experiments, discussed in Section 5.4, show that addressing these challenges is fundamental, and that current state-of-the-art tools cannot effectively address them.

5.2 DIANE

While our goal is to find bugs in IoT devices, given the general unavailability of the code running in them, we focus our analysis on these devices' companion apps. Our key intuition is to identify and use, within these companion apps, those functions that *optimally* produce inputs for the analyzed IoT devices. These optimal functions efficiently produce inputs that are valid (i.e., not discarded by the IoT device) yet under-constrained (i.e., more likely to trigger bugs in the IoT device).

Automatically identifying these functions is a challenging task because the complexity of the companion apps, the usage of native code, and the presence of multiple threads rule out approaches based entirely on static analysis. Consequently, we developed an approach based on a *novel analysis pipeline* built on top of four different analyses: i) static call-graph analysis, ii) network traffic analysis, iii) static data-flow analysis, and iv) dynamic analysis of the function arguments.

In contrast with similar work [21], our approach does not make any assumption on how the app's user interface influences the data sent to the controlled IoT device,

and it avoids app-side sanitization on the generated data. Our analysis does not start by considering UI-processing functions, but, on the contrary, uses a novel "bottom-up" approach. Specifically, we start from identifying low-level functions that potentially generate network traffic, and then we progressively move "upward" in the app call-graph (i.e., from low-level networking functions to high-level UI-processing ones). This approach allows us to identify functions that produce valid yet under-constrained inputs, skipping all the sanitization checks performed by UI-processing functions. We then use these functions, which we call *fuzzing triggers*, to efficiently fuzz the analyzed IoT device, while monitoring it for anomalous behaviors, which indicates when a bug is triggered.

We implemented our approach in a tool named DIANE, depicted in Figure 5.2. DIANE works in two main phases: *Fuzzing Trigger Identification*, and *Fuzzing*. In the *Fuzzing Trigger Identification* phase, DIANE identifies optimal functions within the companion app, that, when invoked, generate under-constrained well-structured inputs for the analyzed device. Then, during the *Fuzzing* phase, these functions are used to generate data that is sent to the analyzed device using a local network connection.

Our approach is independent of the network medium used by the analyzed app. We applied it to both apps communicating with their related IoT device over WiFi and Bluetooth (Section 5.3.2). DIANE fuzzes IoT devices that receive commands through a *local* connection between the device and the companion app. Though some devices

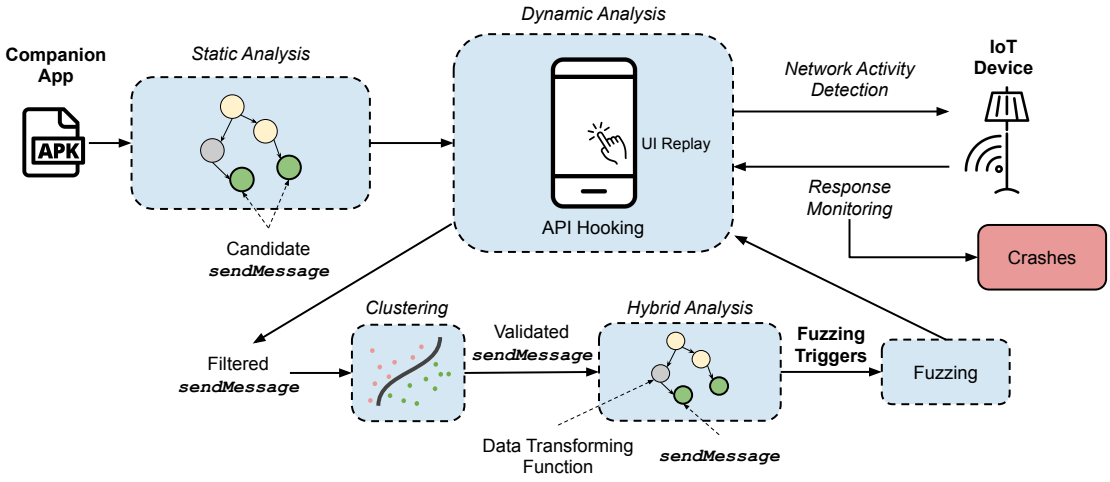


Figure 5.2: Using static analysis, DIANE first identifies candidate *sendMessage* functions. Then, it runs the companion app, replaying a recorded user interaction, to validate the candidate *sendMessage* functions. Next, DIANE uses a hybrid analysis to identify data-transforming functions and, therefore, *fuzzing triggers*. Finally, DIANE fuzzes the validated triggers and identifies crashes by monitoring the device responses.

might receive commands from cloud-based endpoints, research showed that the vast majority of them (95.56%) also allow some form of local communication (e.g., during the device setup phase) [3].

5.2.1 Fuzzing Trigger Identification

Intuitively, *fuzzing triggers* are functions that, in the app's control flow, are located in between the app-side validation logic and any data-transforming (e.g., message serialization) function occurring before sending data over the network. Precisely, given an execution trace from a source of input (e.g., data received from the UI) to the function

sending data over the network, a *fuzzing trigger* is defined as a function that dominates[1] *all* data-transforming functions and post-dominates *all* input-validating functions. We consider the first data-transforming function in the trace a valid *fuzzing trigger*, as it dominates every other data-transforming function (itself included).

Our bottom-up Fuzzing Trigger Identification algorithm is composed of four steps: i) *sendMessage* Candidates Identification, ii) *sendMessage* Functions Validation, iii) Data-Transforming Function Identification, and iv) Top-Chain Functions Collection. Algorithm 4 lists the pseudo-code of our approach.

Step 1: *sendMessage* Candidates Identification. We begin by identifying the functions, in the companion app, that implement the necessary logic to send messages to the IoT device. We call these functions *sendMessage* functions.

Identifying these functions in an automated and scalable way is difficult. Companion apps might rely on ad-hoc native functions directly invoking system calls to implement *sendMessage* functions. Furthermore, we found that these functions might be executed within separate threads, which makes it harder for any analyses (both static or dynamic) to precisely track data flows between the app's UI and *sendMessage* functions. However, our key insight is that the companion app must contain "border" functions, situated between the app core functionality and external components (i.e., the

[1]We refer to the dominance concept of the call graph theory, where a node d dominates a node n if every path from the entry node to n must go through d. Also, we say that a node p post-dominates n if every path from n to an exit node passes through p.

Android framework or native libraries), which, when executed, eventually trigger a message to be sent to the IoT device. Throughout the rest of this Chapter, we consider these border functions our *sendMessage* functions.

In our approach, we first identify *candidate sendMessage* functions by statically analyzing the companion app. We aim at finding all the border methods that might implement the network interactions with the analyzed IoT device (function getBorderMethods in Algorithm 4). Specifically, we collect all the methods that perform (at least) a call to native functions or a call to methods in the Android framework that implement network I/O functionality (see Section 5.3.1 for more details).

Step 2: *sendMessage* Functions Validation. We dynamically execute the app and leverage API hooking to validate the candidate *sendMessage* functions. In order to establish whether a border function is a valid *sendMessage* function we could, in theory, i) dynamically execute the function multiple times and check whether it generated network traffic each time, and ii) prevent the app from executing the function and check whether or not network traffic is still generated. Unfortunately, we found that preventing a function to be executed, as well as forcing the app to execute the same function multiple times, usually causes the app itself to crash. To solve these issues, we adopt a different approach, based on timestamps and machine learning.

First, we dynamically hook all the candidate functions and run the app. When we observe network activity, we register the last executed candidate *sendMessage* function.

In particular, each time a candidate *sendMessage* function is executed, we collect the elapsed time between its execution and the observed network activity. Then, we leverage the K-mean algorithm to cluster the observed elapsed time measures. Specifically, we group our candidates into two clusters (i.e., $k = 2$). To do so, we compute each feature vector as the mean, standard deviation, and mode of the elapsed times of each candidate. The rationale is that functions that cause network activity have a smaller mean and standard deviation, as they are less affected by noise. Finally, among the *sendMessage* candidates, we select those belonging to the cluster having the smallest mean of the elapsed times. Only the *sendMessage* functions within this cluster will be considered in the subsequent steps of our analysis. This approach is represented by the function dynamicFilter in Algorithm 4.

Step 3: Data-Transforming Function Identification. While *sendMessage* functions are intuitively good triggers for performing fuzzing, apps may apply data-transformations in functions executed before a *sendMessage* function. A typical example of a data-transforming function is represented by an encoding method that takes as input a list of integers and serializes it to a sequence of bytes.

As previously explained, *fuzzing triggers* are functions that, in the app's control flow, are located before any data-transforming function. Fuzzing a function located in between a data-transforming function and a *sendMessage* function would likely pro-

duce invalid inputs that are discarded by the IoT device. Thus, to find *fuzzing triggers*, we first need to identify the data-transforming functions applied to the data being sent.

This task presents different challenges. First, the data being sent might be contained in a class field, which is referenced by the *sendMessage* function. This field might be theoretically set anywhere in the app code, including within other threads. Furthermore, for each field, we need to consider its parent classes, as the variable holding the message to be sent might be inherited by a different class.

In our approach, we take into account these issues. We first statically identify the possible variables that hold the data being sent by the considered *sendMessage* function, and the code locations where these variables might be set in the app (function getArgAndObjLocs in Algorithm 4). To achieve this, we create a set S_v containing tuples (v, cl), where v is a variable used by the *sendMessage* (i.e., *sendMessage* arguments or objects referenced within the *sendMessage* body), and cl is the code location where v is set.

Then, we identify data-transforming functions. For each tuple $(v, cl) \in S_v$, we perform a static inter-procedural backward slicing (Line 6 in Algorithm 4) from cl up to any function retrieving values from any UI objects. Then, we divide the computed program slices in function scopes (getFunctionScopes at Line 7). Given a program slice, a function scope is defined as a subsequence $inst_f$ of sequential instructions in the slice that belong to the same function f.

For each collected function scope, we perform a liveness analysis [94]: We consider the variables (i.e., local variables and class fields) referenced within the function scope, and we compute the set Li_f of variables that are live at the beginning of the scope, and the set Lo_f of variables that are live at the end of the scope (Line 8). For example, if a function f is traversed by the slice, the variables that are live at the beginning of the function scope $inst_f$ are f's arguments and the class fields that are read before being written in f. The variables that are live at the end of f's scope are the returned variable and the class fields that f creates or modifies.

To identify data-transforming functions, we leverage the observation that these functions increase the entropy of the data they consume, as explored by related work [21]. Therefore, we hook the functions we identified in a slice, we dynamically run the app, and we calculate the Shannon entropy [65] of the data assigned at runtime to each variable v contained in Li_f and Lo_f (more details about how we calculate the entropy are provided in Section 5.3.3). If v is a primitive variable (e.g., `int`), or a known type (i.e., `String`, `Integer`, `Float`, and `Double`), we convert the data it contains in its byte representation and calculate the Shannon entropy of this list of bytes. Conversely, if v is a class field, we retrieve its class definition and consider each field variable v_c of v whose type is either primitive or known. Then, we compute the entropy of each one of these v_c variables, and add them to either the Li_f set or to the Lo_f set, depending on which live set v belongs.

Finally, we inspect every collected function scope and calculate the quotient d_e between the maximum entropy registered among all the variables in Lo_f and the minimum value of entropy registered among all the variables in Li_f (Line 11). If d_e is greater than a certain threshold T_f (set to 2.0 in our experiments, as previous work suggested [132]), we consider the function f to be a data-transforming function (Line 12).

Step 4: Top-Chain Functions Collection. Data-transforming functions are usually executed in precise sequences to adequately prepare the data to be sent to an IoT device. For instance, a companion app may encode some user data in base64, and then encapsulate it in an HTTP request.

We call a sequence of data-transforming functions a *transformation data chain*, and we refer to the first function in the sequence with the term top-chain function. We say that a top-chain function f *affects* a variable v if modifying the content of f's variables eventually affects v's value.

Of particular interest for us are the top-chain functions that affect *sendMessage* variables. In fact, if we control the variables of these top-chains, we can control the data sent to the analyzed IoT device. In particular, this data is both valid (i.e., accepted by the IoT device) and not affected by unnecessary app-side input validation. As such, top-chain functions that affect *sendMessage* variables represent the optimal *fuzzing triggers* to stimulate the IoT device functionality.

To identify these top-chain functions, we build the dominance tree[2] of each data-transforming function detected at the previous step (Line 13), and select those data-transforming functions that are not dominated by any other data-transforming function (Line 16). Finally, we consider as *fuzzing triggers* the collected top-chain functions.

Note that, if no data-transforming function dominates a *sendMessage* function, we consider the *sendMessage* as a *fuzzing trigger* (Line 14, 15, and 16). This could happen when, for instance, the companion app does not contain data-transforming functions.

Note finally that, in principle, app-side sanitization code might be present in a function within a transformation data chain. We discuss this in Section 5.5.

Example. As a simple example, consider Figure 5.3, which represents one of the data chains we found on the August Smart Lock device. Assuming that we previously identified `sendToDevice` as being a *sendMessage* function, we set { `c`} as the initial set of variables possibly holding data to be sent, and determine the code locations where `c` is set. As `c` is a function argument, we retrieve the *sendMessage* call site (Line 15), and bootstrap a backward program slicing from the call site, up to the function `unlock` (Line 1). This is achieved by following the data-flow of the variable `e` backward: `sendToDevice` uses the variable `e`, which is the result of a call to the function `encrypt`. Then, we continue the slice backward from the end of the function `encrypt` up to its entry point, and back to the `sendCommand` function. Finally, we

[2] A dominance tree is a graph where each node's children are those nodes it immediately dominates.

reach the entry point of this function, and continue the slice considering its caller (i.e., the function `unlock`).

Following the definition of function scopes above stated, this backward slice contains the following function scopes: i) `sendCommand`: Line 15, ii) `encrypt`: Lines from 6 to 9, iii) `sendCommand`: Lines 12 and 13 iv) `unlock`: Line 3, v) `Command` constructor: (code not reported in this example), and vi) `unlock`: Lines 1 and 2. For brevity, in the following we only consider the relevant function scopes: ii) `encrypt`, iii) `sendCommand`, and vi) `unlock`. Their sets of live variables are: `encrypt`: $Li_f = \{b\}$, $Lo_f = \{enc\}$, `sendCommand`: $Li_f = \{cmd\}$, $Lo_f = \{cmd\}$, and `unlock`: $Li_f = \{\}$, $Lo_f = \{cmd\}$.

Once we identify the function scopes in the slice, we run the app and compute the entropy of the data assigned to each of their live variables. Then, we calculate the amount of entropy introduced by each function scope and check whether its value exceeds a threshold T_f.

The function `unlock` does not introduce any entropy, as the set Li_f is empty. In the cases where the set Li_f is empty, we do not consider the function f as a candidate data-transforming function, since it does not take any input.

For the function `encrypt`, the entropy of the data stored in `b` is 5.94, whereas the entropy of the data returned in `enc` is 53.16. Since the entropy delta d_e is greater than our threshold ($d_e = 53.16/5.94 > 2.0$), we consider `encrypt` as a data-transforming

```
1  public boolean unlock() { // unlock request
2    Command cmd = new Command(OP.UNLOCK);
3    return sendCommand(cmd);
4  }
5  /* Encrypts and return its parameters */
6  public byte[] encrypt (Command b){
7    byte[] enc;
8    // ...
9    return enc;
10 }
11
12 public boolean sendCommand (Command cmd){
13   // various checks on the command to send
14   byte[] e = encrypt(cmd);
15   return sendToDevice(e);
16 }
17 /* send a message */
18 public boolean sendToDevice(byte[] c) {/* ... */}
```

Figure 5.3: Example of a simple Transformation Data Chains found on the August Smart Lock.

function. Also, the function `sendCommand` introduces a low amount of entropy ($d_e = 1.03$), and, therefore, it is not considered a data-transforming function. Finally, as the function `encrypt` dominates the function `sendToDevice`, `encrypt` is the only top-chain function, and it is used as the only *fuzzing trigger*.

UI Stimulation. Our approach executes the same app multiple times, being consistent across the different runs. This means that, ideally, we want the app to follow always the same execution paths. To achieve this goal, we require the analyst to run the app once, while DIANE records the generated UI inputs. Then, we automatically replay the same inputs in the subsequent runs, by leveraging RERAN [44]. We do not explicitly handle other sources non-determinism [25], but we found this does not significantly affect our approach.

Fuzzing Intermediate Data-Transforming Functions. In principle, transformation data chains might be arbitrary long. As DIANE's goal is to stimulate the core functional-

ity of IoT devices, our approach ignores intermediate data-transforming functions (i.e., data-transforming functions dominated by a top-chain function) as they generate messages that would likely be discarded by the IoT device. However, as IoT devices might contain bugs also in the procedures used to decode a received message, we provide DIANE with the option to fuzz also all the intermediate data-transforming functions. Likewise, DIANE provides an option to fuzz the *sendMessage* functions directly even when dominated by top-chain functions. In Section 5.4.3, we empirically show that fuzzing the *sendMessage* functions does not lead to the discovery of new bugs, while it slows down the execution of our tool.

5.2.2 Fuzzing

After the first phase of our approach, we obtain a set of *fuzzing triggers*, which are the inputs to our fuzzer.

Test Case Generation. For each *fuzzing trigger*, we generate a set of test cases by mutating the parameters of the identified *fuzzing triggers*, which eventually modify the data sent by a *sendMessage* function. We fuzz the different *fuzzing triggers* one at the time, in a round-robin fashion. To mutate the values of their parameters, we use the following strategies:

- **String lengths:** We change the length of strings in order to trigger buffer overflows and out-of-bound accesses. We generate random strings with different lengths.

- **Numerical values:** We change the values of integer, double or float values to cause integer overflows or out-of-bound accesses. We generate very large values, negative values, and the zero value.

- **Empty values:** We provide empty values, in the attempt to cause misinterpretation, uninitialized variable vulnerabilities, and null pointer dereferences.

- **Array lengths:** We modify the content of arrays by removing or adding elements.

It is important to specify that we do not only fuzz primitive variables (e.g., `int`, `float`), but we also fuzz objects (as explained in Section 5.3.2), by fuzzing their member variables.

Identifying Crashes. As shown by a recent study [87], identifying *all* crashes of network-based services of IoT devices without *invasive* physical access to the devices is challenging. At the same time, getting invasive physical access to IoT devices needs considerable engineering effort [7], since vendors usually prevent this type of access [63, 96].

For these reasons, while fuzzing a device, DIANE automatically analyzes its responses to identify crashes. Specifically, DIANE first performs a normal run of the app and monitor how the device responds during normal activity. Then, while fuzzing, DIANE monitors the network traffic between the app and the device again, and considers an input to be *potentially* crash-inducing, if any one of the following conditions is satisfied.

- **Connection dropped.** If the device abruptly ends an ongoing connection, we consider it as an indication that something wrong happened to the device. Specifically, for TCP connections, we look for cases where the app sent a `FIN` packet and received no response (FIN + ACK), and then sent a sequence of two or more `SYN` packets.

- **HTTP Internal Server Error (500).** Instances where the app and the device communicate through HTTP, and the device returns an Internal Server Error [135] (status code `500`), are considered as a signal that the device has entered in a faulty state.

- **Irregular network traffic size.** If the amount of data exchanged between the app and the device overcomes a threshold S_e, we save the current crash-inducing input. Our intuition is that, when a device enters a faulty state (e.g., due to a crash) it usually becomes temporarily unavailable for the app, thus drastically reducing

the amount of data exchanged. In our experiments, we empirically verified that when the amount of exchanged data was less than 50% (compared to a regular run), something unusual happened to the device. For this reason, we set S_e to be 50%.

- **Heartbeat Monitoring.** While fuzzing a given device, we continuously ping it and monitor its response time. We report any crash-inducing inputs causing the response time to be above a certain threshold T_p. In our experiments, we set T_p to 10 seconds, as we empirically verified that the average response time of an IoT device falls within 1 second under normal conditions.

Finally, we use an additional Android smartphone, which we refer to as the *watchdog device*, to monitor the status of the IoT device from a neutral standpoint (i.e., we do not instrument the companion app on this device). We run the companion app on the watchdog device and automatically replay the previously recorded UI inputs to exercise the different IoT device functionality at regular intervals. A human analyst can then observe whether the functionality exercised by the watchdog device (e.g., pressing the light switch UI button) caused the desired effect on the IoT device (e.g., turning the light on) or not. If an undesired effect is detected, it means that DIANE was able to bring the analyzed device into an invalid state. This feature is not fully automated, since it requires a human analyst to decide.

5.3 DIANE Implementation Details

In this Section, we provide technical details about DIANE's different components. We implemented DIANE in about 4,500 lines of Python code, following the high-level architecture depicted in Figure 5.2. DIANE is implemented on top of pysoot [3], which leverages Soot [124] to translate the companion app's bytecode into an intermediate representation. DIANE currently only handles Android applications.

5.3.1 Static Analysis

To find the initial set of *sendMessage* candidates within a companion app, we analyze its internal representation. In particular, we select all those functions that either contain calls (Soot intermediate-representation `invoke` instructions) to native methods (having the `native` attribute) or calls to methods in the Android framework known to implement network I/O operations (e.g., `java.net.*`, `javax.net.*`, or `android.net.*`). By applying these rules, we obtain a list of functions that, when invoked, potentially send network messages to the IoT device.

5.3.2 Dynamic Analysis

APK Instrumentation. To hook methods of the APK under analysis and to fuzz them, we use Frida [40]. More precisely, each method is hooked and dynamically modified

[3] `https://github.com/angr/pysoot/`

to include additional code. This injected additional code is used to enable fuzzing of the method arguments and of the used class fields and to extract information necessary for our analysis, such as the timestamp when the method is invoked and the contents of its parameters.

Network Interception. DIANE intercepts the network traffic generated by the companion app at runtime. DIANE supports the interception of traffic sent using both the WiFi and Bluetooth interfaces. Note that our approach is independent of the specific network medium and only requires to passively observe the communication channel without accessing the content of the exchanged data. For traffic transmitted over WiFi, DIANE leverages a router and the tool `tcpdump` to capture the packets sent from the smartphone to the IoT device, filtering the IP addresses. Traffic transmitted using the Bluetooth interface is instead captured using the *Bluetooth HCI snoop* Android debugging functionality [12]. Unless otherwise specified, we use the term network activity to refer both to WiFi and Bluetooth network traffic.

Fuzzing Objects. DIANE fuzzes both primitive variables (e.g., `int`, `float`) and class instances. To do this we use pysoot to retrieve the class definition of the considered class instances, and we fuzz each field whose type is either primitive or known (e.g., `java.lang.String`).

5.3.3 Hybrid Analysis

Fuzzing Trigger Identification Details. To implement the *fuzzing triggers* algorithm described in Section 5.2.1, we implemented a static inter-function backward slicer on top of pysoot. Theoretically, the backward slice of a given variable might traverse an arbitrary number of functions. Therefore, to keep our analysis tractable, our backward slicer algorithm adopts a conservative approach.

Specifically, when calculating the backward slice of a variable v, our backward slicer traverses up to N consecutive function calls (we set N to five in our experiments), and it over-approximates data dependencies when a function call is not followed. For instance, if a function call takes v as one of its arguments, and the function call is not followed, we assume that v is data-dependent on all the other arguments. Although this approach might lead our static analysis phase to produce false positives, it does not affect the performance of our tool, since, as explained in Section 5.2.1, we use dynamic analysis to validate the results produced by static analysis.

To build the data-transforming function dominator trees, as explained in Section 5.2.1, we first need to build the companion app call graph. To achieve this, we perform intra-procedural type inference [97] to determine the possible dynamic types of the object on which a method is called. When this fails, we over-approximate the possible targets as all the subclasses of its static type.

Entropy Calculation Details. To find data-transforming functions, DIANE needs to calculate the entropy of each variable v within the live sets Li_f and Lo_f of a function scope f. To achieve this, if v is a primitive variable (e.g., `int`), or a known type (i.e., `String`, `Integer`, `Float`, and `Double`), we convert the data it contains in its byte representation and calculate the Shannon entropy of this sequence of bytes. Note that the entropy is computed on the entire sequence of bytes, rather than the single bytes considered separately.

Conversely, if v is a class object, we use pysoot to retrieve its class definition, and we consider each field variable v_c whose type is either primitive or known. For all these field variables, we compute their entropy as specified above, and we add them to the Li_f set or to the Lo_f set, based on to which live set v belongs.

5.4 Experimental Evaluation

In this Section, we answer two research questions:

1. Is DIANE *able* to find both previously-known and previously-unknown vulnerabilities in IoT devices effectively?

2. Is DIANE *needed* to find vulnerabilities in IoT devices effectively, or can existing (app-based or network-level) fuzzers achieve similar results?

Table 5.1: Summary of our dataset of IoT devices (* account required to operate the device).

DeviceID	Type	Vendor	Model	FirmwareVers.	Android AppPackage Name	App Vers.	Online Account*	Setup Time [Seconds]
1	Camera	Wansview	720P X Series WiFi	00.20.01	`wansview.p2pwificam.client`	1.0.10	✗	219
2	Camera	Insteon	HD Wifi Camera	2.2.200	`com.insteon.insteon3`	1.9.8	✓	427
3	Smart Socket	TP-Link	HS110	1.2.5	`com.tplink.kasa_android`	2.2.0.784	✗	311
4	Camera	FOSCAM	FI9821P	1.5.3.16	`com.foscam.foscam`	2.1.8	✓	406
5	Camera	FOSCAM	FI9831P	1.5.3.19	`com.foscam.foscam`	2.1.8	✓	403
6	Smart Socket	Belkin	Wemo Smart Socket	2.0.0	`com.belkin.wemoandroid`	1.20	✗	211
7	Bulb	iDevices	IDEV0002	1.9.4	`com.idevicesllc.connected`	1.6.95	✗	274
8	Smart Socket	iDevices	IDEV0001	1.9.4	`com.idevicesllc.connected`	1.6.95	✗	276
9	Camera	Belkin	NetCam	Unknown	`com.belkin.android.androidbelkinnetcam`	2.0.4	✓	1,040
10	Bulb	LIFX	Z	2.76	`com.lifx.lifx`	3.9.0	✓	313
11	Smart Lock	August	August Smart Lock	1.12.6	`com.august.luna`	8.3.13	✓	213

To answer the first research question, we first evaluated DIANE precision in detecting *fuzzing triggers* (Section 5.4.2) and then we used it to fuzz 11 different IoT devices (Section 5.4.3). Our system found 11 bugs in 5 devices, including 9 zero-day vulnerabilities, running, in all cases, for less than 10 hours (Section 5.4.8).

To answer the second research question, we first compared our tool with IoT-Fuzzer [21] by running it on the 11 analyzed devices (Section 5.4.4). Our experiment shows that DIANE outperformed IoTFuzzer in 9 devices, and performs as well as IoTFuzzer for the remaining 2 devices. Then, we performed a larger-scale automated study (Section 5.4.5) to measure how often companion apps perform app-side validation, which would limit the efficiency of approaches like IoTFuzzer. Our experiment revealed that 51% of the analyzed apps contain, indeed, app-side sanitization. Finally, we compared DIANE with existing network-level fuzzers (Section 5.4.6), and showed that network-level fuzzers are unable to find bugs in the analyzed devices.

We conclude this Section by presenting a detailed case study about two zero-day bugs DIANE found in one of the analyzed devices (Section 5.4.7).

5.4.1 Dataset & Environment Setup

To evaluate DIANE, we used popular real-world IoT devices of different types and from different brands. Specifically, in October 2018 we searched for "smart home devices" on Amazon and obtained the list of the top 30 devices. Among these, we excluded 5 expensive devices (price higher than 200 USD), 1 device that does not communicate directly with the companion app (the communication passes through a Cloud service), and other 13 devices because they require other appliances (e.g., a smart ceiling fan controller).

Our dataset contains the remaining 11 devices, which are listed in Table 5.1. This dataset encompasses devices of different types (cameras, smart sockets, bulbs, smart locks). Note that the respective companion apps of these devices are quite complex as they contain, on average, over 9 thousand classes, 56 thousand functions, and 766 thousand statements. The complexity of these apps is in line with the complexity of the apps used by the related work [131], which contains the largest dataset of validated IoT apps.

We installed the IoT devices in our laboratory, we deployed DIANE on an 8-core, 128GB RAM machine running Ubuntu 16.04, and we ran the Android companion apps

Table 5.2: Summary and features of our dataset of IoT companion apps. TP indicates a true positive result, FP a false positive result, and NC a result we were not able to classify either as true positive nor false positive. **?** indicates that we could not verify whether an app applied data sanitization. The last three columns indicate the complexity of the apps in terms of number of classes, functions, and statements respectively.

Device ID	Network Protocol	Native Code	Sanity Checks	No. Candidate sendMessage	No. sendMessage	No. Fuzzing Triggers	No. Classes	No. Functions	No. Statements
1	UDP	✓	✓	4 (1 TP, 3 FP)	1 (1 TP)	7 (6 TP, 1 FP)	4,341	31,847	409,760
2	HTTP	✓	✓	12 (8TP, 4FP)	9 (6 TP, 3 FP) ⋆	6 (6 TP)	11,870	76,558	1,180,817
3	TCP + JSON	✗	?	6 (2 TP, 4 FP)	6 (2 TP, 4 FP)	3 (2 TP, 1 FP)	16,461	107,935	1,267,785
4	UDP	✓	✓	10 (2 TP, 7 FP, 1 NC)	2 (2 TP)	2 (2 TP) •	6,859	41,256	615,410
5	TCP	✓	✓	10 (2 TP, 7 FP, 1 NC)	2 (2 TP)	2 (2 TP) •	6,859	41,256	615,410
6	HTTP + SOAP	✓	✗	15 (3 TP, 12 FP)	6 (2 TP, 4 FP)⋆	9 (8 TP, 1 FP)	4,169	30,462	378,733
7	TCP	✓	✓	8 (2 TP, 6 FP)	3 (2 TP, 1 FP)	4 (3 TP, 1 NC)	8,418	52,013	813,444
8	TCP	✓	✓	8 (2 TP, 6 FP)	3 (2 TP, 1 FP)	4(3 TP, 1 NC)	8,418	52,013	813,444
9	TCP	✓	?	6 (3 TP, 3 FP)	1 (1 TP)⋆	1 (1 TP)•	6,010	42,358	467,670
10	UDP	✓	?	9 (1 TP, 8 FP)	3 (1 TP, 2 FP)	0	5,646	33,267	457,719
11	Bluetooth	✓	✓	9 (4 TP, 5 FP)	9 (4 TP, 5 FP)	16 (14 TP, 2 FP)	22,406	108,507	1,411,798
Total		10/11	7/11	97 (30 TP, 65 FP, 2 NC)	45 (25 TP, 20 FP)	54 (47 TP, 5 FP, 2 NC)	101,457	617,472	8,431,990

• *fuzzing triggers* coincide with *sendMessage* functions.

on a Google Pixel and a Google Nexus 5X running Android 8.0. The smartphones, the IoT devices, and the machine running DIANE were connected to the same subnet, allowing DIANE to capture the generated network traffic. To configure each device, we manually performed its initial setup phase, such as registering an account on the device and on the Android companion app.

Table 5.3: Summary of the bugs detected by DIANE, IoTFuzzer, and by existing network fuzzers (BED, Sulley, uFuzz, and bss). No. Generated Alerts indicates the number of unique *fuzzing triggers* for which DIANE automatically generated at least one alert. Time indicates the time required to find all the reported bugs (and the number of fuzzing input generated to find the bugs). No. fuzzed functions indicates the number of functions identified by IoTFuzzer for fuzzing.

	DIANE					IoTFuzzer			Other Fuzzers			
Device ID	No. Generated Alerts	No. Bugs	Zero-day	Vuln. Type	Time [hours] (No. Generated Inputs)	No. Fuzzed Functions	No. Bugs	Time [hours]	No. Bugs			
									BED	Sulley	uFuzz	bss
1	1	1	✓	Unknown	$\leq$ 0.5 (60,750)	• 1	0	N/A	N/A	0	N/A	N/A
2	3	7	✓	Buff overflow	$\leq$ 0.5 (322)	5	2	0.98	0	0	N/A	N/A
3	1	1		Unknown	$\leq$ 1.2 (7,344)	1	1	4	0	0	N/A	N/A
4	1	0		N/A	N/A	• 1	0	N/A	N/A	0	N/A	N/A
5	1	0		N/A	N/A	• 1	0	N/A	0	0	N/A	N/A
6	4	1		Unknown	$\leq$ 10 (34,680)	1	1	$\leq$ 10	0	0	0	N/A
7	3	0		N/A	N/A	N/A	N/A	N/A	0	0	N/A	N/A
8	3	0		N/A	N/A	N/A	N/A	N/A	0	0	N/A	N/A
9	0	0		N/A	N/A	3	0	N/A	0	0	0	N/A
10	1	0		N/A	N/A	N/A	N/A	N/A	N/A	0	N/A	N/A
11	0	† 1	✓	Unknown	2.2 (3,960)	N/A	N/A	N/A	N/A	N/A	N/A	0

• We manually instrumented IoTFuzzer to identify a valid send function.

† Vulnerability discovered through the watchdog device.

5.4.2 Fuzzing Trigger Identification

Table 5.2 shows the results of each step of DIANE's *fuzzing triggers* identification phase: For each IoT device, we report the protocols in use to communicate with the companion app, whether or not the app contains native code, if it sanitizes user inputs, the number of candidate *sendMessage* functions found by DIANE, the number of validated *sendMessage* functions, and the number of *fuzzing triggers*. For each intermediate result, we calculated the number of true positives and false positives, and investigated false negatives.

Since there is no available ground truth, we validated our ability to identify *sendMessage* functions and *fuzzing triggers* by manually reversing (both statically and dynamically) the Android companion apps in our dataset. Specifically, an expert analyzed each app for an average of five hours.

Reverse engineering of real-world apps is known to be difficult. Therefore, while we did our best to fully comprehend the dynamics of these apps, in a few cases we could not verify our results completely, as indicated in the following Sections. We also acknowledge that this manual evaluation cannot completely exclude the presence of false negatives.

To measured DIANE's ability to find *sendMessage* functions precisely, we manually analyzed the *sendMessage* functions returned by the first two steps of our analysis. Specifically, we classified each function returned by the *sendMessage* candidates identification step (Step 1 in Section 5.2.1) and by the *sendMessage* function validation step (Step 2 in Section 5.2.1) as either true positive or false positive (TP and FP in the fifth and sixth columns of Table 5.2). To perform this classification, we hooked each of these functions and manually exercised the IoT device's functionality through its companion app, while monitoring the network traffic. We considered a candidate *sendMessage* function a true positive if: i) We registered network traffic when the companion app invoked the *sendMessage* function, and ii) the code and semantic of the function indicated network functionality. If either of these two conditions were false, we considered

the *sendMessage* function a false positive. There were cases where the app was heavily obfuscated, and we could not establish if the considered *sendMessage* function was indeed sending data (NC in Table 5.2).

As shown in Table 5.2, DIANE was able to remove 45 false positive results during its *sendMessage* function validation step, . For Device IDs 2, 6, and 9 (indicated with $\star$), one might think that we lost some true positives during the validation step. However, this was not the case. After manual verification (using both static and dynamic analyses), we discovered that the missing true positives were just wrappers of other validated true positives. We also looked for false negatives, that is, *sendMessage* functions that were not identified as such. To the best of our ability, we found no such false negatives.

Overall, though we registered some false positives (20 in total), we always identified correctly *sendMessage* functions (i.e., no false negatives). We investigated the false positives and we found that they were due to border functions containing calls to native methods, which were called within (or right before) the correct *sendMessage* functions. As such, their execution times were close to the actual *sendMessage* functions, causing our *sendMessage* validation step to label them as valid *sendMessage* functions. Also, it is important to say that false positive results do not affect the effectiveness of DIANE (i.e., the number of bugs found), rather its efficiency (i.e., the time spent to find those bugs). In fact, considering a non-*sendMessage* function as a *sendMessage* would only

result in identifying additional, wrong *fuzzing triggers* that would not generate any network traffic when fuzzed, thus not affecting the IoT device.

For each true positive *sendMessage* function, we verified that DIANE correctly identified the top-chain functions (i.e., *fuzzing triggers*). *Fuzzing triggers* for Device IDs 4, 5 and 9 (marked with •) coincided with the *sendMessage* functions. This happens in apps that either do not have data-transforming functions, or where the functions that transform the data also embed the send functionality. Consequently, these functions are both *sendMessage* and top-chain functions.

For three apps (Device IDs 3, 9 and 10), we could not trace the data-flow from the identified *sendMessage* functions back up to the UI elements. This was due to imprecisions of the employed reverse engineering tools. Therefore, we could not establish whether they performed app-side data sanitization.

We also investigated false positives and negatives in the identified *fuzzing triggers*. Overall, our transformation data chain identification algorithm generated 5 false positives. In 2 cases, our backward slicer could not find any callers of a given function, and, therefore, our algorithm ended and considered the last detected data-transforming function f a *fuzzing trigger*. After manual verification, we found that the correct *fuzzing trigger*, in both cases, was a caller of function f. Although f is a valid data-transforming function, DIANE cannot assure that it is a top-chain function, as there might be another data-transforming function calling f that dominates f. The remain-

ing 3 false positives were due to the fact that these functions introduced an entropy higher than our threshold, though they were not data-transforming functions. However, we maintained our threshold to 2 as this value is indicated as optimal by related work [132]. As we explained before, these false positives do not influence the effectiveness of DIANE, but only its efficiency.

Finally, we evaluated the false negatives generated by DIANE. To the best of our ability, we did not encounter any false negative while manually reversing the apps.

5.4.3 Vulnerability Finding

Finally, we fuzzed the obtained *fuzzing triggers*, and verified the alerts produced by our tool. Table 5.3 shows the results of our fuzzing. Note that, while DIANE can also use *sendMessage* functions as entry points for fuzzing, it identified all the detected bugs only when leveraging *fuzzing triggers*. We discuss the human effort required to verify the alerts produced by DIANE in Section 5.4.9.

We validated our findings as follows. The seven bugs for Device ID 2 were confirmed by analyzing both the network traffic and the camera firmware. Through the analysis of the firmware, we were able to verify our findings and craft a proof-of-work exploit that stalls the device for an arbitrary amount of time. We reported these bugs to the manufacturer, who confirmed our findings.

As for Device ID 1, after finding the candidate crash input, we verified it, through the app, by observing how the device behaved. We noticed that, after sending the crafted input, the device did not respond anymore, unless it was rebooted. Also, after fuzzing it for 24 hours the device entered a malfunctioning state, and we were unable to correctly restore it, even after multiple factory resets. We then purchased another camera of the same model, and the same result was obtained after 24 hours. We are still investigating to find whether some crash-inducing inputs we provided also cause irreparable damage to the device.

When validating the crash reports for the Device ID 3, we noticed that, after sending the crash-inducing input, the TCP connection was dropped, and the device response time significantly increased. We found that this bug, as well as the bug affecting the Device ID 6, were known vulnerabilities [21].

For Device ID 11 (a popular smart door lock), we noticed that after around two hours of fuzzing the device became unreachable for the watchdog device. Even more interestingly, the device then started to make an intermittent noise, which we realized being "SOS" encoded in morse code[4]. We then reset the door lock, and we observed that it started to show erratic behavior. For example, we noticed that it was not possible to control it through two different Android phones anymore: If the lock status was shown as "online" on one companion app, it would be "unreachable" on the same companion

[4]Audio recording: `https://drive.google.com/file/d/1j9ydwO9CWuC3d-HxcMDeZga6m3sSrg5l/view`

app on another phone. We are still working with the vendor to find the root cause of the problem.

We reported our findings to the appropriate manufacturers and we are in contact with them to disclose all the details.

5.4.4 DIANE vs. IoTFuzzer

To compare our approach to IoTFuzzer [21], we contacted the authors and obtained their tool. We also attempted to purchase the same devices used to evaluate IoTFuzzer, but we could only obtain Device 3 and Device 6, as the remaining ones are only available in China.

IoTFuzzer required manual intervention to be adapted to different devices and companion apps. In particular, we had to i) limit the scope of the analysis (i.e., number of hooked functions) to a subset of Java packages present in the Android apps—to keep the analysis tractable and avoid crashes—and ii) manually specify any encryption functions present in the app. After this manual configuration step, we were able to replicate the results presented in the original paper for the devices we were able to obtain (Device 3 and Device 6). Additionally, IoTFuzzer is based on TaintDroid, whose latest release supports up to Android 4.3 (2012). For this reason, we were not able to analyze Device 10 and Device 11, as their companion apps require newer Android SDK versions.

Our results are reported in Table 5.3. IoTFuzzer crashed Device 3 and 6 (the two devices used in the original paper) and Device 2, but failed to find any bugs for the other 8 devices.

For Device 2, IoTFuzzer identified 5 functions to fuzz. We manually analyzed these functions and found that three of them were false positives, as they were used to save user information on the Android phone. To confirm our findings, we fuzzed these functions and observed that none of them generated network traffic.

Then, we proceeded to fuzz the two remaining functions `HouseExtProperty` and `changeCameraUsernamePassword`. While fuzzing the former function for an hour, we discovered that the generated messages were directed to the vendor's cloud, rather than the actual device, therefore not producing any meaningful fuzzing input for the IoT device.

The `changeCameraUsernamePassword` function is, instead, used to change the credentials on the IoT device. We fuzzed this function for 24 hours, and IoTFuzzer rediscovered 2 of the 7 bugs that DIANE found on this device.

To understand better why IoTFuzzer missed some of the bugs we found, we examined `changeCameraUsernamePassword` (shown in Figure 5.4). This function calls the functions `cam.changeUsername` and `cam.changePassword` to generate the requests to change the username and password, respectively (the first argument of these functions represents the current username of the camera). Also, the variable

`cam` is an internal structure that the app uses to store the details of the camera (e.g., the camera model), and its content is not directly influenced by the data received from the app's UI. On the other hand, both `newUsr` and `newPwd` contain user data, which is passed through the app's UI. As IoTFuzzer fuzzes only the function arguments that contain user data (when a function is invoked), it fuzzes the second and third function arguments, but it does not fuzz the first.

Unfortunately, as we explain in detail in Section 5.4.7, this camera contains a bug that can be exploited if the request generated by the companion app contains a username whose length is larger than a particular buffer size. However, by fuzzing the second two arguments of `changeCameraUsernamePassword` IoTFuzzer only mutate the second parameter of `cam.changeUsername` and `cam.changePassword`—`newUsr` and `newPwd` respectively—and it does not mutate their first parameter (i.e., `cam.user`), which would lead to the discovery of an additional bug. This case highlights a limitation of IoTFuzzer's approach, as it shows that assuming that all the data being sent to the device comes directly from the app's UI is ineffective to find bugs in an IoT device. On the other hand, our bottom-up approach, which bootstraps its analysis from *sendMessage* functions (see Section 5.2), is agnostic with respect to the sources of input, and, therefore, is more generic.

In addition, `changeCameraUsernamePassword` allows one to modify the credentials only for specific camera models (Line 2, `cam.checkCameraModel`).

```
1 boolean changeCameraUsernamePassword(Camera cam, String newUsr, String newPwd) {
2   if(cam.checkCameraModel()) {
3     if(cam.user.compareTo(newUsr) != 0)
4       cam.changeUsername(cam.user, newUsr);
5     if(cam.pwd.compareTo(newPwd) != 0)
6       cam.changePassword(cam.user, newPwd);
7   }
8   //...
9 }
```

Figure 5.4: Fuzzing function found by IoTFuzzer for the Insteon camera (Device ID 2). We report only the relevant code for space reasons.

This means that IoTFuzzer cannot effectively fuzz other camera models. By identifying a *fuzzing trigger* deeper in the control flow, DIANE, instead, bypasses this check and is effective independently from the device version.

For Device IDs 7 and 8, IoTFuzzer caused the app to crash immediately due to the number of hooked functions. We narrowed the analysis to only the package containing the code to interact with the device, but the app would crash regardless. Thus, we could not run IoTFuzzer on these devices.

For Device ID 9, IoTFuzzer identified 3 functions to fuzz. However, we found these functions to be false positives, as they were used to log user data on the smartphone.

For Devices IDs 1, 4, and 5 (marked with • in Table 5.3) IoTFuzzer failed to identify any functions to fuzz. The reason is that to find a function to fuzz, IoTFuzzer has to first find a data flow between a UI element of the app and the Android's socket send function. However, in these devices the "send" functionality is implemented in native code (i.e., these devices do not rely on the Android's send function). As IoTFuzzer cannot identify send functions in native code, it failed to identify what UI events would

```
1 void changeCredentials(String newUsr, String newPwd) {
2   if(this.confirm_credentials()) {
3     if(!this.get_user().equals(newUsr) && !this.get_pwd().equals(newPwd))
4       this.changeUserAndPwd(newUser, newPwd);
5     //...
6   }
7 }
```

Figure 5.5: Fuzzing function found by IoTFuzzer for the Foscam cameras companion app (Device IDs 4 and 5).

eventually generate network traffic, and, therefore, it did not generate any valid fuzzing inputs. DIANE overcomes this limitation by using dynamic analysis, and find the border functions that generate network traffic, as explained in Section 5.2.1.

To help IoTFuzzer and have a direct comparison with our tool, we hard-coded the send functions found by DIANE in IoTFuzzer, and re-ran the analysis for these devices. For Device IDs 4 and 5, IoTFuzzer identified one candidate function to fuzz, which, similarly to Device ID 2, is used by the app to change the device's credentials. This function is depicted in Figure 5.5, and it implements a check (through `confirm_credentials`) that asks the user to provide their credentials in order to proceed. As a result, fuzzing `changeCredentials` did not produce any meaningful input to the camera, as the check would constantly fail. Instead, DIANE identified as a fuzzing trigger the function `changeUserAndPwd`, which is not affected by any checks, and effectively sends commands to the camera when fuzzed. These cases highlight another limitation of IoTFuzzer's approach, as they show that fuzzing the first function in the app's control flow that handles user-provided data is ineffective.

For Device ID 1, IoTFuzzer identified a function called `setUser`, which sends the user's login information to the device. In this case, this function is guarded by a check that forbids the user's password to contain some special characters (e.g., "&"). We fuzzed this function for 24 hours and we did not register any anomaly in the device. Also in this case, DIANE selected a function deeper in the control flow of the app, after any client-side checks. This was necessary to successfully discover a (zero-day) bug.

Overall, DIANE performed as well as IoTFuzzer only in two cases (Device IDs 3 and 6), and it outperformed IoTFuzzer in all the other cases—either because IoTFuzzer was unable to identify any meaningful send functions, or because it did not produce any crash-inducting input.

This evaluation highlights the importance of carefully selecting the right function to fuzz within the companion app, and that app-side sanitization checks hinder the efficacy of a fuzzing campaign. This issue is exacerbated by the frequency in which app-side sanitization is present in companion apps. For instance (as shown in Table 5.2), in our dataset we found that at least 7 out of 11 apps contain sanity checks. We further measure this aspect in Section 5.4.5.

5.4.5 App-side Sanitization and Fuzzing Triggers

App-side Sanitization. To evaluate how common app-side sanitization code is in companion apps, we first manually reverse-engineered the 11 companion apps of the IoT

devices in our dataset. As shown in the *Sanity Checks* column of Table 5.2, at least 7 out of 11 apps contain sanity checks.

As an additional evaluation of this aspect, we performed a large-scale study on the presence of app-side sanitization code in companion apps. For this experiment, we used 2,081 apps, which we gathered from related work [131]. This dataset is ideal for our evaluation as it specifically contains Android companion apps of IoT and smart home devices, which have been collected from the Google Play Store and manually inspected by the authors of the related work. To the best of our knowledge, this is the largest dataset of validated IoT companion apps. Since we did not have access to all the physical devices that these apps interact with, we could not run DIANE against them, and, therefore, we implemented a fully-static automated approach, suitable for a large-scale study.

Specifically, given a companion app, we identified its *sendMessage* functions by locating functions that contained I/O operations (as detailed in Section 5.3.1). We were able to identify *sendMessage* functions for 1,304 of the apps (˜63%). For the remaining apps, we were not able to statically identify any network-related operations, as we could not find, for instance, a socket send operation. Then, we performed an inter-procedural backward slice from every argument of each identified *sendMessage* function, and considered the instructions in each slice. Finally, we counted the comparisons against constant data (e.g., using a string comparison in a `if` statement) in these slices.

In this experiment, we found that 663 (˜51%) companion apps implement sanitization of the data being sent, and that, on average, the variables handled by a *sendMessage* function are affected by 7 checks across the companion app. To validate these results, we randomly selected 100 *sendMessage* functions and found 85 to be true positives, 14 to be false positives (these functions were sending messages to another Android thread), and for 1 of them we could not determine its functionality, as it was heavily obfuscated. Also, we randomly sampled 30 functions that we detected were applying input sanitization code, and found 29 to be true positives: the companion app applied checks on the user data.

These results show how app-side sanitization code is common in companion apps. Note that, this experiment is only an approximation of our approach, which requires the physical devices to be fully effective. Therefore, these results do not aim to evaluate our approach, rather they serve as an indication of the presence of input validation code in mobile apps. Our results are in line with a recently published study [149].

Fuzzing Triggers. We also evaluated how prevalent *fuzzing triggers* are in Android companion apps. As DIANE relies on dynamic analysis to find *fuzzing triggers*, we replaced the parts of our approach that leverage dynamic analysis with symbolic execution. We used the Java support provided by the angr [115] tool to symbolically execute the app's functions in a slice (see Algorithm 4), so to calculate the Shannon entropy. In particular, we concretize the input of a function (i.e., its live variables) with known

values, symbolically execute the function, and observe the values in the output (i.e., its live variables when the function returns). Then, we replicate our approach explained in Section 5.2.1, and calculate the difference of entropy introduced by each function to identify the data-transforming functions.

We sampled 100 apps from the 2,081 aforementioned apps, ran our analysis, and manually verified the results. For 37 apps, our analysis found *fuzzing triggers*, and for the remaining 63, it did not. We investigated our results and found that our analysis correctly identified a *fuzzing trigger* for 25 of the 37 apps, and it produced false positives in the remaining 12 cases. These false positives were due to imprecisions in our inter-procedural backward slicer (i.e., our static analysis could not find the callers of a given function).

On the other hand, in 63 apps our analysis did not find any *fuzzing trigger* because of imprecisions of the symbolic execution. In fact, to keep the analysis tractable, we symbolically execute every function up to 10 minutes and follow up to 2 consecutive function calls (we drop the collected symbolic constraints when a function call is not followed). As such, when the analysis fails to calculate the added entropy of a given function, we stop the analysis.

Overall, we found *fuzzing triggers* for 25% of the analyzed apps. While this number sufficiently demonstrates that such sweet spots are, indeed, present in many apps, we highlight that, in our analysis, this is a lower bound. In fact, our attempt to emulate our

approach using symbolic execution introduces imprecisions that would not occur when using DIANE together with the real devices. Therefore, we expect this number to be even higher in practice. This further emphasizes the need for a system that can identify fuzzing triggers that are located past client-side checks in the companion apps.

5.4.6 DIANE vs. Network-Level Fuzzing

We also compared DIANE to well-known network fuzzers: BED [66], Sulley [13], uFuzz [122] (UPnP endpoints), and bss [112] (Bluetooth fuzzer). Table 5.3 shows the results of the comparison. Note that the labels N/A indicate that the corresponding network fuzzer does not handle the network protocols employed by the corresponding IoT device.

We configured BED and Sulley as indicated by previous work [21], and the remaining tools as suggested by their related web pages. We ran each tool for 24 hours. However, uFuzz finished its fuzzing cycle before the allocated time, and bss was not able to generate input for Device ID 11, as the device does not accept connections outside the companion app.

Overall, no bugs were found by any of these network fuzzing tools. The reason why no network fuzzers triggered any crash is that these fuzzers are general-purpose [98, 138], and they fail to trigger deeper code paths in the devices' firmware. For instance,

```
1  public static String httpRequest(String req, ...){
2    // perform the requested HTTP request
3  }
4  /* Camera class */
5  private Result sendCommand(String cmd, TreeMap t){
6    String fmt = "http://%s/CGIProxy.fcgi?cmd=%s:%s";
7    toSend = String.format(fmt,CAMERA_ENDPOINT, cmd);
8    Iterator it = t.keySet().iterator();
9    while(it.hasNext()) {
10     String key = (String)it.next();
11     String val = (String)t.get(key);
12     toSend += "&" + key + "=" + this.encodeUrlParam(val);
13   }
14   String encUser = this.encodeUrl(this.user);
15   String encPwd = this.encodeUrl(this.passwd);
16   fmt = "&usr=%s&pwd=%s"
17   toSend += String.format(fmt, encUser, encPwd);
18   HttpUtil.httpRequest(toSend,"GET",null,10,10);
19 }
20
21 public boolean changePassword(String user, String newPwd) {
22   TreeMap t = new TreeMap();
23   v0.put("usrName", user);
24   v0.put("newPwd", newPwd);
25   res = this.sendCommand("changePwd", t).resCode;
26   return res != ResCode.SUCCESS ? false : true;
27 }
28
29 boolean changeCameraUsernamePassword(Camera cam, String newUsr, String newPwd) {
   /*...*/ }
```

Figure 5.6: Snippet of code for the Insteon Camera app.

BED only fuzzes HTTP headers without considering the syntax or the semantics of HTTP payloads.

5.4.7 Case Study: Insteon HD Wifi Camera

In this Section, we present a case study regarding two bugs that DIANE found in the Insteon Camera (Device ID 2).

Among the functionality offered by the app, a user can change their credentials (username and password). Figure 5.6 depicts a simplified version of the app's code that accomplishes this task (we omit the code of the function `changeCameraUsername`

`Password` as it is already shown in Figure 5.4). In particular, when the user wants to change their password, the companion app invokes the function `changeCameraUser namePassword` (Line 29). As explained in Section 5.4.4, this function first checks that the camera belongs to a certain camera family, and if so, the app invokes the function `changePassword` (Line 21). This function creates a TreeMap structure containing couples "key:values," which will be placed in the request generated by the app. Then, `changePassword` invokes the `sendCommand` function (Line 5), which is a helper function used to send commands to the camera. This function prepares the request by using the TreeMap, and it eventually calls `httpRequest` (Line 1) to send the request to the device.

For this particular device, we could gather the firmware running on the camera (by sniffing the wireless network during the initial firmware update). Figure 5.7 shows a simplified version of the firmware function used to copy the values of parameters from a given URI. This function acts as an unsafe `strcpy`: it takes as input a destination buffer (allocated by the caller function) and copies the value of a pair "key:value" present in a given URI. This function is called 789 times within the Insteon firmware, and, to the best of our knowledge, for 9 of them, we can trigger a buffer overflow. In particular, when a user wants to change the camera password, the firmware allocates two buffers on the stack, and it uses this function to copy the username and new password values from the URI into the allocated buffers (of 88 and 64 bytes respectively).

```
1 int key_strcpy(char *dst, char *URI, char *key){
2   int len = 0;
3   char* val = get_ptr_val(URI, key, &len);
4   strncpy(dst, val, len);
5   return 0 ;
6 }
```

Figure 5.7: Simplified snippet of code from Insteon firmware.

As a result, if we provide two values for username and password large enough, we can trigger two buffer overflows.

By looking at the code in Figure 5.6 and Figure 5.4, we can see that fuzzing the function `encodeUrl` (Lines 12,14, and 15 in Figure 5.6) allows us to i) skip any app-side validation (Line 2 in Figure 5.4), and ii) trigger both bugs discovered by IoT-Fuzzer (as shown in Section 5.4.4) and two additional bugs due to a long username and password.

DIANE identified 9 different *sendMessage* functions (6 true positives), and 6 *fuzzing triggers* (6 true positives) for the Insteon Camera companion app. Among these, DIANE automatically identified the function `httpRequest` as a *sendMessage* function, and the function `encodeUrl` as a *fuzzing trigger*. When DIANE fuzzed `encodeUrl`, DIANE immediately generated an alert.

Finally, note that the `sendCommand` represents another valid *fuzzing trigger* for `httpRequest`, as it modifies the command being sent. Indeed, DIANE correctly identified `sendCommand` as a further *fuzzing trigger*.

5.4.8 Runtime Performance

We assessed the runtime performance of our tool by measuring the execution time required by the *fuzzing triggers* identification phase. In our experiments, we setup DIANE to run the fuzzing phase for 24 hours. First, we measured the entire execution time required, on average, for DIANE to analyze an app and identify *fuzzing triggers*. DIANE analyzes a given app in slightly less than 150 minutes on average. Figure 5.8 shows the average and standard deviation of the execution time required for each phase of our analysis process. As shown in Figure 5.8, the execution time of DIANE has a high standard deviation. This is due to the following implementation detail: Frida, which we leverage to hook Android APIs and methods at runtime, sometimes fails, causing the running app to crash. This requires automatically restarting the hooking procedure, randomly slowing down DIANE.

5.4.9 Quantifying Required Human Effort

We evaluated the human effort required to use DIANE. In general, DIANE scales linearly with the number of analyzed devices, as it requires the analyst to perform the same steps for each new analyzed device.

DIANE requires human intervention to setup a new IoT device. During this phase, an analyst has to install the IoT device, which involves installing the companion app, configure the device, and, in some cases, register an online account. In addition, during

this phase, DIANE requires the analyst to run the app and test the basic functionality of the IoT device, so that our tool can record the generated UI interactions (see Section 5.2). We measured the time we spent to setup each device, as reported in Table 5.1. On average, we spent 6 minutes and 12 seconds to setup a new IoT device, of which 41 seconds were spent to interact with the device. Note that, an analyst has to take these steps only once per device.

Human effort is also required if the analyst desires to monitor the state of the watchdog device during fuzzing (recall Section 5.2.2). The watchdog device is optional and useful if the analyst wants to detect semantic issues. In this case, the analyst has to check whether the functionality automatically exercised by the watchdog device results in an undesired effect in the IoT device due to our fuzzer triggering a vulnerability (e.g., unauthenticated requests suddenly open a door lock). The frequency of these manual checks depends on the analyst, as they might want to monitor the watchdog device at regular intervals for the whole duration of the fuzzing campaign, or only at the end of it. In our fuzzing campaign we checked the watchdog device approximately every two hours. In our experiments, we needed the watchdog device only to detect the issue for Device ID 11, as explained in Section 5.4.3. The other 10 bugs were automatically detected by DIANE by monitoring the network traffic, as explained in Section 5.2.2.

When a bug is detected, DIANE generates an alert. In this case, an analyst may want to manually reproduce and verify the bug triggering the alert. DIANE allows this

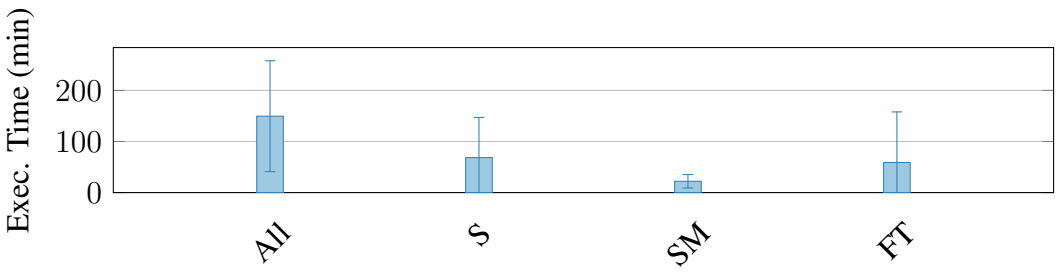


Figure 5.8: Average and standard deviation of the execution time of the phases that DIANE performs (**S** = Setup, **SM** = *sendMessage* functions Identification, **FT** = Fuzzing Trigger Identification).

manual verification, since it produces as output the input triggering the detected bug. With this information in hand, the analyst can use DIANE to send the crashing input to the analyzed IoT device. Then, the analyst can manually check the device functionality to assess if it misbehaves after receiving the crashing input (e.g., the device reboots or does not reply to further requests). In our evaluation, we needed about 6 minutes, on average, to follow this procedure and verify each alert produced by DIANE.

5.5 Limitations and Future Work

While we addressed the major challenges for performing black-box fuzzing of IoT devices, our overall approach and the implementation of DIANE still have some limitations.

Approach. We currently cannot bypass app-side sanity checks when they are implemented in native code, in a data-transforming function or directly in a *sendMessage* function. Though we acknowledge that such checks could be present in any of these

classes of code, we manually verified that none of the apps in our dataset contain sanity checks in any of these categories. In fact, as shown by previous work [1], native code is typically not used to implement the main application's logic, but it is used, instead, in library helper functions Also, note that, differently from previous work, this does not mean that DIANE cannot handle native code at all. In fact, even if the *sendMessage* function is implemented natively, DIANE can identify it and fuzz its *fuzzing triggers*. However, if sanity checks are present in any of the aforementioned classes of code, the fuzzing is less effective.

As any approach based on dynamic analysis, DIANE suffers from limited code coverage, i.e., it cannot identify *fuzzing triggers* that are not executed by the app. To mitigate this limitation, we manually stimulate the apps to trigger most of the available functionality, and we perform our analysis on real smartphones.

Implementation. The current implementation of DIANE cannot fuzz nested Java objects. We plan to address this in future work, as it only requires engineering effort.

Future work. DIANE could be enhanced to automatically discover semantic vulnerabilities (e.g., a smart lock unlocks a door instead of locking it). Currently, this feature is semi-automatic as it requires the analyst to check and interact with the watchdog device.

Algorithm 4 Fuzzing Trigger Identification.

procedure GETTOPCHAIN(*sendMessage*)
  $topChain \leftarrow \{\}$
  for $(v, cl) \in$ getArgAndObjLocs(*sendMessage*) **do**
    $to\_hook \leftarrow \{\}$
    $dtf \leftarrow []$
    $bsl \leftarrow$ getBackwardSlice(v, cl)
    for $(f, inst_f) \in$ getFunctionScopes(bsl) **do**
      $(Li_f, Lo_f) \leftarrow$ livenessAnalysis$(inst_f)$
      $to\_hook \leftarrow to\_hook \cup \{\mathsf{f}, (\mathsf{Li_f}, \mathsf{Lo_f})\}$
    end for
    for $\{f, (ELi_f, Elo_f)\} \in$ getEntropies(to_hook) **do**
      $d_e \leftarrow$ maxVarEntropy(Elo_f)/minVarEntropy(Eli_f)
      appendIfDataTransforming$(dtf, d_e, \{f, Li_f\})$
    end for
    $trees \leftarrow$ getDominatorTrees(dtf)
    $candidates \leftarrow dtf \cup \{sendmessage\}$
    for $f_c \in candidates$ **do**
      if not isDominated$(f_c, trees)$ **then**
        $topChain \leftarrow topChain \cup \{f_c\}$
      end if
    end for
  end for
  return $topChain$
end procedure
procedure FUZZINGTRIGGERIDENTIFICATION($CompanionApp$)
  $fuzzingTriggers \leftarrow \{\}$
  $borderMethods \leftarrow$ getBorderMethods$(CompanionApp)$
  for $s \in$ dynamicFilter(borderMethods) **do**
    $fuzzingTriggers \leftarrow fuzzingTriggers \cup$ getTopChain(s)
  end for
  return $fuzzingTriggers$
end procedure

Chapter 6

Related Work

As studied by Alrawi et al. [3], the security of IoT devices depends on the security of four different layers: (1) the network protocol (2) the device, (3) the companion application, and (4) the cloud endpoint.

Network Protocol Layer. The security of the communications involving an IoT device depends on the security of the employed network protocols. Most IoT systems use a combination of mainly four types of communication protocols: IP [99], ZigBee [37], Z-Wave [2], and Low Energy Bluetooth [43] (or BLE).

Unfortunately, the literature contains a plethora of work showing the weaknesses of these protocols. For instance, Vidgren et al. [125] illustrated how an adversary can take control of IoT devices that rely on the Standard Security level on the ZigBee protocol. Similarly, Ryan [109] showed how an attacker can recover a session key, by exploiting a flaw in the key-exchange protocol in Bluetooth.

Finally, the network protocols class also considers the security of application layers protocols used by IoT devices (e.g., UPnP, SSDP, HTTP, and NTP), which, unfortunately, have been found vulnerable to several attacks [11, 72, 110, 113].

Device Layer. This layer considers the security of the hardware and software components of the IoT device.

As shown by recent work [14, 142], an attacker can take advantage of physical access to an IoT device to cause significant damage, such as obtaining a dump of its memory, modify boot parameters, and extract sensitive information (e.g., root passwords) [23, 78]. In particular, Zhou et al. [151] recently discovered that if certain sensitive hardware information, such as the identifier to uniquely register an IoT device to the vendor, is disclosed, an attacker could successfully take control over the device remotely. Remote attacks are the most insidious, as they allow the attackers to compromise the IoT device without needing physical access to it.

The literature contains a plethora of work to detect vulnerabilities and secure firmware for IoT devices. Dynamic taint analysis [111] (DTA) is a well-known technique for vulnerability detection. However, reduction in performance is one of the main reasons for not integrating DTA into production devices. Techniques based on function summaries [152], instruction coalescing [100], storage optimization [70], and multithreading [84] were developed to improve the performance of DTA techniques. However, resource constraints on embedded devices render traditional DTA techniques in-

feasible [148]. Although techniques such as FirmaDyne [27], SURROGATES [74], and Avatar [145] address this by emulation, custom hardware, and hardware proxying, they either pose strict assumption on the firmware or rely on the presence of debugging ports (e.g., JTAG), which are usually disabled.

Driller [120] uses bounded symbolic execution to generate deep inputs. Dowser [52] and offset-aware fuzzing [104] use a combination of taint analysis and symbolic execution to generate overflow-inducing inputs. VUzzer [103] and GAFuzzing [82] use genetic algorithms to generate high coverage inputs. However, gray-box fuzzing techniques [71, 82, 103] require access to the runtime state of the target program making them unsuitable for embedded devices. DIFUZE [26] uses the interface information extracted using static analysis for fuzzing mobile kernel drivers. However, their techniques are customized to kernel drivers and do not apply to arbitrary binary programs. RPFuzzer [133] provides a fuzzing framework for routers. However, it requires monitoring of the running process, which is not always possible for proprietary routers. IoTFuzzer [21] performs black-box fuzz testing of various IoT devices through the corresponding mobile app. However, it obeys to the app's code constraints on the user input to generate fuzzing inputs (user's data sanitization). FIRM-AFL [150] and FirmFuzz [119] fuzz programs on IoT devices by emulating the corresponding firmware. However, a faithful emulation of firmware is a hard problem. Furthermore, similar to the other fuzzing techniques they suffer from effective input generation.

Most of the static analysis-based techniques focus on specific vulnerability types, such as buffer overflows [77, 92], integer overflows [22, 129], use-after-free [39], authentication bypass [115] and v-table escapes [35]. Few techniques exist to detect general taint style vulnerabilities [31]. However, they suffer from scalability. Unlike KARONTE, none of these techniques handle vulnerabilities that require modeling interaction between multiple binaries. Costin et al. [28] provide a framework that mixes static analysis and emulation to analyze embedded web interfaces. However, their technique is not generic, does not detect previously-unknown memory-corruption vulnerabilities, and relies on various heuristics for emulation.

Although a considerable effort has been done to assess the security of the booting process of classic systems, the booting process of IoT devices has yet to be fully explored. Wojtczuk et al. studied how unprivileged code can exploit vulnerabilities and design flaws to tamper with the SPI-flash content (containing the code that is first executed when the CPU starts), completely breaking the chain-of-trust [140] in Intel systems. Kallenberg et al. achieved a similar goal by exploiting the update mechanisms exposed by UEFI code [67]. Researchers have also shown how the chain-of-trust can be broken on the Mac platform, using maliciously crafted Thunderbolt devices [58, 59]. Other research focused on how Windows bootloader, built on top of UEFI, works and how it can be exploited [16, 107]. Bazhaniuk et al., provided a comprehensive study of the different types of vulnerabilities found in UEFI firmware and propose some mitiga-

tions [10], whereas Rutkowska presented an overview of the technologies available in Intel processors, which can be used to enforce a trusted boot process [108].

BareDroid [88] proposes and implements modifications to the Android boot process to build a large-scale bare-metal analysis system on Android devices. Although with a different goal, in this work, the authors introduce some aspects related to ours, such as difficulties in establishing a chain of trust in Android-based devices and how malware could permanently brick a device. BOOTSTOMP expands and integrates their findings, comparing different implementations and devices.

Companion Application Layer. Companion apps might compromise the security of an IoT device. As these mobile applications are trusted by the IoT devices they interact with, attackers can leverage that trust to attack the device. For instance, Sivaraman et al. [118] showed how to use companion applications to collect information about home devices connected to a local network, to ultimately make them accessible from the Internet. Programming errors on companion apps might be dangerous too. For instance, Max [64] showed how programming errors in companion apps can be used to leak sensitive information about the IoT device, ultimately showing how to use this information to harvest user credentials of the August Smart Lock.

However, researchers have recently been using companion apps mostly to assess the security of IoT devices, as they contain unique information about the device [131]

and the necessary logic to produce well-formed input to ultimately fuzz the device itself [21, 131].

Cloud Endpoint Layer. The cloud endpoints of an IoT device might contain protocol flaws, or other security-related issues, that an attacker can use to compromise the targeted IoT device. Cloud endpoints might dispose of insecure API that an attacker can leverage to, for example, perform privilege escalation of a guest account on a targeted device [64].

Furthermore, cloud endpoints might contain classic security-related issues, such as cross-site-scripting vulnerabilities, never expiring cookies, and no brute force limitation that would allow username enumeartion [61].

Chapter 7

Conclusions

In this dissertation, we presented novel approaches to improve the security of IoT devices from different angles (i.e., finding bugs versus reducing the attack surface of a program) and using different strategies (i.e., using static versus dynamic analyses). We analyzed the different software components present in an IoT device (i.e., bootloaders, OS, and user applications), and studied their peculiarities.

In particular, we analyzed modern mobile device bootloaders and showed that current standards and guidelines are insufficient to guide developers toward creating secure solutions. To study the impact of these design decisions, we implemented a static analysis approach, called BOOTSTOMP, which discovered six previously unknown memory corruption vulnerabilities, as well as two unlock-bypass vulnerabilities.

Then, we studied how the different components of a firmware sample communicate with each other to manage user requests, with particular emphasis on how different assumptions on the data being shared can cause security vulnerabilities. We learned how

tracking these inter-binary communications is of crucial importance to precisely uncover these vulnerabilities, and presented KARONTE, a tool that leverages novel static analysis techniques to drastically reduce the false positives that traditional binary analysis techniques produce when analyzing real-world firmware. Our prototype produced 87 alerts (two orders of magnitude reduction over an approach *not* considering inter-component interactions), among which we identified 46 previously unknown *zero-day* bugs.

Next, we studied how to reduce the attack surface of a program when the source code is not available. To this end, I presented a tool, called BINTRIMMER, which uses a novel abstract domain, which we used to soundly identify and remove useless code within binaries (such as firmware). On average, our tool was able to soundly remove almost 36% of the code, which contained around the 25% of ROP gadgets of the original program.

Finally, we studied the effectiveness of IoT device fuzzers. We found that randomly fuzzing network packets sent to the devices requires knowledge about the data format accepted by a device, which is seldom available when devices use custom firmware. On the other hand, approaches that leverage the UI of the companion mobile app to produce syntactically correct messages are ineffective because of the constraints that the app-side code imposes. Conversely, we proposed a novel approach, called DIANE, that sits in the sweet spot between network-level fuzzing and UI-level fuzzing. DIANE

outperforms the current state-of-the-art approach, and it detected critical bugs (9 zero-days) that cannot be triggered by existing fuzzers.

Overall, we evaluated the performance of the proposed approaches and show that the developed tools are effective in improving the security of firmware for IoT devices.

Bibliography

[1] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna. Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy. In *The Network and Distributed System Security Symposium*, pages 1–15, 2016.

[2] Z.-W. Alliance. About Z-Wave Technology. `https://z-wavealliance.org/about_z-wave_technology/`.

[3] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. Sok: Security evaluation of home-based iot deployments. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2019.

[4] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1), 1999.

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

[6] K. J. Ashton. That 'internet of things' thing. 1999.

[7] Attify. Jtag debugging. `https://blog.attify.com/hack-iot-device/`.

[8] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, Aug. 2010.

[9] L. Bang, A. Aydin, and T. Bultan. Automatically Computing Path Complexity of Programs. In *Proc. of the Joint Meeting on Foundations of Software Engineering*, 2015.

[10] O. Bazhaniuk, Y. Bulygin, A. Furtak, M. Gorobets, J. Loucaides, A. Matrosov, and M. Shkatov. Attacking and Defending BIOS in 2015. In *REcon*, 2015.

[11] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *HotSec*, 2006.

[12] Bluetooth. Debugging Bluetooth With An Android App. `https://blog.bluetooth.com/debugging-bluetooth-with-an-android-app`, 2016.

[13] Boofuzz. boofuzz: Network Protocol Fuzzing for Humans, successor to the venerable Sulley fuzzing framework. `https://github.com/jtpereyda/boofuzz`.

[14] E. Bou-Harb, C. Fachkha, M. Pourzandi, M. Debbabi, and C. Assi. Communication security for smart grid distribution networks. *IEEE Communications Magazine*, 51(1):42–49, January 2013.

[15] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[16] Y. Bulygin, A. Furtak, and O. Bazhaniuk. A tale of one software bypass of Windows 8 Secure Boot. *Black Hat USA*, 2013.

[17] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):65, 2016.

[18] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communication of the ACM*, 56(2), 2013.

[19] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, San Jose, CA, USA, 2012.

[20] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[21] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2018.

[22] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie. Brick: A Binary Tool for Run-Time Detecting and Locating Integer-Based Vulnerability. In *Proc. of the Availability, Reliability and Security (ARES)*, 2009.

[23] I. Clinton, L. Cook, and S. Banik. A survey of various methods for analyzing the amazon echo (2016), 2018.

[24] L. Cojocar, J. Zaddach, R. Verdult, H. Bos, A. Francillon, and D. Balzarotti. PIE: Parser Identification in Embedded Systems. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2015.

[25] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna. Obfuscation-resilient privacy leak detection for mobile apps through differential analysis. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2017.

[26] J. Corina, A. Machiry, C. Salls, Y. Shoshitaishvili, S. Hao, C. Kruegel, and G. Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138. ACM, 2017.

[27] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large-scale analysis of the security of embedded firmwares. In *USENIX Security Symposium*, pages 95–110, 2014.

[28] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 437–448, New York, NY, USA, 2016. ACM.

[29] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'77, Los Angeles, California, 1977.

[30] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL'78, Tucson, Arizona, 1978.

[31] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[32] A. Cui and S. J. Stolfo. A quantitative analysis of the insecurity of embedded network devices: Results of a wide-area scan. In *Proceedings of the 26th Annual*

Computer Security Applications Conference, ACSAC '10, pages 97–106, New York, NY, USA, 2010. ACM.

[33] S. Datt. The information explosion: Trends in technology 2011 review. *The Journal of Government Financial Management*, 2011.

[34] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[35] D. Dewey and J. T. Giffin. Static Detection of C++ Vtable Escape Vulnerabilities in Binary Code. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[36] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, 1996.

[37] S. Farahani. *ZigBee Wireless Networks and Transceivers*. Newnes, Newton, MA, USA, 2008.

[38] J. Feist, L. Mounier, S. Bardin, R. David, and M.-L. Potet. Finding the Needle in the Heap: Combining Static Analysis and Dynamic Symbolic Execution to Trigger Use-After-Free. In *Proc. of the Workshop on Software Security, Protection, and Reverse Engineering (SSPREW)*, 2016.

[39] J. Feist, L. Mounier, and M.-L. Potet. Statically Detecting Use After Free on Binary Code. *Journal of Computer Virology and Hacking Techniques*, 10(3), 2014.

[40] Frida. Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. `https://frida.re/docs/android/`.

[41] GitHub. ARM Trusted Firmware. `https://github.com/ARM-software/arm-trusted-firmware`, 2017.

[42] D. Giusto, A. Iera, G. Morabito, and L. Atzori. *The internet of things: 20th Tyrrhenian workshop on digital communications*. Springer Science & Business Media, 2010.

[43] C. Gomez, J. Oller, and J. Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[44] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 72–81. IEEE Press, 2013.

[45] Google. `https://support.google.com/nexus/answer/6172890?hl=en`, 2016.

[46] Google. Verifying Boot. `https://source.android.com/security/verifiedboot/verified-boot.html`, 2017.

[47] A. Gotlieb, M. Leconte, and B. Marre. Constraint Solving on Modular Integers. In *ModRef Worksop, associated to CP'2010*, Saint-Andrews, United Kingdom, Sept. 2010.

[48] P. Granger. Static analysis of arithmetical congruences. 30:165–190, 01 1989.

[49] GSMA. Anti-theft Device Feature Requirements, 2016.

[50] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, Chaoyang District, Beijing, Sept. 2019. USENIX Association.

[51] M. Gyung Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[52] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proc. of the USENIX Security Symposium*, 2013.

[53] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, page 2005.

[54] C. Heffner. binwalk - firmware analysis tool designed to assist in the analysis, extraction, and reverse engineering of firmware images. `https://github.com/ReFirmLabs/binwalk`, 2014.

[55] Hex-Rays. IDA Pro. `https://www.hex-rays.com/products/ida/index.shtml`, 2017.

[56] T. Hickey, Q. Ju, and M. H. Van Emden. Interval arithmetic: From principles to implementation. *J. ACM*, 48(5):1038–1068, Sept. 2001.

[57] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999.

[58] T. Hudson, X. Kovah, and C. Kallenberg. ThunderStrike 2: Sith Strike. *Black Hat USA*, 2015.

[59] T. Hudson and L. Rudolph. Thunderstrike: Efi firmware bootkits for apple macbooks. In *Proceedings of the 2015 ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015.

[60] IBM. IBM X-Force: Stolen Credentials and Vulnerabilities Weaponized Against Businesses in 2019. https://newsroom.ibm.com/2020-02-11-IBM-X-Force-Stolen-Credentials-and-Vulnerabilities-Weaponized-Against-Businesses-in-2019?_ga=2.223706392.2070857419.1601855701-1259169858.1601855701.

[61] S. R. Inc. Hello barbie initial security analysis. http://static1.squarespace.com/static/543effd8e4b095fba39dfe59/t/56a66d424bf1187ad34383b2/1453747529070/HelloBarbieSecurityAnalysis.pdf.

[62] B. Insider. Hackers once stole a casino's high-roller database through a thermometer in the lobby fish tank. https://www.businessinsider.de/hackers-stole-a-casinos-database-through-a-thermometer-in-the-lobby-fish-tank-2018-4?r=UK&IR=T, 2018.

[63] T. Instruments. Jtag fuse flow. https://e2e.ti.com/support/microcontrollers/msp430/f/166/t/18936?JTAG-FUSE-BLOW.

[64] Jmaxxz. Backdooring the frontdoor. DEFCON, 2016. https://doi.org/10.5446/36251.

[65] L. Jost. Entropy and diversity. *Oikos*, 113(2):363–375, 2006.

[66] Kali. bed - A network protocol fuzzer. https://tools.kali.org/vulnerability-analysis/bed.

[67] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell. Extreme privilege escalation on Windows 8/UEFI systems. *BlackHat, Las Vegas, USA*, 2014.

[68] Katyusha. Katyusha rest and soap fuzzer. `https://github.com/lpredova/Katyusha`.

[69] R. Katzev. Car sharing: A new approach to urban transportation problems. *Analyses of Social Issues and Public Policy*, 3(1):65–86, 2003.

[70] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. libdft: Practical Dynamic Data Flow Tracking for Commodity Systems. In *Proc. of the ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[71] M. E. Khan, F. Khan, et al. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Sciences and Applications*, 3(6), 2012.

[72] P. Kintis, Y. Nadji, D. Dagon, M. Farrell, and M. Antonakakis. Understanding the privacy implications of ecs. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 343–353. Springer, 2016.

[73] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[74] K. Koscher, T. Kohno, and D. Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT*, 2015.

[75] U. W. Kulisch. Complete Interval Arithmetic and Its Implementation on the Computer. In *Numerical Validation in Current Hardware Architectures: International Dagstuhl Seminar, Dagstuhl Castle, Germany, January 6-11, 2008. Revised Papers*, pages 7–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[76] 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.

[77] D. Larochelle, D. Evans, et al. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proc. of the USENIX Security Symposium*, 2001.

[78] J. Lau, B. Zimmerman, and F. Schaub. Alexa, are you listening?: Privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):102, 2018.

[79] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed Systems Security*, NDSS '11, San Diego, CA, USA, 2011.

[80] K. Lee, Y. Lee, H. Lee, and K. Yim. A brief review on jtag security. In *2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*, pages 486–490, July 2016.

[81] J. Lerch, B. Hermann, E. Bodden, and M. Mezini. FlowTwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases. In *Proc. of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2014.

[82] G.-H. Liu, G. Wu, Z. Tao, J.-M. Shuai, and Z.-C. Tang. Vulnerability analysis for x86 executables using genetic algorithm and fuzzing. In *Proceedings of the 2008 Convergence and Hybrid Information Technology*, volume 2 of *ICCIT '08*, pages 491–497. IEEE, 2008.

[83] Micron Technologies. eMMC Security Features, 2016.

[84] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu. TaintPipe: Pipelined Symbolic Taint Analysis. In *Proc. of the USENIX Conference on Security Symposium*, 2015.

[85] Mitre. LK bootloader security vulnerability, CVE-2014-9798. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-9798`.

[86] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Proc. of the ISOC Network and Distributed System Security Symposium (NDSS)*.

[87] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[88] S. Mutti, Y. Fratantonio, A. Bianchi, L. Invernizzi, J. Corbetta, D. Kirat, C. Kruegel, and G. Vigna. Baredroid: Large-scale analysis of android apps on real devices. In *Proceedings of the 2015 Annual Computer Security Applications Conference*, ACSAC 2015, New York, NY, USA, 2015.

[89] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Programming Languages and Systems*, pages 115–130. Springer, 2012.

[90] N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys & Tutorials*, 04 2019.

[91] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, 42(6):89–100, June 2007.

[92] M. Neugschwandtner, P. Milani Comparetti, I. Haller, and H. Bos. The borg: Nanoprobing binaries for buffer overreads. In *Proceedings of the 2015 ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 87–97, New York, NY, USA, 2015. ACM.

[93] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2005.

[94] F. Nielson, H. Riis Nielson, and C. Hankin. *Principles of Program Analysis*. 01 1999.

[95] A. Outler. Have you paid your linux kernel source license fee? `https://www.xda-developers.com/have-you-paid-your-linux-kernel-source-license-fee/`, March 2013.

[96] K. U. P. W. M. O. P. K. U. P. C. J. O. K. U. S. M. C. S. K. U. Paczkowski, Lyle W. (Mission Hills. Jtag fuse vulnerability determination and protection using a trusted execution environment, April 2015.

[97] J. Palsberg and M. I. Schwartzbach. *Object-oriented type inference*, volume 26. ACM, 1991.

[98] P. Pokorny and M. Royal. Dumb fuzzing in practice. 2012.

[99] D. I. Program. Internet Protocol. `https://tools.ietf.org/html/rfc791`.

[100] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2006.

[101] Qualcomm. (L)ittle (K)ernel based Android bootloader. `https://www.codeaurora.org/blogs/little-kernel-based-android-bootloader`.

[102] D. A. Ramos and D. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the 2015 USENIX Conference on Security Symposium*, SEC'15, Washington, DC, USA, 2015.

[103] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[104] S. Rawat and L. Mounier. Offset-Aware Mutation Based Fuzzing for Buffer Overflow Vulnerabilities: Few Preliminary Results. In *Proc. of the Software Testing, Verification and Validation Workshops (ICSTW)*, 2011.

[105] S. Rawat, L. Mounier, and M.-L. Potet. Static taint-analysis on binary executables, 2011.

[106] F. Riggins and S. Fosso Wamba. Research directions on the adoption, usage and impact of the internet of things through the use of big data analytics. volume 2015, 01 2015.

[107] rol. ring of lightning. `https://rol.im/securegoldenkeyboot/`, 2016.

[108] J. Rutkowska. Intel x86 considered harmful, 2015.

[109] M. Ryan. Bluetooth smart: The good, the bad, the ugly... and the fix. *BlackHat USA, Las Vegas, USA*, 2013.

[110] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72. ACM, 2010.

[111] E. J. Schwartz, T. Avgerinos, and D. Brumley. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*, 2010.

[112] SecuriTeam. BSS (Bluetooth Stack Smasher) Fuzzer. `https://securiteam.com/tools/5NP0220HPE/`.

[113] J. Selvi. Bypassing http strict transport security. *Black Hat Europe*, 2014.

[114] R. Sen and Y. N. Srikant. Executable Analysis using Abstract Interpretation with Circular Linear Progressions. *Formal Methods and Models for Co-Design, ACM/IEEE International Conference on*, 00(undefined):39–48, 2007.

[115] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[116] 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 *Proc. of the IEEE Symposium on Security and Privacy (SP)*, 2016.

[117] A. Simon. *Value-Range Analysis of C Programs: Towards Proving the Absence of Buffer Overflow Vulnerabilities*. Springer Science & Business Media, 1 edition, 2010.

[118] V. Sivaraman, D. Chan, D. Earl, and R. Boreli. Smart-phones attacking smart-homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '16, pages 195–200, New York, NY, USA, 2016. ACM.

[119] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer. FirmFuzz: Automated IoT Firmware Introspection and Analysis. In *Proc. ACM CCS Workshop on IoT Security and Privacy (IoT S&P)*, 2019.

[120] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, NDSS '16, San Diego, CA, USA, 2016.

[121] R. Uehara and Y. Uno. Efficient Algorithms for the Longest Path Problem. In *Proc. International Symposium on Algorithms and Computation (ISAAC)*, 2005.

[122] UFuzz. UFuzz, or Universal Plug and Fuzz, is an automatic UPnP fuzzing tool. `https://github.com/phikshun/ufuzz`.

[123] Vaas, Lisa. Smartphone anti-theft "kill switch" law goes into effect in California, 2015.

[124] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *International conference on compiler construction*, pages 18–34. Springer, 2000.

[125] N. Vidgren, K. Haataja, J. Patino-Andres, J. Ramirez-Sanchis, and P. Toivanen. Security threats in zigbee-enabled systems: Vulnerability evaluation, practical experiments, countermeasures, and lessons learned. In *2014 47th Hawaii International Conference on System Sciences*, pages 5132–5138, Los Alamitos, CA, USA, jan 2013. IEEE Computer Society.

[126] VxWorks. VxWorks, the industry's leading real-time operating system. `https://www.windriver.com/products/vxworks/`.

[127] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'17, San Diego, CA, USA, 2017.

[128] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 2015 USENIX Conference on Security Symposium*, SEC'15, Washington, DC, USA, 2015.

[129] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[130] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, Anaheim, CA, USA, 2008.

[131] X. Wang, Y. Sun, S. Nanda, and X. Wang. Looking from the mirror: Evaluating iot device security through mobile companion apps. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1151–1167, Santa Clara, CA, Aug. 2019. USENIX Association.

[132] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In M. Backes and P. Ning, editors, *Computer Security – ESORICS 2009*, pages 200–215, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[133] Z. Wang, Y. Zhang, and Q. Liu. Rpfuzzer: A framework for discovering router protocols vulnerabilities based on fuzzing. *KSII Transactions on Internet & Information Systems*, 7(8), 2013.

[134] H. S. J. Warren. *Hacker's Delight*. Addison-Wesley, Boston, Toronto, London, 2003.

[135] M. web docs. 500 internal server error. `https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500`.

[136] H. Wen, Q. Zhao, Q. A. Chen, and Z. Lin. Automated Cross-Platform Reverse Engineering of CAN Bus Commands From Mobile Apps. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2020.

[137] D. B. West. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ, 1996.

[138] T. Wilson. Evaluation of fuzzing as a test method for an embedded system. 2018.

[139] Wired. A Legion of Bugs Puts Hundreds of Millions of IoT Devices at Risk. `https://www.wired.com/story/ripple20-iot-vulnerabilities/`.

[140] R. Wojtczuk and C. Kallenberg. Attacking UEFI boot script. In *31st Chaos Communication Congress (31C3)*, 2014.

[141] Wsfuzzer. Web services fuzzing tool for http and soap. `https://sourceforge.net/projects/wsfuzzer/files/`.

[142] J. Wurm, K. Hoang, O. Arias, A. Sadeghi, and Y. Jin. Security analysis on consumer and industrial iot devices. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 519–524, Jan 2016.

[143] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.

[144] R. Yonck. Connecting with our connected world. *The Futurist*, 2013.

[145] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, 2014.

[146] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, Feb 2014.

[147] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation.

[148] N. Zhang, S. Demetriou, X. Mi, W. Diao, K. Yuan, P. Zong, F. Qian, X. Wang, K. Chen, Y. Tian, et al. Understanding iot security through the data crystal ball: Where we are now and where we are going to be. *arXiv preprint arXiv:1703.09809*, 2017.

[149] Q. Zhao, C. Zuo, D.-G. Brendan, G. Pellegrino, and Z. Lin. Automatic uncovering of hidden behaviors from input validation in mobile apps. 2020.

[150] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, Santa Clara, CA, Aug. 2019. USENIX Association.

[151] W. Zhou, Y. Jia, Y. Yao, L. Zhu, L. Guan, Y. Mao, P. Liu, and Y. Zhang. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1133–1150, Santa Clara, CA, Aug. 2019. USENIX Association.

[152] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting Sensitive Data Leaks Using Application-level Taint Tracking. *ACM SIGOPS Operating Systems Review*, 45(1), 2011.