# Towards 1-day Vulnerability Detection using Semantic Patch Signatures

Alexis Challande



INSTITUT POLYTECHNIQUE DE PARIS

NNT: 2022IPPAX096

Thèse de doctorat

# Towards *1-day* Vulnerability Detection using Semantic Patch Signatures

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à École Polytechnique

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Informatique

Thèse présentée le 11 octobre 2022, par

**Alexis Challande**

Composition du Jury :

| | |
|---|---|
| Stefano Zacchiroli<br>Professeur, Télécom Paris | Président |
| Aurélien Francillon<br>Professeur, Eurecom | Rapporteur |
| Christine Morin<br>Directrice de Recherche, Inria | Rapporteure |
| Thomas Clausen<br>Professeur, École Polytechnique | Examinateur |
| Sébastien Josse<br>Chercheur, Direction Générale de l'Armement | Examinateur |
| Sarah Zennou<br>Directrice de Recherche, Airbus | Examinatrice |
| Robin David<br>Ingénieur R&D, Quarkslab | Encadrant industriel |
| Guénaël Renault<br>Agence nationale de la sécurité des systèmes d'information<br>Professeur chargé de cours, École Polytechnique | Directeur de thèse |

Arrakis teaches the attitude of the knife - chopping off what's incomplete and saying: "Now, it's complete because it's ended here."

---

Frank Herbert

C'est beau un jardin qui ne pense pas encore aux hommes.

---

Jean Anouilh

# Résumé


## Contexte

Cette thèse étudie la recherche de vulnérabilités de type *1-jour* dans des systèmes de fichiers arbitraires.

**Définition.** *Une vulnérabilité* 1-jour *est une vulnérabilité pour laquelle un correctif publiquement disponible existe depuis au moins* 1 jour.

L'identification des vulnérabilités dans les logiciels représente un des piliers sécuritaires pour protéger les systèmes d'information. Malgré un travail de recherche continu effectué par les communautés scientifiques et industrielles, de nouvelles failles sont régulièrement identifiées. Par exemple, la National Vulnerability Database liste plus de 18 000 CVEs pour l'année 2020 et la tendance n'est pas à la baisse. De plus, la lutte entre attaquant·e·s et défenseur·e·s est fondamentalement asymétrique, puisqu'une erreur sur une ligne de code dans un projet pouvant en contenir des millions peut résulter en une compromission totale.

Après l'identification d'une faille sur un logiciel (ou un matériel), ses responsables peuvent proposer un correctif corrigeant la vulnérabilité. Une vulnérabilité pour laquelle un correctif est disponible cesse alors d'être une vulnérabilité *0-jour* et devient une vulnérabilité *1-jour*. En revanche, on observe un temps de propagation entre le moment où le correctif est disponible et celui où il est appliqué sur l'ensemble des systèmes ; il est donc utile de déterminer si un correctif est présent sur un système pour établir son exposition à un risque.

Ce problème se décompose en deux parties complémentaires, la première, comment identifier si un correctif a été appliqué sur un système et la seconde, comment caractériser ce correctif. Le problème de l'identification (*matching*) a été étudié dans la littérature notamment sous l'aspect de la recherche d'artefacts équivalents. Le problème de caractérisation d'un correctif est plus nouveau, et relativement peu exploré.

Il faut néanmoins noter que l'absence d'un correctif sur un système n'implique pas que ce dernier soit vulnérable. En effet, il est possible que la portion de code concernée ne soit pas utilisée, ou que d'autres contre-mesures soient en place pour empêcher l'exploitation. De même, la présence d'un correctif n'est pas suffisante pour déterminer la sécurité d'un système : ce correctif peut-être incomplet en laissant ouvert des chemins d'exploitation, ou plus simplement inapproprié.

## Contributions

Les contributions de cette thèse se déclinent en quatre axes.

La première contribution est la formalisation du problème de la recherche de correctifs dans un système de fichier arbitraire, dénommé le FMP. Cette formalisation étend la notion de test de présence de correctif (*patch presence test*) établie par Zhang *et al.* [143] à l'échelle d'un système. Cette contribution est complétée par l'établissement d'une stratégie en trois étapes pour résoudre ce problème à l'aide de signatures de correctifs. Cette stratégie, la *F-S-M* (pour Filtrage-Sélection-*Matching*) réduit par couches successives l'espace de recherche, permettant une résolution efficace du problème dans des systèmes de fichiers contenant des milliers de programmes.

Nous proposons également une mise en oeuvre de cette stratégie à travers un outil que nous avons développé : **QSig**. Cet outil est constitué de deux composants. Un premier, le générateur, crée la signature d'un correctif en analysant automatiquement les différences entre deux programmes binaires. Le second, le détecteur, applique des signatures générées précédemment à un système de fichiers, en utilisant la *F-S-M*. Comme elles sont utilisées à chaque étape, les signatures de correctifs sont donc des éléments cruciaux de la résolution du FMP. Celles que nous proposons sont fondées sur des invariants sémantiques du code binaire.

Pour évaluer **QSig**, nous introduisons un nouveau de jeu de données utilisant les bulletins de sécurité publiés tous les mois par Google pour le système Android. Ces bulletins précisent notamment l'identifiant du *commit* corrigeant une vulnérabilité, permettant à notre jeu de données d'être précis à cette granularité. Comme la recherche de vulnérabilités est également effectuée sur du code binaire, nous proposons une sous-partie de ce *dataset* sous forme d'artefacts précompilés via une méthode automatique. Ce jeu de données a été publié pour être accessible à la communauté.

Finalement, notre dernière contribution est la construction de graphes de dépendances unifiés (formalisés par Fan *et al.* [40]) utilisant le système de construction de l'*Android Open Source Project* pour établir les dépendances entre différentes cibles de compilation statiquement. Ces liens de dépendances sont ensuite exploités pour adapter **QSig** à la détection de correctifs dans des bibliothèques statiquement compilées sur le système Android.

Toutes nos contributions participent à la création d'un ensemble logiciel permettant de rechercher efficacement des correctifs de vulnérabilités sur des systèmes de fichier. Nous montrons également que l'approche choisie est pertinente et efficace en la comparant à différents travaux de l'état de l'art.

# Contents

Chapter 1

# Introduction

## Contents



Information systems are a key component of modern world and are used in most economic fields. Tasked with an ever-growing set of demands and supported by ever-improving hardware, they have evolved from simple and large devices (mainframes) to complex machines. Indeed, computer systems range from well-known general-purpose devices, like smartphones or desktops to specialized objects, called embedded systems, like routers, industrial systems, or household appliances.

The missions handled by such devices may be critical for the safety and security of many users. For example, the ransomware epidemic severely disrupted hospitals over the last years [18]. These attacks were even responsible for real-life casualties with the first report of a death directly attributed to one of them [65]. Countries also rely on their cyber infrastructure security and physical actions are now with a cyber part. For example, a cyberattack took down satellite communications in Ukraine in the hours before the 2022 invasion. Journalists in Western states have attributed this attack to Russia [110].

Cyberattacks usually exploit a bug in a codebase. We define bugs with security implications with regard to a security model as *vulnerabilities* (Section 2.1). Codebases are composed of internal components and external dependencies. Indeed, to reduce development costs and improve delivery speed, developers of new software often borrow or use external components. However, using someone's else code requires trusting their third-party authors. These problematics are now called *supply chain* issues. Indeed,

from an attacking point of view, a bug location, within the main code or inside a dependent component, does not matter. For instance, the *Log4j* vulnerability [69] affected a common logging library and spread to hundreds of projects. Several strategies exist to battle software vulnerabilities. We detail some of them below, but this list is far from exhaustive.

- Focus on the security aspects during the software development cycle using a Secure Software Development Framework (SSDF). The best option to prevent vulnerabilities is to avoid their introduction. It is worth considering security during the design process and before the device release. Indeed, a defect found afterward in a component not upgradable is impossible to fix. For example, Positive Technologies discovered a vulnerability affecting a read-only memory in Intel CSME which was unfixable [101]. The suggested mitigation strategy was to disable the feature[1].
- Using a Host-based Intrusion Detection System (HIDS) [96] to monitor the host internals and the network interfaces to prevent exploitation attempts from reaching applications. By monitoring system activity [79], a defense system can block malicious activity and protect vulnerable applications without modifying them. Legacy applications with discontinued support can be protected using this strategy.
- Apply available updates to each software. Most vendors provide support for their software using updates. These updates may add functionalities, but also fix (security) bugs. Some fast-evolving software like web browsers provides an automated update mechanism. However, this behavior is not suitable in contexts where each update must be vetted to ensure it maintains backward compatibility. Moreover, embedded devices may either lack an update mechanism or it may require manual intervention which makes fleet-wide upgrades more complex. Finally, updates may not be available for a device as vendors only provide support for a limited time.

These strategies are complementary, and stakeholder should consider them together (Swiss cheese model) because the layer multiplication reduces the residual risk. Tackling the full spectrum of vulnerability defenses would be impracticable for a single work. While the research of defenses against new vulnerabilities (*0-day*) draws massive attention from the community, the study of known vulnerabilities is much less explored. Still, a patch fixing a security issue is not immediately deployed to each instance running the software. Hence, studying vulnerabilities with existing patches, denoted *1-day* vulnerabilities, is needed. Chapter 2 provides a more detailed definition of *1-day* vulnerabilities.

Detecting whether a software is patched against a known *1-day* vulnerability is crucial for assessing system exposures to known threats. Companies managing fleets of devices using heterogenous firmware versions are impacted by this problem. Indeed, knowing whether commercial firmwares they rely on are properly patched against known vulnerabilities is crucial for their security posture. It can also be used as a security baseline:

[1] Arguably, a problem when the feature is needed.

before managing company secrets, a device must have at least the patches for a vulnerability set. Of note, detecting patches can also be valuable from an attackers point of view (e.g. nation state, penetration testing companies). Tasked to extract information from a device, knowing that the patch for a vulnerability is missing could pave the way for a potential compromise.

Overwhelming attacks also exploit *1-day* vulnerabilities. For instance, the infamous WannaCry ransomware weaponized a *1-day*, Eternal Blue [90]. Even if Microsoft released patches[2], the worm spread across unpatched systems. Cyence estimated the cost of the hack at $4 billion [61].

*1-day* vulnerabilities affect every software and the ones running on embedded devices are no exceptions. Their *firmware*, a software specifically tailored for embedded devices [82], is usually written in low-level languages facilitating hardware control. Because these languages (C, C++, or proprietary ones) allow the developer a fine-grained control over memory, they require a greater discipline to prevent coding issues. On the opposite, higher-level counterparts use mechanisms abstracting the memory management from the user, offering stronger safety guarantees[3]. The extensive control of the underlying hardware offered by low-level languages comes with a drawback: programming errors can quickly have serious security impacts.

Unfortunately, the embedded systems security state is subpar [24], and even high-end devices may lack some elementary countermeasures [1]. Assessing the security of such devices creates new challenges compared to traditional software. Indeed, the device diversity, the numerous architectures, and the interactions between hardware and software complicate analyses, even straightforward. Additionally, challenges arise for even accessing and extracting the device firmware. We consider this problem out of this work scope.

We summarize the problem for finding whether a patch has been applied to a device in Figure 1.1. The black box represents the missing link for answering this question. Of note, it is crucial to remember that probing the patch presence will only answer on the patch *presence*.

> *"Not finding a patch on a device is insufficient to conclude that a device is* vulnerable *to a particular vulnerability."*

For example, the device may not use the vulnerable code or even be shipped with it (Table 1.1). With this restriction in mind, the terms *missing* and *not found* are used interchangeably in this manuscript when referring to patches. Moreover, patches

[2] And provided patches for end-of-life systems.
[3] But at the cost of relying on their toolchain.

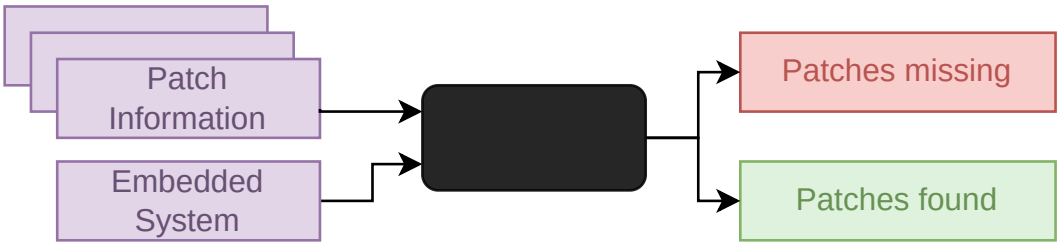


Figure 1.1: Main Problem Illustrated

| **Patch Found** | **Device Status** |
|---|---|
| Yes | Not Vulnerable |
| No | Unable to conclude |
| Partially | Probably Vulnerable |

Table 1.1: Possible Answers to the Patch Presence Test on a Device

for vulnerabilities are not atomic objects. For example, they can spread over multiple functions. If a patch is only partially found on a device, we conclude that the device remains vulnerable to the vulnerability because we consider that a fix must be complete to be effective.

The most common *embedded system* is the smartphone and its Operating System (OS) distribution is a duopoly. Apple with iOS represents 28% of the worldwide market share, and Android from Google the remaining 71% [115]. While both systems share the same objectives, their conception is different. Apple develops both the hardware and the software while keeping iOS closed. On the contrary, Android has an open-source core: Google develops the software and provides licenses to various OEM to drive its adoption[4]. This model resembles Microsoft's for Windows. Because Android runs on heterogeneous devices, it is a representative use case to study embedded systems. We detail the reasons supporting our choice in Chapter 2.

Android devices are often lacking patches due to discontinued support from vendors. Moreover, even if a device is at its latest version, vendors may miss patches. In 2008, SRLabs demonstrated the *hidden patch gap* prevalence in the Android ecosystem [112]: in their study, most vendors were missing at least one patch included in the claimed Security Patch Level (SPL). On another hand, some vendors also backport security fixes

[4]Google is also a manufacturer with the Pixel line.

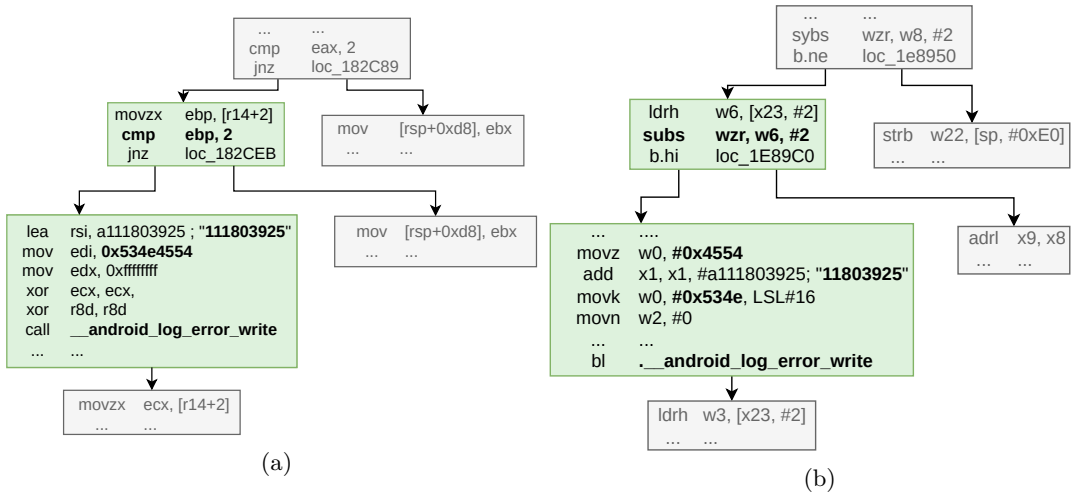


Figure 1.2: CVE–2018–9506 associated CFG for x64 (a) and aarch64 (b)

to their devices. Detecting the program version is thus insufficient to assess whether the program is patched [144].

## 1.1 Motivating Example: CVE-2018-9506

**Listing 1.1** CVE–2018–9506 Patch Extract

```
    @@ -660,6 +662,13 @@
    msg.browse.p_browse_pkt = p_pkt;
  } else {
+    if (p_pkt->len < AVRC_AVC_HDR_SIZE) {
+      android_errorWriteLog(0x534e4554, "111803925");
+       [ ... skip ...]
+      return;
+    }
    msg.hdr.ctype = p_data[0] & AVRC_CTYPE_MASK;
```

CVE–2018–9506 [92] fixes a possible out-of-bound read in Android's bluetooth stack. An extract of the vulnerability patch adding a length check is listed in Listing 1.1. The Control Flow Graph (CFG) represents all paths that might be traversed by a program through its execution [131]. Both the `x64` and `aarch64` versions are shown in 1.2a and 1.2b. While the CFG of both versions is similar on the excerpt, the number of blocks varies between the two functions (167 and 154 basic blocks). Moreover, the mnemonics used are architecture dependent, and creating a mapping from one set to another is rather impractical.

Google fixed the vulnerability in the Android Security Bulletin from October 2018 (SPL 2018-10-01) before any exploitation attempt was reported. Nonetheless, device manufacturers may not have updated every device to fix this vulnerability, either by installing the newer SPL, or backporting the fix. This delay opens an exploitation window for attackers. Thus, it is valuable to elaborate a strategy for detecting whether this patch (or others) was applied to a device.

The main objective of this thesis is to construct a solution reducing this patch to its main characteristics, then devise a strategy allowing to search it efficiently and accurately on a complete filesystem.

## 1.2 Thesis Objectives

The growing number of vulnerabilities requires constant attention from stakeholders to develop and provide patches to affected users. Deploying these patches is a necessary step for ensuring a device safety. However, due to the *patch propagation delay*, end users are vulnerable within this patching window and might remain vulnerable permanently if the patch is not propagated.

This thesis objective is to delve into how to automate the research of *1-day* vulnerabilities in embedded systems. An idea that first comes to mind when talking about automation in cyber security, and more broadly in computer science, is to consider machine learning algorithms. First coined in 1959 [109], this term and its associated domain have seen a large expansion since the 2000' with new techniques and an increase in computational power. If the study of techniques, their underlying statistics models, and inner workings are out of scope for this work, we highlight that most algorithms used today require a vast amount of data to work.

Another objective of this thesis is characterizing *1-day* vulnerabilities. A formal definition is given in Chapter 2, but it does not cope well with automated analysis. Elaborating a precise overview of *1-day* vulnerabilities is a first step before searching them in systems. Current research on *1-day* characterization has either considered them on too high granularity (e.g. at a function-level) [3] or as taxonomies, which are not suitable for automated research [12].

A final objective of this research work is to elaborate, from the characteristics of *1-day* vulnerabilities, a mechanism to store them and retrieve them in real-world firmwares. This work could enhance the security of embedded systems as it would give end users the list of vulnerabilities unpatched on their system along with an explanation of the induced risks.

## 1.3 Contributions

In this thesis, we tackle the challenges of detecting *1-day* vulnerabilities patches in filesystems. For doing so, we present four contributions to reach our objectives:

- We propose a formalization of the Firmware Matching Problem (FMP) which aims to detect whether a patch has been applied to a firmware.
- We introduce a three-step solution, *F-S-M*, for *Filtering-Selecting-Matching* based on successive pruning stages to solve the Firmware Matching Problem.
- We improve the State of the Art by providing an implementation of our solution inside **QSig**, an open-source system developed to answer FMP.
- **QSig** is extensively tested using a large dataset we constructed from Android's vulnerabilities. We also provide this dataset in open-source for broader usage within the community.
- Finally, we extend our approach to detect *1-day* in statically embedded code by designing a specialized filtering step for the Android platform. It also demonstrates our approach modularity in various workflows.

### Publications

Parts of this work have been published in academic conferences, listed below.

- Exploitation du graphe de dépendance d'AOSP à des fins de sécurité in *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC '21), June 2–4, 2021, Rennes, France.*
- Building a Commit-level Dataset of Real-world Vulnerabilities in *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy (CODASPY '22), April 24–27, 2022, Baltimore, MD, USA.*
- (submitted) Patch Detection using Binary Only Semantic Signatures in *27th European Symposium on Research in Computer Security (ESORICS) 2022.*

## 1.4 Thesis Outline

This thesis is organized into seven chapters. The present one introduces the manuscript and the next one presents the necessary background to understand the context of this work (Chapter 2).

Chapter 3 starts with a major patch analysis study on Android and formalizes the *Firmware Matching Problem*. It then introduces our solution, *F-S-M*, solving the *patch presence test* at a firmware scale. Finally, it presents our open-source system, **QSig**, which implements the *F-S-M*.

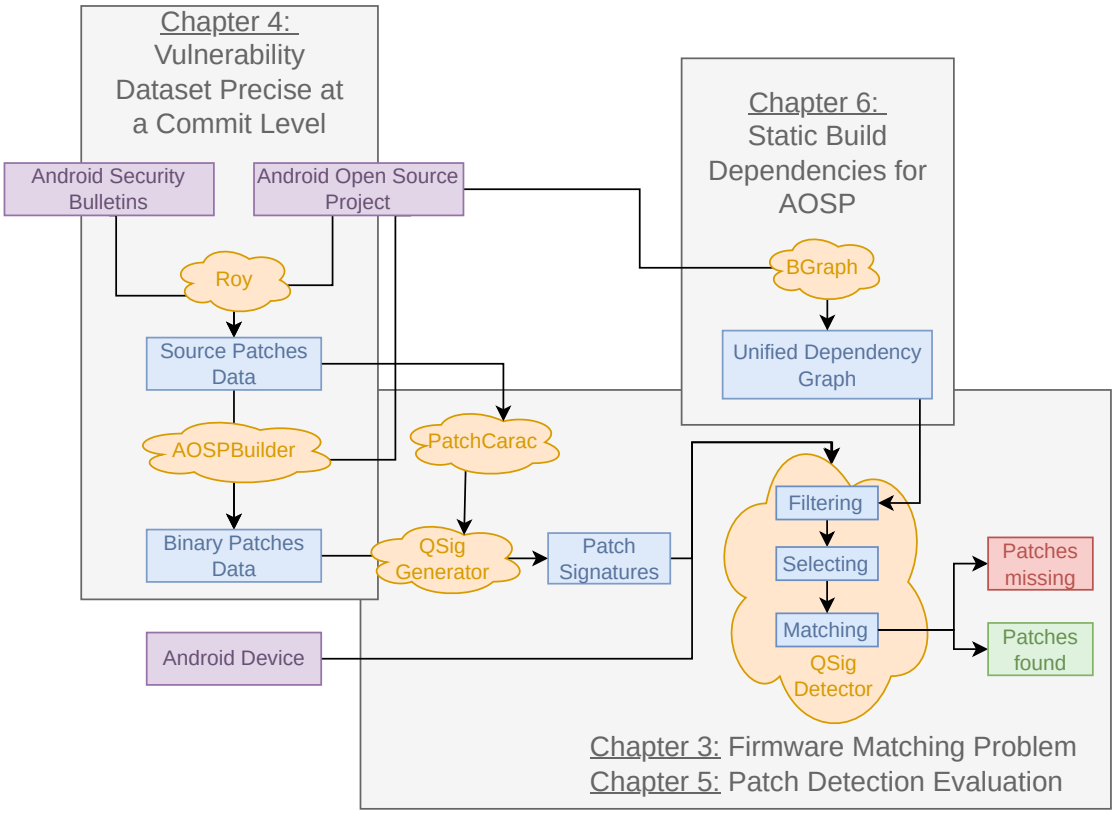


Figure 1.3: Thesis Outline and Contributions

In Chapter 4, we introduce a vulnerability dataset, precise at a commit level, to help the community create an accurate testing ground for vulnerability research applications.

We use this dataset for the patch analysis presented in Chapter 3, and to conduct the **QSig** evaluation which is presented in Chapter 5.

To follow compilation dependency chains, we develop our *Build Graphs* in Chapter 6. Applied to the Android Open Source Project (AOSP), these graphs allow us to solve the *Firmware Matching Problem* filtering step, presented in Chapter 3, and to find vulnerabilities affecting statically embedded libraries.

Eventually, we conclude the thesis in Chapter 7 and give an outlook on the future of the *patch presence test* for generic devices.

To help the reader understand the articulations between these works and replace them in the final objective context, we use the schema presented in Figure 1.3 as a thread. The clouds on it represent the different components mentioned across the chapters.

Chapter 2

# Background

**Contents**



This chapter introduces key concepts used in this thesis and helps the comprehension of this manuscript. It starts with reminders on the security issues management in software information systems in Section 2.1. Section 2.2 presents generalities on *embedded systems* and firmwares, and Section 2.3 focuses on *Android*. Finally, we discuss various code representation techniques in Section 2.4.

## 2.1 Security Issues in Software Information Systems

Every running information system is susceptible to bugs, a deviation from the expected behavior. Some of them introduce security risks for the program users and they are therefore classified as *security bugs*. A threat actor exploits a weakness, called vulnerability, to compromise a system [124]. While the most common vulnerabilities stem from

*security bugs*, other vectors are possible (e.g. hardware, humans), but we do not consider them in this work. Hence, for the remaining of this document, the terms *vulnerability*, and *security bugs* are used interchangeably.

### 2.1.1 Vulnerability Taxonomy

Identifying vulnerabilities in both software and hardware stacks is a critical part of the current global security edifice foundation. Despite a continuous research effort, from academic and industrial communities, new critical vulnerabilities are discovered regularly. For example, the National Vulnerability Database (NVD) [120] reported more than 30,000 Common Vulnerabilities and Exposures (CVE) for the year 2021 alone (more than 83 per day), and the trend is not slowing (93 per day in 2022). The fight between attackers and defenders is fundamentally asymmetric: a mistake in a single line of code from a project which may contain millions of them could result in a complete system compromise.

The search of new vulnerabilities, denoted as *0-day*, is attracting constant attention and resources from security practitioners. Indeed, such vulnerabilities are highly marketable (Spectre [68], Meltdown [77], Hertzbleed [130]) and used in security conferences as a measure of success. They also have monetary value. A company may offer a bug bounty program rewarding researchers reporting security issues, and *0-day*-brokers provide a gray market to sell (weaponized) vulnerabilities. Payouts can reach millions of dollars [104], supporting a complete ecosystem within the cybersecurity world. As already mentioned in the previous chapter, let us not forget that vulnerabilities are also used as cyber weapons (i.e. WannaCry).

A *0-day* is associated with a defined software and a version (or a range of versions). This stems directly from the research itself because an attacker is usually looking at a target. However, due to code reuse across projects and the usage of open-source components, two projects may share a portion of their codebase. Thus, a vulnerability affecting the former could also affect the latter. While the research towards finding new *0-day* is massive, the study of their impact is much less explored. The deployed systems' exposition, the patch propagation delay, or their extension towards other contexts (different versions, hardwares, ...) are important pillars for the whole ecosystem security.

### 2.1.2 Reporting Security Defects Using CVEs

Common Vulnerabilities and Exposures (CVE) is the *de facto* standard for publishing advisories on vulnerabilities. The program's mission is to identify, define, and catalog publicly disclosed cybersecurity vulnerabilities [81]. This program is led by the MITRE Corporation since 1999, and financed by the U.S. Department of Homeland Security. MITRE does not issue every CVE. Their partners, CVE Numbering Authority (CNA), are also allowed to create new CVE within their scope. In May 2022, MITRE listed

216 partners ranging from well-known companies (e.g. Airbus, Google), Open-Source foundations (e.g. Debian, Fedora), or national agencies (e.g. from Spain or Switzerland).

Each CVE entry abides by a schema. Its last version (5), available on the organization's GitHub repository[1], is presented below. Only some fields are required:

- `dataType`: always `CVE_RECORD`;
- `dataVersion`: the current version of the standard;
- `cveId`: the CVE-ID, a unique identifier;
- `assignerOrgId`: the organization to which the CVE was originally assigned;
- `state`: either `PUBLISHED` or `REJECTED`.
- `cnaPublishedContainer`: information on the CNA publishing the advisory.

Every other field is optional, but the ones below are usually filled:

- `description`: summary of the vulnerability in plain text;
- `reference`: complimentary resources to understand the vulnerability;
- `datePublished`: the record date.

As of May 2022, the CVE database contained more than 234,000 vulnerability entries. The first is `CVE-1999-0001` in 1999 and the last was `CVE-2022-30114`. A CVE entry does not mean a vulnerability is fixed, but only that it is disclosed to a CNA.

To assess the severity of a vulnerability, the NVD created the Common Vulnerability Scoring System (CVSS), an open framework to grade vulnerabilities. Scores range from 0 to 10 and increase with the vulnerability severity. This scoring system considers exploitability metrics (e.g. attack vector, the need of user interaction), and impact metrics (e.g. on integrity, availability, confidentiality) to compute the final score. This score cannot be computed automatically as it depends on the analyst's interpretation of the vulnerability. Therefore, CVSS's scores are updated to reflect discoveries or better understandings of the vulnerability impact.

### 2.1.3 Security Patches and Patch Propagation Delay

**Definition 2.1.** *A software **patch** is a set of changes between two versions [129].*

[1] https://github.com/CVEProject/cve-schema

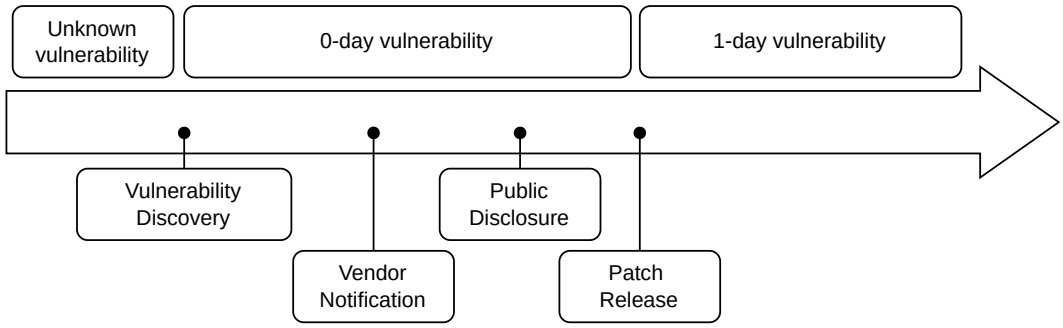


Figure 2.1: Vulnerability Lifecycle

Patches are natural in a software lifecycle. They can add features, remove unneeded parts, or fix bugs. Security patches address bugs with security implications: vulnerabilities. In versioning systems (e.g. git), each commit is considered as a patch. A commit represents a modification of the program state and can modify multiple files. In this thesis, we only consider *security patches* and use the terms *patches* or *security patches* equally. As software code is organized in functions, we provide a more tractable definition of patches below, which will be used in the next chapters.

**Definition 2.2.** *A **patch** is a finite collection of function changes.*

**Definition 2.3.** *A* **1-day** *is a vulnerability for which a patch has been released.*

A vendor may be notified of a vulnerability, either by an external party through bug bounties or through a coordinated disclosure, by internal or external audits, or through public advisories. After their notification, some vendors may release a patch fixing the bug and mitigating the risk [46]. While in a best-case scenario, all reported vulnerabilities are fixed, a vendor may choose to ignore the notification as its interests are not always aligned with those of its users.

A vulnerability, for which there exists a patch, ceases to be a *0-day* and becomes a *1-day*, as at least *1 day* has passed since the patch release. The literature also uses the term $n$-days, with $n$ the number of days since the patch. In this manuscript, we prefer to use *1-day* for every vulnerability associated with an existing patch, because the number of days is irrelevant to our work.

The existence of a patch is a necessary, but insufficient, step to ensure every instance of the program is corrected. Indeed, several reasons (software incompatibility, time constraints) prevent stakeholders from applying a patch promptly. The delay between a patch release and its application to every impacted software is called the *patch propagation delay*. During this time window, end users are at risk because their software is vulnerable to a publicly known issue. The different steps of a vulnerability life are listed in Figure 2.1.

### 2.1.4 Defenses Strategies Against 1-day Vulnerabilities

A prerequisite for building and maintaining an appropriate security posture is to know the different components of an information system. Executive Order 14,028 in the United States [54] mandates every purchaser of a software to require a Software Bill of Materials (SBOM), listing all the components (and their versions) used in the product. This requirement helps customers establish their software and hardware inventory used in their information systems.

The first line of defense against *1-day* vulnerabilities is to regularly update software. However, this task is far from trivial. Clément Elbaz identified three main obstacles when deploying a security patch [38]:

- The inherent difficulty in establishing the list of software deployed in the current installation.
- The need to understand whether the deployed software is indeed vulnerable to a vulnerability.
- The compatibility between the update and the usage of the software.

The *1-day* exposition problem must still be addressed when applying software updates is impracticable. In such cases, positioning the defense at a higher level (e.g. system, network) is also a valid option. Intrusion Detection Software (IDS), either based on signatures or behavior can be placed on the host or on the network, to analyze and detect malicious activity before reporting or blocking it. Nonetheless, relying solely on those tools is not ideal as they may hinder regular user activity or fail to detect an attack.

## 2.2 Embedded Systems

For years, vulnerability research was mostly focused on desktop systems such as *Windows*, *macOS*, and *Linux*. However, the rise of mobile systems has shifted the focus towards mobile platforms and embedded devices (routers, VoIP phones, cameras, household appliances) that are even more numerous.

### 2.2.1 Internet of Things Ubiquity

Embedded devices are omnipresent in every modern life aspect. They are used in most complex systems, like transportation (e.g. cars, planes, rockets), wearable devices (e.g. watches, armbands), household appliances (e.g. fridges, dishwashers), network devices (e.g. routers, gateways), or healthcare (e.g. insulin pumps, hearing aids). Nowadays, the prefix *smart* can be used together with almost any object, as long as it uses

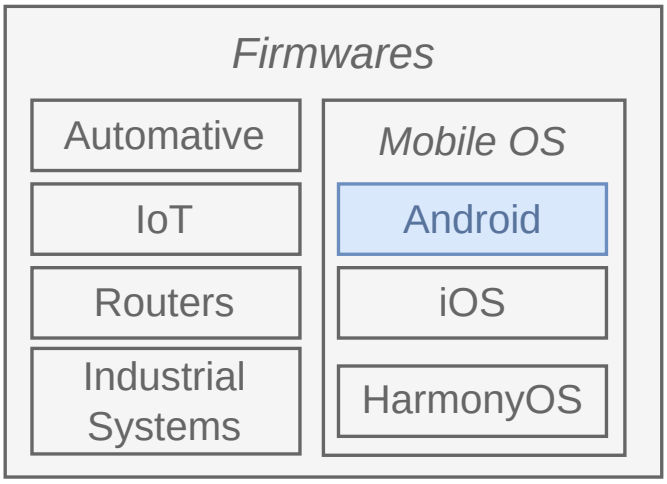


Figure 2.2: Firmware Overview

some *intelligence*[2]. A precise definition of an embedded device is thus complex to establish.

Embedded systems are often insecure [24, 123]. This problem may stem from the short *time-to-market* cycle for products where security is not a selling factor. It discourages manufacturers from investing time and resources in this area. Therefore, they represent valuable targets for attackers because they are often connected and less supervised [5] than regular devices.

To perform their tasks, embedded devices run some software called *firmware*. Firmware is different from general-purpose software as they are (1) tightly coupled with the underlying hardware, and (2) installed by the manufacturer instead of the user [82]. If some devices are running a well-known OS (e.g. Linux), *flat* firmware are also used, where the whole code comes as a binary blob.

Some embedded devices have reached unprecedented adoption levels in history: smartphones. In 2021, Odea reported more than 6 billion smartphone subscribers [95]. We decided to use phone devices OS, and specifically, Android, as a representative example for embedded devices analyses.

### 2.2.2 Android

*Android*[3] is today the most common OS in the world [94]. It runs on various devices of all forms factors from smartphones to household appliances, televisions, cars, and

[2]Or even just some digital features.
[3]https://www.android.com/

numerous smart objects. The heart of Android, AOSP, developed by Google, is based on a Linux kernel and is openly available. AOSP is described in Section 2.3. This heart is insufficient to power every device, as Google cannot include drivers for every hardware configuration. Therefore, OEM extend the base image of the system to their needs. These modifications can modify the system even deeply to create a customized user experience, usually called *Android ROMs*.

The freedom for vendors to modify and adapt Android to all devices is a double-edged sword. On one hand, it helped to grow one of the largest mobile ecosystems by allowing manufacturers to compete on the hardware level while outsourcing the software to Google. The model chosen by Google for Android resembles the one established by Microsoft on the desktop. Both manufacturers simply license[4] their system and do not manufacture hardware. The line is finally blurring as Google has become a manufacturer with its Pixel line (and the acquisition of Nest)[5]. Google's model contrasts with Apple's model, as Apple controls both the software (*iOS*) and the hardware of their mobile devices.

On the other hand, one of the biggest challenges for Android security resides in this version fragmentation. To differentiate from other manufacturers and increase their market share, vendors are offered few choices. They can either modify the hardware (new optics, bigger screens, ...) or customize the user interface to propose new functionalities. These changes may be deeply interleaved with code owned by Google and increase the difficulty of updating a device, or simply may be adding new vulnerabilities as uncovered by Project Zero [63].

The update process of Android is tedious as multiple parties are involved. The update is created by Google and released as a new Android version. Then, the chipset manufacturer, the vendor, and sometimes, the carrier, validate that the update does not break the device. To reduce friction, Google acts in two ways. First, through the Android Compatibility Definition Document (CDD), which lists the requirements needed for a device to use the brand *Android*. The CDD is distributed with the Compatibility Test Suite (CTS), a suite of unit tests to verify automatically that the device meets the requirements and serves as a guide for device manufacturers. By enforcing that every device abides by the CDD rules, Google enforces a common baseline for its OS. Second, Google limits the risk induced by Original Equipment Manufacturer (OEM) modifications by drawing a clear boundary between the code under their control and the one of other stakeholders. Since Android 8, the project Treble [58] creates a new interface between vendor components and generic AOSP code. However, Treble is still relatively recent, and its requirements are neither always well implemented nor checked [102].

[4]The license is only required to run the Google Mobile Services (GMS) who are not required for Android *per se*.

[5]The analogy remains valid as Microsoft also launched a Surface product line.

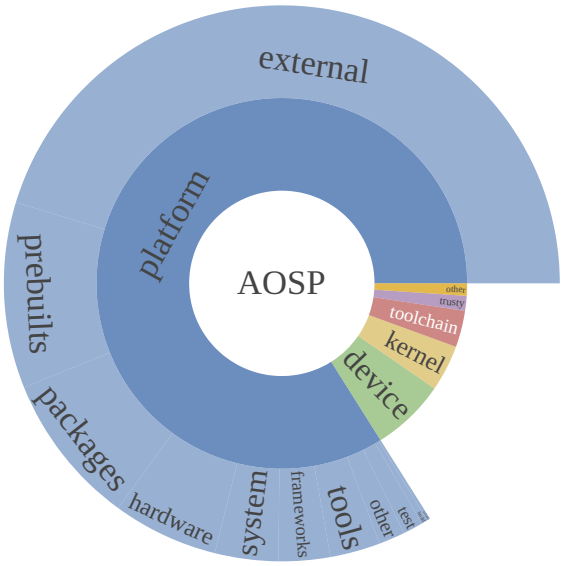


Figure 2.3: Repartition of AOSP Projects

## 2.3 Deep Dive Inside AOSP

AOSP is not monolithic: it is composed of about 2,200 projects[6], ranging from well-known open-source projects (e.g. Linux, curl) to more specific ones (e.g. device descriptions, Dalvik). AOSP is an information security goldmine, and this section describes its inner workings.

### 2.3.1 Overview

AOSP is led by Google, which maintains and develops Android. Although Android is open-source, each version is first developed privately by Google (with its partners) before being opened for a yearly release. This organization allows Google to steer Android development towards its choices, handle confidential information shared by OEM, and focus public attention on the latest released Android version.

As previously stated, AOSP is an aggregation of projects which repartition is depicted in Figure 2.3. A large majority of them come from *external* sources. For example, the copy of `chromium`, some Python packages (i.e. `numpy`, `requests`), Rust crates, and many others are embedded directly inside AOSP. The external projects live independently from their *upstream* parent. Google regularly propagates upstream changes. However, this process creates a delay, and every dependency is not always up to date in AOSP root tree. For example, in March 2022, `curl`'s AOSP version was behind the official one[7] by one minor version.

[6] As of March 2022
[7] `https://curl.se/`

AOSP is almost fully self-contained, and the requirements for building the system are low. This is possible due to the presence of *prebuilt* packages to perform the build. For instance, a binary version of `Python` is directly versioned in the source tree. The *packages* ensemble gathers all initial APK. Not all of them end up on end-user devices as OEM are keen to replace them with their variants.

### 2.3.2 Terminology, Manipulation, and Usage

Android releases are sanctioned by versions. The first releases were named after treats (Gingerbread, Lollipop) but since Android 10, they just have incremental numbers (Android 10, Android 12). Each version is associated with an Application Programming Interface (API) level. It determines the list of features available for external developers. Finally, each build for a device is associated with a unique *build ID*. This identifier includes the Android version, the build date, an individual increment, and the SPL number.

Some *build IDs* are also named as tags (e.g. *android-12.0.0_r32*)[8]. As tags are associated with a device, multiple ones coexist at the same time, even for different versions. The tag order is also to be considered with caution, as a higher tag number does not necessarily mean it was released after. For example, *android-10.0.0_r1* was released in September 2019 while *android-9.0.0_r61* was released in October 2020.

A tag is associated with a list of projects. Before July 2021, this list was stored in a *Manifest*, an XML file associating each project with its path in the Android repository. The precise version of each sub project is determined by the release tag (e.g. AOSP's `curl` has an *android-12.0.0_r31* tag). Since July 2021, Google has started a migration to *superprojects* that use `git` superprojects and submodules. This migration is still a work-in-progress and Google keeps the backward compatibility between the two methods. Thus, the transition is transparent for the user.

The *Manifest* lists all projects used by a version. However, manipulating and synchronizing each project individually is tedious. Therefore, to ease the usage, Google released `repo`[9], a tool abstracting the inner workings of AOSP. It prepares the build layout for further manipulation. In consequence, the changes induced by superprojects may deprecate `repo` in the coming years.

### 2.3.3 A Huge Scale Project: AOSP by the Numbers

AOSP sources are contained in its *mirror*, which encompasses all the `git` repositories used at least once in the project. The *mirror* weighed about 1 TB in March 2022 and is ever-growing. It is composed of 2204 different git projects. Half of this space

[8] This is not the case for every build ID, and some are missing from the official list.
[9] https://gerrit.googlesource.com/git-repo/

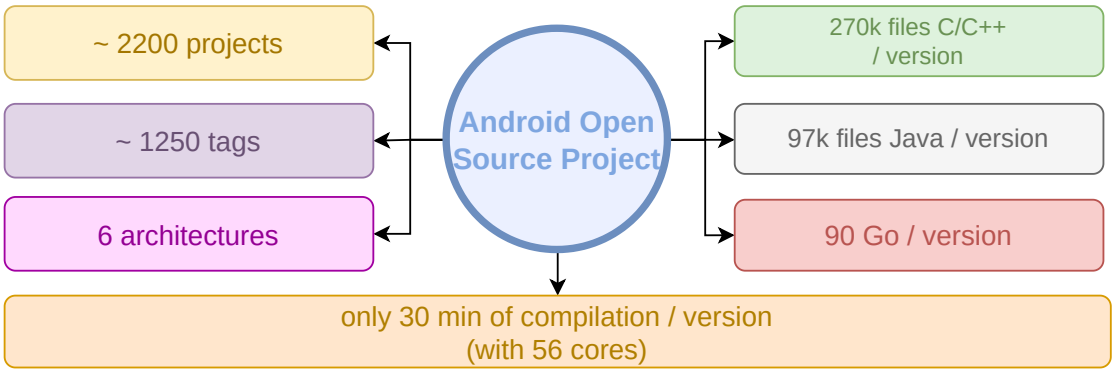


Figure 2.4: Key Figures in AOSP

(417 Gb) is used to store device data (notably kernels) for various devices. The most common are the Google ones (the Pixel line), but some for Asus or Motorola[10] also lie in this repository. More interestingly, kernels for generic devices are also available. They can run on every hardware compatible with Treble and help developers to test their applications with the newer Android release. Another sizeable chunk in the *mirror* is the `prebuilts` one, where most build dependencies for AOSP are stored. This allows to bootstrap the project easily as compilers, Software Development Kit (SDK), and build generators are already present as pre-compiled binaries.

Working with the full mirror is rather impracticable. Most of the time, a user is only interested in a single version (e.g. tag) of Android. The last version, *android-12.0.0_r32* weights 90 Gb and contains ≈990,000 files. Numerous languages coexist in AOSP: C/C++ (about 27% of the files), Java (10%), LLVM IR (4%), Python (3.5%), Go (1.5%), or Rust (1%). All languages with more than 1 million lines are listed in Figure 2.5. Each project contains the information needed to build itself using AOSP build system Soong (detailed in Chapter 6).

Building an Android version from the source code is straightforward but is lengthy: about 30 minutes on a 56 cores machine[11]. The build output weights around 70 additional gigabytes. The output is immediately usable with an emulator for testing purposes, and *flashable* on a device using `fastboot`[12].

### 2.3.4 AOSP as a Typical Complex Embedded System

As Android mostly powers embedded devices (e.g. smartphones, watches, cars, ...), it is an appropriate starting point for looking at embedded systems generally. Nonetheless,

[10] Previously owned by Google.
[11] While the size of Android version increase, the compilation time seems to decrease.
[12] Part of the Android platform-tools https://developer.android.com/studio/releases/platform-tools

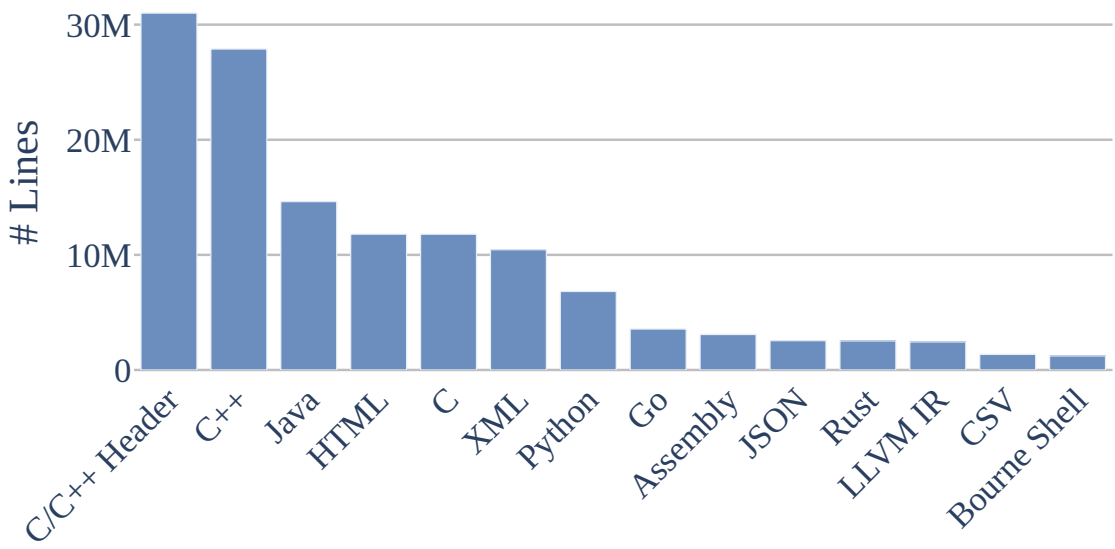


Figure 2.5: Languages With More Than 1 million Code Lines

multiple differences exist between Android and generic embedded systems. The most notable of these are:

- Android is based on Linux and provides a filesystem abstraction.
- Android runs on multiple hardwares and implements a Hardware Abstraction Layer (HAL).
- Android is extensible with applications and offers external developers an SDK.

Nonetheless, Android and AOSP display similarities with embedded systems:

- Android powers *mobile* devices with limited power.
- Android is connected through various mediums (i.e. WiFi, cellular data, Bluetooth, NFC).

Moreover, using Android to study embedded systems offers some advantages. With 2,200 projects and close to 1 million files, AOSP is a gigantic open-source codebase. For example, its size allows studying how an analysis scales on a large codebase.

The ecosystem around Android also makes working with the project scale manageable. Google developed a tools suite to download, search, and build AOSP. For example, the build system allows compiling both old Android versions and newer ones on the same system. Although this tooling was initially created for the platform developers and not for security analysis, it is adaptable for various needs. Android's ecosystem vitality

makes the imminent disappearance of the current tooling unlikely. On the other hand, it is challenging to keep up with the rapid pace of change.

Finally, from a security viewpoint, Google's handling of security reports is compelling, especially for *1-day* vulnerability research. The monthly SPL lists all the vulnerabilities fixed during the last update and it is achievable to leverage this information to construct a dataset from this data. Our research on this aspect is detailed in Chapter 4.

Because Android is a modified Linux system, it uses a filesystem abstraction. Thus, the analyses we develop work not only on Android but also on any system offering this abstraction level. For example, the patch detection system introduced in Chapter 5 could work on Windows. However, this can represent a limitation because it is not transposable for smaller devices using *flat* binaries.

## 2.4 Representation of Code

To reason about programs, the scientific community uses various code representations. While providing a complete overview of these representations would be interesting, it is not relevant for the remaining of our work. Instead, we briefly present the Code Property Graph (CPG) introduced by Yamaguchi *et al.* in 2014 [141], because it is used in the following chapters.

CPG are a combination of several code representations that we detail below:

- An Abstract Syntax Tree (AST).
- A Control Flow Graph (CFG).
- A Program Dependency Graph (PDG).

### Abstract Syntax Tree

An Abstract Syntax Tree (AST) is an ordered tree representing the code syntax. This intermediate representation is generally built early by code parsers to encode how code expressions and statements are nested. They are distinguished from *concrete* syntax trees because they no longer represent syntax artifacts irrelevant to the code comprehension (i.e. grouping parentheses). AST are not only used to derive other code representations, but also to perform code comparison [27], evolution [84], and summarization [21].

### Control Flow Graphs

A Control Flow Graph represents a single procedure code as a directed graph, where the nodes are the basic blocks and the edges the jumps. CFG are the standard code representation in disassemblers (e.g. `IDA` [55]) and have been used for various applications

like code similarity [117] or malware signatures [15]. In a CFG, each edge is labeled with the jump predicate, either `True`, `False`, or `Unconditional`. CFG examples are listed in Figure 1.2a and 1.2b.

### Program Dependency Graph

**Definition 2.4.** *A **Program Dependency Graph (PDG)** represents a program as a graph in which nodes are statements and predicate expressions, and the edges represent both the data values on which the node's operation depends, as well as the control conditions on which the execution of the operation depends[43].*

The PDG is a code representation introduced by Ferrante *et al.* [43]. Because it explicitly represents both data and control dependencies, the graph can be used to detect parallelism or perform program slicing [28]. It has also been used to detect software vulnerabilities [74] and detect code clones [4].

### Code Property Graph

Finally, a Code Property Graph is a property graph combining the three previous representations inside a single graph. The definition proposed by Yamaguchi *et al.* is as follows:

**Definition 2.5.** *A **Code Property Graph** $G = (V, E, \lambda, \mu)$ is a directed edge-labeled attributed multigraph constructed from the AST $A$, the CFG $C$ and the PDG $P$:*

- *$V$ is the set of nodes and $V = V_A$.*
- *$E$ is the set of directed edges and $E = E_A \cup E_C \cup E_P$*
- *$\lambda : E \to \Sigma$ is an edge labeling function and $\lambda = \lambda_A \cup \lambda_C \cup \lambda_C$*
- *$\mu : (V \cup E) \times K \to S$ with $K$ a set of properties keys and $S$ the set of property values is a function to assign properties to nodes or edges.*

This model can be used to search for new vulnerabilities, and more broadly, to reason about specific program properties. The main tool to manipulate CPG is *graph traversals*, which are chainable functions used to iterate over the graph's nodes.

**Definition 2.6.** *A **traversal** is a function $\mathcal{T} : \mathcal{P}(V) \to \mathcal{P}(V)$ that maps a set of nodes to another set of nodes according to a property graph $G$, where $\mathcal{P}$ is the power set of $V$ [141].*

*Traversals* are used to encode *semantic* queries inside the graphs. For instance, it is possible to write a query that searches every function having at least one condition. In Chapter 5, we use CPG to characterize AOSP patches by looking at the differences between the CPG generated from a project vulnerable and the one generated from a fixed version.

**Listing 2.1** Exemplary Code Sample (from [141])

```
void foo()
{
    int x = source();
    if (x < MAX)
    {
        int y = 2 * x;
        sink(y);
    }
}
```

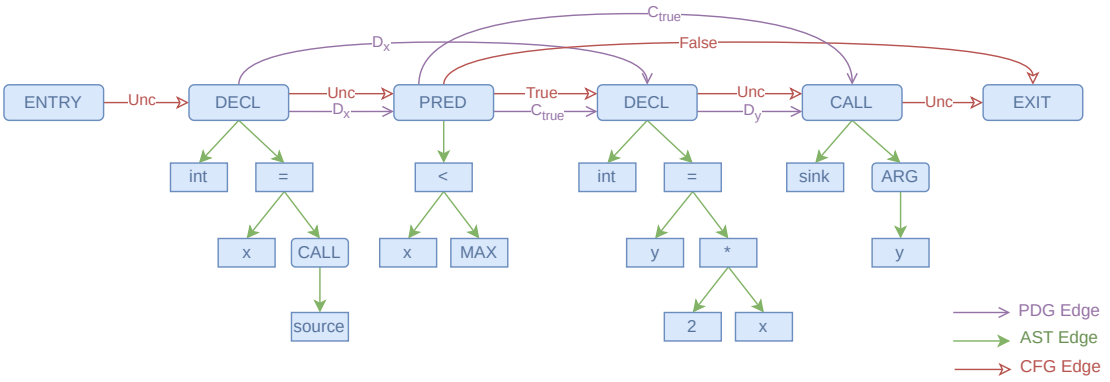


Figure 2.6: Example of CPG (adapted from [141])

To illustrate a CPG, we reproduce in Figure 2.6 the CPG derived from the code in Listing 2.1.

Chapter 3

# Firmware Patch Matching Problem

## Contents

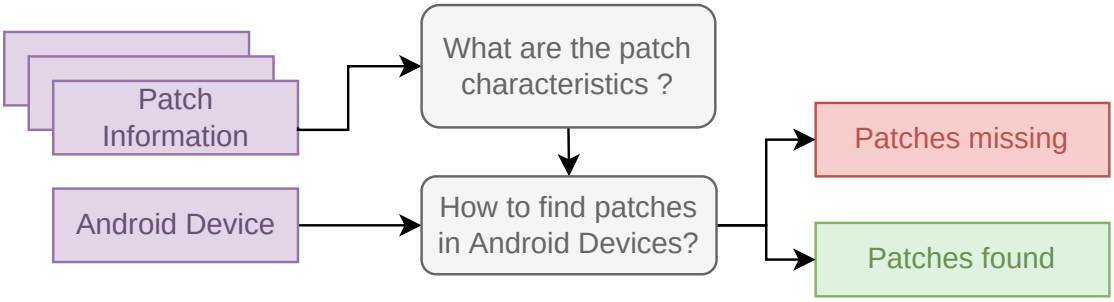


Figure 3.1: How to Find Patches in Android Devices

As highlighted in Chapter 1, detecting if a piece of software is patched against *1-day* vulnerabilities is crucial to assess system's exposure to known threats. Indeed, because of the patch propagation delay or discontinued support, recurrent security patches are insufficient to prevent these vulnerabilities.

This chapter answers the two main questions illustrated in Figure 3.1. First, it starts with a survey of existing works on the *patch presence test* [143]. We conduct an extensive state-of-the-art review before exposing the remaining open challenges to improve this problem solution.

A necessary step for researching generic patches on systems is to define patches and their main characteristics. Section 3.2 leverages a CPG diffing approach for characterizing AOSP patches. This study allows establishing patches' common traits that could be leveraged in a patch probing system.

Then, we introduce a formalization of the FMP and its generalization to the Firmware **Patch** Matching Problem ($\Xi-$FMP) which extends the *patch presence test* at a firmware scale (Section 3.3). We also devise a strategy, *F-S-M*, to solve the FMP in three consecutive steps. We continue this chapter by discussing how to build signatures adapted to *F-S-M*. Finally, in Section 3.5, we describe our system, **QSig**, implementing the *F-S-M* to solve the $\Xi-$ FMP.

## 3.1 State of the Art

### 3.1.1 Introduction

**Definition 3.1.** *The **patch presence test** checks whether a specific patch has been applied to an unknown target, assuming the knowledge of the affected function(s) and the patch itself [143].*

| | | Input | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Year | Source | Binary | Approach Type | Diffing | Automatic Signature | Machine Learning | Intermediate Representation |
| SPAIN [139] | '17 | | ✔ | Static | BinDiff | ✔ | ✘ | ✘ |
| FIBER [143] | '18 | ✔ | ✔ | Static | N/A | ✔ | ✘ | ✔ |
| 1dVul [98] | '19 | | ✔ | Hybrid | BinDiff | ✔ | ✘ | ✘ |
| PATCHECKO [116] | '20 | | ✔ | Hybrid | N/A | ✔ | ✔ | ✘ |
| BINXRAY [138] | '20 | | ✔ | Static | Custom (basic blocks) | ✔ | ✘ | ✘ |
| BScout [29] | '20 | ✔ | ✔ | Static | N/A | N/A | ✔ | ✘ |
| PDiff [60] | '20 | ✔ | ✔ | Static | N/A | ✔ | ✘ | ✘ |
| VIVA [135] | '21 | | ✔ | Static | Custom (basic blocks) | ✔ | ✘ | ✘ |
| QuickBCC [59] | '21 | | ✔ | Static | diaphora | ✘ | ✘ | ✔ |
| PMatch [70] | '21 | ✔ | ✔ | Static | diaphora | ✔ | ✔ | ✘ |
| P1OVD [72] | '22 | ✔ | ✔ | Static | N/A | ✔ | ✘ | ✘ |

Table 3.1: Survey of Various Patch Presence Works.

The main approach to overcome vulnerabilities is patching. However, it is challenging to ensure that patch propagation follows the code propagation, especially in a timely manner. Hence, known unpatched vulnerabilities, *1-day* vulnerabilities, are a serious threat. Thus, solutions need to accurately scan complete codebases and perform a *patch presence test* [143].

Intuitively, the first idea is to perform a *vulnerable code search* using techniques searching for unpatched code on source code [66, 75, 134], at an image level [8, 35], or on binary code [31, 56, 42]. As differences between patched and vulnerable code are often subtle, these tools cannot distinguish between vulnerable and fixed code and are unreliable to answer the patch presence.

Table 3.1 surveys works addressing the *patch presence problem* and lists their main characteristics. In the following paragraphs, we detail each of these characteristics in the following paragraphs.

### 3.1.2 Inputs: Binaries Only or with Source

| Input | SPAIN | FIBER | 1dVul | PATCHECKO | BINXRAY | BScout | **PDiff** | VIVA | QuickBCC | PMatch | **P1OVD** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Source | | ✔ | | | | ✔ | ✔ | | | ✔ | ✔ |
| Binaries | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |

Table 3.2: Inputs Types

Approaches listed in Table 3.1 derive a signature from each patch. A first option is to use information gathered from the source code to generate this signature, even if the final objective is to search for patches in binaries [29, 70, 60, 72, 143]. This helps to precisely pinpoint the changes induced by the patch but asks to also identify the source-to-binary mapping: how a specific source code line is translated to assembly.

P1OVD [72] uses this approach. It generates a PDG for both source and binary code and creates a mapping for each node of the former to the latter. After identifying root instructions, a notion formalized by Zhang *et al.* [143], the authors perform a function symbolic execution and extract the AST for each root instruction. Their *patch presence test* is two-fold: first, a structural matching is performed to check AST similarities, and secondly an equation matching is performed verifying condition equivalences using Z3 [32].

PDiff [60] uses the patch source code information: it compiles the kernel with and without the patch with debug symbols. They detect the affected function names by parsing the patch file and use the symbols to detect the function at the binary level. Then, they generate a patch summary by extracting digests from patch-affected paths and comparing these digests for the *patch presence test*.

Relying on the source code for patches restricts the solution's applicability to vulnerabilities in open-source components. However, it allows targeting more specific changes. It is also possible to generate signatures directly from the difference between two binaries [98, 116, 135, 59, 138].

### 3.1.3 Static or Hybrid Approaches

| Approach Type | SPAIN | FIBER | **1dVul** | **Patchecko** | BinXRay | BScout | PDiff | Viva | QuickBCC | PMatch | P1OVD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Static | ✔ | ✔ | | | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| Hybrid | | | ✔ | ✔ | | | | | | | |

Table 3.3: Approach Types

Binary analysis techniques are divided into two main categories: static and dynamic. While the frontier between the two types is blurry, we define a technique as static if it does not directly execute the binary code. Finally, we define an approach as hybrid if it combines both static and dynamic techniques.

Most works presented in Table 3.1 are static. Indeed, static analyses are easier to bootstrap because they do not require configuring an execution environment. However, both 1dVul [98] and Patchecko [116] leverage hybrid approach.

1dVul [98] *target-oriented input generation* combines fuzzing [49] and symbolic execution [67] to generate an input crashing the vulnerable program. A candidate is labeled as patched only if it does not crash when fed the generated input.

Patchecko [116] also uses a hybrid approach. It starts with a deep neural network to determine whether two candidate functions are similar. Then, it validates its findings using a dynamic analysis checking if the two traces are semantically equivalent.

While more accurate, dynamic methods are slower and require additional work to set up an execution environment for unknown code. Moreover, executing code for embedded devices firmwares without the hardware is an open problem [83].

### 3.1.4 Diffing Approaches

| Diffing | SPAIN | FIBER | 1dVul | PATCHECKO | **BinXRay** | BScout | PDiff | **Viva** | QuickBCC | PMatch | P1OVD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| BinDiff | ✔ | | ✔ | | | | | | | | |
| diaphora | | | | | | | | | ✔ | ✔ | |
| Custom | | | | | ✔ | | | ✔ | | | |
| N/A | | ✔ | | ✔ | | ✔ | ✔ | | | | ✔ |

Table 3.4: Diffing Solutions

When starting with two binaries, one of the first tasks to generate a signature for a patch is detecting within the binary where the patch is located. This can be reduced to a *binary diffing* problem in a simplified case: both binaries are similar and differ only in a few locations. Moreover, most approaches use debug symbols because generating the signature is an offline process [70, 116].

Thus, the diffing problem resolution is often offloaded to other specialized tools like BinDiff [147] in SPAIN [139] and 1dVul [98], or diaphora [62] in QuickBCC [59] and PMatch [70].

Another approach is to implement a customized diffing approach. For instance, BINXRAY [138] generates a one-to-one basic block mapping between the fixed and vulnerable versions. It uses the mapping to recover basic block traces that changed between the two versions and store them inside a signature. BINXRAY detector checks whether the traces found in the target binary are closer to the fixed or vulnerable trace sets.

VIVA [135] also implements a customized algorithm to perform the diffing. To generate a mapping of blocks from the vulnerable to the fixed binary, it uses a context-sensitive hash-based algorithm. Then, VIVA uses a program slicing technique to build traces that contain the vulnerability execution path. Of note, VIVA follows function calls when generating a trace to defeat function inlining.

### 3.1.5 Signatures Types

| Signatures | **SPAIN** | FIBER | 1dVul | PATCHECKO | BINXRAY | **BScout** | PDiff | VIVA | QuickBCC | PMatch | P1OVD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Patch Based | | ✔ | ✔ | ✔ | ✔ | | ✔ | ✔ | ✔ | ✔ | ✔ |
| Patch Summary | ✔ | | | | | | | | | | |
| No Signature | | | | | | ✔ | | | | | |

Table 3.5: Signatures Types

SPAIN [139] starts by diffing two binaries. However, it then searches for security changes within the difference. For such changes, SPAIN summarizes the patch pattern using semantic traces containing the newly added instructions data flow. Thus, their signatures are not patch signatures *per se*, but patch pattern signatures that find similar patches in other codebases.

Contrary to the other approaches presented here, BScout [29] checks for the presence of a whole patch in Java executable *without* generating a signature. It directly uses the patch and measures the patch line proportion present in the target. However, BScout only works on Java executable.

Other approaches presented in Table 3.1 generate signatures for each searched patch.

### 3.1.6 Usage of Machine Learning

| Machine Learning | SPAIN | FIBER | 1dVul | PATCHECKO | BINXRAY | BScout | PDiff | VIVA | QuickBCC | **PMatch** | P1OVD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Yes | | | | ✔ | | ✔ | | | | ✔ | |
| No | ✔ | ✔ | ✔ | | ✔ | | ✔ | ✔ | ✔ | | ✔ |

Table 3.6: Usage of Machine Learning

Machine Learning is used for numerous tasks in computer science. Because it shows promising results for the *binary diffing* problem [80], several approaches leverage this

tool for the *patch presence problem* [29, 70, 116].

PMatch [70] leverages an unsupervised Natural Language Processing (NLP) algorithm to generate the binary code semantic representation. More precisely, it computes an embedding from the normalized disassembly using a continuous bag of words neural network. The Smooth Inverse Frequency (SIF) [6], a sentence embedding technique, transforms the instruction embedding into a block embedding. Then, they use the similarity score between the target and fixed embeddings as a patch detector.

### 3.1.7 Architectures Support

| Intermediate Representation | SPAIN | **FIBER** | 1dVul | PATCHECKO | BINXRAY | BScout | PDiff | VIVA | **QuickBCC** | PMatch | P1OVD |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Yes | | ✔ | | | | | | | ✔ | | |
| No | ✔ | | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ | | ✔ | ✔ |

Table 3.7: Intermediate Representation and Architecture Support

Methods targeting binary code must consider the multiple assembly languages. Two main architectures coexist, Intel x86, mostly used for desktops and servers, and ARM, for mobile and embedded devices. Historically, they both used 32-bit address size, but the hardware evolution has now led to 64-bit address size. A solution for the *patch presence problem* can be generic or tailored to a specific architecture.

Another option is to generate a signature on an Intermediate Representation (IR). Using an IR allows developing methods that work for every architecture supported by the IR. Moreover, it entails writing optimizations or normalizations passes only once. In FIBER [143], Zhang et Qian identify root instructions as Data Flow Graph (DFG) leaves and use them in their signature. Since they perform their analyses on VEX, the IR from Valgrind [85], their design is architecture agnostic from the start.

QuickBCC [59] generates its signature by hashing VEX IR instructions strands that changed between the two vulnerable and fixed versions of the binary. The similarity is performed using *n-gram similarity* metrics on the removed and added instructions strands sets. In their work, they also leverage VEX static single assignment property to reduce the strand generation algorithm complexity.

All other approaches listed in Table 3.1 work directly on the assembler language. Thus, they often support only a single architecture.

### 3.1.8 Conclusion

In this section, we established a survey of the current *State of the Art* in the *patch presence test* domain. Multiple approaches have been presented over the last years using different techniques. However, multiple challenges remain open such as:

- How to perform a cross-architecture matching?
- How to extend the *patch presence test* to complete firmware?

## 3.2 Patch Characterization

Understanding what security fixes are made of is important to identify them in other workflows (e.g. on silent fix detection, or in patch signature generation). Little research has been made to formalize and characterize such changes on source trees. In this section, we introduce a taxonomy of vulnerability fix commits profiles.

### 3.2.1 Formalization

Let define a project $P$ as a sorted sequence of commits.

$$P = \{c_0, c_1 \ \ldots, c_{i-1}, c_i, \ \ldots\}$$
$$\Delta_i = P^{i-1}..P^i$$

where $c_i$ is a commit with parent $c_{i-1}$. We denote $P^i$ the project state after commit $i$ and $\Delta_i$ the difference between $P^{i-1}$ and $P^i$. This model is simplified and, for example, ignores branching mechanisms. Yet, it remains sufficient for our needs.

In the CPG (defined in Chapter 2) of a project $P$, some vertices represent the program functions $f$. If we extract the CPG induced by a function root node, we denote it $G_P^f$. This is a slight notational abuse[1] but it simplifies the following formalization.

While $G_P^f$ is also a CPG, let us only consider $V$ its set of vertices. Let us denote the node types as $\Phi = \{String, Constant, Call...\}$ and define $\psi$, a labeling function associating a type to a node.

$$\begin{array}{rccl} \psi\colon & V & \longrightarrow & \Phi \\ & v & \longmapsto & \{\text{String}, \text{Constant}, ...\} \end{array}$$

Let us assume a vulnerability fixed in $P$ by the commit $c_i$. We denote $G_{fix}$ the graph $G_{P^i}$, i.e. the CPG of project $P$ right after the application of $c_i$. Thus, $G_{P^{i-1}}$ corresponds to a vulnerable state of $P$ regarding this vulnerability, and will be denoted $G_{vuln}$.

[1] We should construct the collection of subgraphs $\{G_P^f : f \text{ function in } G_P\}$

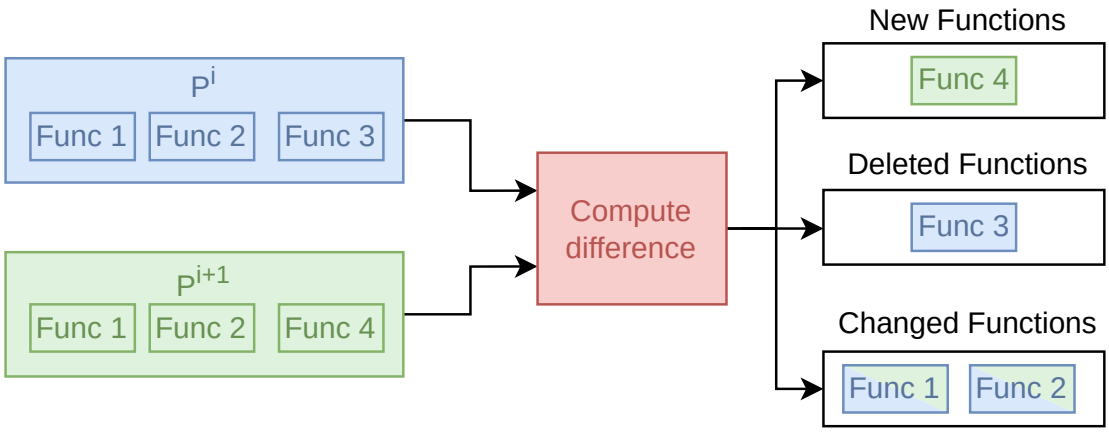


Figure 3.2: Computing $\mathbb{N}$, $\mathbb{R}$ and $\mathbb{D}$

We are now interested in the difference between $G_{vuln}$ and $G_{fix}$. To express this difference formally we consider, for each function $f$ in $G_P$, the following set

$$\mathbb{D}^f = \left\{(\text{add}, \psi(v)) : v \text{ a vertice in } G^f_{fix} \setminus G^f_{vuln}\right\} \cup \left\{(\text{del}, \psi(v)) : v \text{ a vertice in } G^f_{vuln} \setminus G^f_{fix}\right\}$$

where the first subset corresponds to function $f$ nodes added during the commit $c_i$ and the second one to the deleted ones. Note that this set is empty when no change appears for $f$ in $c_i$. To be exhaustive, we also consider cases where functions are added or removed and define the two following sets:

$$\mathbb{N} = \{f : f \in G_{fix} \setminus G_{vuln}\}\,, \quad \mathbb{R} = \{f : f \in G_{vuln} \setminus G_{fix}\}\,.$$

To summarize, the changes of $P$ between $c_{i-1}$ and $c_i$ are represented by the tuple $(\mathbb{N}, \mathbb{R}, \mathbb{D})$ for the new ($\mathbb{N}$), removed ($\mathbb{R}$), and changed ($\mathbb{D}$) functions where $\mathbb{D} = \{\mathbb{D}^f : f \in G_{fix} \cap G_{vuln} \text{ and } \mathbb{D}^f \neq \emptyset\}$.

### 3.2.2 Patch Analysis using a Code Property Graph

To analyze a patch using the method described in the section above, the first step is generating two CPG [141] for the project: one for the vulnerable state and another one for the fixed state.

The second step is computing the tuple $(\mathbb{N}, \mathbb{R}, \mathbb{D})$. Computing the first two elements is straightforward as illustrated in Figure 3.2, but the third needs more attention. To check whether the two graphs are equal, a possible method is to compute a graph isomorphism between them. However, in the generic case, the problem is *NP*-complete. Since we are using this step for filtering, we bound the computation time and consider the timeout as a marker of change.

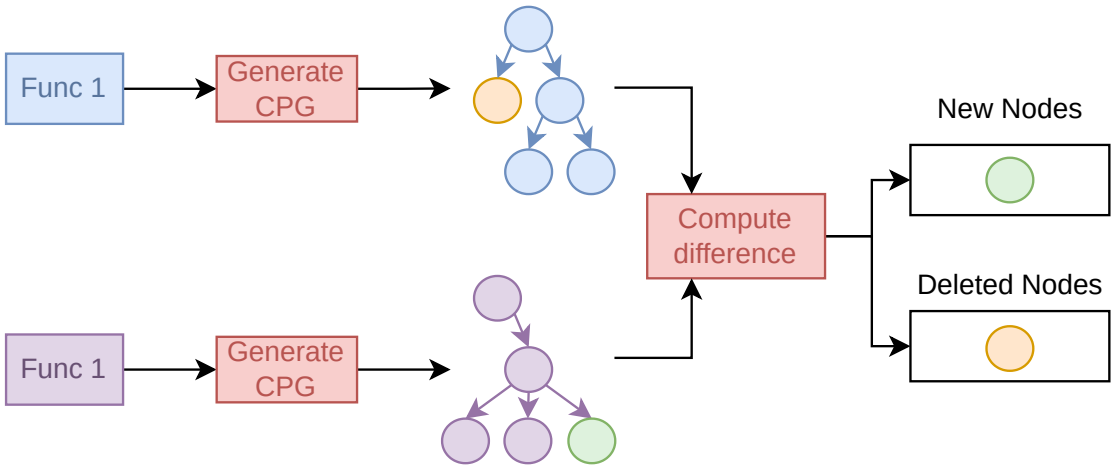


Figure 3.3: Using CPG To Find the Difference For $\mathbb{D}$

Finally, the third and last step considers every function marked as changed. For each of those, we compute the $\mathbb{D}^f$ by labeling each new node using $\psi$. By only considering node types $\Phi$ and not their specific value (i.e. a constant), we lose some information on the patch but keep part of its semantics. The types $\Phi$ are determined using the heuristics based on the features listed in Table 3.8.

### 3.2.3 Results Analysis

We use the CVE dataset presented in Chapter 4 as testing data to understand more deeply the security changes nature in AOSP.

**Feature Pertinence** At least one occurrence of the features listed in Table 3.8 is found in more than 99% of the patches. We manually reviewed patches not reporting any of those and found some unusual changes. For example, the patch for CVE–2016–2464 [88] only changes a comparison sign from $\geq$ to $>$. This modification does not fallback in any of our features. Yet, this feature set is precise enough to characterize almost all patches.

**Patch Locality** The correction for a vulnerability is typically local. 76% of the patches in the dataset are only modifying a single file and 94% are modifying fewer than 5 different files (Figure 3.4). This locality is preserved at the function level as depicted in Figure 3.5. 58% of the patches change a single function, and this percentage rises to 85% when we consider patches modifying fewer than 5 functions. These results are unsurprising, and they align with Li and Paxson [71]. As a security patch aims at fixing a defect in the program behavior, it can be pinpointed to a specific location (e.g. adding a missing bound check). However, changes required to fix a vulnerability may be large in certain cases. For instance, the patch for CVE–2016–3839 [89] changes 100 distinct functions because it wraps calls to API functions.

| Feature | Explanation | Example |
|---|---|---|
| `CAST` | Cast of a variable | `(long) x` |
| `ASSIGNMENT` | Assignment of a variable | `int x = 2` |
| `CONDITION` | Condition | `x >= 5` |
| `CONSTANT` | Constant | `0b1` |
| `CALL` | Call to a function | `printf()` |
| `GOTO` | Goto statement | `goto label` |
| `STRING` | Program string | `"foo"` |
| `LOOP` | Loop structure | `for ()` |
| `SWITCH` | Switch structure | `switch {}` |

Table 3.8: Details of PatchAnalysis features

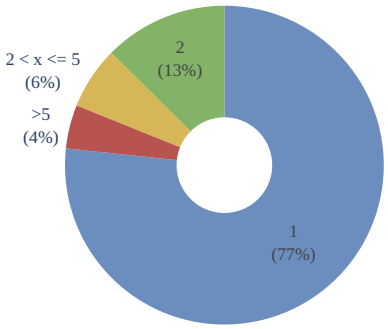


Figure 3.4: Number of Files Affected by a Patch

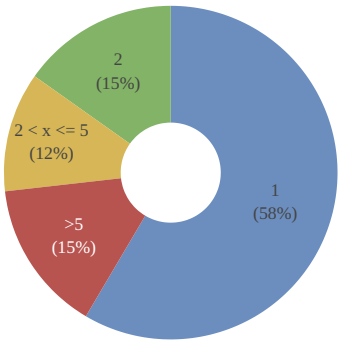


Figure 3.5: Number of Functions Affected by a Patch

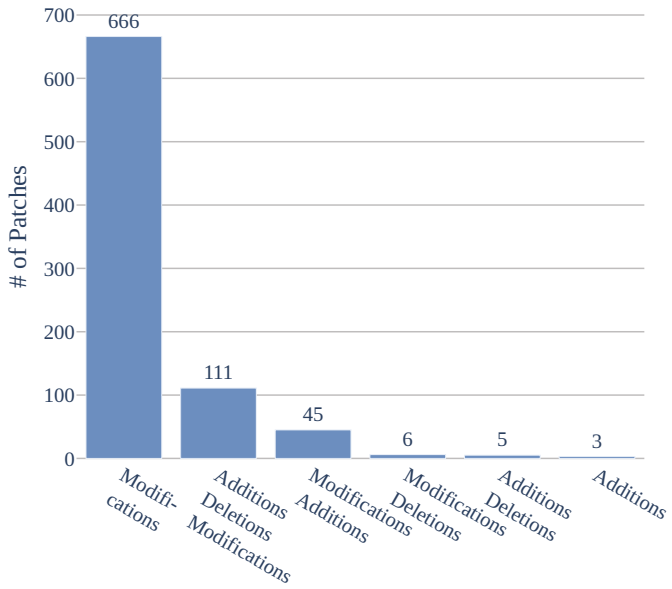


Figure 3.6: Patch Shape

**Patch Shape** In our dataset, 99% of patches modify at least one function. This change is associated with the creation of new functions in 19% of the patches. However, none of the patches were only deleting code. The results are highlighted in Figure 3.6.

**Patch Classification** We classify each patch according to the additions and deletions they display. The results are depicted in Figure 3.7 and Figure 3.8. For 66% of the patches, the fix included at least a new call. This is, by far, the most common addition to fix a vulnerability. Following the calls, new conditions (42%), new assignments (36%), and new constants (34%) are the most frequent changes. An interesting category is the prevalence of string addition in patches for Android. For 20% of the patches, a new string is added. This is a specificity of Android as the string often represents a bug identifier

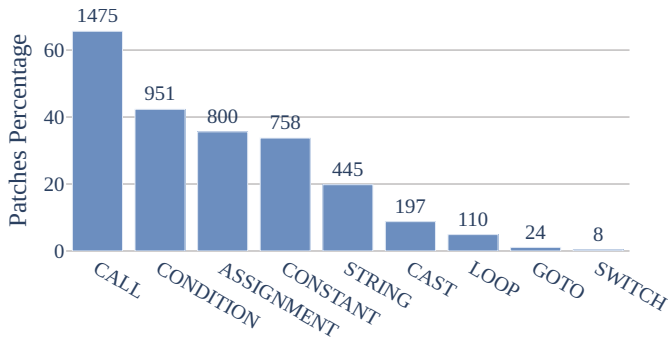


Figure 3.7: Features Occurrence in Patches for Additions

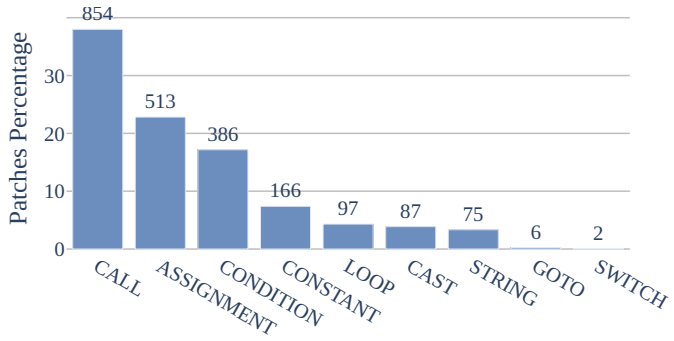


Figure 3.8: Features Occurrence in Patches for Deletions

**Listing 3.1** CVE–2020–0101 Patch Extract

```
@@ -264,8 +264,12 @@
   CHECK_INTERFACE(ICrypto, data, reply);
-  uint8_t uuid[16];
-  data.read(uuid, sizeof(uuid));
+  uint8_t uuid[16] = {0};
+  if(data.read(uuid, sizeof(uuid)) != NO_ERROR) {
+    android_errorWriteLog(0x534e4554,"144767096")
+    reply->writeInt32(BAD_VALUE);
+    return OK;
+  }
```

and is used by SafetyNet[2]. The categories in Figure 3.7 are not mutually exclusive as a patch may comprise multiple additions. The Listing 3.1 displays such a case. In this example, the patch adds two constants, two calls, and one condition.

In this section, we analyzed the difference induced by the patch at a source level and found patterns that could help classify most patches. However, the source code is often missing for a firmware and binary only patch characterization could still be useful.

## 3.3 Firmware Matching Problem

Due to the patch propagation delay or discontinued support, recurrent security patches are insufficient to prevent *1-day* vulnerabilities on end user devices. Finding which patches are missing is crucial to assess a system's exposure to known security threats. In this section, we introduce and formalize the Firmware Matching Problem (FMP) and expose our strategy to design an efficient solution.

[2] https://developer.android.com/training/safetynet/

### 3.3.1 Problem Layout

In this section, a *firmware* $\mathcal{W}$ is defined as a finite collection $\mathcal{W} = \{P_0, \ldots, P_n\}$ of programs where a *program* $P$ is abstracted as a collection $P = \{f_0, \ldots, f_m\}$ of functions. Clearly, firmwares are composed of more than binary programs such as configuration or data files. However, these do not impact our analyses, and therefore are safely ignored in this rest of this chapter.

**Definition 3.2.** *The Firmware Matching Problem (FMP) is stated as follows. For a given firmware $\mathcal{W}$ and a function specific version $f_s$, find the largest subset $\mathcal{P} \subset \mathcal{W}$ such that $\forall P \in \mathcal{P}, f_s \in P$. The problem also asks to identify precisely the function $f_s$ positions in $P$.*

If the function *specific version* of Definition 3.2 is a function after the patch application, the FMP can be extended as a *patch presence test* systemization to a firmware. Thus, with a slight notation simplification, we consider the FMP in this context and search if a function is patched.

To solve this problem, we will need some clues to determine the presence of a specific function version. More precisely we define the following:

**Definition 3.3.** *For a given function specific version $f_s$, a corresponding* signature $\mathcal{S}$ *is a collection of artifacts that could identify $f_s$ among a large set of functions.*

The practical definition of signatures is central to answer the FMP. Indeed, they represent the patch fingerprint, and they are the only element used to assess a patch presence. To guide the patch probing on a system, the signatures should contain information on both the patch itself but also on the patch surroundings. In the following example, we present some possible signature artifacts.

**Example 3.1.** *For a given function $f_s$, artifacts could be:*

- *its enclosing binary name (to find candidate binaries);*
- *the function name itself or the calls to external libraries (to find candidate functions within a binary);*
- *a set of constants used by a function (to determine its version)*

### 3.3.2 F-S-M: Filtering, Selecting, and Matching

A naive process for solving the FMP would test every function in every program in the given *firmware* $\mathcal{W}$. Such an approach is impracticable for a firmware containing thousands of binaries. We thus introduce another strategy addressing the problem which relies on three consecutive steps (*Filtering*, *Selecting*, and *Matching*), named *F-S-M* and described hereafter, pruning successively the search space. The strategy is depicted in Figure 3.9.

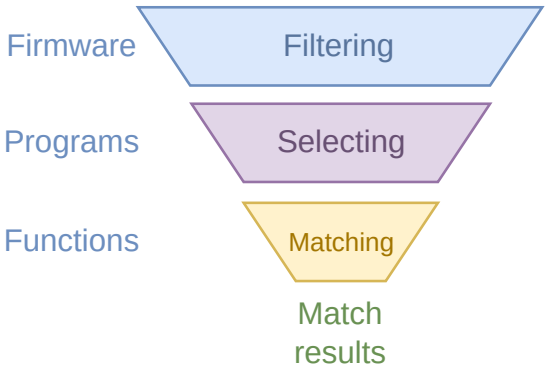


Figure 3.9: *F-S-M*: Solving the FMP in Three Steps

**Filtering** The *filtering* is our first step for solving the FMP. It aims at identifying the programs containing the target function $f_s$ in a given firmware. This step is crucial to efficiently address the FMP because $f_s$ is present in, at most a few programs of $\mathcal{W}$. This step is straightforward when the vulnerability affects a function in an executable or a shared library but requires additional work when it affects a static library (e.g. CVE–2018–9497).

This filtering step is formalized in Equation 3.1. It takes as input $(\mathcal{W}, \mathcal{S})$ where $\mathcal{S}$ is the signature corresponding to $f_s$ that has to be identified in $\mathcal{W}$. The *filtering* function returns a set $\mathcal{P} \subset \mathbb{P}$ listing the candidate binaries. Thus, a signature must hold information about the program containing the function to complete this step. Of note, $2^{\mathbb{P}}$ represents the power set of $\mathbb{P}$.

$$\begin{aligned} \text{Filtering}: \quad \mathbb{W} \times \mathbb{S} &\longrightarrow 2^{\mathbb{P}} \\ (\mathcal{W}, \mathcal{S}) &\longmapsto \mathcal{P} = \{P_0, \ldots, P_n\} \end{aligned} \tag{3.1}$$

**Selecting** The *selecting* stage is our solution's second step. For each of the binaries $P_i \in \mathcal{P}$ filtered by the first step, it *selects* a binary function subset for the final matching step. Within the programs set $\cup_{P \in \mathbb{P}} P$, the selector searches the target function $f_s$. Thus, by denoting $\mathbb{F}$ as the set of all the functions, this step is formalized in (3.2) where the output is a set $\mathcal{F}$ of candidate functions.

$$\begin{aligned} \text{Selecting}: \quad 2^{\mathbb{P}} \times \mathbb{S} &\longrightarrow 2^{\mathbb{F}} \\ (\mathcal{P}, \mathcal{S}) &\longmapsto \mathcal{F} = \{f_0, \ldots, f_m\} \end{aligned} \tag{3.2}$$

Because at this stage, the function version is still unknown, the selector must contain function invariants that remain valid for any function version. Designing this part is delicate because missing the target functions yields inconclusive matching results. Con-

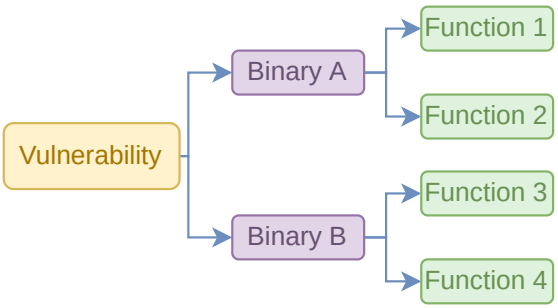


Figure 3.10: Typical Patch Breakdown

versely, selecting too many functions increases the likelihood of getting false positive results.

**Matching** The final FMP stage is performing the matching step. From the functions selected at the *selecting* step, the *matcher* determines whether the candidate function is patched. This last phase is the core of a solution to the FMP thus various strategies have been used to address it in the literature. If we denote the result set as $\mathbb{R}$ then the matching can be described as the function (3.3). Recall that a result $\mathcal{R}$ is not only a boolean but some information concerning the functions $f_s$ localization in the $P_i$.

$$\begin{aligned} \text{Matching}: \quad 2^{\mathbb{F}} \times \mathbb{S} &\longrightarrow \mathbb{R} \\ (\mathcal{F}, \mathcal{S}) &\longmapsto \mathcal{R} \end{aligned} \tag{3.3}$$

In conclusion, our process can be synthesized by the application of the three steps in Equation 3.4. As one can see again, the signature importance is reflected in its use at every step of our proposal.

$$\text{Matching}(\text{Selecting}(\text{Filtering}((\mathcal{W}, \mathcal{S}), \mathcal{S}), \mathcal{S})) \tag{3.4}$$

### 3.3.3 Firmware Patch Matching Problem

**Definition 3.4.** *A **patch** is a finite collection of function changes. They can span over multiple binary files.*

The FMP focuses on matching a single function in a firmware. Nevertheless, patches for real-world vulnerabilities are more complex and may require changes in multiple functions over multiple files. A generic patch breakdown is presented in Figure 3.10. Thus, the underlying problem is to find a patch on a given firmware.

**Definition 3.5.** *We define the $\Xi-$FMP for a firmware $\mathcal{W}$ and a* patch $\mathcal{F} = \{f_{s_1}, \ldots, f_{s_k}\}$. *It finds the largest subset $\mathcal{P} = \{P_0, P_1, \ldots P_n\} \subset \mathcal{W}$ such that $\forall P \in \mathcal{P}$, $\exists f \in \mathcal{F}$ such that*

**Listing 3.2** Solving FPMP from FMP

```
def solve_FPMP(W, F):
  for f_t in F:
    list_P_f = Filtering(W, S_{f_t})
    for P in list_P_f:
      sel = Selecting(P, S_{f_t})
      for f in sel:
        sol = sol + Matching(P, S_{f_t})
```

*$f \in P$. The problem also asks to identify precisely the targeted functions $f_{s_i}$ positions in the corresponding programs $P$.*

The $\Xi$ – FMP is a generalization of the FMP. Thus, we establish a strategy to solve the $\Xi$ – FMP by applying our FMP solution to each *patch* function. This algorithm is described in Listing 3.2 where $\mathcal{S}_f$ denote the signature corresponding to a function in $\mathcal{F}$.

### 3.3.4 State of the Art and the FMP

The *patch presence test* is a well-studied problem in literature. At the beginning of this chapter, we surveyed several state-of-the art studies. However, none of them fully tackle neither the $\Xi$ – FMP problem nor FMP. Table 3.9 displays each work position on the challenges raised by the two problems. Thus, additional work is still required to provide a complete solution to the problem.

## 3.4 Vulnerability Signature

Signatures are involved in every step of our procedure to solve the $\Xi$ – FMP. Their design is paramount for our algorithm efficiency. The following section describes their implementation. We illustrate how to build signatures for a $\Xi$ – FMP solution in a practical context with the CVE–2018–9506 [92], already used as the motivating example in Chapter 1.

### 3.4.1 Semantic Function Invariants

To be relevant in various syntactic contexts (e.g. different architectures, compilers...), a signature should consider changes resilient across syntactic transformations. Table 3.10 lists various target features at a binary level and their associated resilience through various binary transformations. For illustration, the call graph is stable between different compilers (unless a function is inlined). On the contrary, the number of basic blocks is compiler and architecture dependent but remains when stripping the binary from its symbols.

| | FMP | | | FPMP |
|---|---|---|---|---|
| | Filtering | Selecting | Matching | Multiple Functions |
| SPAIN [139] | ✗ | ✓ | ✓ | ✓ |
| FIBER [143] | ✗ | ✗ | ✓ | ✗ |
| 1dVul [98] | ✗ | ✗ | ✓ | N/A* |
| PATCHECKO [116] | ✗ | ✓ | ✓ | ✗ |
| BINXRAY [138] | ✗ | ✓ | ✓ | ✗ |
| BScout [29] | ✗ | ✓ | ✓ | ✓ |
| PDiff [60] | ✗ | ✗ | ✓ | ✓ |
| VIVA [135] | ✗ | ✓ | ✓ | ✗ |
| QuickBCC [59] | ✗ | ✓ | ✓ | ✗ |
| PMatch [70] | ✗ | ✗ | ✓ | ✗ |
| P1OVD [72] | ✗ | ✓ | ✓ | ✗ |
| **QSig** | ✓ | ✓ | ✓ | ✓ |

Table 3.9: FMP in the State of the Art

*1dVul generates a crashing input.

| | Compiler | Static Dynamic | Arch. | Stripped |
|---|---|---|---|---|
| Strings | ✓ | ✓ | ✓ | ✓ |
| Constants | ≈ (spilling) | ✓ | ≈ (splitting) | ✓ |
| Call Graph | ≈ (inlining) | ✓ | ✓ | ✓ |
| Conditions | ✓ | ✓ | ✓ | ✓ |
| Names (exports) | ✓ | ✗ | ✓ | ✓ |
| Addresses | ✗ | ✗ | ✗ | ✓ |
| # Basic Blocks | ✗ | ✓ | ✗ | ✓ |
| # Instructions | ✗ | ✓ | ✗ | ✓ |
| Function names | ✓ | ✓ | ✓ | ✗ |

Table 3.10: Reliability of Selected Features Through Various Binary Transformations

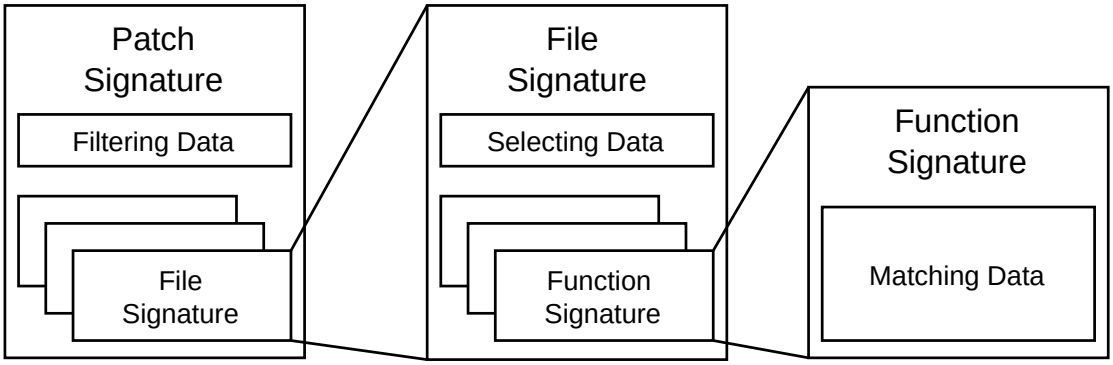


Figure 3.11: Nested Signature Structure

The features listed in Table 3.10 presents some resemblance with the features described in Section 3.2. Indeed, we reuse all features that translate from the source code to the binary code. However, some features (e.g. `cast`) are challenging to distinguish at the binary level since variable types are lost during the compilation. We also consider binary only features (e.g. the number of basic blocks) that do not exist at the source level.

The term *semantic* refers to the meaning of language constructs, as opposed to their form (i.e. syntax). While the features described above do not utterly understand the construct meanings, they are syntax agnostic. Thus, with a *slight* notation abuse, we call them *semantic features*.

### 3.4.2 Signature Layout

To tackle the $\Xi$ – FMP using the *F-S-M* solution, we need to store in a signature the information to solve each step. Thus, the signature structure is nested and can be seen as matryoshka dolls for each abstraction level as depicted in Figure 3.11.

For the outmost layer, the patch level, the signature includes data on how to solve the *filtering* step, i.e. identifying whether the candidate binary is valid. For the file layer used in the *selecting* step, the signature contains function invariants that are identical in both target function versions (the vulnerable and fixed one). Finally, at the function layer, the information to solve the *matching* step is stored.

## 3.5 QSig: An Implementation Solving the FMP

We implement our proposal solution to solve the $\Xi$ – FMP in **QSig**. This section overviews this two-fold tool, describes the signature content, and details its relevant implementation particularities. **QSig** is open-sourced and available on GitHub [19].

**Definition 3.6.** *A **target binary** $T$ is a candidate binary for a signature $S$.*

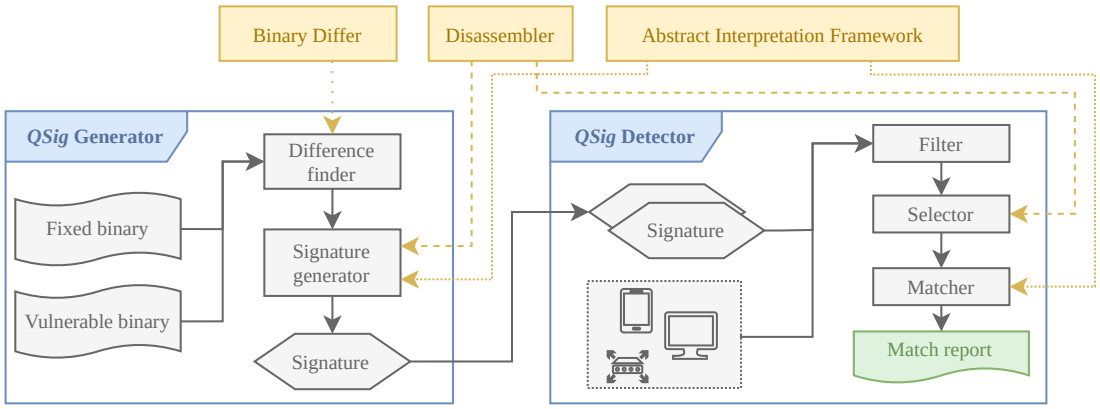


Figure 3.12: **QSig** Architecture

**Definition 3.7.** *A **target function** $f$ is a candidate function within a target binary.*

In summary, **QSig** performs the following algorithm. A signature is generated from the difference between a function in a vulnerable state $f_v$ and fixed state $f_p$. While applying a signature $S$ on a target binary $T$, we first try to find $f$ among the functions in $T$ and then assess the state of $f$ to verify whether the patch has been applied.

### 3.5.1 Tool Architecture and Overview

**QSig** is a system composed of two complementary components depicted in Figure 3.12. The first one, the *generator*, takes as input two binaries, the same program in a vulnerable and patched version, and generates a signature from the difference between the two. The second component, the *detector* applies the *F-S-M* solution on a firmware.

As our features are based on the patch semantic, it is possible to generate a signature from an architecture (e.g. `x64`) and to match a binary on another one (e.g. `aarch64`). Adding other architectures requires little engineering effort and our tool already supports `x86`, `x64`, `ARM`, and `aarch64`.

### 3.5.2 Filtering Binaries

The signature includes data on how to solve the outmost layer: the *filtering* step which identifies whether a candidate binary is suitable. Our default filtering algorithm is simple yet effective: it only considers the name and the target binary type. In general, using the target name is enough for the filtering part. For instance, if an analyst searches a patch for a `curl` vulnerability, probing for the `curl` binary presence on the firmware is usually sufficient.

| Feature | Explanation |
|---|---|
| `NAME` | Function name (if present) |
| `INDEX` | Function index in the binary |
| `SIZE` | Function size |
| `STRINGS` | Strings used |
| `CONSTANTS` | Constant profile |
| `LIBRARY CALLS` | List of calls to external libraries |

Table 3.11: Features to Locate a Function Inside A Binary

However, this straightforward process fails when the patch affects a static vulnerability (Chapter 6). In such case, a name-based filtering step does not work because the static library and final target names differ. **QSig**'s modular system allows us to change it for Android's phone firmwares and to use BGraph (Section 6.7).

### 3.5.3 Selecting Functions

Before identifying whether a target function $f$ has been patched, the first step is to locate this function inside a binary. Recall that the binaries we are dealing with are usually without symbols, so the function names are missing and cannot be used. However, we can leverage similar invariants as those used in the *matching* part. We also add the following ones, described hereafter.

- `NAME`: The function name is an accurate marker of a function but is only present for exported functions.
- `INDEX`: A compiler has no reason to reorder the code during the compilation so the index (i.e. its position inside the binary) should remain stable.
- `SIZE`: A function size is a rough estimate of its complexity and only marginally changes when functions are slightly updated.
- `LIBRARY CALLS`: calls to external libraries represent a good estimate of function behavior and are effortless to recover.

Precisely identifying the function location inside a binary is paramount for the final results. Missing the target function in this step leads to incorrect results at the end of the process. On the contrary, selecting too many functions increases the risk of false positives matches since **QSig**'s signatures are somewhat generic.

**Listing 3.3** Small Difference Function

```
def small_difference(x: int, y: int) -> float:
    """Returns a number between [0, 1]"""

    if x == y:
        return 1.0
    else:
        return min(abs(x), abs(y)) / max(abs(x), abs(y))
```

The features used by the selector are divided into three categories based on their return value type. Each return type is handled differently to produce a similarity score for this specific feature.

- When the return type is a number (i.e. function size, index), we use the `small_difference` method displayed in Listing 3.3.
- When the return type is a string (e.g. function name), we compute the score for the feature using a *Levenshtein* distance.
- For features returning sets of values, we use an inclusion index.

For every function, each feature is evaluated and assigned a score in the $[0, 1]$ interval using one of the three functions presented and then aggregated. This yields a final score for each function and the $n$ highest ranked are selected. The algorithm is described in Listing 3.4. We discuss the features pertinence along with the choice of $n$, i.e. the number of selected functions, in the Evaluation Section 5.2.

### 3.5.4 Matching a Function Version

Intuitively, a characteristic representing the function semantics remains across compilation transformation. For example, the newly added call to `android_errorWriteLog` is present in both CFG (Figure 3.13). We leverage those invariants to create a signature and detect whether a target binary is patched.

**QSig** uses the features underlined in Table 3.10. To pick **QSig** features, we leverage the patch analysis results at the source code level presented in Section 3.2. Recall that the most frequent changes were the additions of calls (66%), conditions (42%), variable assignments (36%), and constants (34%). If new strings were found in only 20% of the patches, they are effortlessly recoverable in binary code and thus make an uncomplicated addition. Of note, these features are also relevant for our use case because they are resilient against most natural binary transformations.

**Listing 3.4** Selecting Algorithm

```
def selector(target: Function, candidate: Program, n: int) -> list[Function]:
  """Select the *n* highest ranked functions in candidate"""

    # Iterate over every function in candidate
    scores = {}
    for function in candidate:
        scores[function] = (
          inclusion_index(function.strings, target.strings)
          levenshtein(function.name, target.name)
          inclusion_index(function.calls, target.calls)
          inclusion_index(function.constants, target.constants)
          small_difference(function.size, target.size)
          small_difference(function.index, target.index)
        )

    # Sort the functions by score (highest first)
    scores = sorted(scores.items(), key=itemgetter(1), reverse=True)

    # Select the n first
    selected_functions = []
    for function, _ in scores[:n]:
        selected_functions.append(function)

    return selected_functions
```

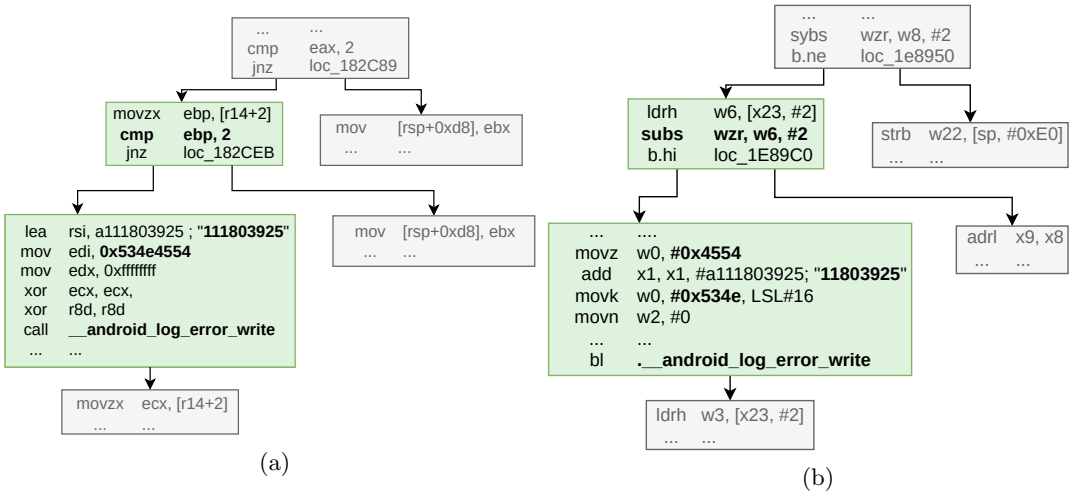


Figure 3.13: CVE–2018–9506 associated CFG for x64 (a) and aarch64 (b)

| General | |
|---|---|
| Binary | libbluetooth.so |
| Functions | 9,322 |
| **All constants** | |
| Functions | 7,288 |
| Unique profile | 5,468 |
| Percentage | 75% |
| **(`0x1000`, `0xEFFF`)** | |
| Functions | 1,889 |
| Unique profile | 1,774 |
| Percentage | 93% |

Table 3.12: Constant Prevalence in Binary Code

### Strings

Strings are a common code characteristic and are recoverable easily in binary code. While they can be obfuscated in some adversarial settings, our application domain does not consider this threat.

Finding the difference between the strings in $f_v$ and $f_p$ is immediate from the disassembly. To increase the search efficiency, utilities (e.g. `strings`) can also find the strings used by a binary program without disassembling.

**QSig**'s signature for the `string` feature contains the strings present only in the fixed version. For the example in Figure 3.13a, it would be the string `"111803925"` which is added by the patch.

**Limits.** Overall, this feature is stable and resilient. However, it fails when a patch adds a string already contained in the vulnerable function. Indeed, it may be impossible to count them as the new occurrence may be merged with the previous one as a compiler optimization.

### Constants

Binary code uses constants extensively. Table 3.12 illustrates the constant distribution in a typical binary. Moreover, they are resilient against most of the binary transformations listed in Table 3.10. However, there is an issue when considering constants from

`ARM`. Since the assembly has fixed length instructions, constants may be split into two parts and then reconstructed. An example of such behavior is displayed in Figure 3.13b, where the constant `0x534e4554` is loaded in two parts. The first part loads the lower 16 bits with `movz` and the second part uses `movk` with a left shift. To circumvent this limitation and to still recover constants across architectures, we *normalize* every 32 bits constant in two 16-bit parts.

Another limit while searching and finding constants is the prevalence of small constants (e.g. close to `0` or `0xFFFF`). Indeed, lower constants are used for syntactic reasons such as resetting a register, allocating space on the stack or shifting values. In those contexts, they do not convey enough information to be used for matching purposes. The constants unique profile rate represents the percentage of functions in a binary having a unique signature for their constant distribution. When considering all constants (e.g. from `0` to `0xFFFF`), this rate is at 75% as illustrated in Table 3.12. However, when restricted to the smaller interval, the rate reaches 93%.

For each normalized constant present in either $f_v$ or $f_p$, **QSig** signature contains a tuple with:

- its value, i.e. `0x1234`
- its occurrence count in $f_v$;
- its occurrence count in $f_p$.

Listing 1 displays the algorithm to match constants. It uses the `normalize` function splitting 32 bits constants into 16 bits constants.

**Algorithm 1** Procedure to Match Constants

```
def match_constants(f, S):
    # Mapping of constants with occurrence
    counter = Counter(cst for cst in normalize(f_constants))

    for (cst, count_vuln, count_fix) in S:
        # Constant added in fix
        if count_vuln == 0:
            r = counter[cst] > 0
        else: # When found in both version
            r = count_vuln < counter[cst] <= count_fix

        if r is False:
            return False

    return True
```

### Calls

A binary Call Graph (CG) [108] is a directed acyclic graph representing inter-procedural relationships. The graph nodes represent functions and an edge from $A$ to $B$ exists only if $A$ calls $B$. Thus, for each node, we define an in-degree (resp. out degree) which is its number of predecessors (resp. successors). Multiple calls towards $B$ from within $A$ do neither change the function degrees nor the graph.

The call feature highlights changes made to the call graph during a patch and we store information about each newly called function. Recall that we do not have the binary symbols to help the identification of callee functions. To retrieve those, we leverage the features presented in 3.5.3.

Matching of new calls raises two main problems. The first one appears when the patch adds a call to a function previously called in the same function. Sometimes, two distinct calls will appear in the binary code, but compilers may merge them. The second one is to deal with *thunk* (small functions used as trampolines). Indeed, since we store the degrees of a function, a *thunk* will have an out degree of 1 towards its target. To cope with this issue, we consider the degrees of the underlying function (by dereferencing the *thunk* function).

**QSig** signature contains a tuple for each new call containing:

- the callee information;
- the in-degree of the callee function;
- the out degree of the callee function;
- the caller count, i.e. the number of times this function is called by the caller.

Listing 2 displays the algorithm to match calls in **QSig**. The `get_degrees` function is a helper function to retrieve the degrees of a function.

### Conditions

**Definition 3.8.** *We call a* condition *a comparison between two elements, themselves called the condition terms. The second term may be implicit (e.g. 0).*

If the syntax of conditions varies across different assembly languages, the compared element types are themselves a semantic invariant. For example, in the diff presented in Chapter 1 (Listing 1.1), the condition `if (p_pkt->len < AVRC_AVC_HDR_SIZE)` compares a function argument (`p_pkt`) with a constant (`AVRC_AVC_HDR_SIZE`). The assembly code generated by this code keeps the same properties, i.e. `cmp ebp, 2` compares the value of `ebp`, a value set from `r14` which is a function argument, and `2`, a constant.

**Algorithm 2** Procedure to Match Calls

```
def match_calls(f, S):
  all_degrees = {(target_in, target_out) for target in f_calls}
  calls_count = {f_calls.count(target) for target in f_calls}

  for callee in S_calls:
    if (callee_in, callee_out) not in all_degrees
       and callee_count not in calls_count:
      return False

  return True
```

We consider four origin types for the terms.

- `CONSTANT`: A constant value.
- `CALL`: A call return value.
- `ARGUMENT`: A function argument.
- `UNKNOWN`: Default origin if not enough information is available.

To distinguish between different called functions, constant values, and arguments, each origin type is kept with a value. Moreover, a term may be *tainted by* several sources. For example, the addition of a call return value with a function parameter. Thus, a term origin label is a sequence of origin types.

Let us use Figure 3.14 which depicts a CFG for the start of a function, as a support for our examples.

- Comparison (1) compares the return value of the call to `func_2` with a hard-coded constant. Therefore, the labels associated with this comparison will be `{(CALL 0), (CONSTANT 10)}`.
- Comparison (2) compares the value of two registers. Analyzing the data flow of these two registers unveils their link to the function arguments, held into `edi` and `esi` register. Therefore, the labels are `{(ARGUMENT 0), (ARGUMENT 1)}`.
- Comparison (3) compares `eax` and a memory cell. As the content of this cell is unknown, we are unable to determine this term's origin. Therefore, the labels are `{(CALL 0), (UNKNOWN)}`.

To recover the origin of an element, we developed a relaxed and unsound Abstract Interpretation (AI) framework which is detailed in 3.5.5.

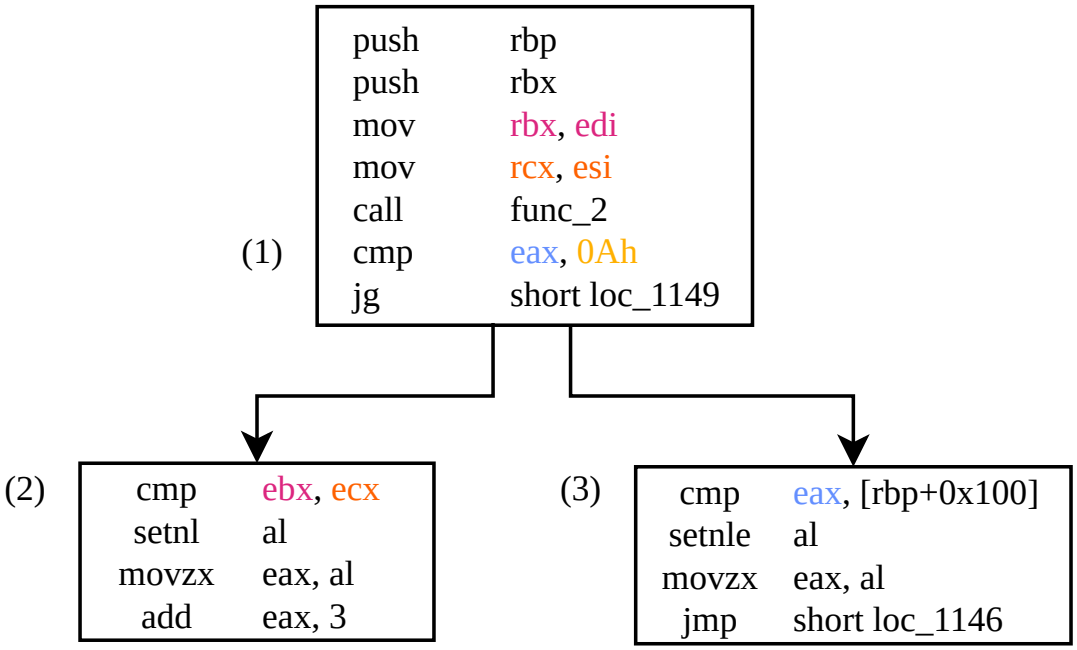


Figure 3.14: CFG Annotated with Origin-tainting

**QSig** signature contains a tuple with the following information for each new condition:

- the first term origin and value;
- the second term origin and value;
- the number of conditions of the same type.

Listing 3 displays the algorithm to match conditions. It starts by trying to find an appropriate mapping from signature labels to target function labels (function `yield_mapping`). This function is needed because it allows to also tackle patches adding a function argument or where the calls order is different.

### 3.5.5 A Relaxed Abstract Interpretation Framework

Abstract Interpretation (AI) is a general purpose static analysis framework [26] leveraging data flow analysis to propagate domain informations on program locations. These domains encode arbitrary properties (i.e. sign, intervals). By design, AI computes a semantic invariant of a program on such domains[3]. We decided to use an AI framework to leverage its efficient data flow fixpoint algorithm that we needed for our origin tainting system. By relying on a semantic solution, we thwart the syntactic-based approaches' limitations using name pattern recognition.

[3]Defining precisely AI is out of this work scope.

**Algorithm 3** Procedure to Match Conditions

```
def match_conditions(f, S):
  sig_counter = Counter(cond for cond in S_conditions)
  for mapping in yield_mapping(f_labels, S_labels):
    counter = Counter()

    for sig_cond in sig_counter:
      for target_cond in f_conditions:
        if equal(sig_cond, target_cond, mapping):
          counter[sig_cond] += 1

    if is_included(counter, sig_cond):
      return True

  return False
```

Implementing an AI framework from scratch is a tremendous task. Thus, **QSig** uses the off-the-shelf `BinCAT`[4] [11], a mature and maintained framework. However, AI strives for *soundness*: a result is guaranteed to be correct. It will quickly discard information, using a widening operator, to accelerate convergence.

On the other hand, we wish our tainting algorithm to work with partial information. Moreover, our signature detection system is opportunistic and does not need to abide by safety constraints used in critical industries. On a side note, as both parts of the system, the generator, and the detector, use the AI framework, an error is not problematic if repeated.

At the top of existing algorithms, we performed the following relaxations tailored for our needs, using the tool mechanisms. They might break soundness but improve the coverage and efficiency of the algorithm for our purposes.

- Skip function calls and only taint the return value: this relaxation allows scaling the system because the execution time of the algorithm is now linear with the instructions count. This is implemented using the `fun_skip` mechanism in `BinCAT`.
- Avoid following backward edges and immediately widen the state: AI deals with loops by trying to find a fixed point. However, it is not always feasible and thus a widening (e.g. taking an upper bound) may be required, at the cost of precision. This could have been implemented using the `unrolling` mechanism already present in the framework, but was performed using a new function `has_ip`.
- Ignore silently non-decoded instructions: an AI framework does not necessarily implement every instruction in the set, and some may be left undecoded or unim-

[4] With minor modifications.

plemented. We decide to simply skip these instructions to improve coverage. This relaxation needed additional work as the `skip` mechanism present in the framework needs to be configured before the execution. We modified the framework to recover from a failed decoding instruction, and advance the instruction pointer to the next one. For architectures with variable length instructons, this process is optimistic.

For **QSig**, we use the tainting domain of `BinCAT` containing 5 elements $(\bot, U, S\,\mathbf{of}\,T, \top)$, where:

- $U$ is `Untainted`;
- $S\,\mathbf{of}\,T$ is a set of possible tainting sources;
- $\bot$ is *bottom*;
- $\top$ is *top*.

Each program location (memory cell and register) is initially untainted and may end up tainted by multiple sources.

Our algorithm starts by creating a tainting source [25] for every function argument location. We then create an initial state fully unconstrained (every value set at $\top$) to reach all possible states. The target function is then fully emulated using the relaxation aforementioned.

## 3.6 Conclusion

This chapter introduced our thesis first two key contributions. The first one is a formalization of the Firmware Matching Problem along with a strategy to solve it, the *F-S-M*. The second one is our system presentation, **QSig**, which implements the *F-S-M* to tackle the problem at a firmware scale using semantic patch signatures.

If we complete Figure 3.1 with the developments presented in this chapter, we obtain Figure 3.15 where **QSig**'s two parts and the **PatchAnalysis** have been added.

However, to assess **QSig** strengths, we need to run experiments on sufficient data. To this extent, we introduce in the following chapter our dataset based on AOSP vulnerabilities.

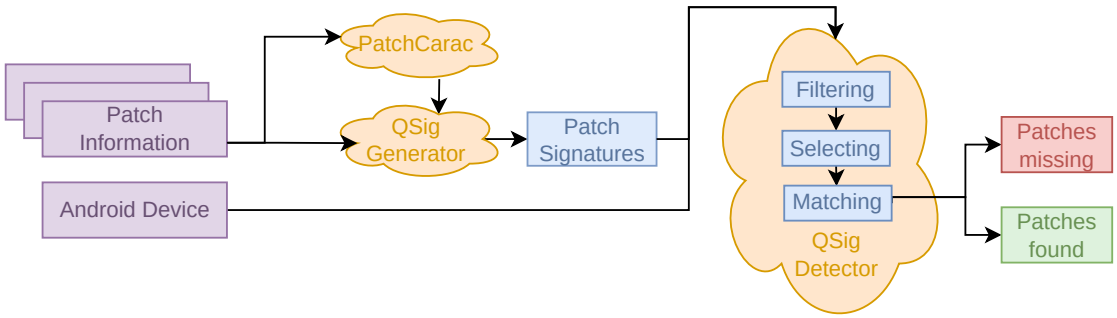


Figure 3.15: Finding Patches Using **QSig**

Chapter 4

# Vulnerability Dataset Precise at a Commit Level

**Contents**



To test the approach developed in the previous chapter and benchmark state-of-the-art, we need exhaustive and representative data to conduct experiments. This chapter starts by exploring solutions used in the literature to assess source and binary security research before exposing the remaining open challenges. Then, we introduce how to create a common base leveraging Android Security Bulletins at the source level and precise at a commit granularity to tackle them. Finally, we extend this source-code

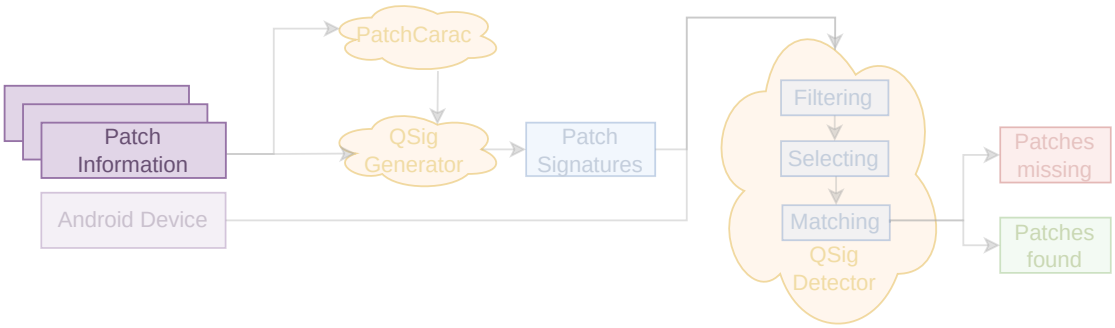


Figure 4.1: Creating a Dataset for *1-day* Detection

information into binary artifacts to allow binary only testing.

In the context of this thesis, the dataset is used both to gather enough information for understanding drawing a typical patch identity card from real-world vulnerabilities, and to test **QSig**'s pertinence and applicability.

## 4.1 Vulnerability Datasets

**Definition 4.1.** *A **dataset** is a structured artifact collection of data aimed at helping to solve a set of problems.*

Vulnerability research or detection requires a representative dataset of vulnerabilities for validating their approach and estimating its pertinence[1]. Contrary to the *Computer Vision* field where the MNIST dataset [33] is a standard, there exists no universal benchmark in the information security domain.

Datasets are multipurpose projects. On one hand, they allow researchers to check how their hypotheses react in real but controlled settings. On the other hand, they also permit benchmarking various solutions to compare them to the same baseline. In this section, we discuss several approaches available to a security practitioner.

### 4.1.1 Standard Tests Suites

The National Institute of Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE) project is dedicated to improve software assurance. The Juliet Test Suite [14, 13] is a collection of synthetic test cases for C/C++, Java, and C# built by the National Security Agency (NSA) and part of the SAMATE initiative.

[1] Conan Doyle wrote *"It is a capital mistake to theorize before one has data."*

| | |
|---|---|
| CWE | 118 |
| Variants | 1,327 |
| Test files | 46,747 |
| C LoC | 4,969,879 |
| C++ LoC | 3,487,852 |

Table 4.1: Juliet 1.3 Breakdown

The tests cover over a hundred Common Weakness Enumeration (CWE). Originally developed for security vendors, its usage has also spread to academia [45, 125] as a baseline for static analysis tools comparison.

To comprehend the scale of Juliet dataset, we list some figures in Table 4.1. Each CWE contains multiple test cases, subdivided themselves into variants. In each test, both the *good* (i.e. correct) and *bad* (i.e. incorrect) behavior coexist. It is possible to use Juliet test cases with Static Application Security Testing Tools (SAST), but also to compile them and use binary only tools.

Using test suites has several advantages over using code from *natural* sources:

- The dataset includes every problem instance under consideration.
- The ground truth is perfectly known in the beginning and creates a baseline. It requires no further processing for understanding if the match is a *false positive*.
- Since the tests are crafted, the flaw locations are known. Thus, it is possible to report a miss even if all tools failed.

However, test suites still suffer from important limitations. Due to their synthetic nature, the authors knowledge limits the test suite exhaustivity. If they forget about a vulnerability class, the test suite cannot cover it. More importantly, defects may not represent real problems. For instance, in one of the test cases displayed in Listing 4.1, the integer overflow impact could be disputed since the result is only printed. Finally, the test suite cases are necessarily short and cannot be representative of real-world software complexity that can reach millions of code lines.

### 4.1.2 Synthetic Datasets

To test and compare bug finding works, another approach has been to inject artificial bugs into real-world programs. This approach corrects a major limitation of *test suites* where programs are unrealistic. By injecting them deep inside the control flow and making them hard to trigger, it ensures that they are only discoverable by design and not accidentally.

**Listing 4.1** Excerpt of `CWE190_Integer_Overflow__int64_t_fscanf_square_17`

```
for(j = 0; j < 1; j++)
{
    {
        /* POTENTIAL FLAW: if (data*data) > LLONG_MAX,
        this will overflow */
        int64_t result = data * data;
        printLongLongLine(result);
    }
}
```

A prominent work using this approach is Lava [34]. They find an execution point where a user-controlled input (e.g. a buffer) is used by a potential attack point (e.g. a call to `memcpy`). By altering the original code line, they can inject a bug that can only be triggered using a determined precondition. Their approach applies to various bug types although they only implement one (out-of-bound memory accesses).

Magma [53] enhances this approach and addresses Lava's limitations by *forward porting* bugs from older to newer software versions. This allows to increase the bug density and diversity because they come from real sources. Magma covers 118 bugs from seven programs but lacks Proof of Vulnerabilities (PoV) for 55% of them, making it hard to use as a ground truth for experiments.

The Cyber Grand Challenge (CGC) [30] was a Defense Advanced Research Project Agency (DARPA) challenge to create Cyber Reasoning System (CRS) to automatically identify software flaws, formulate patches, and deploy them on a network in real-time. Its corpus represented realistic programs for a simplified Linux system. Darpa distributed each challenge with an input crashing the program (Proof of Vulnerability (PoV)). After the challenge final, this corpus served as a ground truth for several experiments [142, 111, 73, 127]. However, because crafted only for a simpler OS, its programs are succinct. Thus, they represent an excellent first step, yet insufficient to bridge the gap for Commercial-Off-The-Shelf (COTS) binaries.

### 4.1.3 Using Real Vulnerabilities

To create a ground truth for experiments, a potential solution is to convert a vulnerability list into a dataset. All works presented in this section share this approach and start with the CVE database from NVD, a project supported by the NIST.

Akram and Ping [3] highlighted in 2020 the need to develop a vulnerability benchmark to *"help users and developers to have a deep understanding of the security flaws and weaknesses in software systems"*. They parse the CVE list, and for each entry, search

for a link to a fixing commit. If they find a link to a supported platform, they analyze the page to retrieve the patch. Their approach has several advantages. Most notably, it is language agnostic and encompasses every project using CVE to report defects. However, the published dataset only contains vulnerable files, affected by a CVE but not the corrected file and is only partially published [2].

CVEFixes [10] listen to JSON vulnerability feeds from NVD server to monitor new vulnerabilities. Its authors process each entry with a link toward its patch to associate it with its CWE. A database stores the approach's results and is freely reproducible from their GitHub[3]. While language agnostic, they require information on the fixing commit which limits their project. This also stands for Akram and Ping work.

These automated methods rely on the processed data quality. On the other side, Ponta *et al.* [100] manually curated a dataset of 624 public vulnerabilities, affecting 205 open-source Java projects used by SAP. By manually selecting and verifying each vulnerability, they ensure a high-quality dataset that suppresses the inconsistencies in the NVD feed.

The CVE benchmark from the Open Security Software Foundation [9] not only manually selects vulnerabilities but also annotate each source code to pinpoint the *responsible line* that caused the vulnerability. This manual process is tedious and not always feasible: it is easier to pinpoint the place of a buffer overflow than the code causing a race condition. The initial release only supported Javascript and TypeScript vulnerabilities, but they are extending it to other languages (C, C++, ...).

Finally, in PatchDB, Wang *et al.* [129] not only collect patch information from NVD metadata, but also develop several methods to increase their dataset size. Their first which relies on a *nearest link search* algorithm on patch features. It classifies every patch as *security* or *non security* and a security expert manually reviews the candidates to include them in their dataset. Their second method uses an oversampling algorithm to generate artificial patches at the source code level.

### 4.1.4 Conclusion

This section has presented approaches that considered how to systemize testing new security experiments. These approaches are either synthetic or natural, on source code, binaries, or both, and with an explicit ground truth or a constructed one. However, they either lack exhaustivity, target specific languages, or are challenging to maintain over time. We present our approach solving these issues in the next section.

[2] https://github.com/znd15/file_level_granularity
[3] https://github.com/secureIT-project/CVEfixes

| Type | Categories example |
|---|---|
| General | Framework, Media Framework, System, ... |
| *SoC* | Qualcomm Model, MediaTek Kernel, ... |
| Linux | Kernel ALSA, Kernel USB, ... |

Table 4.2: Components Examples from Bulletins

## 4.2 Rationale of Using AOSP for a Dataset

In the previous section, we described existing vulnerability datasets. However, they all suffer from limitations, which are addressed with our Commit Level Dataset, like synthetic code, handcrafted vulnerabilities, small size...

Using AOSP and the Android Security Bulletins offers several advantages compared to previous works to build a vulnerability dataset.

- AOSP is the heart of a largely entirely open-source OS and the bulletins cover most of its components. Notably, it creates a coherent vulnerability set because each of them affects the same system.
- Billions of users use Android, and vulnerabilities fixed in each version potentially affect millions of customers. This code runs in the real-world contrary to synthetic datasets where vulnerabilities are artificially created.
- Moreover, the vulnerabilities in AOSP are *de facto* representative because researchers discover them in the codebase. Thus, they are neither more complex nor simpler.
- Finally, as Android Security Bulletins are published regularly, vulnerabilities in our dataset are always up to date and will automatically contain examples for new vulnerabilities classes if they affect the system.

## 4.3 Android CVE Data Aggregation

This section details the architecture developed to build Androids CVE list. While all the needed information is already available online, our work focuses on automatically aggregating data present in numerous locations to expose them in a structured manner.

### 4.3.1 Android's Security Bulletin

Since August 2015, Google publishes advisories on issues fixed in the last Android release, through the Android Security Bulletin Monthly Release. Initially called Nexus

| CVE | References | Type | Severity | Updated AOSP Version |
|---|---|---|---|---|
| CVE-2021-0640 | A-123455677 | EoP | High | 9, 10, 11 |
| CVE-2021-0645 | A-123455677 | EoP | High | 11 |
| CVE-2021-0646 | A-123455677 | EoP | High | 8.1, 9, 10, 11 |

Table 4.3: August 2021 Bulletin Extract

Security Bulletin and only targeting Nexus devices[4], they have been renamed as Android Security Bulletins in May 2016. Google's bulletins are usually divided into two SPL, themselves divided into categories to facilitate their understanding. Examples of categories are listed in Table 4.2. Finally, each category contains a list of vulnerabilities. A list entry contains the following information:

- the CVE identifier, e.g. CVE–2021–0640;
- the vulnerability type, e.g. Elevation of Privilege;
- the severity, e.g. High;
- the updated AOSP versions, e.g. 11;
- a direct link to the fixing commit if the component is open-source.

Table 4.3 shows an August 2021 bulletin extract.

Google is not the only OEM publishing monthly security bulletins, and various manufacturers also publish similar bulletins (e.g. Samsung, LG, Oppo). For example, LG's bulletins contain the monthly Android Security Bulletin, and add information on vulnerabilities fixed for their devices with some details. Oppo bulletins only list CVE identifiers.

### 4.3.2 Crawling Security Bulletins

We created **Roy** to crawl Android Security Bulletins. Its architecture is depicted in Figure 4.2. For each new bulletin since the last run, **Roy** recovers the new vulnerabilities list. If a link towards a fix commit exists, **Roy** also parses the changes provided by the commit (e.g. changed files). To reduce the workload, **Roy** works in an incremental process: an already parsed bulletin is never reanalyzed. Thus, the parser complexity remains stable over time because we solely need to maintain one for the last bulletin version.

[4]Google-branded devices but built by other manufacturers.

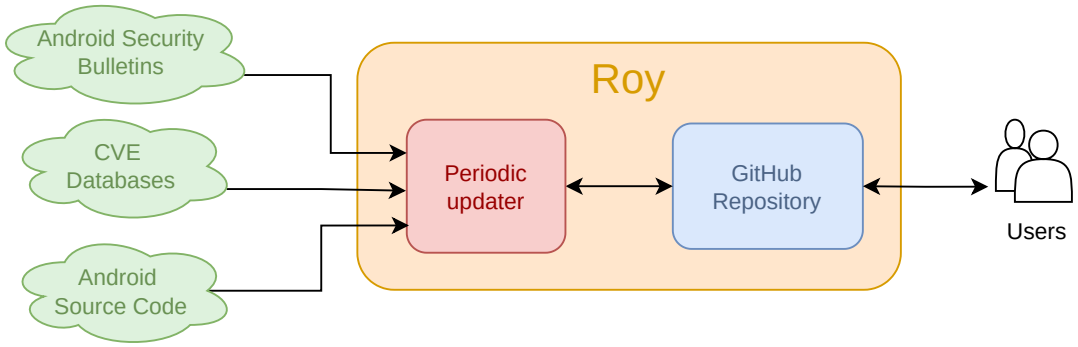


Figure 4.2: **Roy**'s Architecture

While thorough, bulletins are not complete, and we can augment the data with additional information, available from other sources. For example, using, CVE-Search [37], an open access CVE database, it is possible to augment the data for each entry.

Crawling web pages is challenging as they are not primarily designed for automated consumption. The main predicament lies in the ever-changing bulletin format. Keeping the parser up to date requires a regular maintenance effort. Furthermore, the bulletin data is unstable, and fields are subject to change, usually just restricting the available information. For instance, bulletins after June 2017 remove the `discovered date` field. To keep our entries consistent, we use additional sources to retrieve the information.

## 4.4 Generating Binary Artifacts

Not all the code running on a device is open-source, and vulnerabilities equally affect closed source components. Therefore, developing and testing solutions (e.g. Dynamic Application Security Testing Tools (DAST), SAST or CVE-checkers) working in *binary only* environments is paramount. This section describes how we use the commit precise dataset to build a binary level dataset.

Thanks to the Android Security Bulletins, we know the precise commit in which a vulnerability is fixed. We assume that before this *fixing* commit, the project is in a vulnerable state, and in a *fixed* state starting from the commit application in the tree. This is depicted in Figure 4.3. To be more precise, the project is in a *vulnerable* state only during the time between the commit introducing the vulnerability and the commit fixing it. Since automatically detecting whether a project is vulnerable is tedious, we assume here that the project is at least vulnerable since the previous version. This assumption is reasonable, as we are only considering vulnerabilities with a CVE identifier affecting released projects.

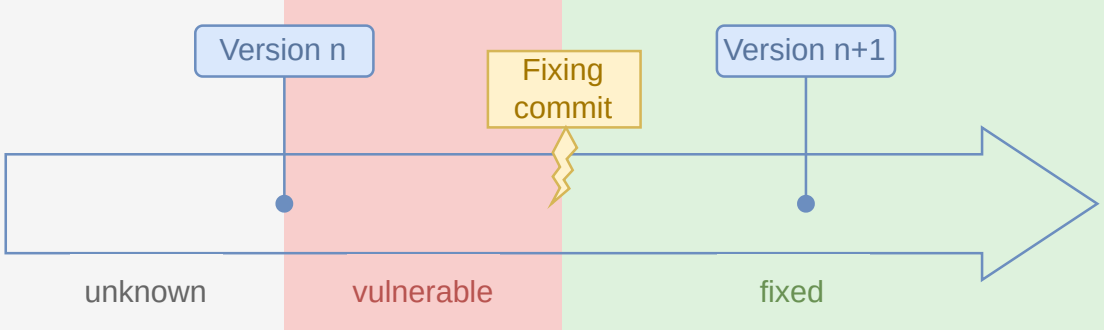


Figure 4.3: Project State Around the Fixing Commit

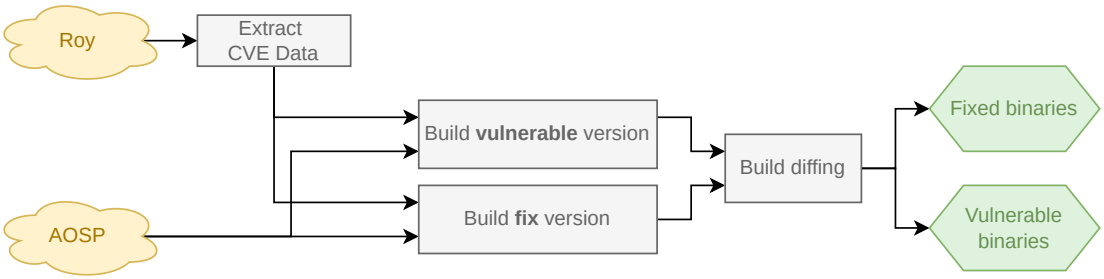


Figure 4.4: **AOSPBuilder** Workflow

Let us consider two arbitrary points of a project life, one during the vulnerable state and one during the fixed state. If we compute the difference between these two program states, the vulnerability patch will be among the changes. However, extracting from a larger change set the modifications fixing the vulnerability is tedious. To overcome this issue, we choose to take our points closer to the fixing commit: one direct parent of the commit for the vulnerable state, and the fixing commit itself for the fixed state. This allows to precisely pinpoint the changes at the binary level coming from the vulnerability patch.

### 4.4.1 Automated AOSP Building

AOSP is a target well suited to provide pre-compiled binaries for the vulnerabilities.

- The project is open-source, and provides a documented build system.
- Thanks to **Roy** and the Android Security Bulletins, we have vulnerability information, precise to a commit level.
- AOSP targets multiple architectures, allowing the generation of multiple binaries for a single vulnerability.

To automate the process, we developed a solution, named **AOSPBuilder** whose workflow is depicted in Figure 4.4. It compiles the binaries at the commits just before and just after the vulnerability fix. The inputs to **AOSPBuilder** is a fixing commit hash, obtained from **Roy**. Then, it builds the project both with the fixing commit, and without it. Finally, we only keep the binaries that differ between the two builds. To reduce noise introduced by compiler optimizations, we keep the same settings between the two builds.

### Building AOSP at a Specific Commit

Building AOSP at a specific commit is challenging. First, building past project states (i.e. not versions) is demanding. To be successful, we need to reconstruct the whole dependency chain needed by the project (compiler version, libraries used...) at that time. Even if AOSP is self-contained, for both its toolchain and all its project dependencies, this remains challenging. As we need to identify the correct version of each built project dependency. A second problem arises from either bogus commits preventing the project compilation, or fixes applied to former development branches. In this case, we cannot obtain binaries differing by precisely one commit, and we use other strategies detailed in the following paragraph. Finally, even a single Android version in AOSP remains huge. Building parts of it repeatedly is costly in time, computational power, and space.

### Build Diffing

After building the project twice, for the fixed version and for the vulnerable one, we search for the changes between the two versions in the compilation artifacts. For this, we leverage a common build system optimization designed to spare resources: a target is only recompiled if one of its dependencies has changed. By rebuilding *in place*, we leverage this property for our build diffing system. We describe its algorithm in Listing 4 where $P_{vuln}$ (resp. $P_{fix}$) represent the program artifacts built with the vulnerable commit (resp. fixing commit). These two sets are a superset of the project binaries as AOSP interdependent projects make it necessary to compile multiple ones to obtain the expected target.

### Vulnerability Building Strategies

We use two strategies to build the binaries of a project in both the vulnerable and fixed versions. The first is to checkout the project at a vulnerable commit, e.g. a fix commit parent, build the project, then checkout to the fix commit itself and rebuild the project. This is the preferred strategy since it is the most precise. However, it is unsuited if the project compilation fails in its vulnerable state. The second strategy uses the opposite approach. It builds the project at the version state (one from Android), and then reverts the fixing commit on the project before compiling again in a vulnerable state. If reverting the commit succeeded, this strategy fixes the problem of the first strategy.

**Algorithm 4** Build Diffing Algorithm

```
1: procedure BuildDiff(P)
2:     Build P_vuln
3:     H_vuln ← {(f, hash(f)) |∀f ∈ P_vuln}
4:     Build P_fix
5:     H_fix ← {(f, hash(f)) |∀f ∈ P_fix}
6:     for all f in H_fix ∪ H_vuln do
7:         if H_fix[f] ⁄H_vuln[f] then
8:             Save f_vuln and f_fix
9:         end if
10:    end for
11: end procedure
```

The success rate of compiling vulnerabilities is around 65%. Indeed, the multiple problems listed above are not always solvable automatically and would require manual intervention. Nonetheless, it helped us to create a large dataset we detail in the following paragraphs.

#### Dataset Artifacts

For each vulnerability, we consider the build process to be complete, if it produces the relevant binaries in both forms (fixed and vulnerable) for each of the four architectures (`x86`, `x86_64`, `arm`, `arm64`). We also keep the binaries with debug symbols (i.e. unstripped) if they are available.

Finally, we use heuristics relying on the source code using *ctags universal* [107] to guess the names of every function affected by a patch. While these heuristics are unreliable for edge cases (e.g. a change in a macro, or incomplete support of C++ templates), they give a baseline for further analyses. More importantly, compiler optimizations (e.g. function inlining, tail call) may change the function layout and merge affected functions inside others at the binary level. To improve the identification of functions changed between the two builds, we also use `BinDiff` [147]. Because we are close to a perfect setting, `BinDiff` results allow to precisely identify the changes inside the binaries.

Figure 4.5 presents an artifact extract for a precompiled vulnerability. We prefix every file by its `SHA256` hash to prevent name collisions.

## 4.5 Dataset Overview and Analysis

In this section, we detail our datasets, both at a source level and at the binary level. We also discuss some of their usages and limitations.

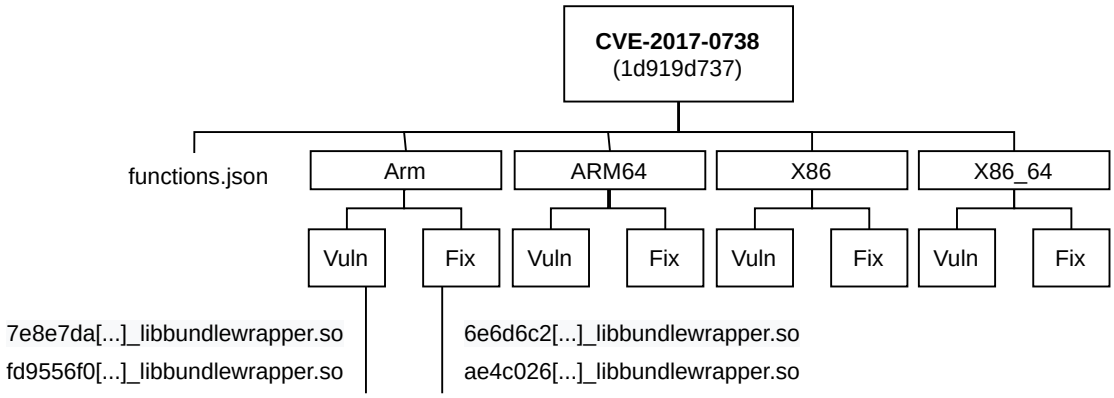


Figure 4.5: Extract of Artifacts for `CVE-2017-0738`

### 4.5.1 At Source Level

The source-level dataset contains the 3,903 CVE listed in the bulletins between August 2015 and March 2022. Their repartition is detailed in Figure 4.6. The closed source vulnerabilities affect Qualcomm's, NVIDIA's, or Google's components. We retrieved the fix commit id for 1,359 open-source vulnerabilities (64%). We support `gitiles` [50], the platform used by Google for the AOSP's versioning, while other platforms such as GitHub, kernel.org, or CodeAurora would need additional engineering effort. 35% of the vulnerabilities present in our dataset have a CVSS score of at least 9.0. Their CVSS score cumulative distribution is presented in Figure 4.7.

Figure 4.8 shows the CVE distribution over the years. While the bulletins start in 2015, we considered the `Published Date` entry for each CVE. As they are fixed in subsequent patches, some earlier CVE are nonetheless listed. The report numbers decrease for closed source projects after 2017-2018, explaining the drop on the graph. The data for 2022 is shown until March 2022 Android Security Bulletin.

A single commit is enough to fix most vulnerabilities, as illustrated in Figure 4.9. This is the case for 84% of the dataset. However, a patch may be split into numerous commits. The one for CVE-2015-3873 [87] affecting `libstagefright` in Android reports 20 fixing commits.

### 4.5.2 At Binary Level

To provide a set of binaries to the community, we managed to precompile hundreds (612) of vulnerabilities. Complementary to our source-level dataset, this helps practitioners bootstrap their research for binary only works or cross-architecture experiments. This dataset is also available on GitHub.

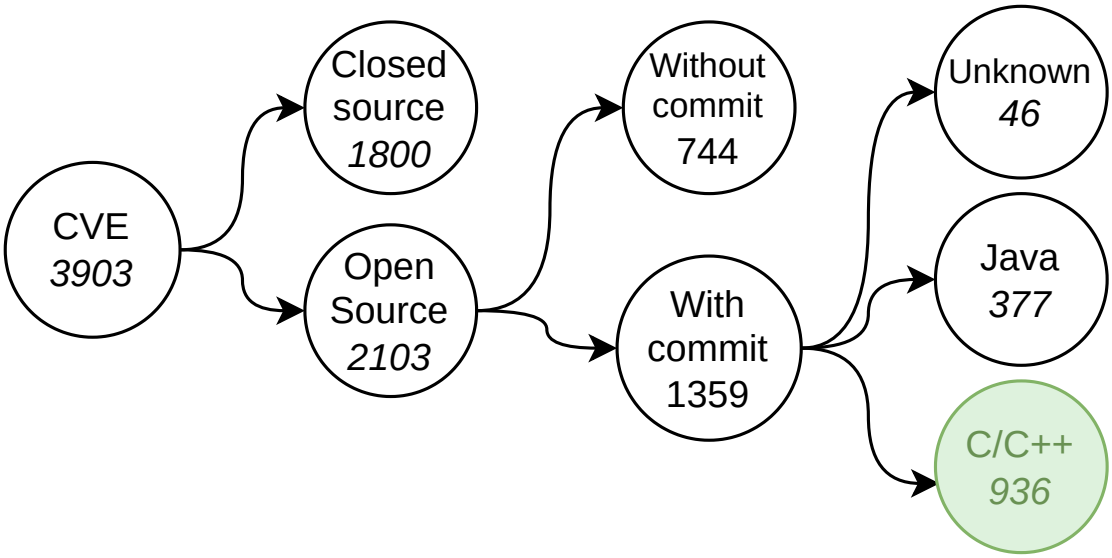


Figure 4.6: Vulnerability Dataset Repartition

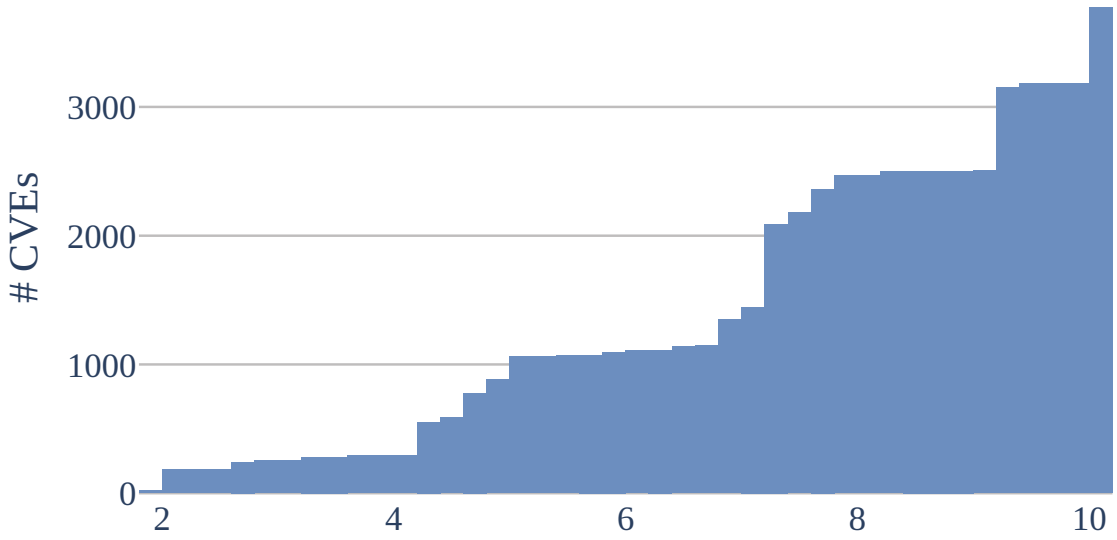


Figure 4.7: Cumulative CVSS Scores

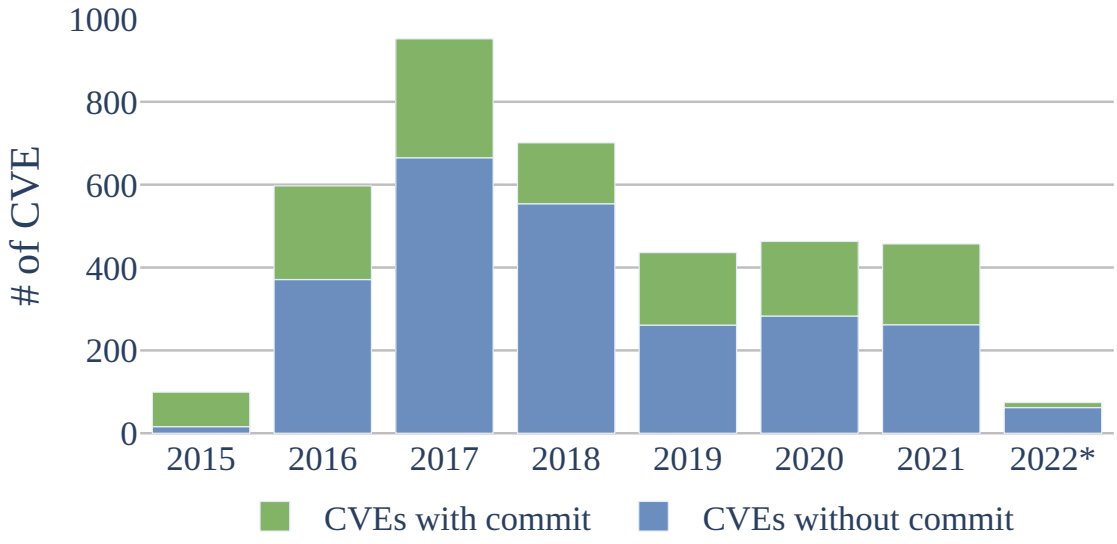


Figure 4.8: CVE Evolution Over Time

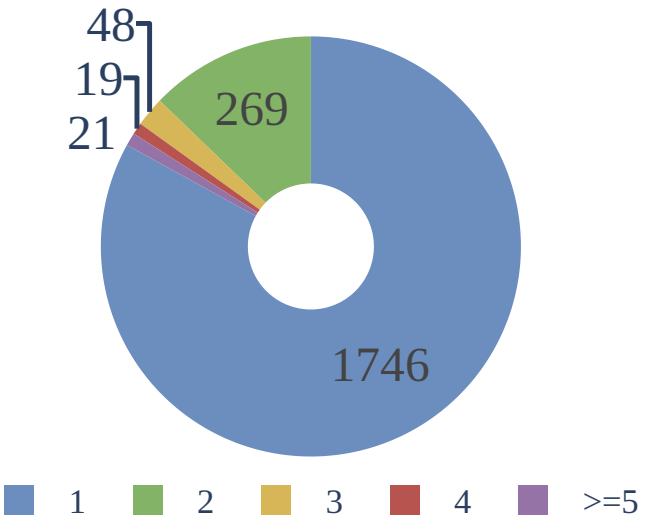


Figure 4.9: Fixing Commits per Vulnerability

| Category | Mean | Median | Standard dev. |
|---|---|---|---|
| Unstripped | 12.3 MiB | 2.7 MiB | 40.2 MiB |
| Stripped | 17.7 MiB | 623.8 KiB | 128.2 MiB |

Table 4.4: Binaries Sizes in Dataset

| Name | Count |
|---|---|
| libbluetooth.so | 953 |
| bluetooth.default.so | 748 |
| libnfc-nci.so | 650 |
| libstagefright.so | 421 |
| net_test_btif | 417 |
| net_test_stack | 299 |
| hevcdec | 268 |
| libaudioflinger.so | 242 |
| libbinder.so | 234 |
| libmedia.so | 229 |

Table 4.5: Most Common Binaries

Based on file extensions, our dataset contains shared libraries (35%), object files (37%), and executables (13%). The largest file is `libv8` (1.6 Gb), the static library of `v8`, a JavaScript engine. Several metrics are also listed in Table 4.4.

Table 4.5 lists the top 10 most common files found in the dataset. They mostly represent complex OS parts (e.g. wireless communications, and media management), particularly targeted by attackers. For instance, the most common file is `libbluetooth.so`, found 953 times.

### 4.5.3 Potential Usages

We provide this dataset, both at the source and binary level to support data driven software security research. In the following paragraphs, we list several types of research that could benefit from such a dataset. Although not exhaustive, we believe it is still relevant to illustrate its applicability.

**Patch Characterization** Our vulnerability dataset lists issues affecting various components of the same OS. It enables to conduct research aiming at characterizing what a common patch looks like and drawing an identity card. For example, Farhang [41] or Li [71] could have benefited from our data. We also presented in, Section 3.2, our experiment to characterize patches using CPG.

**Silent Fix Detection** Not all security patches are labeled as so. A frequent practice is to silently fix a vulnerability without an explicit notice [128, 145]. While appealing to vendors to avoid bad publicity or to delay the development of *1-day* exploits, silent patches create security risks for dependent projects. Indeed, keeping an up-to-date dependency chain is hard, and project managers must prioritize between all the dependencies updates. By not labeling an update as security related, the update can be overlooked and thus leaving end users at risk. Our dataset enables training solutions both using the commit message (e.g. using Natural Language Processing algorithms) or analyzing the commit impact on the code to better detect hidden security fixes.

**Cross-architecture Binary Diffing/Matching** An important and common task for reversers is to understand what changed between two versions of a binary or to port analysis results from one architecture to another. As our dataset contains the same binaries in different architectures, it enables research on cross-architecture binary matching [137, 140], where an application maps two code snippets from different architectures together if they have the same semantics. Because they are compiled from the same source code, two binaries for the same vulnerability share their semantics. Moreover, some binaries in our dataset are present multiple times in different versions. They are interesting examples for cross-architecture binary diffing [78, 36], where the objective is to find a mapping between functions in the first binary to the second one. These applications are an interesting extension of our dataset because they are not using the fact that some binaries are vulnerable while others are fixed.

**Patch Detection** Finally, our dataset creates a base to test tools aiming at solving the *patch presence problem* [66, 143, 138]. By isolating the difference between the vulnerable and fixed version of the same binary, it enables the creation of signatures to detect if a target binary has been patched. This is one of the main usages of this dataset, and we provide in Chapter 5 an explanation on how to use it in this context.

**Decompilation** To improve decompilation, approaches have been made to treat it as a NLP problem [47, 64]. However, these neural networks require huge datasets for their training which are not always available. Using ours for addressing these problems is attainable, as both source code and binaries are available.

### 4.5.4 Dataset Limitations

**Component Diversity** The main limitation of our dataset stems from the components used: they are all open-source. While numerous components are open, some parts (e.g. low-level firmware code, and drivers for specific hardware) remain closed, and are thus not considered in our dataset. This blind spot may lead to a lack of examples for complete classes of vulnerabilities.

**Single Point of Failure** Our dataset exclusively relies on Google's commitment to regularly publishing Android Security Bulletins. If Google stops the publication of new bulletins, or restricts their accessibility, our dataset will stop growing because as that prevent us from updating it with the latest vulnerabilities. Thus, our data relevancy would slowly fade as it would become outdated. Nonetheless, the current dataset and until the potential last Google bulletin, will remain usable.

**Data Quality** Our dataset implies that the commit referenced as a vulnerability fixing commit is complete, i.e. completely fixes the vulnerability, and minimal, i.e. it does not fix any other problem nor add functionalities. Detecting if both assertions hold is left unchecked, as orthogonal to this research. Automated patch verification is a different subject that could leverage studies on *Automatic Exploit Generation* [16, 57, 136].

## 4.6 Conclusion

None of the limitations hamper our approach validity. On the contrary, our dataset could serve as a baseline for building future works to overcome these limitations. For instance, our data are extensible by analyzing other bulletin providers or considering the CVE feed from the NVD.

Constructed from the interaction between AOSP and the Android Security Bulletins, our dataset features about 3,900 different vulnerabilities, with more than 1,300 associated with their fixing commit. This is one of the world's largest vulnerability datasets and it is effortless usable for various researches in multiple contexts.

In the context of this thesis, the work presented in this chapter serves two objectives.

1. It lets us gather a large ensemble of patches at both the source and binary level. This allows conducting an extensive study on real-world patches for vulnerabilities affecting most of an OS already presented in Chapter 3.
2. It permits us to generate some *signatures* for *1-day* patches and we use them as inputs for our **QSig** patch probing system.

Thus, the final work schema is upgraded in Figure 4.10 with the two new components developed in this chapter: **Roy** and **AOSPBuilder**.

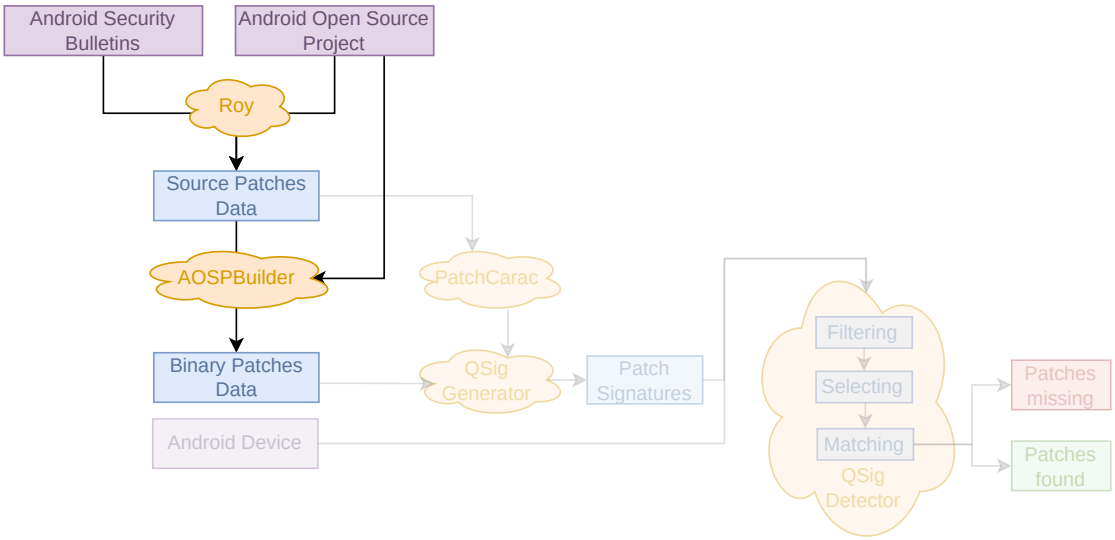


Figure 4.10: Workflow Updated with the Dataset Usage

Chapter 5

# Patch Detection using Binary Only Semantic Signatures

**Contents**



**QSig**, a semantic patch signature detection system, was introduced in Chapter 3. This system is composed of two components: a signature generator generating a signature from the difference between two binaries and a detector applying a signature onto a filesystem using the *F-S-M* strategy also presented in Chapter 3.

In this chapter, we discuss how **QSig**'s parameters can be tuned, and the default settings used by the system. Then, we assess the pertinence of **QSig**'s features, the system accuracy and efficiency by comparing it to various other approaches from the literature. Notably, we demonstrate **QSig**'s ability to truly considering the patch semantic by working in a cross-architecture setting: signatures generated on `x64` are used

to match `aarch64` binaries.

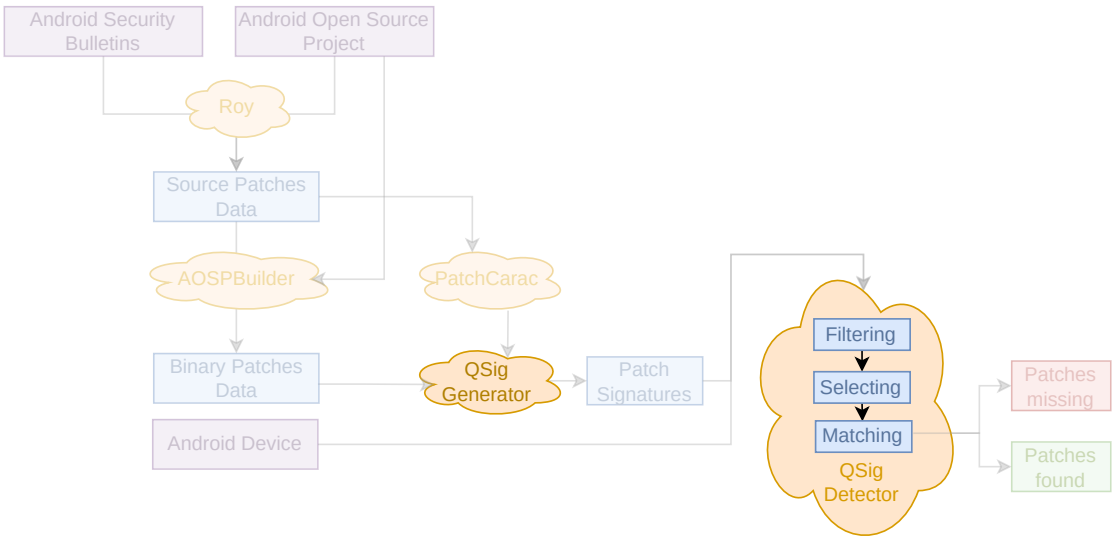


Figure 5.1: Patch Detection Evaluation

This chapter does not improve the schema presented in Figure 5.1 because it only evaluates the solution presented before.

## 5.1 Parametrization

**QSig** is a modular system, and its hyperparameters are adjustable to different contexts. We discuss in this section our default choices.

### 5.1.1 Selector Parameter

The *selector* selects $n$ functions from a target binary as candidates for the final matching step. If $n = |\{f \in T\}|$, i.e. we select all functions, the correct one is *certainly* in the selected ones. However, **QSig** matching signatures are not specific enough to only trigger on the correct candidate (e.g. several functions may exhibit a comparison between their first argument and a specific constant). On the contrary, if $n = 1$, the chance of missing the correct function increases. Thus, we need to carefully choose $n$.

To help us choose the *selector* default parameters, we used the dataset presented in Section 5.2.2. It consists of vulnerabilities detected in a Pixel 4 phone firmware image[1].

[1] A more precise description of the dataset is available in Section 5.2.2.

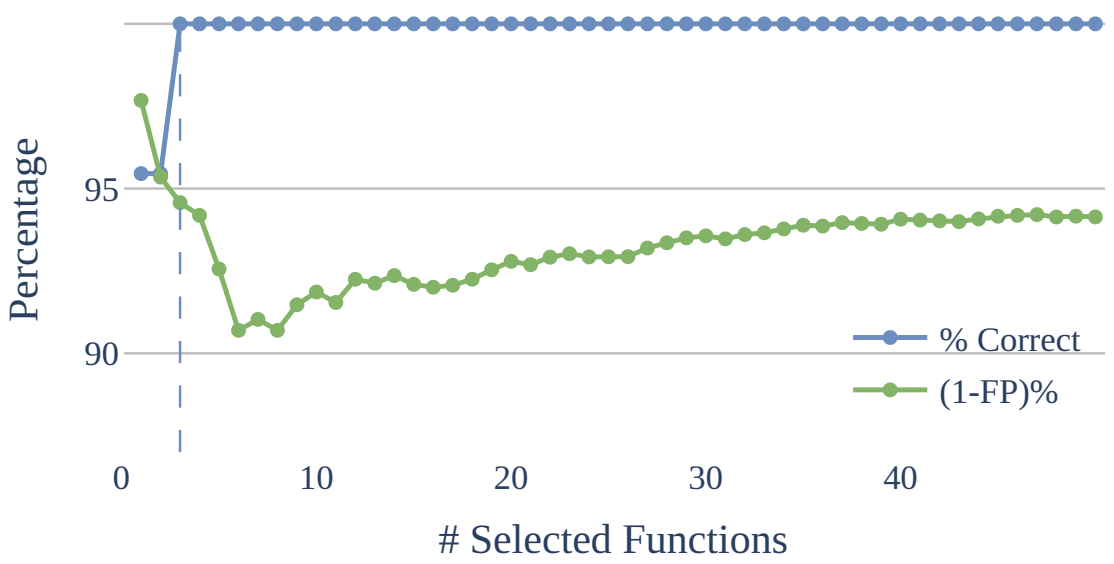


Figure 5.2: Precision for a Given $n$

When $n = 1$, the selected function is correct for 95% of the signature, and we have a few false positives (FP).By selecting 50 functions in each binary, the correct function is in the selected set in 100% of the cases, but 126 false positives occurs also. Therefore, we chose $n = 3$ in the evaluation. In this example, the correct one is always selected, and we only have 7 FP as illustrated in Figure 5.2.

A potential improvement for **QSig** would be to consider a dynamic $n$ that changes with the binary function count or the signature expressivity. Intuitively, selecting fewer functions in a smaller binary and more when the signature expresses characteristics for all features would potentially improve **QSig** accuracy. This is not yet developed but will be integrated into a future iteration of **QSig**.

### 5.1.2 Matchers Parameters

**QSig** matcher uses the four features underlined in Table 3.10 (strings, calls, constants, conditions) but patches may only exhibit a subset of them. If the running example presented in Figure 1.2 shows a complete patch to illustrate the features choices, it is not *typical*.

The analysis presented in Section 3.2 showed that 79% of the patches in AOSP had at least one of these features at a source level which hints the features' pertinence. For the 577[2] vulnerabilities considered, **QSig** finds a difference and generates a signature

[2]See Section 5.1.5

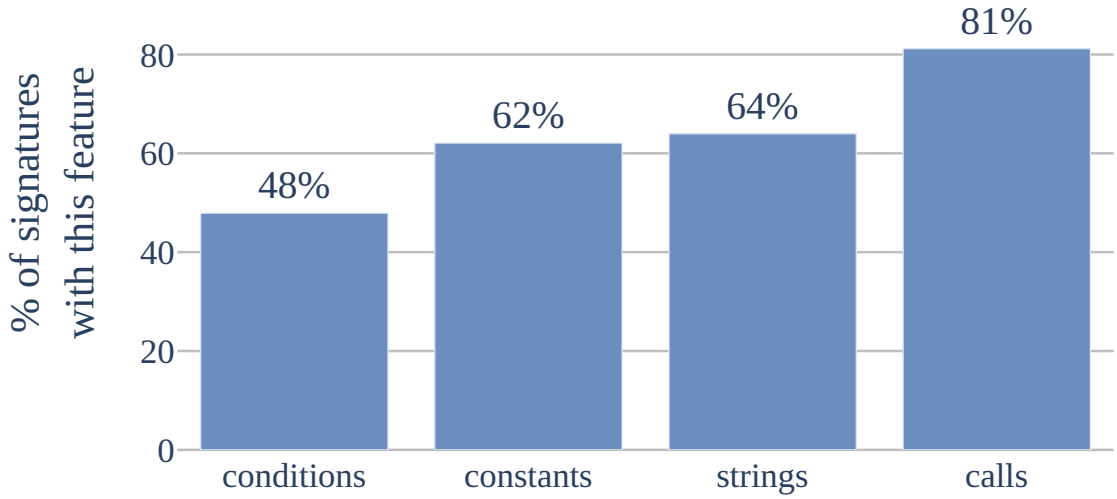


Figure 5.3: Feature Types

| Function | Valid | Invalid | Success Rate |
|---|---|---|---|
| any | 25 | 1 | 96% |
| majority | 21 | 5 | 81% |
| all | 7 | 19 | 27% |

Table 5.1: Choice Function Selection for Matcher

for 74% of the cases. The most common change signed is the `CALL` difference, which is present in 81% of the signatures as depicted in Figure 5.3.

Only a subset of AOSP patches exhibits the four features. Only 55% of the signatures exhibit strictly more than one feature. The exact distribution is depicted in Figure 5.4.

### 5.1.3 Matcher Results Combination Choice Function

**QSig** uses four features in the matching phase. However, they are not always all present. Thus, **QSig** considers each feature as an independent test function, and aggregates the results afterward. This strategy allows considering contradictory results, when the test results for two features are contradictory.

We evaluate three aggregation strategies:

- `any`: accepts a result if *any* feature reported a positive match;

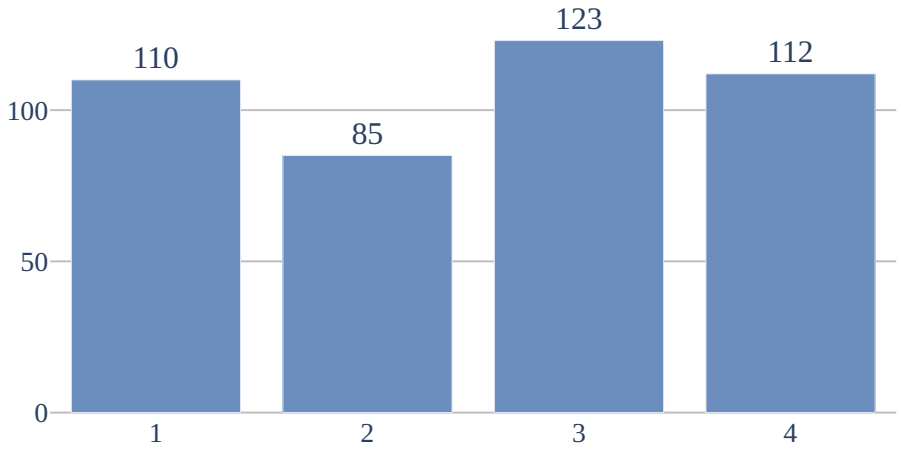


Figure 5.4: Number of Features

- `all`: accepts a result if *all* features reported a positive match;
- `majority`: accepts a result if at least *half* features reported a positive match.

Table 5.1 displays the results of the experiment on a dataset part. A result is valid if the match result using this strategy was the expected one. The best combination function is `any`. If our features are hard to detect and may have changed (e.g. a compiler may have decremented a constant used by a condition), the presence of at least one is a sufficient indicator of the patch presence.

### 5.1.4 Complete Patch Presence Test

A vulnerability patch may spread over multiple functions, in different binaries. 21% (309) of the signatures generated in Section 5.1.5 affect at least two binaries. 645 of the 2,109 impacted files present modifications in at least two functions. The CVE–2016–2412 patch includes 81 functions modifications.

While **QSig** inner matcher works at a function level, its objective is to assess whether a firmware is fully patched for a vulnerability. Thus, if a patch is made of multiple functions, to assess the end user security, we need to find the change in every affected function. Therefore, we combined function matcher results with the `all` strategy at a patch level.

### 5.1.5 Signature Generation

To validate **QSig**, we generate a signature on all 612 *vulnerability signatures* compiled in Chapter 4. The first step is to identify which functions have changed at the binary level. For this task, we use `BinDiff` [147], which resolves the binary alignment problem by comparing the call graphs of two executables [44].

| Architecture | CVE Signatures | Functions | Success Rate |
|---|---|---|---|
| X86 | 377 | 1072 | 61% |
| X64 | 371 | 1273 | 60% |
| ARM | 401 | 1069 | 65% |
| ARM64 | 339 | 938 | 55% |
| **Unique** | **459** | 1652 | **74%** |

Table 5.2: Signature Generation

For each compiled vulnerability, we search for changed functions with `BinDiff`. Since we have the symbols on the vulnerable and fixed binary, we only look at functions matched with a high confidence but a similarity score strictly below one. Using `BinDiff`, we found differences in 85% of the vulnerabilities (577).

Finally, we generate one signature for each architecture in a vulnerability. The generation results are presented in Table 5.2. In this experiment, **QSig** generated signatures for 74% of the vulnerabilities for which `BinDiff` reported a difference.

## 5.2 QSig Evaluation

To assess the usability of **QSig** for real-world scenarios, we test it in various contexts.

- On the CGC dataset for comparing it against 1dVul [98].
- On unmodified Pixel 4 firmware images.
- On a Debian Live image, a test case used by QuickBCC [59].
- On a stock AOSP version compiled for `x86`, the only architecture supported by PMatch [70].

This also demonstrates **QSig** versatility and adaptability to different use-cases.

### 5.2.1 Precision

**QSig** aims at distinguishing vulnerable from patched functions. Since 1dVul [98] uses the CGC binaries [30] for their work evaluation, we also use it for assessing **QSig** precision.

| | Correct | Incorrect | Success Percentage |
|---|---|---|---|
| Patched functions | 250 | 3 | 99% |
| Vulnerable functions | 212 | 41 | 84% |
| **Total** | 462 | 44 | 91% |

Table 5.3: **QSig**'s Accuracy

| | Total | **QSig** | 1dVul | Increase |
|---|---|---|---|---|
| Changed functions | 348 | 253 | 209 | +21% |
| Patch detected | 348 | 250 | 130 | +92% |

Table 5.4: Comparison of **QSig** and 1dVul

In the CGC dataset, a binary is present in both a vulnerable and in a patched version. We check the match result between our signature and the two versions. A result is *correct* if the two following statements are true.

1. The signature matches the patched version.

2. The signatures do not match the vulnerable version.

Table 5.3 shows the test results. **QSig**'s signatures are correct in 91% of the cases and only 3% of patched functions are not found.

**QSig** uses a static and semantic approach to solve the FMP. On the opposite, 1dVul [98] leverages a dynamic solution to answer the same problem. We compare both approaches' result on 1dVul's dataset as their tool is not available.

The 1dVul authors report generating 209 target branches for the dataset, from the 126 binaries on the 348 functions changed at a binary level. **QSig** generates a signature for 253 functions (21% improvement) demonstrating our approach's effectiveness. Moreover, 1dVul only generates an input for 130 targets to detect a patch where **QSig** finds 250 of them (92% more). These results are summarized in Table 5.4.

Not only **QSig** outperforms 1dVul, it also demonstrates that **QSig**'s static approach is more effective to answer the *patch presence test*.

| | |
|---|---|
| Before 2020.01 | CVE–2018–9547 CVE–2018–9506 CVE–2019–1996 CVE–2019–2009 CVE–2019–2133 CVE–2019–2134 CVE–2019–2179 CVE–2019–2187 CVE–2019–2202 CVE–2019–2220 CVE–2020–0006 CVE–2020–0007 |
| After 2020.01 | CVE–2020–0018 CVE–2020–0037 CVE–2020–0070 CVE–2020–0072 CVE–2020–0105 CVE–2020–0256 CVE–2020–0257 CVE–2020–0385 |

Table 5.5: Vulnerabilities List

| Feature | TP | TN | FP | FN | Pr. | Rec. | NA |
|---|---|---|---|---|---|---|---|
| Strings | 22 | 12 | - | - | 1 | 1 | 9 |
| Constants | 19 | 3 | - | - | 1 | 1 | 21 |
| Calls | 10 | 3 | 4 | 15 | 0.71 | 0.40 | 11 |
| Conditions | 1 | 2 | - | - | 1 | 1 | 40 |
| **Match** | 28 | 12 | 1 | 2 | 0.97 | 0.93 | - |

Table 5.6: **QSig**'s Matching Results (aarch64 to aarch64)

TP: True Positive TN: True Negative FP: False Positive FN: False Negative
Pr.: Precision Rec.: Recall NA: Not Applicable

### 5.2.2 Assessing Android Phone Firmware

We assess **QSig** adequation for real-world workloads with the following experiment: we search 40 patches on a Google Pixel 4 with an off-the-shelf firmware version.[3]. We split those 40 patches from 20 vulnerabilities into two groups:

- Group 1: Patches expected to be on the firmware.
- Group 2: Control group composed of patches posterior at the update.

The vulnerabilities used are listed in Table 5.5.

Table 5.6 and Table 5.7 presents the matching results for both *same* and *cross* architecture settings. It validates **QSig**'s approach to find patches with great accuracy. Moreover, the results are similar in the two settings and **QSig** maintains precision for cross architecture matching (0.97 vs. 0.81).

The test results for the *call* features are below the three other ones. Indeed, recovering the changes in the call graph is tedious. We can only rely on the functions in and out

[3] `QQ1D.200105.002, Jan 2020`

| Feature | TP | TN | FP | FN | Pr. | Rec. | NA |
|---|---|---|---|---|---|---|---|
| Strings | 21 | 12 | - | 1 | 1 | 0.95 | 14 |
| Constants | 13 | 3 | - | 1 | 1 | 0.93 | 31 |
| Calls | 2 | 6 | 3 | 23 | 0.40 | 0.08 | 14 |
| Conditions | 2 | 4 | - | 4 | 1 | 0.33 | 38 |
| **Match** | 21 | 13 | 5 | 9 | 0.81 | 0.70 | - |

Table 5.7: **QSig**'s Matching Results (x64 to aarch64)

| | TP | TN | FP | FN | Pr. | Rec. |
|---|---|---|---|---|---|---|
| **QSig** | 21 | 13 | 5 | 9 | 0.81 | 0.70 |
| PMatch | 4 | 12 | 0 | 26 | 1.0 | 0.13 |

Table 5.8: **QSig**'s Matching vs PMatch

degrees and we must also identify the source or targets of these calls to accurately match them which is error prone in our context.

Moreover, we also compare **QSig** results with PMatch [70] using the default settings provided by PMatch authors' implementation using the same patches as the previous test. Since PMatch only supports `x86` assembly code, we compile the Pixel 4 Android version for this architecture. To provide a fair comparison, we implement the PMatch *filtering* and *selecting* steps ourselves and only draw a comparison on the *matching* stage. The results are displayed in Table 5.8. PMatch only discovers 4 patches while **QSig** discovers 21 of them. However, PMatch has a better precision because it does not yield *any false positive* results.

### 5.2.3 QSig's Efficiency

Another key point to consider when developing a solution is to reduce the friction for end users. In our use case, reducing the feedback delay when assessing whether a device is fully patched is crucial. For this experiment, we compare **QSig** with QuickBCC [59], another tool for finding patches. Since their code is not available, we reproduce their experiment and search for 5 patches in a Debian 9 live image and list the results in Table 5.9. Scanning the Debian 9 live ISO image for 5 vulnerabilities takes about 3 min on a single core[4] without caching any results. Our results are three magnitudes faster than QuickBCC [59]. For the dry run, QuickBCC needed 533 minutes to match their

[4]Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz

| | Dry Run | | Cached | |
|---|---|---|---|---|
| | QuickBCC | **QSig** | QuickBCC | **QSig** |
| Run Time | 8h 53m 34s | 3m 09s | 3m 24s | 2m 11s |
| Preprocessing time | 8h 53m 19s | 1m 9s | 1m 34s | 8s |
| Preprocessing time (per file in ms) | 5,692 | 13 | 19 | 1 |
| Matching time (s) | 15 | 108 | 110 | 117 |
| Matching time (per file in ms) | 3 | 20 | 3 | 21 |
| Matching time (per signature in ms) | 0.01 | 3.95 | 0.01 | 4 |

Table 5.9: Comparison of **QSig**'s and QuickBCC Efficiency

signatures on 5,500 binaries from Debian 9 live image while **QSig** only needed 3 min.

Disassembling using `IDA` Pro [55] takes about half (52%) of the total time. This time is paid only once and the disassembly for subsequent runs is cached. Still, **QSig** is approximately 50% faster than QuickBCC after caching the preprocessing.

Nonetheless, QuickBCC generates one more signature than does **QSig**. CVE–2018–19841's patch [91] modifies how an array is accessed. Because this change is subtle in binary code and undetected by **QSig** features, our system cannot generate a signature for this patch. Thus, the results in Table 5.9 only consider 5 vulnerabilities.

### 5.2.4 Results Stability Over Time

To demonstrate **QSig** stability over time, we select a vulnerability from each bulletin between October 2019 and March 2020 and generate their respective signature. We then download the six Google Pixel 4 images for the same period (October to March) and try to match all signatures for each of them. We report in Table 5.10 a match with a check and the absence of a match with a circle. The color indicates if the answer is correct. **QSig**'s results are perfect. It does not find a patch before its application and finds them in every subsequent version after the patch release.

| | 2019 | | | 2020 | | |
|---|---|---|---|---|---|---|
| | Oct. | Nov. | Dec. | Jan. | Feb. | Mar. |
| CVE–2019–2187 (Oct. 2019) | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| CVE–2019–2202 (Nov. 2019) | ○ | ✔ | ✔ | ✔ | ✔ | ✔ |
| CVE–2019–2220 (Dec. 2019) | ○ | ○ | ✔ | ✔ | ✔ | ✔ |
| CVE–2020–0006 (Jan. 2020) | ○ | ○ | ○ | ✔ | ✔ | ✔ |
| CVE–2020–0018 (Feb. 2020) | ○ | ○ | ○ | ○ | ✔ | ✔ |
| CVE–2020–0037 (Mar. 2020) | ○ | ○ | ○ | ○ | ○ | ✔ |

Table 5.10: **QSig**'s Results Over Time

## 5.3 Limitations and Discussion

### 5.3.1 Threats to Validity

#### Adversarial Transformations

In Table 3.10, we consider only the features' resilience against *natural* function transformations that are not specifically forced but due to different software versions or regular software evolution. Adversarial transformations, specifically targeted towards hiding the features **QSig** searches, would vastly undermine **QSig**'s utility.

It is uncomplicated to create such transformations. For instance, encoding the strings in the binary is enough to break the `string` feature. However, we dismiss this limitation by stating that vendors have no interest in hiding their patches.

#### Signature

**QSig**'s signatures rely on four binary semantic invariants and the system generates a signature when a change is reported for at least one of them. This is the case for 74% of the vulnerabilities in our dataset. Naturly, when **QSig** does not find differences, it does not generate a signature. For instance, CVE–2016–2464 [88] patch modifies a comparison sign from $\geq$ to $>$. This change can be translated in multiple ways at a binary level (e.g. a mnemonic change, a constant decrement) and is not supported by **QSig**'s features.

To cover the remaining patches where **QSig** was unable to generate a signature, additional features must be developed. This is the main axis for **QSig** improvements and

a path for future works.

#### Tainting Algorithm

The tainting algorithm we developed is a substantial improvement over a syntactic approach like VIVA [135] because it follows the instruction semantics.

However, the algorithm only analyzes a single function, and ignores calls during the execution. This relaxation improves the algorithm performance. Nonetheless, this yields incorrect results in some cases. For example, in a call to `memcpy(*dest, *src, n)`, the taint should be propagated from the `src` to the `dst` argument. Failing to propagate the taint results in an important loss of information.

To partially solve this problem, a solution would implement a stub library to propagate taint results for widely used functions (e.g. the `libc`).

#### Patch Completeness

While **QSig** assesses a patch presence, it cannot draw any conclusion about the vulnerable status of the analysis target. Finding a patch is insufficient to conclude that the target is not vulnerable. Indeed, the correction may not completely fix the vulnerability [71]. On the contrary, not finding the patch does not prove the system's vulnerability. For instance, the patch may affect an unused or removed code portion.

This patch validity problem is orthogonal to this research. Hybrid approaches like 1dVul [98] check if they manage to crash the candidate binary with their generated PoV. However, they require extensive settings to execute real-code in a controlled environment.

## 5.4 Conclusion

In this chapter, we discussed **QSig** parametrization choices and evaluated them against various approaches in the literature. **QSig** is faster than other approaches while yielding more accurate results than other works. We also demonstrated it truly captured the patch semantic by performing seamlessly cross architecture matching.

**QSig** is a versatile system and can be adapted to specific workflows. In this chapter, we successfully applied it for three OS: Linux, Android, and Decree. In the following chapter, we introduce a novel *filtering* step to improve the detection of patches on Android devices. Finally, **QSig** is open-source and available on GitHub[5].

[5] https://github.com/quarkslab/qsig

Chapter 6

# Static Build Dependencies for AOSP

## Contents



AOSP's build system, *Soong* [52], allows analyzing component interdependencies by constructing and using Unified Dependency Graph (UDG) [40]. Considered from a se-

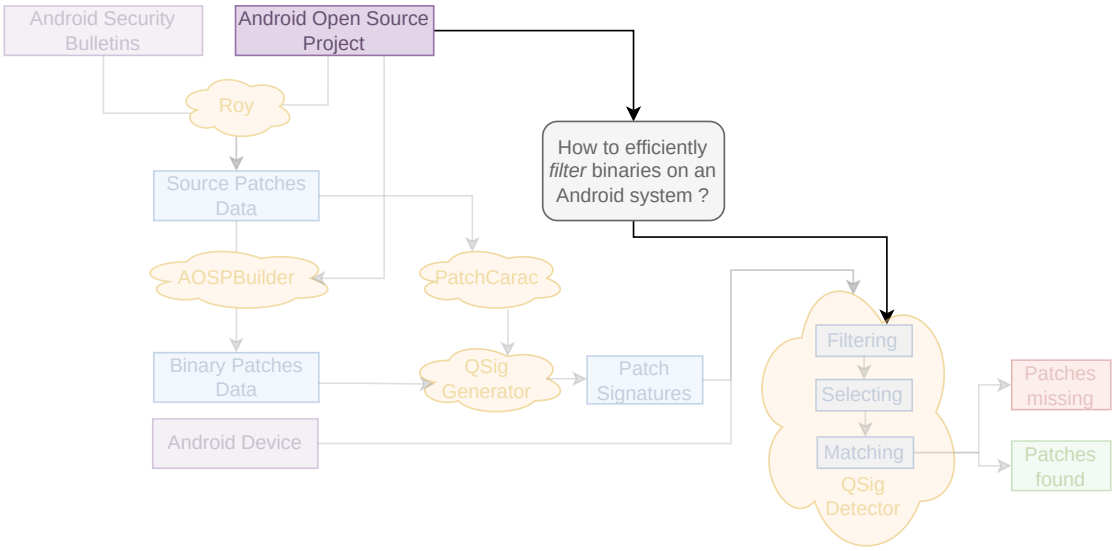


Figure 6.1: Build Dependencies for the Patch Detection Problem

curity perspective, these dependencies enable understanding vulnerabilities propagation through the system. This chapter describes the challenges faced during this graph construction and its potentials usages.

In the context of this thesis, we use these *Build Graphs* to enhance the *filtering* step of **QSig** when searching for patches in Android devices. Providing this information to **QSig** allows the system to also detect patches applied to statically embedded libraries. This use case is depicted in Figure 6.7 and detailed in Section 6.7.

## 6.1 Build Graphs

Modern software and large projects resort using build systems. Build systems automate not only the creation of software binaries but also sibling processes such as packaging or running tests. They offer an abstraction for developers for cumbersome and error prone tasks. The historical first build system was the GNU AUTOTOOLS with the `make` command. Several other tools were adapted for various workflows over the years: either generalist tools such as BAZEL or tools tailored to specific languages such as GRADLE for Java. In this manuscript, we are interested in exploiting the build scripts to understand the dependencies in large systems.

To drive these build systems, developers write *build scripts* in the appropriate format (i.e. Makefile). However, developing these build scripts is substantial and error-prone and bugs in build scripts are common. In Seo *et al.* study [113], dependency issue was the most common build failure reason (50% of C++ build-errors). Debugging these

**Listing 6.1** Extract of a Compilation Database

```
[
    { "directory": "/home/user/llvm/build",
        "arguments": ["/usr/bin/clang++", "-Irelative", "-DFOO=bar", "-c", "-o",
↪   "file.o", "file.cc"],
        "file": "file.cc"
    },

    { "directory": "/home/user/llvm/build",
        "command": "/usr/bin/clang++ -Irelative -DFOO=bar -c -o file.o file.cc",
        "file": "file2.cc" },
]
```

errors is tedious, as they stem from inconsistencies between *declared dependencies* and *actual dependencies* [40]. Thus, techniques to help developers have been introduced.

### 6.1.1 Compiler Builtins

Analysis tools based on the C/C++ AST need full information on how to parse the source code. During the compilation, a compiler will generate this information, but it is not easily extractable by other tools. To improve reusability, compiler engines or build systems may generate a `compilation database`, a record of which compile options are used to build the files in a project. For instance, this option is available in CMake since version 2.8.5 or in Ninja since version 1.2. An extract of a compilation database is presented in Listing 6.1[1].

A compilation graph is not a Unified Dependency Graph (UDG), and it requires further post processing to transform the former into the latter. Nonetheless, it already provides the information required to recover the dependencies between build targets and source files. However, generating a compilation database using an unsupported toolchain is demanding. Therefore, additional techniques, not relying on this file, have also been proposed in the literature.

### 6.1.2 Static Dependency Graph

Another option to recover build dependencies is to analyze the content of the build scripts. This approach is implemented in SYMAKE [118], which uses a symbolic evaluation algorithm to create a *symbolic dependency graph*. SYMAKE allows users to detect code smells and improve their confidence while refactoring some codebase parts. However, since SYMAKE only analyze *Makefile*, it is to only finding declared dependencies and cannot reason about missing ones.

[1] Example from https://clang.llvm.org/docs/JSONCompilationDatabase.html.

Parsing soundly build definitions is not always feasible. For example, *Makefile* rules can be dynamic (e.g. depending on an unknown input set) or non-deterministic (e.g. depending on another command output). Furthermore, it implies writing a parser for each supported build system. SYMAKE is the sole work statically analyzing build scripts. Every other approach used either dynamic or hybrid techniques.

### 6.1.3 Dynamic Dependency Graph

As analyzing statically build scripts is tedious, other approches instrumented the building environment. MAKAO [2] uses the build script output to recover every target dependency before converting them to a graph but its definition is not formalized. This approach scales well because MAKAO generates a graph for an entire Linux kernel build but is tailored for `make` output.

Licker and Rice [76] infer a dependency graph by tracing system calls. Their approach works for any build system because any persistent data must be used as an argument to a system call. However, their implementation has some limitations. For example, Java build systems use the compiler as a library and not in a separate process. Thus, every project source file is an input and every generated artifact an output of the same node. It therefore creates an under constrained graph.

### 6.1.4 Hybrid Dependency Graph

Detecting missing or redundant dependencies between the declared definitions and the real build system requires analyzing both parts statically and dynamically.

Sotiropoulos *et al.* [114] build on the ideas of Licker and Rice [76] and instrument calls to library functions using `strace`. Their approach is tailored for *Puppet*, a configuration management tool. Their approach detects faults such as *Missing Ordering Relationships* when a resource is accessed before its creation or *Missing Notifier* when a dependency link is missing between a configuration file and its configured service. Their detection system searches for discrepancies between the instrumentation results and the build definitions parsing.

Another hybrid approach is from Fan *et al.* [40]. They formalize the UDG and implemented its creation in a tool named VERIBUILD. An UDG is defined as follows:

**Definition 6.1.** *A Unified Dependency Graph (UDG) is a directed graph* $UDG = (V, E)$ *where* $V$ *is the set of nodes and* $E$ *is the set of edges.*

- $V = V_T \sqcup V_F$ *where*
  - $V_T$ *is the target node set (e.g. libraries, binaries);*
  - $V_F$ *is the file node set (e.g. source and headers files).*

- $E = E_{SD} \sqcup E_{SS} \sqcup E_{DA} \sqcup E_{DS}$
  - $E_{SD}$ *(Static Dependency): specified in build scripts*
  - $E_{SS}$ *(Static Spawn): output files for a target*
  - $E_{DA}$ *(Dynamic Access): files accessed during the build*
  - $E_{DS}$ *(Dynamic Spawn): files built during the build*

The tool works in two successive steps. Step one analyzes the build scripts to generate a dependency graph. In step two, this graph is then augmented by the build instrumentation output. Mixing these two techniques allows them to detect whether a dependency link is not present both statically and dynamically.

### 6.1.5 Conclusion

In this section, we introduced several works to generate build dependency graphs. While compilation databases could have served as a baseline, they were not used by the works in the literature which usually instrumented the build system to recover the build dependency graph. However, if the byproduct of the compilation is not needed, this process has several flaws.

- Large project compilation usually takes time[2].
- The artifacts generated by a compilation take disk-space.
- Compiling a project requires preparing a valid build environment with all the project's dependencies.

## 6.2 Definitions

**Definition 6.2.** *A **compilation target** (or unit) is the output of a non-default rule in Soong (e.g. an executable, a shared library, a static library...).*

**Definition 6.3.** *A compilation unit **dependency chain** is composed of the set of all compilation targets and source files used to build the unit.*

**Definition 6.4.** *A compilation target is **affected** by a source file if this source file is present in the target compilation dependency chain.*

Given a source file, knowing which compilation targets it affects allows establishing the impact of a vulnerability affecting this source file. On the other side, knowing a source file is used in multiple easily reachable components could justify an additional security assessment during an audit.

[2] https://xkcd.com/303/

The dependency chain for a compilation target is not necessarily complex. For example, if a single source file is enough to generate an executable, it is effortless to recover the chain. Dependencies induced by static libraries are much harder to detect. While it is possible to determine them manually by looking at each module dependency, this approach is both tedious and not scalable. Another approach is to instrument the build system or to analyze debug symbols at the compilation termination. These two options have a common prerequisite: they need to *perform* the compilation first, which takes time, and which implies an appropriate build environment is available.

Thus, the problem is to find an automated solution to establish build dependencies without compiling them. In 2020, Fan *et al.* [40] formalized the notion of a Unified Dependency Graph (UDG), which we refine below.

**Definition 6.5.** *A simplified* ***Unified Dependency Graph*** *is a directed graph* $UDG = (V, E)$ *where:*

- $V$ *is the set of nodes such that* $V = V_T \sqcup V_F$ *with* $V_T$ *the compilation targets set and* $V_F$ *the source files set.*
- $E$ *is the set of dependency links directed from dependency to dependent.*

Fan *et al.* definition distinguished various edge dependency types. However, due to the *Soong* particularities, our study only considers static dependencies[3]. Thus, we simply consider the edges as directed links between two nodes.

Fan *et al.*'s method [40], VeriBuild, analyzed *Makefile* for various open-source projects. However, Android's build system, *Soong*, is unique and the techniques to build the UDG must be adapted.

## 6.3 Android Build System: Soong

### 6.3.1 A Build System Tailored for AOSP

Since Android 7 (Nougat), the Android build system is named *Soong* [52]. It replaces the previous system based on *GNU autotools* and *Makefile*. While adaptable to other contexts, only AOSP uses this build system now. Note that *Soong* is not the only build system currently developed by Google: it is also actively developing Bazel.

*Soong* is a Go program and is part of AOSP[4]. It implements all the build logic complexity and orchestration in its modules. *Soong* bootstraps itself: the information to

[3] Indeed, in *Soong*, runtime dependencies must be explicitly stated in the module definition.
[4] https://android.googlesource.com/platform/build/soong

| | *GNU autotools* | *Soong* |
|---|---|---|
| File | Makefile | blueprint |
| Syntax | "Makefile" | "JSON-like" |
| Compilation unit | rule | module |
| Imports Childrens | Manual | Automatic |
| External calls | Yes | No |

Table 6.1: Comparison Between *GNU autotools* and *Soong*

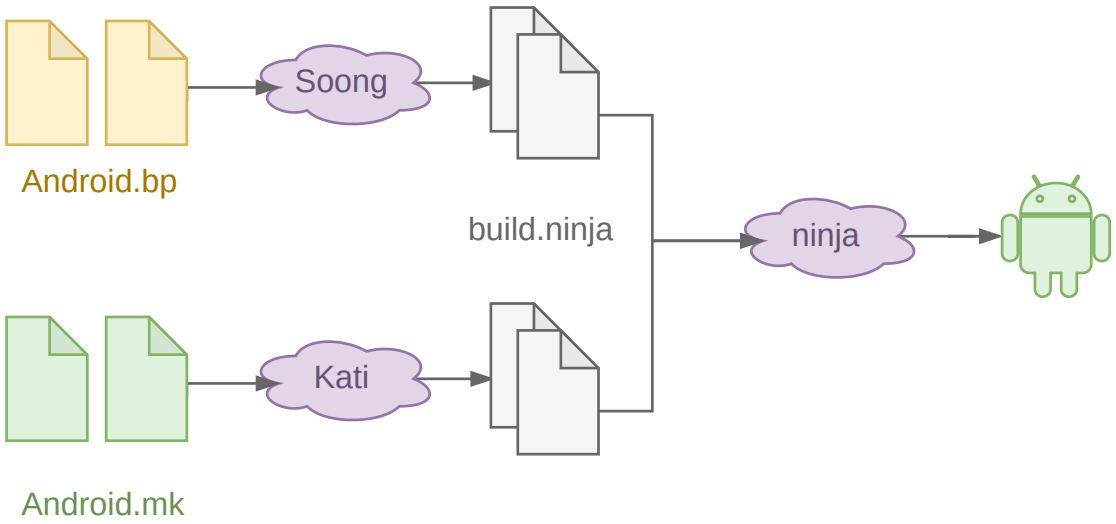


Figure 6.2: AOSP Build System

build *Soong* is also described in a *blueprint* file. The Soong output is not an Android image, but a *ninja* [39] file that will then be used to perform the actual compilation[5].

During the build systems transition, all remaining Makefiles are read with Kati[6], a `make` clone developed by Google to improve its performance in Android context. As expected, Kati outputs also a ninja file that is combined with the one generated by *Soong* before building the system. The process is illustrated in Figure 6.2.

[5] *ninja* is a small build system originally developped to improve Google Chrome compilation speed. It is driven by *ninja* files that are not supposed to be hand written but generated by a *build generator* like CMake.

[6] `https://android.googlesource.com/platform/build/kati`

**Listing 6.2** Inheritance in *Soong*

```
1  cc_defaults {
2      name: "dexlayout-defaults",
3      defaults: ["art_defaults"],
4      host_supported: true,
5      shared_libs: [
6          "libbase",
7      ],
8  }
9
10 art_cc_binary {
11     name: "dexlayout",
12     defaults: ["dexlayout-defaults"],
13     srcs: ["dexlayout_main.cc"],
14     shared_libs: [
15         "libdexfile",
16         "libprofile",
17         "libartbase",
18         "libart-dexlayout",
19     ],
20 }
```

### 6.3.2 Blueprints

*Blueprint* files contain *Soong*'s compilation directives. The name comes from `blueprint`[7], a now-retired build system framework developed by Google. Soong's blueprints are uncomplicated because Soong handles most build logic. Their syntax resembles a mix between JSON and Protocol buffers [51] and is close to Bazel `BUILD` files. To help the reader understand the terminology, we draw a parallel in Table 6.1 between *GNU autotools* and *Soong*.

Lines 1 and 10 in Listing 6.2 define the module type. This will be used by *Soong* to select the appropriate building rules. Soong implements many types, either based on the source language (e.g. sh, python), the target destination (e.g. source or host), or the build system configuration.

Each module in *Soong* must have a unique name (Lines 2 and 11). It is used to derive the module output final name and to reference them in other modules. For instance, on a Linux platform, the module displayed in Listing 6.4 will generate the library `libpldump.so`.

*Soong* offers an inheritance mechanism for *blueprints* writers with the `defaults` keys. If a module defines defaults, *Soong* initializes the child with the parents information and replaces or merges those overwritten in the child definition. In Listing 6.2, the `dexlayout`

[7] `https://github.com/google/blueprint`

**Listing 6.3** Maps for Conditionals Expressions in *Soong*

```
cc_library_static {
    name: "libc_common_static",
    arch: {
        x86: {
            srcs: ["arch-x86/static_function_dispatch.S"],
        },
        arm: {
            srcs: ["arch-arm/static_function_dispatch.S"],
        },
    },
    ...
}
```

**Listing 6.4** Module Definition in *Soong*

```
cc_library_shared {
    name: "liblpdump",
    defaults: ["lp_defaults"],
    shared_libs: [ "libbase", "liblog", "liblp"],
    static_libs: ["libjsonpbparse"],
    srcs: ["lpdump.cc",
           "dynamic_partitions_device_info.proto"]
  }
```

target will have the `libbase` as a dependency because it inherits from `dexlayout-defaults`. Of note, it is possible to chain defaults, and to specify multiple defaults for a single module.

*Soong* offers a variable pattern that works similarly to the C preprocessing macro: the variable content is replaced before its usage. Variables are immutable[8]. While they can be used for various purposes (e.g. configuration), their usage in AOSP is limited: 500 occurrences in the 3,643 blueprints for *Android 10*.

Finally, because the build complexity is handled by the engine with high-level language constructs, conditionals are not possible in *blueprints*. Instead, they are converted to map properties as illustrated by Listing 6.3 when the source files are architecture dependent.

### 6.3.3 Dependencies in Soong

Listing 6.4 displays a blueprint extract defining the rule to compile `liblpdump`.

[8] With one exception: they can be appended before their first reference.

A blueprint module definition lists all runtime dependencies. In Listing 6.4, the module defines three dynamically linked library with the `shared_libs` key (Line 4). On a compiled binary, it is possible to list these dependencies (i.e. by using `ldd`). Statically linked library are enumerated with the `static_libs` key (Line 5). However, this information is not easily recoverable on the final binary.

The only external prerequisite for statically analyzing a *blueprint* is the file system layout. Indeed, *Android.bp* files use the `glob` operator to list files. However, the file content is superfluous, and the file list is sufficient. In Listing 6.4, the source dependencies are listed Line 6.

## 6.4 BGraph Construction

### 6.4.1 From UDG to BGraph

Our UDG applied to AOSP is called BGraph, for *Build-Graph*. It represents dependency links between compilation targets. As for a regular UDG, nodes are compilation targets and source files. The edges represent relations between two nodes and are directed in the usage direction: from the dependency to the dependent. Because all AOSP components use *Soong*, the BGraph represents system-wide component dependencies.

As runtime dependencies must be explicit in Soong, we are confident that the graph created from the blueprint analysis is complete. This allows working statically, and avoids the build complexity environment and the source code itself. Indeed, we only require build files.

Using a graph representation allows us to extract the information using standard algorithms developed on this data structure. For instance, the potential targets of a source file are the node set for which there exists a path in the graph. Figure 6.3 shows the Listing 6.4 as a graph.

To generate the UDG for AOSP, we parse every build file (e.g. `Android.bp`), recover their module definitions, and add an edge between modules listed in each dependency keys.

### 6.4.2 Results

As previously stated, the migration to Soong has only started in Android 7 (2016). We decided to generate BGraph for every tag in AOSP since. It represents about 350 versions from `7.0.0_r1` to `11.0.0_r31`. We build a new graph for each version as the dependencies are tied to a version. Moreover, this allows us to understand the differences between the two versions.

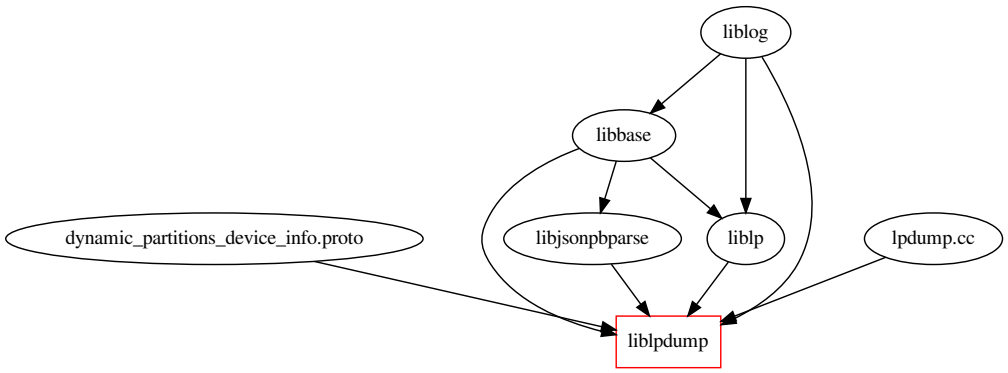


Figure 6.3: Listing 6.4 as a Graph

This migration remains a work in progress. For example, in `android-12.0.0_r32`, there are still 1,170 *Android.mk* files for 7,740 *Android.bp*. Our tool does not analyze the latter and ignores dependencies defined in such files. However, the precision of BGraph will only increase as much as the migration progress.

The compilation target repartition for Android 11 is depicted in Figure 6.4. The most common type is `test` because tests are present as companions for most targets. Worth noticing, about 5% of the targets build code for the host, i.e. the computer building AOSP.

## 6.5 BGraph: A Tool to Create and Query Graphs

BGraph[9] is the tool we developed to generate and query dependency graphs for AOSP (also named BGraph). It is an open-source Python module, with both a Text User Interface (TUI) and an API.

### 6.5.1 Creating One (or More) Graph

To create a graph, BGraph needs access to an AOSP mirror. While the method is applicable to every codebase using *Soong*, we tailor our implementation for AOSP. After selecting a version, BGraph will automatically download the blueprints for every project. Downloading only the blueprints allows saving valuable disk space (125 Mb instead of 80 Gb). Finally, each blueprint is analyzed, their content combined, and then transformed into a graph.

[9] https://github.com/quarkslab/bgraph

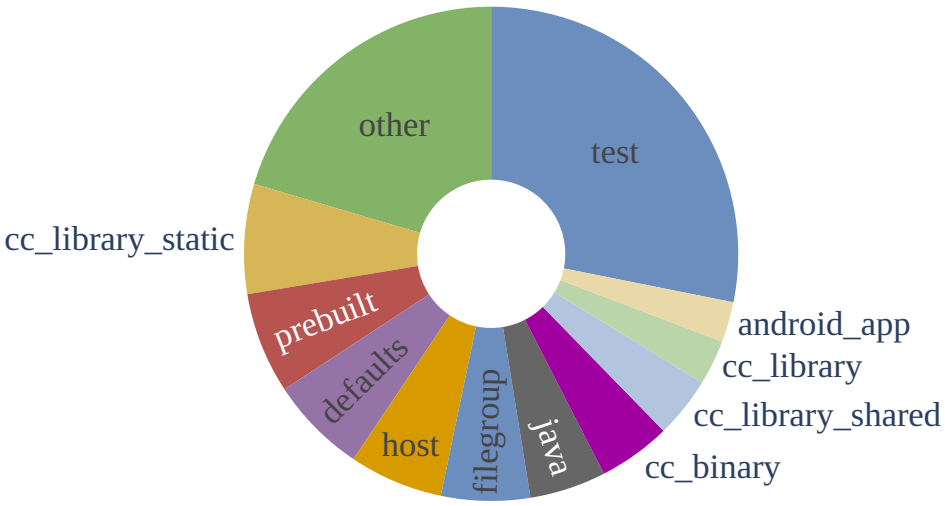


Figure 6.4: Compilation Target Repartition in Android 11

### 6.5.2 Query a Graph

BGraph TUI implements a `query` command to retrieve information inside the graph. Two main types of queries are pre-implemented, other types need to be performed using the API. The first query searches for every dependent target from a source file. And the second one performs the inverse operation, looking for the source files *used* by a target. Both query complexity is linear in the number of nodes in the graph. To ease reusability and integration into other toolchains, BGraph outputs the result in various formats (JSON, DOT, text).

## 6.6 Use Cases

### 6.6.1 Diffusion of CVE–2020–0471

CVE–2020–0471 [93] was fixed into January 2021 Android Security Bulletin. This flaw allowed an attacker to inject packets inside a Bluetooth connection and could lead to an Elevation of Privilege (EOP). Commit `ca6b0a21`[10] fixed the vulnerability by changing the file `packet_fragmenter.cc`. We will not perform a vulnerability in-depth analysis, but only consider its potential impact in AOSP.

`packet_fragmenter.cc` is used while building the static library `libbt-hci`. Since static libraries are not visible in the imports of a binary, a classical dependency system would be limited in its analysis. However, using BGraph allows circumventing this limitation because it also resolves static dependencies.

[10] https://android.googlesource.com/platform/system/bt/+/ca6b0a211eb39ba85eed60ea740c85d1122fc6bc

**Listing 6.5** Example of BGraph Query

```
1  % bgraph query graphs/android-11.0.0_r31.bgraph --src 'packet_fragmenter.cc'
2          Dependencies for source file
3              packet_fragmenter.cc
4
5  Target       | Type              | Distance
6  =============|===================|==========
7  libbt-hci    | cc_library_static | 1
8  libbluetooth | cc_library_shared | 2
9  libbt-stack  | cc_library_static | 2
10 Bluetooth    | android_app       | 3
```

Listing 6.5 displays this request. In the command output, we observe that only the shared library `libbluetooth.so` (line 8) is an entry point on the system. Moreover, the Bluetooth application in AOSP also uses the library, and thus the vulnerable code. If in this case, the dependency relationship seems trivial, BGraph allows performing such requests automatically, and to verify every impacted binary.

### 6.6.2 Looking for Interesting Targets

In this example, we show how to use BGraph to list the most used files in AOSP, and *in fine*, which target would be interesting to analyze more thoroughly.

| Target | # Nodes | Description |
|---|---|---|
| libbase | 3025 | Classical functions |
| fmtlib | 3027 | Alternative to *stdio* and *iostreams* |
| liblog | 3340 | Log library |

Table 6.2: Most Used Dependencies in AOSP

By using BGraph API, we can compute the graph size induced by each source file in AOSP. The graph size scales with the number of targets using this file. A large graph means that a file is deeply intricate within AOSP. Table 6.2 lists the three most used libraries in AOSP. Someone looking for a code review could prioritize these libraries as the impact of a vulnerability in one of them could be disastrous. For example, the Log4J vulnerability [22] affected a widely used logging library in Java and had a tremendous impact.

While giving already interesting and exploitable results, it is also possible to refine this query to exclude modules targeting the host or to filter the target languages.

**Listing 6.6** Function to Check If a Vulnerability Is Static

```
def is_static_lib_vuln(
    graph: bgraph.BGraph, vuln: Cve
) -> bool:
    # Find the first target in the graph
    _, targets = bgraph.viewer.find_target(
        graph, vuln.file, radius=1
    )

    # Resolve node types
    node_types = set(
        bgraph.viewer.get_node_type(
            graph.nodes[targets[0]], all_types=True
        )
    )
    return 'cc_library_static' in node_types
```

## 6.7 Detecting Patches in Statically-Linked Code

### 6.7.1 Finding Vulnerabilities in Static Libraries

**Definition 6.6.** *A **static vulnerability** is a vulnerability affecting a library that will be statically embedded.*

To illustrate one of BGraph strengths, and to create the dataset for one of the experiment in Chapter 5, we want to list every static vulnerability in AOSP. Static vulnerabilities are usually challenging to patch, because they require a dependency management process.

To find every AOSP static vulnerability, we devise the following strategy:

1. List AOSP vulnerabilities.
2. For each vulnerability, find affected source files.
3. For each affected source file, list the immediate children in their BGraph. These children represent direct inclusion links: from a source to a binary target.
4. Accept the CVE if one of the children is a static library.

Thanks to the work presented in Chapter 4, we already have an exploitable list of all vulnerabilities affecting AOSP along with the list of changed files for each patch. The immediate children are listed with the `find_target` method in Listing 6.6 and the `radius` parameter restrict the induced graph size. The module type defines the compilation target types. Thus, we search whether one of them represents a static library.

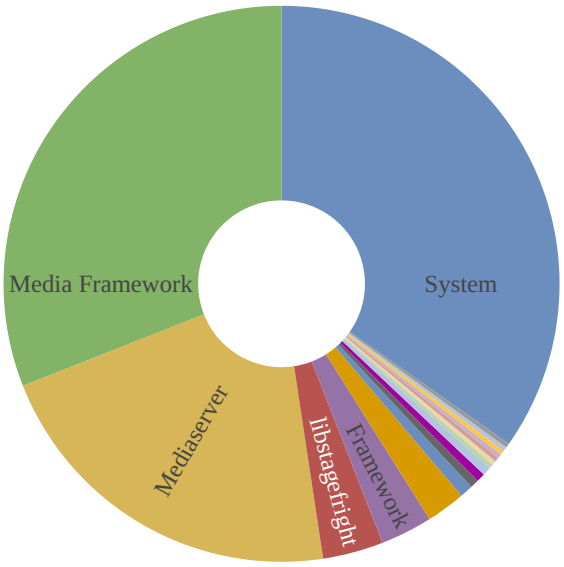


Figure 6.5: Static Vulnerabilities Repartition

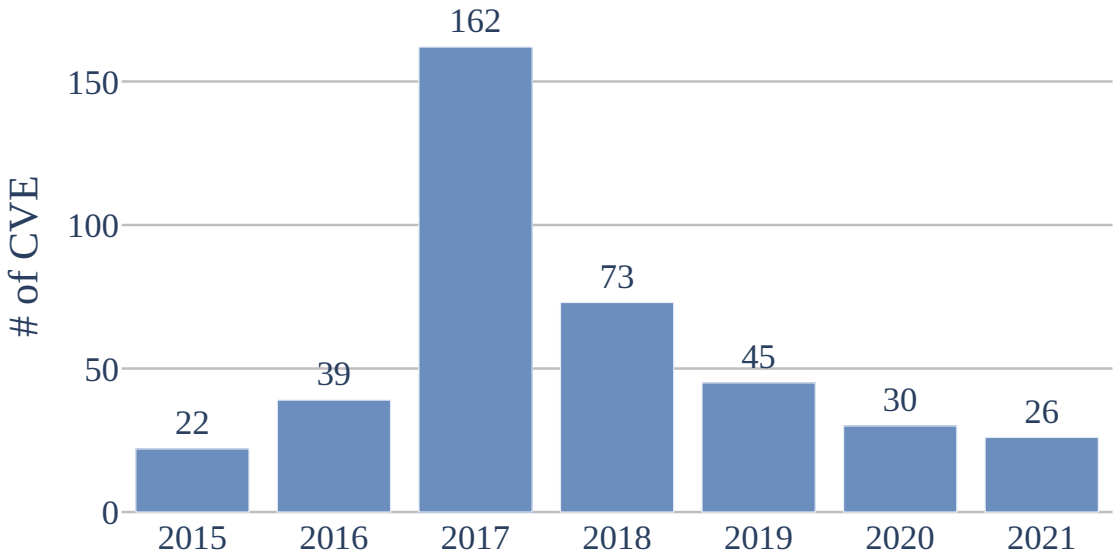


Figure 6.6: Static Vulnerabilities per Year

| Feature | TP | TN | FP | FN | Pr. | Rec. | NA |
|---|---|---|---|---|---|---|---|
| Strings | 19 | 60 | - | 3 | 1 | 0.86 | 48 |
| Constants | 25 | 64 | 2 | 8 | 0.93 | 0.76 | 31 |
| Calls | 9 | 61 | 6 | 29 | 0.60 | 0.24 | 25 |
| Conditions | 6 | 15 | 1 | - | 0.86 | 1 | 108 |
| **Match** | 35 | 70 | 5 | 20 | 0.88 | 0.64 | - |

Table 6.3: Detection in Static Libraries

The query returns 351 vulnerabilities, affecting mostly the Media Framework, the Media Server, and the System components. The results are depicted in Figure 6.5. The vulnerability number remains stable over the years (except in 2017). This is explained because the number of vulnerabilities reported in 2017 is much higher as seen in Figure 4.8.

### 6.7.2 Detection of Patches in Static Libraries

Detecting patches in statically embedded libraries is challenging and left aside by other works in the literature. The main issue stems from finding in which targets the library has been included. Nonetheless, they are prevalent but less updated than standalone components.

We implement in **QSig** a filtering step using BGraph. A candidate binary is accepted by the filter using a similar procedure as the one presented in Listing 6.6. To demonstrate **QSig** effectiveness at finding such patches, we searched in six shared libraries from Android[11] 84 vulnerabilities patches. Of those 84, the target version contains 35 of them and 49 are posterior to the release. The 84 vulnerability patches relate to 130 functions. The results are reported in Table 6.3. **QSig** detects the patch with a precision of 87% while maintaining recall at 64%.

This result is only achievable because **QSig** implements the *F-S-M* strategy and each component is easily customizable.

[11] Version 8.1.0_r23

## 6.8 Discussion

### 6.8.1 Limitations

#### Build System Exhaustivity

BGraph's results rely on the *Soong* build system exhaustivity. However, this assertion does not entirely hold. AOSP's components using the old *GNU autotools* implementation are omitted in the UDG. Thus, a vulnerability affecting a file of such component would not be tractable across the system. This limitation will be progressively erased with the migration completion.

#### Incomplete Blueprint Support

Blueprint files implement a complete language albeit not complex. Still, some language features are currently unsupported in our implementation. For example, the support of mapping is missing in our implementation. Mappings are used to tailor the file selection for a specific architecture. This could be solved with additional engineering effort.

#### Combining with Other Approaches

Finally, combining BGraph with other approaches based on `Makefile` is unfeasible since the detail level is too disparate. For instance, the target type is implicit in a `Makefile` and usually guessed using the target extension but explicit in Soong.

### 6.8.2 Conclusion

The work presented in this chapter introduced a new solution to solve the dependency propagation problem: finding in which compilation target will end up a source file. In AOSP context, our BGraph creates a UDG statically from build file definitions and uses it for answering user defined queries.

Of note, BGraph is an open-source tool and is available on Quarkslab's GitHub [20].

The Figure 6.7 is updated the BGraph usage as a *filtering* step for **QSig**.

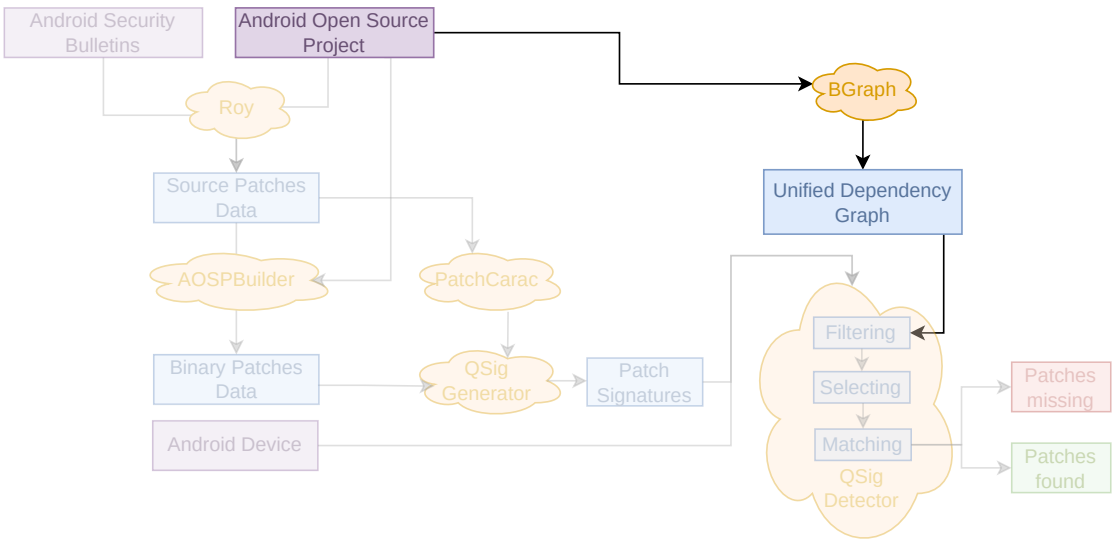


Figure 6.7: Unified Dependency Graph for Patch Detection

Chapter 7

# Conclusion & Perspectives

In this closing chapter, we conclude this thesis by summarizing its contributions. Then, we present opportunities opened by our work.

## 7.1 Conclusion

This thesis focused on a particular spot of the information systems defense: how to ensure a system is protected against known threats? To answer this question, security teams must know which vulnerabilities are affecting their system and the defenses already in place. Indeed, due to the *patch propagation delay*, a patch availability is not enough to ensure its presence in a system.

These vulnerabilities are called *1-day* vulnerabilities, because at least *one* day has passed after the patch release. To understand this vulnerability class, we conducted an extensive study of their common traits, and their associated patches. We characterized them by reducing their patch as a set of change types. This work helped us to find the most common changes performed by developers to fix a vulnerability. We leveraged this information to create patch signatures.

To understand a system's exposure to *1-day* vulnerabilities, the next step is to perform a patch presence test: detecting which patches have been applied to a system, down at the binary level. Indeed, relying on reported version numbers is insufficient to detect the security level. Thus, this is a challenging problem, and we established a three-step strategy, the *F-S-M*, to solve it swiftly and efficiently.

Because our work was conducted in an industrial context, a theoretical approach is unsuited to tackle the problem in a real-world assignment. Thus, we proposed an implementation of our solution in our open-source system **QSig**. **QSig** searches patches

on any filesystems using patch signatures based on semantic invariants derived from our patch analysis.

In Chapter 5, we extensively tested **QSig** performances for real-world workloads using a vulnerability dataset we created. This dataset provides vulnerability information at two levels: a first one at the source level, with commit precise information, and a second one with precompiled binaries in multiple architectures. While we use the dataset to generate patch signatures and assess **QSig** results, it is independent of our work and allows multiple security workflows. Hence, we open-sourced it.

Finally, to improve the state of the art at detecting patches applied to statically embedded libraries, we developed a novel filtering step based on build dependencies graph for Android devices. This demonstrated both the pertinence of our three-step strategy and the extensibility of **QSig**.

#### Community Contributions

The works presented in this thesis are often accompanied by various open-source tools or resources. We list them below.

- BGraph (`https://github.com/quarkslab/bgraph`)
  BGraph is a tool designed to generate dependencies graphs from Android.bp Soong files.
- QSig (`https://github.com/quarkslab/qsig`)
  QSig generates semantic patch signatures from the difference between two binaries and applies them onto a filesystem.
- Dataset (`https://github.com/quarkslab/aosp_dataset`)
  Large Commit Precise Vulnerability Dataset based on AOSP CVEs.
- *Quokka* (`https://github.com/quarkslab/quokka`)
  A Fast and Efficient Binary Exporter for `IDA`.

## 7.2 Perspectives

In this section, we discuss some perspectives our work opens.

### 7.2.1 Extending our Vulnerability Dataset

Our dataset relies on Google's commitment to provide regular and accurate security bulletins because it is our dataset's main information source. While they steadily published bulletins for the last seven years, nothing prevents them from stopping their effort. To improve our dataset resilience, a potential improvement would be to monitor other

security bulletins, either from different manufacturers (i.e. Samsung) or for different systems (i.e. Debian). This would require additional engineering effort but would increase the dataset representativeness of security vulnerabilities.

Moreover, to provide precompiled binaries, we developed a build automation protocol around AOSP's build system. Nonetheless, this protocol is laborious and error-prone. However, Google provides an open access to Android's Continuous Integration (CI)[1]. It presents the result of every AOSP compilation and offers to download the compilation artifacts.

With additional engineering efforts and a better understanding of the CI interface, it should be possible to obtain the binaries required for our dataset directly from Google's server. This would avoid the compilation and project synchronization burden while still providing accurate results.

### 7.2.2 From Filesystems to Raw Firmwares

Analyzing generic embedded devices is challenging for numerous reasons. The work presented in this thesis focused on *filesystems* analysis. This requirement was adapted for Android devices as they run modified Linux systems. However, smaller and heavily specialized devices may leave out this abstraction level and only run flat firmware. To handle such firmwares, an analyst requires a proper disassembly which is one of the substantial tasks listed by Wright *et al.* [133]. For instance, even finding the base address requires adapted tooling and techniques analysis [103, 146].

Moreover, analyzing a COTS device usually requires extracting the firmware. While in Android case, the firmware is usually accessible, Vasile *et al.* [121] survey existing techniques to perform this task on more generic devices. These techniques range from finding the firmware or an update on the manufacturer's website to physical attacks by using debug protocols (USART, JTAG). Thus, they are hard to automate and scale, which is an additional challenge when performing a security assessment of heterogeneous device fleets.

In **QSig**'s context and to broaden its applicability, the remaining tasks would be to remove the *filtering* part as it is not adapted to flat firmware and to extend the toolchain to handle the architectures used by the devices under analysis.

### 7.2.3 Towards Semantic Queries

Our method to detect *1-day* patches on binaries detects artifacts presence in binaries. Our most advanced feature, the condition term tracking, goes one step further and

[1] https://ci.android.com/

reasons on the dataflow graph. However, newer code representations are generating a more accurate and complete representation of the code semantics.

CodeQL [48] and Joern [141] work in two phases, they first generate a code database. Then, they provide an interface to query this database[2][3]. Nonetheless, these tools bear a constraint: they work only at the source code level.

By extending one of them to handle binary code (either directly or through an IR), it should be possible to encode the patch presence test for a vulnerability as a query on the code representation. For instance, we could encode the fix for the vulnerability presented in Chapter 1 (Listing 1.1) as a request. This request would check if the field at offset `2` in the structure `p_pkt` is compared with the constant `2`. Such an approach would allow writing more precise queries on the patch semantic but present some research challenges. The first one is to obtain a code representation precise enough to propagate type information or value inference. The second one would be to understand how to generate such queries automatically.

### 7.2.4 Patch Application and Device Security

Detecting *1-day* in systems is insufficient to evaluate the end users security completely. First, the patch presence test only answers whether a patch is present, but its absence does not necessarily induce a risk on a device (e.g. it may not use the vulnerable code). Moreover, the information that a vulnerability is present only helps stakeholders to make an informed decision. Our work provides information that must be acted upon. Indeed, knowing its presence does not prevent exploitation attempts and additional steps are required to protect a device. These remediation steps could range from non-technical options (i.e. avoiding using a vulnerable device) to more technical ones (i.e. network exploitation protection systems, code firewalling, or sandboxing) and in straightforward scenarios, applying an update containing the missing patch.

Finally, even finding a patch does not necessarily ensure that a device is protected against a vulnerability. Indeed, patches may only partially correct a flaw in a program [71], and additional work is required to ensure a patch is completely fixing a vulnerability. Automatic exploit generation approaches [136, 7] could be a starting point: generating an exploit from the vulnerable code and applying it to a fixed binary to ensure it is properly patched, extending the approach in 1dVul [98]. However, this is a challenging problem because the exploit must reliably trigger the bug and fail only when the vulnerability is fixed.

[2]CodeQL's queries are written in `QL` and Joern's in Scala.

[3]Other solutions such as Semgrep [106] or weggli [132] skip the database generation but performs pattern matching on the AST and do not reason on the code semantic.

### Final Words

The common best practice is to consider that security vulnerabilities should be fixed as soon as possible to protect a system. However, patching hinders a project maintainability [105] because it adds complexity to a codebase, and mitigation can reduce the performances of critical applications. Moreover, developers' time is a scarce resource and the time spent on fixing issues is not allocated elsewhere. Before jumping on an immediate fallacy, a more thorough study on the security vulnerability **and** fix impact should be conducted. Sometimes, the best option is probably not to fix the vulnerability and accept the risk.

# Glossary

**Ξ – FMP** Firmware **Patch** Matching Problem.

***0-day*** A vulnerability publicly announced for which there exist no patch.

***1-day*** A vulnerability for which there exist a patch.

**AI** Abstract Interpretation.

**AOSP** the Android Open Source Project.

**API** Application Programming Interface.

**APK** Android Package Kit.

**AST** Abstract Syntax Tree.

**CDD** Android Compatibility Definition Document.

**CFG** Control Flow Graph.

**CG** Call Graph.

**CGC** Cyber Grand Challenge.

**CI** Continuous Integration.

**CLI** Command Line Interface.

**CNA** CVE Numbering Authority.

**COTS** Commercial-Off-The-Shelf.

**CPG** Code Property Graph.

**CRS** Cyber Reasoning System.

**CTS** Compatibility Test Suite.

**CVE** Common Vulnerabilities and Exposures.

**CVSS** Common Vulnerability Scoring System.

**CWE** Common Weakness Enumeration.

**DARPA** Defense Advanced Research Project Agency.

**DAST** Dynamic Application Security Testing Tools.

**DFG** Data Flow Graph.

**EOP** Elevation of Privilege.

**FMP** Firmware Matching Problem.

**GMS** Google Mobile Services.

**HAL** Hardware Abstraction Layer.

**HIDS** Host-based Intrusion Detection System.

**IDS** Intrusion Detection Software.

**IR** Intermediate Representation.

**NIST** National Institute of Standards and Technology.

**NLP** Natural Language Processing.

**NSA** National Security Agency.

**NVD** National Vulnerability Database.

**OEM** Original Equipment Manufacturer.

**OS** Operating System.

**PDG** Program Dependency Graph.

**PoV** Proof of Vulnerability.

**SAMATE** Software Assurance Metrics And Tool Evaluation.

**SAST** Static Application Security Testing Tools.

**SBOM** Software Bill of Materials.

**SDK** Software Development Kit.

**SIF** Smooth Inverse Frequency.

**SPL** Security Patch Level.

**SSDF** Secure Software Development Framework.

**TUI** Text User Interface.

**UDG** Unified Dependency Graph.

# Bibliography


[1] Ali Abbasi et al. “Challenges in Designing Exploit Mitigations for Deeply Embedded Systems”. In: *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. June 2019, pp. 31–46. DOI: 10.1109/EuroSP.2019.00013.

[2] Bram Adams et al. “MAKAO”. In: *2007 IEEE International Conference on Software Maintenance*. Oct. 2007, pp. 517–518. DOI: 10.1109/ICSM.2007.4362678.

[3] Junaid Akram and Luo Ping. “How to Build a Vulnerability Benchmark to Overcome Cyber Security Attacks”. In: *IET Information Security* 14.1 (Jan. 2020), pp. 60–71. ISSN: 1751-8709. DOI: 10.1049/iet-ifs.2018.5647.

[4] Hakam W. Alomari and Matthew Stephan. “Clone Detection through srcClone: A Program Slicing Based Approach”. In: *Journal of Systems and Software* 184 (Feb. 2022), p. 111115. ISSN: 0164-1212. DOI: 10.1016/j.jss.2021.111115.

[5] Ghaida Alqarawi et al. “Internet-of-Things Security and Vulnerabilities: Case Study”. In: *Journal of Applied Security Research* 0.0 (Feb. 2022), pp. 1–17. ISSN: 1936-1610. DOI: 10.1080/19361610.2022.2031841.

[6] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. “A Simple but Tough-to-Beat Baseline for Sentence Embeddings”. In: (Apr. 2017). URL: https://oar.princeton.edu/handle/88435/pr1rk2k (visited on 05/10/2022).

[7] Thanassis Avgerinos et al. “Automatic Exploit Generation”. In: *Communications of the ACM* 57.2 (Feb. 2014), pp. 74–84. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2560217.2560219.

[8] Michael Backes, Sven Bugiel, and Erik Derr. “Reliable Third-Party Library Detection in Android and Its Security Applications”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. New York, NY, USA: ACM, Oct. 2016, pp. 356–367. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978333.

[9] Bas van Schaik and Kevin Backhouse. “FPs Are Cheap. Show Me the CVEs!” In: *BlackHat EU* (Dec. 2020). URL: https://www.blackhat.com/eu-20/briefings/schedule/#fps-are-cheap-show-me-the-cves-21345.

[10] Guru Prasad Bhandari, Amara Naseer, and Leon Moonen. "CVEfixes: Automated Collection of Vulnerabilities and Their Fixes from Open-Source Software". In: *arXiv:2107.08760 [cs]* (July 2021). DOI: 10.1145/3475960.3475985. arXiv: 2107.08760 [cs].

[11] Biondi, Philippe et al. "BinCAT: Purrfecting Binary Static Analysis". In: *Symposium Sur La Sécurité Des Technologies de l'Information et Des Communications*. 2017. Rennes, France, June 2017. URL: https://www.sstic.org/2017/presentation/bincat_purrfecting_binary_static_analysis/.

[12] Matt Bishop and David Bailey. *A Critical Analysis of Vulnerability Taxonomies:* tech. rep. Fort Belvoir, VA: Defense Technical Information Center, Sept. 1996. DOI: 10.21236/ADA453251.

[13] Paul E. Black. "A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs". In: *Journal of Research of the National Institute of Standards and Technology* 123 (Apr. 2018), p. 123005. ISSN: 2165-7254. DOI: 10.6028/jres.123.005.

[14] Frederick Boland and Paul Black. "The Juliet 1.1 C/C++ and Java Test Suite". In: 45 (Oct. 2012). DOI: 10.1109/MC.2012.345.

[15] Guillaume Bonfante, Matthieu Kaczmarek, and Jean-Yves Marion. "Control Flow Graphs as Malware Signatures". In: (May 2007), p. 6.

[16] David Brumley et al. "Automatic Patch-Based Exploit Generation Is Possible: Techniques and Implications". In: *2008 IEEE Symposium on Security and Privacy (Sp 2008)*. Oakland, CA, USA: IEEE, May 2008, pp. 143–157. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.17.

[17] David Brumley et al. "BAP: A Binary Analysis Platform". In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, July 2011, pp. 463–469. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_37.

[18] Chad Pinson, Jon Rajewski, and Stephanie Snyder. *The Ransomware Epidemic*. Oct. 2020. URL: https://www.aon.com/cyber-solutions/thinking/client-alert-the-ransomware-epidemic/ (visited on 05/19/2022).

[19] Challande. *QSig: Semantic Patch Signatures for Filesystems*. Quarkslab. Aug. 2022. URL: https://github.com/quarkslab/qsig.

[20] Alexis Challande, Guenael Renault, and Robin David. *Exploitation Du Graphe de Dépendance d'AOSP à Des Fins de Sécurité*. June 2021. URL: https://github.com/quarkslab/bgraph (visited on 05/24/2022).

[21] Qiuyuan Chen, Han Hu, and Zhaoyi Liu. "Code Summarization with Abstract Syntax Tree". In: *Neural Information Processing*. Ed. by Tom Gedeon, Kok Wai Wong, and Minho Lee. Vol. 1143. Cham: Springer International Publishing, Dec. 2019, pp. 652–660. ISBN: 978-3-030-36801-2. DOI: 10.1007/978-3-030-36802-9_69.

[22] Chen Zhaojun. *CVE-2021-44228*. Dec. 2022. URL: https://cve.circl.lu/cve/CVE-2021-44228 (visited on 03/17/2022).

[23] Christian Blichmann. *BinExport*. Google. Nov. 2021. URL: https://github.com/google/binexport (visited on 06/03/2022).

[24] Andrei Costin, Apostolis Zarras, and Aurélien 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. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 437–448. ISBN: 978-1-4503-4233-9. DOI: 10.1145/2897845.2897900.

[25] Patrick Cousot. "Abstract Semantic Dependency". In: *International Static Analysis Symposium*. Springer, Oct. 2019, pp. 389–410. DOI: 10.1007/978-3-030-32304-2_19.

[26] Patrick Cousot and Radhia Cousot. "Abstract Interpretation and Application to Logic Programs". In: *The Journal of Logic Programming* 13.2-3 (July 1992), pp. 103–179. DOI: 10.1016/0743-1066(92)90030-7.

[27] Baojiang Cui et al. "Code Comparison System Based on Abstract Syntax Tree". In: *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. Oct. 2010, pp. 668–673. DOI: 10.1109/ICBNMT.2010.5705174.

[28] Ron Cytron et al. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Transactions on Programming Languages and Systems* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/115372.115320.

[29] Jiarun Dai et al. "BScout: Direct Whole Patch Presence Test for Java Executables". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1147–1164. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/dai.

[30] DARPA. *Cyber Grand Challenge*. Oct. 2017. URL: https://www.darpa.mil/about-us/timeline/cyber-grand-challenge (visited on 09/19/2021).

[31] Yaniv David, Nimrod Partush, and Eran Yahav. "Statistical Similarity of Binaries". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2016*. Santa Barbara, CA, USA: ACM Press, June 2016, pp. 266–280. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908126.

[32] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, Apr. 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[33] Li Deng. "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]". In: *IEEE Signal Processing Magazine* 29.6 (Nov. 2012), pp. 141–142. ISSN: 1558-0792. DOI: 10.1109/MSP.2012.2211477.

[34] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 110–121. ISBN: 978-1-5090-0824-7. DOI: 10.1109/SP.2016.15.

[35] Ruian Duan et al. "Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. Dallas, Texas, USA: ACM Press, Nov. 2017, pp. 2169–2185. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134048.

[36] Yue Duan et al. "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing". In: *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, Feb. 2020. ISBN: 978-1-891562-61-7. DOI: 10.14722/ndss.2020.24311.

[37] Alexandre Dulaunoy and Pieter-Jan Moreels. *Cve-Search - a Free Software to Collect, Search and Analyse Common Vulnerabilities and Exposures in Software*. CIRCL. Sept. 2015. URL: https://cve.circl.lu/ (visited on 10/05/2020).

[38] Elbaz, Clément. "Reacting to "N-Day" Vulnerabilities in Information Systems". PhD thesis. Rennes 1, Mar. 2021. URL: http://www.theses.fr/2021REN1S021.

[39] Evan, Martin. *Ninja, a Small Build System with a Focus on Speed*. Nov. 2020. URL: https://ninja-build.org/ (visited on 06/08/2022).

[40] Gang Fan et al. "Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, July 2020, pp. 463–474. ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397388.

[41] Sadegh Farhang et al. "Hey Google, What Exactly Do Your Security Patches Tell Us? A Large-Scale Empirical Study on Android Patched Vulnerabilities". In: *arXiv:1905.09352 [cs]* (May 2019). arXiv: 1905.09352 [cs]. URL: http://arxiv.org/abs/1905.09352 (visited on 09/24/2020).

[42] Qian Feng et al. "Extracting Conditional Formulas for Cross-Platform Bug Search". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. Abu Dhabi, United Arab Emirates: ACM Press, Apr. 2017, pp. 346–359. ISBN: 978-1-4503-4944-4. DOI: 10.1145/3052973.3052995.

[43] Jeanne Ferrante. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Transactions on Programming Languages and Systems* 9.3 (July 1987), p. 31. URL: https://dl.acm.org/doi/10.1145/24039.24041.

[44] Halvar Flake. "Structural Comparison of Executable Objects". In: *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6.7, 2004, Proceedings*. Ed. by Ulrich Flegel and Michael Meier. Vol. P-46. LNI. GI, July 2004, pp. 161–173. URL: https://dl.gi.de/20.500.12116/29199.

[45] Lori Flynn, William Snavely, and Zachary Kurtz. "Test Suites as a Source of Training Data for Static Analysis Alert Classifiers". In: *arXiv:2105.03523 [cs]* (May 2021). arXiv: 2105.03523 [cs]. URL: http://arxiv.org/abs/2105.03523 (visited on 04/22/2022).

[46] Stefan Frei et al. "Large-Scale Vulnerability Analysis". In: *Proceedings of the 2006 SIGCOMM Workshop on Large-scale Attack Defense - LSAD '06*. Pisa, Italy: ACM Press, Sept. 2006, pp. 131–138. ISBN: 978-1-59593-571-7. DOI: 10.1145/1162666.1162671.

[47] Cheng Fu et al. "Coda: An End-to-End Neural Program Decompiler". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach et al. Dec. 2019, pp. 3703–3714. URL: https://proceedings.neurips.cc/paper/2019/hash/093b60fd0557804c8ba0cbf1453da2Abstract.html.

[48] GitHub. *CodeQL*. GitHub. Feb. 2008. URL: https://codeql.github.com/ (visited on 06/03/2022).

[49] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. "Grammar-Based Whitebox Fuzzing". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. June 2008, pp. 206–215.

[50] Google. *Gitiles - Git at Google*. Google. Aug. 2012. URL: https://gerrit.googlesource.com/gitiles (visited on 09/22/2021).

[51] Google. *Protocol Buffers*. Google. Jan. 2022. URL: https://developers.google.com/protocol-buffers/.

[52] Google. *Soong*. Google. Aug. 2016. URL: https://android.googlesource.com/platform/build/soong/+/refs/heads/master/README.md.

[53] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4.3 (Nov. 2020), pp. 1–29. ISSN: 2476-1249. DOI: 10.1145/3428334.

[54] Hernandez, Paul. *Executive Order 14028, Improving the Nation's Cybersecurity*. Text. Apr. 2021. URL: https://www.nist.gov/itl/executive-order-improving-nations-cybersecurity (visited on 03/02/2022).

[55] Hex-Rays. *IDA Pro - Interactive Disassembler*. Hex-Rays. May 2022. URL: www.hex-rays.com/idapro/ (visited on 08/01/2021).

[56] He Huang, Amr M. Youssef, and Mourad Debbabi. "BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Abu Dhabi United Arab Emirates: ACM, Apr. 2017, pp. 155–166. ISBN: 978-1-4503-4944-4. DOI: 10.1145/3052973.3052974.

[57] Shih-Kun Huang et al. "Software Crash Analysis for Automatic Exploit Generation on Binary Programs". In: *IEEE Transactions on Reliability* 63.1 (Mar. 2014), pp. 270–289. ISSN: 0018-9529, 1558-1721. DOI: 10.1109/TR.2014.2299198.

[58] Iliyan Malchev. *Here Comes Treble: A Modular Base for Android*. Blog. May 2017. URL: https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html (visited on 06/28/2021).

[59] Hajin Jang et al. "QuickBCC: Quick and Scalable Binary Vulnerable Code Clone Detection". In: *ICT Systems Security and Privacy Protection*. Ed. by Audun Jøsang, Lynn Futcher, and Janne Hagen. Vol. 625. Cham: Springer International Publishing, June 2021, pp. 66–82. ISBN: 978-3-030-78119-4. DOI: 10.1007/978-3-030-78120-0_5.

[60] Zheyue Jiang et al. "PDiff: Semantic-based Patch Presence Testing for Downstream Kernels". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. Virtual Event USA: ACM, Oct. 2020, pp. 1149–1163. ISBN: 978-1-4503-7089-9. DOI: 10.1145/3372297.3417240.

[61] Jonathan Berr. *"WannaCry" Ransomware Attack Losses Could Reach $4 Billion*. May 2017. URL: https://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/ (visited on 06/09/2022).

[62] Joxean Koret. *Diaphora, the Most Advanced Free and Open Source Program Diffing Tool*. Aug. 2021. URL: https://github.com/joxeankoret/diaphora (visited on 08/01/2021).

[63] Jurczyk Mateusz. *2002 - Samsung Android Multiple Interactionless RCEs and Other Remote Access Issues in Qmage Image Codec Built into Skia - Project-Zero*. Jan. 2020. URL: https://bugs.chromium.org/p/project-zero/issues/detail?id=2002 (visited on 03/03/2022).

[64] Deborah S. Katz, Jason Ruchti, and Eric Schulte. "Using Recurrent Neural Networks for Decompilation". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Mar. 2018, pp. 346–356. DOI: 10.1109/SANER.2018.8330222.

[65] Kevin Collier. *Baby Died Because of Ransomware Attack on Hospital, Suit Says*. Sept. 2021. URL: https://www.nbcnews.com/news/baby-died-due-ransomware-attack-hospital-suit-claims-rcna2465 (visited on 05/11/2022).

[66] Seulbae Kim et al. "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery". In: *2017 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, May 2017, pp. 595–614. ISBN: 978-1-5090-5533-3. DOI: 10.1109/SP.2017.62.

[67] James C King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[68] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P'19)*. May 2019. DOI: 10.1109/SP.2019.00002.

[69] Jennifer Korn. *The Log4j Security Flaw Could Impact the Entire Internet. Here's What You Should Know*. Dec. 2021. URL: https://www.cnn.com/2021/12/15/tech/log4j-vulnerability/index.html (visited on 05/31/2022).

[70] Zhe Lang et al. “PMatch: Semantic-based Patch Detection for Binary Programs”. In: *2021 IEEE International Performance, Computing, and Communications Conference (IPCCC)*. Austin, TX, USA: IEEE, Oct. 2021, pp. 1–10. ISBN: 978-1-66544-331-9. DOI: 10.1109/IPCCC51483.2021.9679443.

[71] Frank Li and Vern Paxson. “A Large-Scale Empirical Study of Security Patches”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, Oct. 2017, pp. 2201–2215. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134072.

[72] Hongyi Li et al. “P1OVD: Patch-Based 1-Day Out-of-Bounds Vulnerabilities Detection Tool for Downstream Binaries”. In: *Electronics* 11.2 (Jan. 2022), p. 260. ISSN: 2079-9292. DOI: 10.3390/electronics11020260.

[73] Yuekang Li et al. “Steelix: Program-State Based Binary Fuzzing”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 627–637. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106295.

[74] Zhen Li et al. “SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities”. In: *IEEE Transactions on Dependable and Secure Computing* (Jan. 2021), pp. 1–1. ISSN: 1941-0018. DOI: 10.1109/TDSC.2021.3051525.

[75] Zhen Li et al. “VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis”. In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. Los Angeles California USA: ACM, Dec. 2016, pp. 201–213. ISBN: 978-1-4503-4771-6. DOI: 10.1145/2991079.2991102.

[76] Nandor Licker and Andrew Rice. “Detecting Incorrect Build Rules”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 1234–1244. DOI: 10.1109/ICSE.2019.00125.

[77] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: (Jan. 2018). DOI: 10.1145/3357033.

[78] Bingchang Liu et al. “$\alpha$Diff: Cross-Version Binary Code Similarity Detection with DNN”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering - ASE 2018*. Montpellier, France: ACM Press, Sept. 2018, pp. 667–678. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238199.

[79] Ming Liu et al. "Host-Based Intrusion Detection System with System Calls: Review and Future Trends". In: *ACM Computing Surveys* 51.5 (Nov. 2018), 98:1–98:36. ISSN: 0360-0300. DOI: 10.1145/3214304.

[80] Andrea Marcelli et al. "How Machine Learning Is Solving the Binary Function Similarity Problem". In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, p. 18. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli.

[81] MITRE Corporation. *FAQs | CVE*. May 2022. URL: https://www.cve.org/ResourcesSupport/FAQs (visited on 05/02/2022).

[82] Marius Muench. "Dynamic Binary Firmware Analysis: Challenges & Solutions". PhD thesis. EURECOM, Sept. 2019. URL: https://tel.archives-ouvertes.fr/tel-03143960.

[83] Marius Muench et al. "Avatar $^2$ : A Multi-Target Orchestration Platform". In: *Proceedings 2018 Workshop on Binary Analysis Research*. San Diego, CA: Internet Society, Feb. 2018. ISBN: 978-1-891562-50-1. DOI: 10.14722/bar.2018.23017.

[84] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. "Understanding Source Code Evolution Using Abstract Syntax Tree Matching". In: (May 2017), p. 5. DOI: 10.1145/1082983.1083143.

[85] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. New York, NY, USA: Association for Computing Machinery, June 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746.

[86] NSA. *Ghidra Software Reverse Engineering Framework*. National Security Agency. May 2022. URL: https://github.com/NationalSecurityAgency/ghidra (visited on 06/03/2022).

[87] NVD. *CVE-2015-3873*. Oct. 2015. URL: https://cve.circl.lu/cve/CVE-2015-3873 (visited on 04/19/2022).

[88] NVD. *CVE-2016-2464*. Apr. 2016. URL: https://cve.circl.lu/cve/CVE-2016-2464 (visited on 09/18/2020).

[89] NVD. *CVE-2016-3839*. Aug. 2016. URL: https://cve.circl.lu/cve/CVE-2016-3839 (visited on 04/21/2022).

[90] NVD. *CVE-2017-0144*. Mar. 2017. URL: https://cve.circl.lu/cve/CVE-2017-0144 (visited on 08/01/2021).

[91] NVD. *CVE-2018-19841*. Dec. 2018. URL: https://cve.circl.lu/cve/CVE-2018-19841 (visited on 09/18/2020).

[92] NVD. *CVE-2018-9506*. Oct. 2018. URL: https://cve.circl.lu/cve/CVE-2018-9506 (visited on 08/01/2021).

[93] NVD. *CVE-2020-0471*. Jan. 2021. URL: https://cve.circl.lu/cve/CVE-2020-0471 (visited on 03/17/2022).

[94] O'Dea. *Global Market Share Held by Smartphone Operating Systems from 2009 to 2017*. Feb. 2020. URL: https://www.statista.com/statistics/263453/global-market-share-held-by-smartphone-operating-systems/ (visited on 06/28/2021).

[95] O'Dea. *Number of Smartphone Subscriptions Worldwide from 2016 to 2026*. June 2021. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 06/28/2021).

[96] Yang-jia Ou et al. "The Design and Implementation of Host-Based Intrusion Detection System". In: *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*. Apr. 2010, pp. 595–598. DOI: 10.1109/IITSI.2010.127.

[97] Chengbin Pang et al. "SoK: All You Ever Wanted to Know About X86/X64 Binary Disassembly But Were Afraid to Ask". In: *2021 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2021, pp. 833–851. ISBN: 978-1-72818-934-5. DOI: 10.1109/SP40001.2021.00012.

[98] Jiaqi Peng et al. "1dVul: Discovering 1-Day Vulnerabilities through Binary Patches". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Portland, OR, USA: IEEE, June 2019, pp. 605–616. ISBN: 978-1-72810-057-9. DOI: 10.1109/DSN.2019.00066.

[99] PNF Software. *JEB Decompiler by PNF Software*. PNF Software. May 2022. URL: https://www.pnfsoftware.com/ (visited on 06/03/2022).

[100] Serena Elisa Ponta et al. "A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, May 2019, pp. 383–387. ISBN: 978-1-72813-412-3. DOI: 10.1109/MSR.2019.00064.

[101] Positive Technologies. *Positive Technologies: Unfixable Vulnerability in Intel Chipsets Threatens Users and Content Rightsholders*. Mar. 2020. URL: https://www.ptsecurity.com/ww-en/about/news/unfixable-vulnerability-in-intel-chipsets-threatens-users-and-content-rightsholders/ (visited on 06/07/2022).

[102] Andrea Possemato et al. "Trust, But Verify: A Longitudinal Analysis Of Android OEM Compliance and Customization". In: *2021 IEEE Symposium on Security and Privacy (SP)*. May 2021, pp. 87–102. DOI: 10.1109/SP40001.2021.00074.

[103] Quarkslab. *Binbloom*. Quarkslab. June 2022. URL: https://github.com/quarkslab/binbloom (visited on 06/29/2022).

[104] Ransbotham, Mitra, and Ramsey. "Are Markets for Vulnerabilities Effective?" In: *MIS Quarterly* 36.1 (Mar. 2012), p. 43. ISSN: 02767783. DOI: 10.2307/41410405.

[105] Sofia Reis, Rui Abreu, and Luis Cruz. *Fixing Vulnerabilities Potentially Hinders Maintainability*. Sept. 2021. arXiv: 2106.03271 [cs]. URL: http://arxiv.org/abs/2106.03271 (visited on 06/03/2022).

[106] returntocorp. *Semgrep*. returntocorp. June 2022. URL: https://github.com/returntocorp/semgrep (visited on 06/29/2022).

[107] Reza Jelveh. *Universal Ctags*. Universal Ctags. Sept. 2021. URL: https://github.com/universal-ctags/ctags (visited on 09/17/2021).

[108] B.G. Ryder. "Constructing the Call Graph of a Program". In: *IEEE Transactions on Software Engineering* SE-5.3 (May 1979), pp. 216–226. ISSN: 1939-3520. DOI: 10.1109/TSE.1979.234183.

[109] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3.3 (July 1959), pp. 210–229. DOI: 10.1147/rd.33.0210.

[110] David E. Sanger and Kate Conger. "Russia Was Behind Cyberattack in Run-Up to Ukraine War, Investigation Finds". In: *The New York Times* (May 2022). ISSN: 0362-4331. URL: https://www.nytimes.com/2022/05/10/us/politics/russia-cyberattack-ukraine-war.html (visited on 05/11/2022).

[111] Sanjay Rawat et al. *VUzzer: Application-aware Evolutionary Fuzzing NDSS Symposium*. Feb. 2017. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/ (visited on 05/03/2022).

[112] Security Research Labs. *The Android Ecosystem Contains a Hidden Patch Gap*. Blog. Apr. 2018. URL: https://www.srlabs.de/bites/android-patch-gap (visited on 01/28/2022).

[113] Hyunmin Seo et al. "Programmers' Build Errors: A Case Study (at Google)". In: *Proceedings of the 36th International Conference on Software Engineering*. Hyderabad India: ACM, May 2014, pp. 724–734. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568255.

[114] Thodoris Sotiropoulos et al. "A Model for Detecting Faults in Build Specifications". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3428212.

[115] Statcounter. *Mobile Operating System Market Share Worldwide*. Apr. 2022. URL: https://gs.statcounter.com/os-market-share/mobile/worldwide/2020 (visited on 05/17/2022).

[116] Pengfei Sun et al. "Hybrid Firmware Analysis for Known Mobile and IoT Security Vulnerabilities". In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Valencia, Spain: IEEE, June 2020, pp. 373–384. ISBN: 978-1-72815-809-9. DOI: 10.1109/DSN48063.2020.00053.

[117] Xin Sun et al. “Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph”. In: *ICT Systems Security and Privacy Protection*. Ed. by Nora Cuppens-Boulahia et al. Vol. 428. Berlin, Heidelberg: Springer Berlin Heidelberg, June 2014, pp. 142–155. ISBN: 978-3-642-55414-8. DOI: 10.1007/978-3-642-55415-5_12.

[118] Ahmed Tamrawi et al. “SYMake: A Build Code Analysis and Refactoring Tool for Makefiles”. In: *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. Sept. 2012, pp. 366–369. DOI: 10.1145/2351676.2351749.

[119] Trail of Bits. *McSema*. Trail of Bits. Apr. 2021. URL: https://github.com/lifting-bits/mcsema (visited on 06/03/2022).

[120] U.S. National Institute of Standards and Technology. *National Vulnerability Database*. May 2022. URL: https://nvd.nist.gov/vuln/ (visited on 05/19/2022).

[121] Sebastian Vasile, David Oswald, and Tom Chothia. “Breaking All the ThingsA Systematic Survey of Firmware Extraction Techniques for IoT Devices”. In: *Smart Card Research and Advanced Applications*. Ed. by Begül Bilgin and Jean-Bernard Fischer. Lecture Notes in Computer Science. Cham: Springer International Publishing, Mar. 2019, pp. 171–185. ISBN: 978-3-030-15462-2. DOI: 10.1007/978-3-030-15462-2_12.

[122] Vector 35. *Binary Ninja*. Vector 35. May 2022. URL: https://binary.ninja.

[123] John Viega and Hugh Thompson. “The State of Embedded-Device Security (Spoiler Alert: It’s Bad)”. In: *IEEE Security Privacy* 10.5 (Sept. 2012), pp. 68–70. ISSN: 1558-4046. DOI: 10.1109/MSP.2012.134.

[124] “Vulnerability (Computing)”. In: *Wikipedia* (Jan. 2022). URL: https://en.wikipedia.org/w/index.php?title=Vulnerability_(computing)&oldid=1064173061 (visited on 03/02/2022).

[125] Andreas Wagner and Johannes Sametinger. “Using the Juliet Test Suite to Compare Static Security Scanners:” in: *Proceedings of the 11th International Conference on Security and Cryptography*. Vienna, Austria: SCITEPRESS - Science and and Technology Publications, Aug. 2014, pp. 244–252. ISBN: 978-989-758-045-1. DOI: 10.5220/0005032902440252.

[126] Fish Wang and Yan Shoshitaishvili. “Angr - The Next Generation of Binary Analysis”. In: *2017 IEEE Cybersecurity Development (SecDev)*. Sept. 2017, pp. 8–9. DOI: 10.1109/SecDev.2017.14.

[127] Jinghan Wang et al. “Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 1–15. ISBN: 978-1-939133-07-6. URL: https://www.usenix.org/conference/raid2019/presentation/wang.

[128] Xinda Wang et al. “Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS”. In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2019, pp. 485–492. DOI: 10.1109/DSN.2019.00056.

[129] Xinda Wang et al. “PatchDB: A Large-Scale Security Patch Dataset”. In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. June 2021, pp. 149–160. DOI: 10.1109/DSN48987.2021.00030.

[130] Yingchen Wang et al. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on X86”. In: *Proceedings of the USENIX Security Symposium (USENIX)*. Aug. 2022.

[131] Wikipedia. “Control-Flow Graph”. In: *Wikipedia* (Feb. 2022). URL: https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1072228345 (visited on 07/04/2022).

[132] Felix Wilhelm. *Weggli*. Google Project Zero. June 2022. URL: https://github.com/googleprojectzero/weggli (visited on 06/29/2022).

[133] Christopher Wright et al. “Challenges in Firmware Re-Hosting, Emulation, and Analysis”. In: *ACM Computing Surveys* 54.1 (Jan. 2022), pp. 1–36. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3423167.

[134] Yang Xiao et al. “MVP: Detecting Vulnerabilities Using Patch-Enhanced Vulnerability Signatures”. In: Aug. 2020, p. 18. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/xiao.

[135] Yang Xiao et al. “VIVA: Binary Level Vulnerability Identification via Partial Signature”. In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Honolulu, HI, USA: IEEE, Mar. 2021, pp. 213–224. ISBN: 978-1-72819-630-5. DOI: 10.1109/SANER50967.2021.00028.

[136] Luhang Xu et al. “Automatic Exploit Generation for Buffer Overflow Vulnerabilities”. In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. Lisbon: IEEE, July 2018, pp. 463–468. ISBN: 978-1-5386-7839-8. DOI: 10.1109/QRS-C.2018.00085.

[137] Xiaojun Xu et al. “Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17* (Oct. 2017), pp. 363–376. DOI: 10.1145/3133956.3134018.

[138] Yifei Xu et al. “BinXRay: Patch Based Vulnerability Matching for Binary Programs”. In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, July 2020, pp. 376–387. ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397361.

[139] Zhengzi Xu et al. "SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. Buenos Aires: IEEE, May 2017, pp. 462–472. ISBN: 978-1-5386-3868-2. DOI: 10.1109/ICSE.2017.49.

[140] Yinxing Xue et al. "Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation". In: *IEEE Transactions on Software Engineering* 45.11 (Nov. 2019), pp. 1125–1149. ISSN: 0098-5589, 1939-3520, 2326-3881. DOI: 10.1109/TSE.2018.2827379.

[141] Fabian Yamaguchi et al. "Modeling and Discovering Vulnerabilities with Code Property Graphs". In: *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2014, pp. 590–604. ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.44.

[142] Insu Yun et al. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

[143] Hang Zhang and Zhiyun Qian. "Precise and Accurate Patch Presence Test for Binaries". In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 887–902. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/zhang-hang.

[144] Zheng Zhang et al. "An Investigation of the Android Kernel Patch Ecosystem". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, p. 18. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng.

[145] Yaqin Zhou and Asankhaya Sharma. "Automated Identification of Security Issues from Commit Messages and Bug Reports". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. Paderborn, Germany: ACM Press, Sept. 2017, pp. 914–919. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3117771.

[146] Ruijin Zhu et al. "Determining Image Base of Firmware for ARM Devices by Matching Literal Pools". In: *Digital Investigation* 16 (Mar. 2016), pp. 19–28. ISSN: 1742-2876. DOI: 10.1016/j.diin.2016.01.002.

[147] Zynamics. *BinDiff*. Zynamics/Google. June 2021. URL: https://www.zynamics.com/bindiff.html.

Appendix A

# Quokka

In this appendix, we expose the motivations that led us to develop a binary exporter and present the tool itself. The tool is open-sourced and available on GitHub[1].

## A.1 Motivation

Analyzing binary programs often requires disassembling them: it is a core component of multiple works from malware analysis to code similarity measurement. For instance, in the work presented in Chapter 5, both **QSig**'s signature generation step and the matching step are using as inputs disassembled binaries. Security practitioners have developed numerous tools and frameworks over the years both open-source [126, 17, 119, 86] or commercial [55, 122, 99]

Correctly disassembling is challenging [97]. Indeed, it is insufficient for a disassembler to convert a sequence of bytes into meaningful assembly instructions. It also needs to recover references between code and data, functions boundaries, typical language structures (i.e. jumps or virtual tables) or CFG reconstruction. While studying a disassembler inner workings is out of this work scope, we highlight that disassemblers are complex software that combine both algorithms (i.e. linear sweep, recursive descent) and heuristics to perform their tasks.

However, disassemblers are inadequate to either perform heavy custom analyses on a disassembled binary and to analyze multiple binaries at the same time. Indeed, a disassembler instance running in the background may use few hundreds of megabytes in RAM effectively wasting resources if their functionalities are not used anymore. Moreover, their API may be complicated to use. If only the *output* of the disassembler is needed for further analysis, and it should be possible to extract this result to run *offline* queries using a binary export.

[1] https://github.com/quarkslab/quokka

| Disassembler | Exporter | Description |
|---|---|---|
| IDA | BinExport | Exporter from Zynamics |
| | Ghidra-IDA | Official Ghidra plugin to export a project from IDA to Ghidra |
| Ghidra | McSema | Exporter for McSema lifter |
| | BinExport | BinExport port for Ghidra |
| | Ghidra | Built-in exporter from Ghidra |

Table A.1: List of Different Binary Exporters

**Definition A.1.** *A **binary export** is defined as a file which stores data on the disassembled binary which is usable outside the disassembler.*

## A.2 Existing Binary Exporters Review

We survey in Table A.1 various existing exporters. The most common one in BinExport [23], a binary exporter created by Zynamics and used for `BinDiff` [147]. It supports two backends, both `IDA` and `Ghidra` and generates a binary file in a `Protobuf` format. It exports most information but lacks support for most data fields (i.e. type, size, content).

`Ghidra` provides an official exporter plugin for `IDA`. The plugin generates an XML file to extract some information from `IDA` and to import the project in `Ghidra` to continue the analysis. Because the objective is to reuse the result in another disassembler, the export does not contain any data on the instruction themselves.

Finally, McSema is an executable lifter: it translates native machine code to LLVM IR. The first step uses `IDA` to disassemble the target binary and generates a `Protobuf` with the information extracted from the disassembler. However, as the tool final objective is to generate LLVM bitcode, it uses a second tool to translate instructions. Thus, the first export does not contain information on the instruction themselves other than their address.

None of the existing exporters completely answer the following properties.

- Exhaustivity: It must export as much data from the disassembly process as possible.
- Efficiency: An exporter must be fast to be usable even if it is a one-step process.
- Compactness: It should be compact to save some disk space.

Thus, there exist a need to create a solution solving the three problems. However, it necessarily involves tradeoff: the most compact exporter would not export anything and the most exhaustive one would not be as efficient as another exporting fewer data.

## A.3 *Quokka*: A Fast and Exhaustive Binary Exporter

*Quokka* is an `IDA` Plugin developed to address the limitations observed in other binary exporters. It aims at being fast while thorough. We first list the exported features before briefly discussing its architecture.

### A.3.1 Exported Features

The features exported by *Quokka* listed in Table A.2 are compared with both BinExport and Ghidra builtin export. On a general point of view, *Quokka*'s export is more exhaustive than the two other tools: it exports every item exported by at least one of them.

### A.3.2 *Quokka*'s Architecture

*Quokka* is an `IDA` plugin composed of about 3,500 C++ LoC which targets `IDA`'s last two versions. Its inner component is a state machine which iterates over the binary address space to generate a `Protobuf` file [51].

To keep the export size minimal, we used the following principles through the plugin development:

- Data deduplication: no data should be stored twice. For instance, mnemonics are stored in a list and referenced by their index in this list.
- Integers should be stored as offsets. For instance, addresses are always referenced as offsets to the program base address.
- Most common values should be set at a non-writable value. `Protobuf`s never write *on the wire* some values (`0`, `false`).

## A.4 Evaluation

### A.4.1 Dataset

To compare *Quokka* with BinExport, we select typical binaries present on our systems at the time of the writing with properties listed below:

- We consider multiple architectures as the exporter should be architecture agnostic.
- We want to consider various binary file formats to test their support.

| | | BinExport | Ghidra XML | *Quokka* |
|---|---|---|---|---|
| Metadata | Name | ✓ | ✓ | ✓ |
| | Architecture | ✓ | ✓ | ✓ |
| | ISA | ✓ | ✓ | ✓ |
| | Compiler | ✓ | ✓ | ✓ |
| Layout | Segments | ✓ | ✓ | ✓ |
| | Code Layout | ≈ | ✓ | ✓ |
| Symbols | Name | ✓ | ✓ | ✓ |
| | Value | ✓ | ✓ | ✓ |
| | Type | ✗ | ✓ | ✓ |
| Data | Address | ✓ | ✓ | ✓ |
| | Type | ✗ | ✓ | ✓ |
| | Size | ✗ | ✓ | ✓ |
| | Name | ✗ | ✓ | ✓ |
| Graphs | Call Graph | ✓ | ✗ | ✓ |
| | Control Flow Graph | ✓ | ✗ | ✓ |
| Comments | Address | ✓ | ✓ | ✓ |
| | Type | ✓ | ✓ | ✓ |
| | Content | ✓ | ✓ | ✓ |
| Functions | Name | ✓ | ✓ | ✓ |
| | Type | ✓ | ✓ | ✓ |
| | # Arguments | ✗ | ✓ | ✓ |
| Instruction | Mnemonic | ✓ | ✗ | ✓ |
| | Operand | ✓ | ✗ | ✓ |
| | Operand Type | ✗ | ✗ | ✓ |
| | Bytes | ✓ | ✗ | ✓ |
| | Address | ✓ | ✗ | ✓ |
| | Expressions | ✓ | ✗ | ✓ |
| | Xref (code, data) | ✓ | ✗ | ✓ |
| Basic Block | Address | ✓ | ✗ | ✓ |
| | Instructions | ✓ | ✗ | ✓ |
| | Type | ✗ | ✗ | ✓ |
| | Content | ✓ | ✓ | ✓ |
| Strings | Address | ✓ | ✓ | ✓ |
| | Content | ✓ | ✓ | ✓ |
| Data Structures | Structures | ✗ | ✓ | ✓ |
| | Enumeration | ✗ | ✓ | ✓ |

Table A.2: Comparison of Exporter Features

| Binary Name | Architecture | Format | Binary size |
|---|---|---|---|
| MachO-OSX-x86-ls | x86 | MachO | 34.86 kB |
| pe-Windows-x86-cmd | x86 | PE | 294.50 kB |
| elf-Linux-x86-bash | x86 | ELF | 792.14 kB |
| elf-Linux-lib-x86.so | x86 | ELF | 1.08 MB |
| delta_generator | x86 | ELF | 16.49 MB |
| wpa_supplicant | x86 | ELF | 21.64 MB |
| MachO-OSX-x64-ls | x86_64 | MachO | 38.66 kB |
| pe-Windows-x64-cmd | x86_64 | PE | 337.00 kB |
| x64_delta_generator | x86_64 | ELF | 15.28 kB |
| elf-Linux-x64-bash | x86_64 | ELF | 904.82 kB |
| elf-Linux-lib-x64.so | x86_64 | ELF | 1.09 MB |
| ctags | x86_64 | ELF | 4.59 MB |
| ts3server | x86_64 | ELF | 7.73 MB |
| mdbook | x86_64 | ELF | 10.67 MB |
| llvm-opt | x86_64 | ELF | 33.83 MB |
| clang-check | x86_64 | ELF | 46.83 MB |
| crackmips | MIPS-32 | ELF | 25.54 kB |
| busybox-mips | MIPS-32 | ELF | 352.48 kB |
| elf-Linux-Mips4-bash | MIPS-32 | ELF | 882.38 kB |
| HelloWorld-MachO-2 | armv7 | MachO | 89.64 kB |
| HelloWorld-MachO | armv7, armv8 | MachO | 299.06 kB |
| elf-Linux-ARMv7-ls | armv7 | ELF | 88.68 kB |
| elf-Linux-ARM64-bash | armv8 | ELF | 827.54 kB |
| busybox-powerpc | PPC-32 | ELF | 1.10 MB |
| dex38.dex | - | DEX | 11.48 kB |
| classes.dex | - | DEX | 3.53 MB |

Table A.3: Datasets Binaries

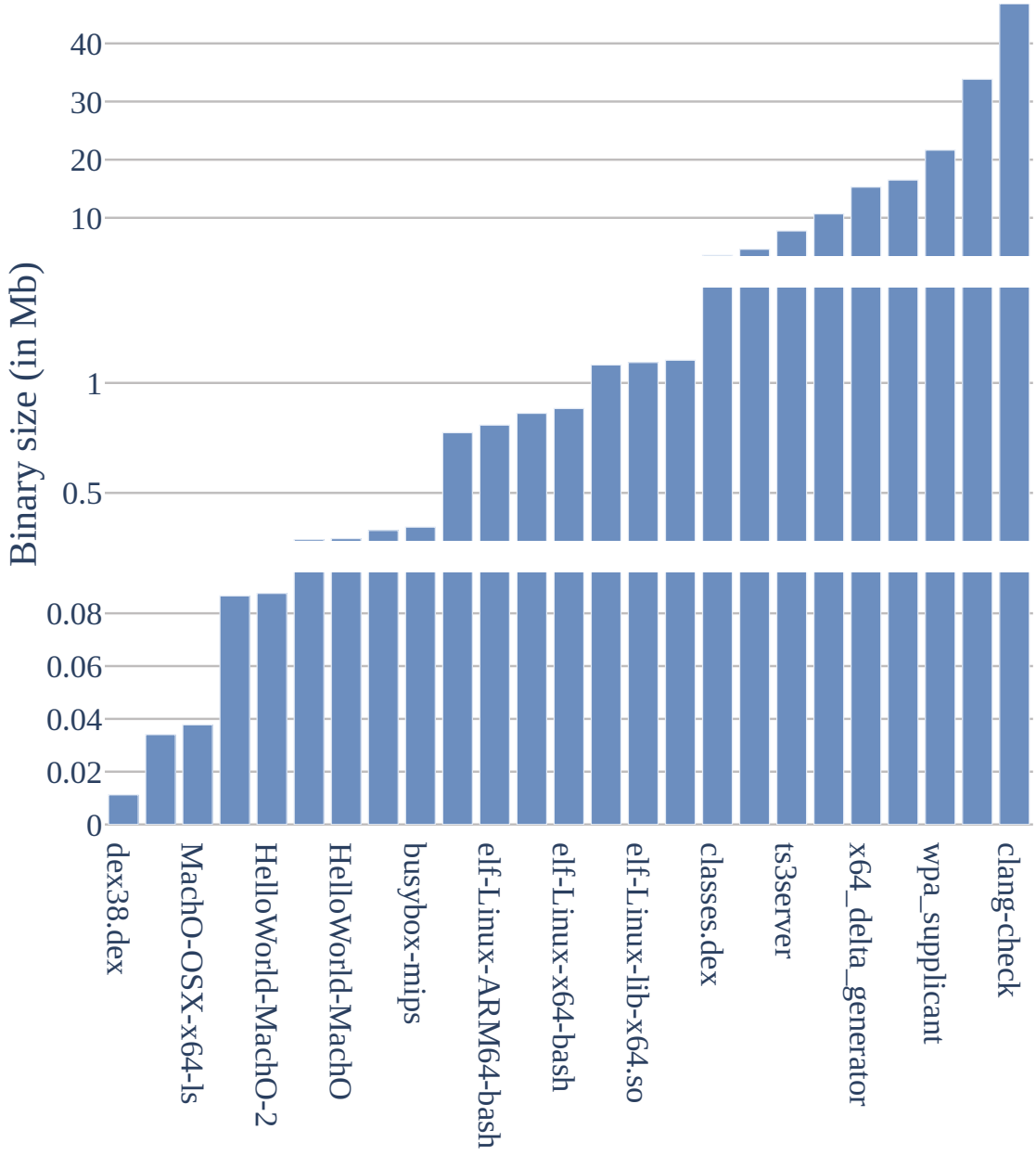


Figure A.1: Samples Sizes

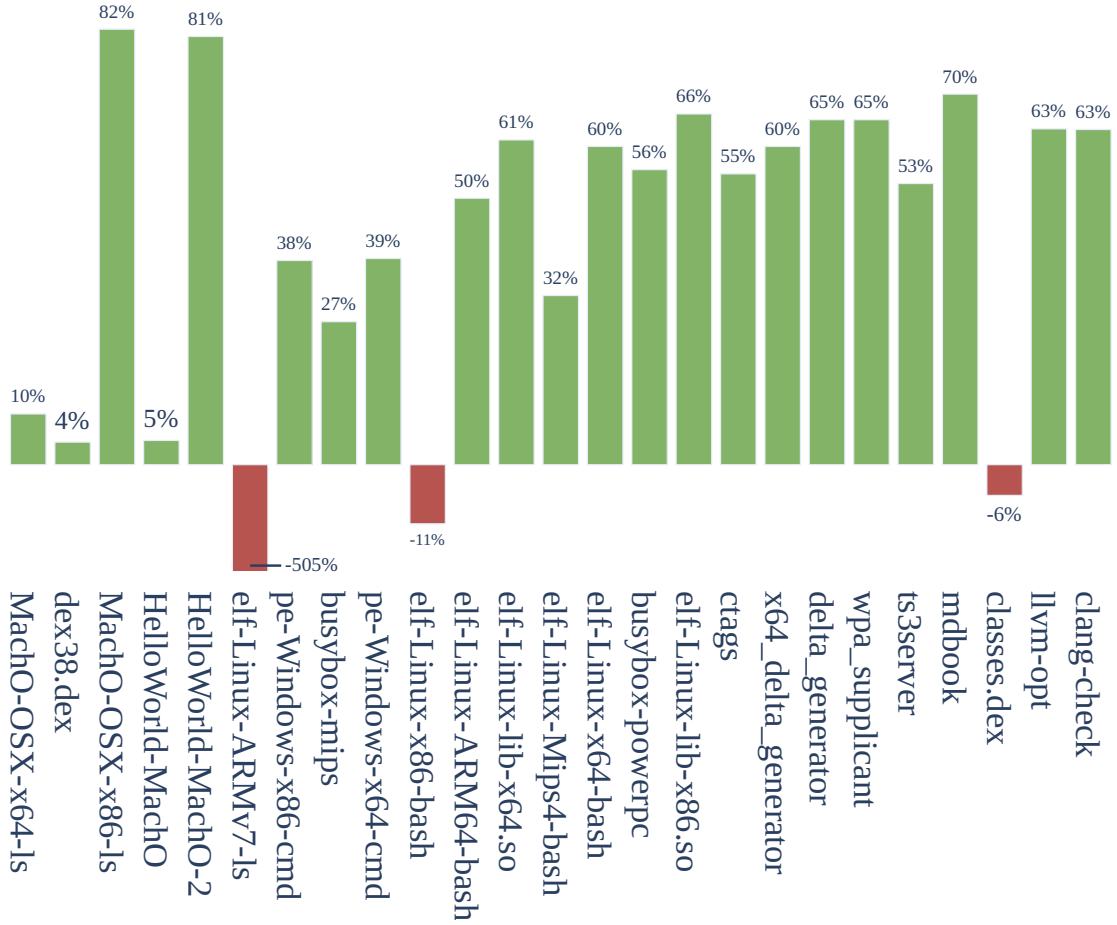


Figure A.2: Duration: *Quokka* vs BinExport

As the export time and result size is a factor of the initial binary size, we select binaries from 25 kB and up to 46 MB. In Figure A.1, we illustrate the binary sizes of the samples we use in the next experiments.

### A.4.2 Efficiency

We first assess *Quokka*'s efficiency by looking at the difference on the export duration between BinExport and *Quokka* for our samples. While exporting more features, *Quokka*'s optimizations are successful as the exporter is faster: 63% improvement for the largest binary and a median 54% improvement for the dataset. Figure A.2 displays the improvement percentage between *Quokka* and BinExport.

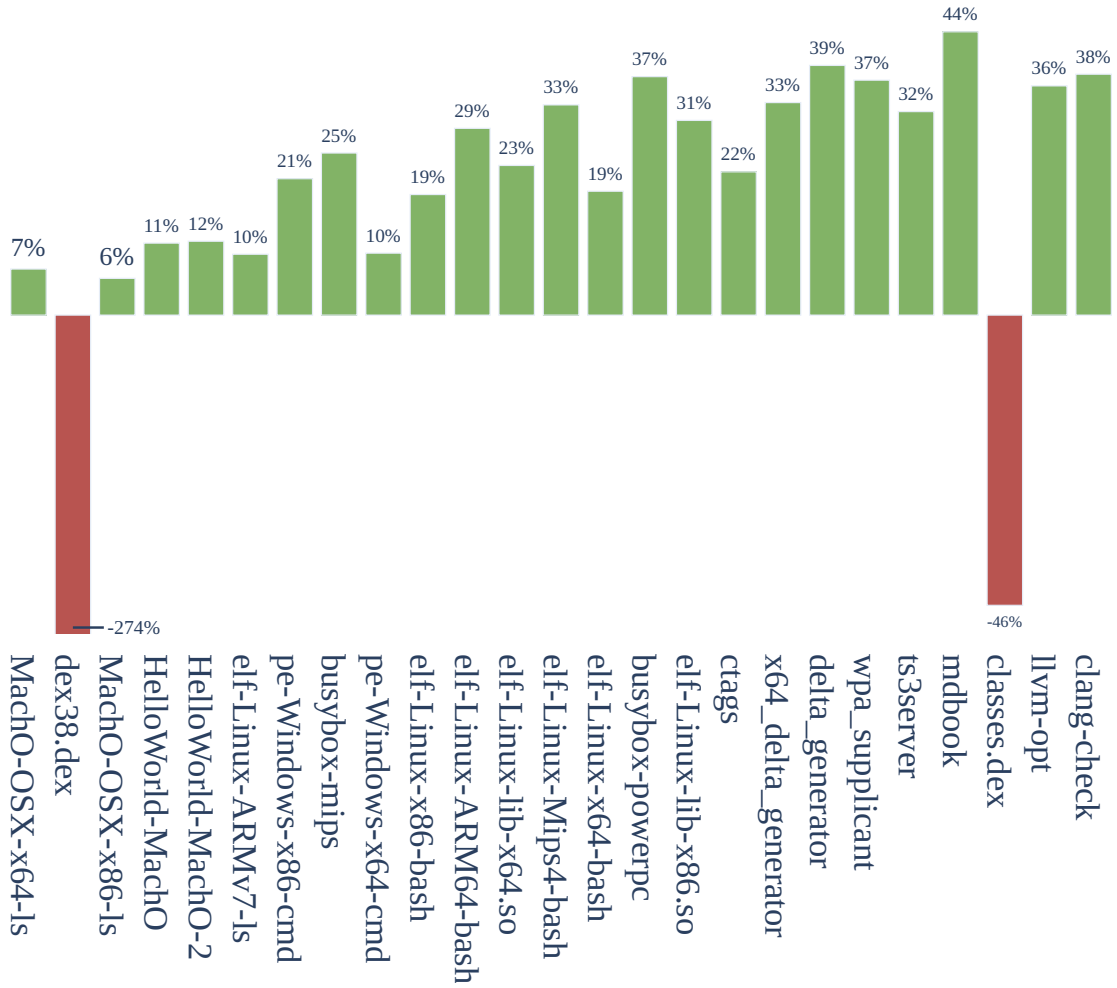


Figure A.3: Size: *Quokka* vs BinExport

The extensive comparison of the exported files for `elf-Linux-ARMv7-ls` did not yield to any conclusive explanation for the duration difference.

### A.4.3 Compactness

We also assess *Quokka*'s export compactness. Recall that *Quokka* exports strictly more data than BinExport. However, thanks to the optimization on size presented in the previous section, *Quokka* manages to be more compact for most binaries. Indeed, the median improvement is 22%.

*Quokka* export files are smaller for each sample except the two DEX files. This is explained by two reasons. First, *Quokka* exports the layout (i.e. if a region is used for code or data) which change often for DEX, requiring multiple objects. Second, *Quokka*

also exports data structure wich takes up to 4.8 MB in `classes.dex`, explaining the 5 MB difference between the two formats.

## A.5 Conclusion

*Quokka* is a fast and complete binary exporter for `IDA` Pro. It timely generates compact binary file containing most `IDA` intelligence and allows reusing the result in an offline setting.

*Quokka* is used extensively in Chapter 5. Indeed, the signature is generated on the exported disassembly. We load the two exports at the same time to iterate on both programs at the same time and uncover their differences. The matching is also performed on the exported disassembly. This allows writing complex queries on the call graph or the function's content seamlessly.

While thorough, *Quokka* still lacks the export of some features listed below.

- Type information: `IDA` allows users to define types and to apply type information onto the disassembly. This information helps the reverser understanding the data flow inside the binary and can be used for other tools to perform analyses (pointer analysis, liveness analysis). Thus, it could be valuable to expose them outside the disassembler.
- Decompilation output: `IDA`'s SDK offers an API to manipulate the decompilation output, i.e. a C code generated from the disassembly. Exporting the decompiled code would broaden the usage possibilities of *Quokka*.

As `IDA` already offer API to manipulate and query such data, exporting these elements only require further engineering work. However, working with the disassembler API is time-consuming.

At the moment, *Quokka* is an `IDA` plugin. Another improvement for the tool would be to also accept other backends (such as `Ghidra` or `Binary Ninja`). Thus, it could act as an IR where an analyst query workflow could be written once, and the backend changed depending on the disassembly selected.

Appendix B

# Using **QSig**

We introduced **QSig** in Chapter 3 and developed its evaluation in Chapter 5. However, we did neither describe the tool internals nor how to use it. We address this shortcoming in this appendix.

## B.1 Usage

**QSig** is usable both as a command-line tool and as a library. In this section, we detail both usages.

### B.1.1 As a Command Line Tool

Using **QSig** with its Command Line Interface (CLI) interface is best when performing simple queries on a supported Android device or on a single binary. Listing B.1 displays its help message. The tool offers five subcommands described below.

- The `detect` command applies a single signature onto a unique file. This command is useful to run tests or to quickly check if a target binary is patched against a vulnerability. Listing B.2 displays a command usage example.
- The `detector` command is **QSig** detector main command. It applies a list of signatures on a complete file system.
- The next two commands `generate` and `generate-multiple` are self-explanatory: they generate patch signatures from vulnerabilities. While the first one only processes a single vulnerability, the second one generates a signature for every vulnerability in a user-specified folder.
- Finally, the last command, `info` displays information on previously generated signatures. This allows to quickly inspect a signature set. `info` output is displayed in Listing B.3.

**Listing B.1 QSig** Command Line Interface

```
$ python -m qsig -help
Usage: qsig [OPTIONS] COMMAND [ARGS]...

  QSIG CLI - Use to generate signature or match firmwares images

Options:
  -d, -debug           Activate debug output  [default: False]
  -q, -quiet           Silence output  [default: False]
  -b, -bench           Activate benchmark output  [default: False]
  -install-completion  Install completion for the current shell.
  -show-completion     Show completion for the current shell, to copy it or
                        customize the installation.

  -help                Show this message and exit.

Commands:
  detect             Apply a signature onto a file.
  detector           Detect if a patch has been applied to a firmware
                     image...
  generate           Generate a signature based on a CVE directory
  generate-multiple  Generate signature for every CVE found in a directory.
  info               Dump info on the signatures.
```

**Listing B.2** `detect` Command Example

```
$ python -m qsig detect libbluetooth.so CVE-2018-9506.sig
INFO: libbluetooth.so was matched with the signature
      (using ['strings', 'constants'])
INFO: Complete chunk match for libbluetooth.so
INFO: CVE Match for CVE-2018-9506 on libbluetooth.so
```

**Listing B.3** `info` Command Example

```
$ python -m qsig info CVE-2019-2134.sig
INFO: Signature CVE-2019-2134 (bfa3d8) :
    File phNxpExtns_MifareStd (OBJECT):
        Chunk : STRINGS CONSTANTS CALLS
    File libnfc_nci_jni (LIBRARY):
        Chunk : STRINGS CONSTANTS CALLS CONDITIONS
```

**Listing B.4** Custom Detector

```
from firmextractor.fs import ExecutableFile
from qsig.detector import Detector

def get_strings(fw_file: ExecutableFile) -> set[str]:
    """Returns `fw_file' strings"""
    ...

class MyDetector(Detector):
    """Simple File Detector to assess whether a candidate
       has a specific string.
    """
    def __init__(self, match_string: str):
        """Constructor"""
        self.match_string = match_string

    def accept(self, fw_file: ExecutableFile) -> bool:
        """Check whether `fw_file' is accepted"""
        return True

    def match(self, fw_file: ExecutableFile) -> bool:
        """Perform the match."""
        return self.match_string in get_strings(fw_file)
```

### B.1.2 As a Library

**QSig** interface is insufficient to answer every query, and a user may want to change some of **QSig**'s behavior more deeply. Thus **QSig** is also available as a library. However, its usage is more complex.

Listing B.4 demonstrates how to implement a custom detector for **QSig**. While this one only searches a string inside the program ones, it is possible to add more complex ones.

**Listing B.5** Custom Detector

```
from pathlib import Path
from firmextractor.fs import FileSystem

def search_string(file_system: Path, string: str):
    file_system = FileSystem(file_system)
    detector = MyDetector(string)

    for exec_file in file_system.elf_files():
        if detector.accept(exec_file):
            print(f"Match Result: {cve_detector.match(exec_file)}")
```

Using this new detector is straightforward and Listing B.5 shows how to instantiate and use it on a filesystem[1].

## B.2 Code Internals

### B.2.1 Code Layout

**QSig** is written in Python and its core composed of 4,637 LoC divided into three modules. Figure B.1 displays the code layout and we briefly describe the role of each module below.

- `sig`: This module is responsible for storing and retrieving the signatures, both for the generator and the detector. Notably, it manages the serialization using `Protobuf`. Finally, it also handles the `BinCAT` integration: it generates the config file required by the AI framework and retrieves the results.
- `generator`: As expected, this module handles the signature generation. It takes as input a list of tuples `(vulnerable_binary, fixed_binary)` from an architecture, and generates a signature from the difference between each pair. The module is organized in three files, one at the CVE level, one at the file level and one for the function level.
- `detector`: Finally, the detector module handles the detection part in **QSig**. A `Detector` is a class responsible for matching a signature. It is possible to instantiate multiple `Detector` at the same time to search for multiple patches in a single batch. Of note, the `Detector` also follows the signature layout and is divided into three parts.

### B.2.2 Dependencies

As specified in Chapter 3, **QSig** offloads part of its workflow to internal and external tools:

- `IDA` for the disassembling and *Quokka* to manipulate the resulting disassembly.
- A modified `BinCAT` to perform the condition's term taint tracking.
- Probotuf to store the signatures.
- BGraph for the filtering step on Android phone firmwares.

## B.3 Conclusion

This appendix briefly presented how to use **QSig** and some of its internals. The code itself is open-source and available on Quarkslab's GitHub [19] with some documentation.

[1] We removed the error handling on the listing to lighten it.

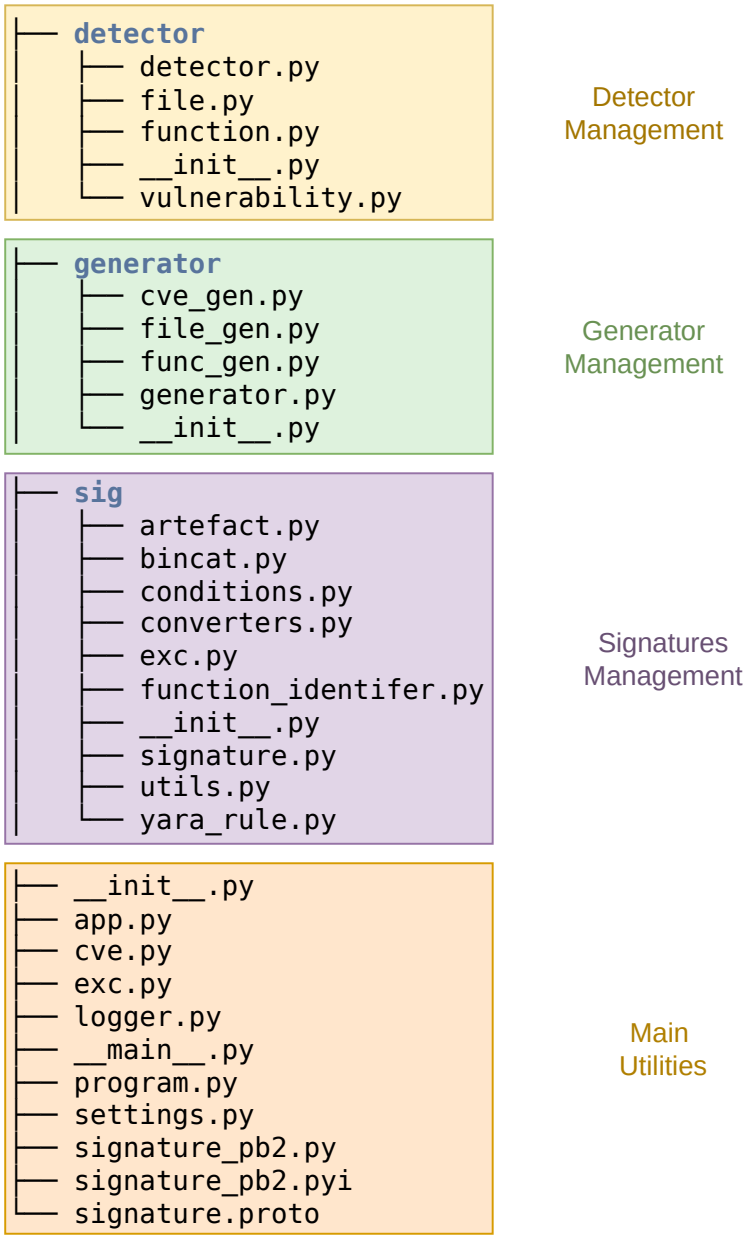


Figure B.1: **QSig**'s Code Layout

DOCTORAL SCHOOL

**Titre:** Vers la recherche de vulnérabilités 1-jour à partir de signatures sémantiques de correctifs

**Mots clés:** Vulnérabilités *1-jour*, Signature sémantiques de patch, Detection de correctif

**Résumé:**
Pour maintenir la sécurité des systèmes d'information, déployer les mises-à-jour proposées dès qu'elles sont disponibles est une bonne pratique encouragée par l'ensemble des acteurs de la sécurité informatique. Effectivement, l'exploitation des vulnérabilités de type *1-jour* (dénommées ainsi car il en existe un correctif depuis au moins 1 journée), peut être dévastatrice comme EternalBlue ou Shellshock ont pu l'illustrer.
L'objectif de cette thèse est de proposer des méthodes et leur application pratique pour détecter si ces correctifs sont bien appliqués au plus bas niveau, i.e. dans le code binaire. Ceci est indispensable pour avoir une vision fiable de la protection d'un système.
Pour atteindre cet objectif, nous avons établi plusieurs jalons. Le premier consiste en une étude approfondie d'un correctif type, avant de formaliser un cadre de recherche de ces derniers à l'échelle d'un système complet.
Nous proposons ensuite l'implémentation d'une solution logicielle construisant automatiquement des signatures sémantiques de correctifs de vulnérabilités et cherchant ces signatures dans des systèmes de fichiers.
Finalement, nous testons cette solution en conditions réelles (i.e. détection de correctifs dans des images du système d'exploitation Android) et montrons la pertinence de cette solution.

**Title:** Towards *1-day* Vulnerability Detection using Semantic Patch Signatures

**Keywords:** *1-day* Vulnerability, Semantic Patch Signature, Patch Detection

**Abstract:**
To maintain the security of information systems, deploying the proposed updates as soon as they are available is a good practice encouraged by all the computer security actors. Indeed, the exploitation of *1-day* vulnerabilities (so called because a patch has been available for at least 1 day) can be devastating as EternalBlue or Shellshock have illustrated.
The objective of this thesis is to propose methods and their practical application to detect if these patches are well applied at the lowest level, i.e. in the binary code. This is essential to have a reliable view of a system protection.
To achieve this goal, we have established several milestones. The first one consists in an in-depth study of a typical patch, before formalizing a framework for searching for them at the scale of a complete system.
We then propose the implementation of a software solution that automatically builds semantic signatures of vulnerability patches and searches for these signatures in filesystems.
Finally, we test this solution in real conditions (i.e. detection of patches in images of the Android operating system) and show the relevance of our approach.

**Institut Polytechnique de Paris**
91120 Palaiseau, France