PhD-FSTM-2022-101
Faculty of Science, Technology and Medicine

Dissertation

Presented on the 28/09/2022 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Timothée RIOM
Born on 6th July 1992 in Orléans, France

A Software Vulnerabilities Odysseus: Analysis, Detection, and Mitigation

Dissertation Defense Committee

Dr. Jacques KLEIN, Dissertation Supervisor
Associate Professor, University of Luxembourg, Luxembourg

Dr. Tegawendé Francois d'Assise BISSYANDE, Chairman
Associate Professor, University of Luxembourg, Luxembourg

Dr. Yves LE TRAON, Vice-Chairman
Full Professor, University of Luxembourg, Luxembourg

Dr. Olivier BARAIS,
Full Professor, Université de Rennes 1, France

Dr. Pierre GRAUX,
Associate Professor, University of Lille, France

Abstract

Programming has become central in the development of human activities while not being immune to defaults, or bugs. Developers have developed specific methods and sequences of tests that they implement to prevent these bugs from being deployed in releases. Nonetheless, not all cases can be thought through beforehand, and automation presents limits the community attempts to overcome. As a consequence, not all bugs can be caught.

These defaults are causing particular concerns in case bugs can be exploited to breach the program's security policy. They are then called vulnerabilities and provide specific actors with undesired access to the resources a program manages. It damages the trust in the program and in its developers, and may eventually impact the adoption of the program. Hence, to attribute a specific attention to vulnerabilities appears as a natural outcome. In this regard, this PhD work targets the following three challenges:

(1) The research community references those vulnerabilities, categorises them, reports and ranks their impact. As a result, analysts can learn from past vulnerabilities in specific programs and figure out new ideas to counter them. Nonetheless, the resulting quality of the lessons and the usefulness of ensuing solutions depend on the quality and the consistency of the information provided in the reports.

(2) New methods to detect vulnerabilities can emerge among the teachings this monitoring provides. With responsible reporting, these detection methods can provide hardening of the programs we rely on. Additionally, in a context of computer performance gain, machine learning algorithms are increasingly adopted, providing engaging promises.

(3) If some of these promises can be fulfilled, not all are not reachable today. Therefore a complementary strategy needs to be adopted while vulnerabilities evade detection up to public releases. Instead of preventing their introduction, programs can be hardened to scale down their exploitability. Increasing the complexity to exploit or lowering the impact below specific thresholds makes the presence of vulnerabilities an affordable risk for the feature provided. The history of programming development encloses the experimentation and the adoption of so-called defence mechanisms. Their goals and performances can be diverse, but their implementation in worldwide adopted programs and systems (such as the Android Open Source Project) acknowledges their pivotal position.

To face these challenges, we provide the following contributions:

- We provide a manual categorisation of the vulnerabilities of the worldwide adopted Android Open Source Project up to June 2020. Clarifying to adopt a *vulnerability* analysis provides consistency in the resulting data set. It facilitates the explainability of the analyses and sets up for the updatability of the resulting set of vulnerabilities. Based on this analysis, we study the evolution of AOSP's vulnerabilities. We explore the different temporal evolutions of the vulnerabilities

affecting the system for their severity, the type of vulnerability, and we provide a focus on memory corruption-related vulnerabilities.

- We undertake the replication of a machine-learning based detection algorithms that, besides being part of the state-of-the-art and referenced to by ensuing works, was not available. Named VCCFinder, this algorithm implements a Support-Vector Machine and bases its training on Vulnerability-Contributing Commits and related patches for C and C++ code. Not in capacity to achieve analogous performances to the original article, we explore parameters and algorithms, and attempt to overcome the challenge provided by the over-population of unlabeled entries in the data set. We provide the community with our code and results as a replicable baseline for further improvement.
- We eventually list the defence mechanisms that the Android Open Source Project incrementally implements, and we discuss how it sometimes answers comments the community addressed to the project's developers. We further verify the extent to which specific memory corruption defence mechanisms were implemented in the binaries of different versions of Android (from API-level 10 to 28). We eventually confront the evolution of memory corruption-related vulnerabilities with the implementation timeline of related defence mechanisms.

Appliquer son interêt et ses reflexions autant au travail lui même qu'à son produit.

Albert Camus, *L'Homme Révolté*

Acknowledgement

This PhD work results from four years at the University of Luxembourg, first in the Serval team led by Prof. Yves Le Traon, then at Trux, with Ass. Prof. Jacques Klein. More than an achievement, it is more a journey that was possible foremost by the people surrounding me and supporting me all along.

I need to first thank the fortitude of my supervisor, Ass. Prof. Jacques Klein, for giving me the opportunity to work in the ideal context of the Trux team, and from whom I could learn to seek for high standards in scientific research. I also shall thank his kindness and his endurance all along the path. To that regard, I shall also underline the benevolence Ass. Prof. Tegawendé Francois d'Assises Bissyandé demonstrated during four years, as for his sagacity.

I also wish to thank the persons from whom I could count on advice invariably; both Dr. Kevin Allix and Prof. Alexandre Bartel hold experience and proficiency that guided my work. If the reader is to find any quality in this work, I extensively owe them for it.

This doctoral journey was also the opportunity to work with remarkable and distinguished researchers. Hence I need to say it was my pleasure to work with Dr. Arthur Sawadogo, and Dr. Li Li, and my honour to do so with Prof. Nahouel Moha and Prof. Yves Le Traon.

From my experience at both Serval and Trux from the SnT, I must thank the researchers that surrounded me, accompanied me, and sometimes condoned me, as it was the case for Dr. Jun Gao, Dr. Pingfan Kong, Dr. Ludovic Mouline, Dr. Niklas Kolbe, Ms. Nadia Daoudi, Mr. Jordan Samhi, Mr. Cedric Lothritz, Dr. Mederic Hurier, Dr. Anil Koyuncu, Dr. Mathieu Jimenez. I found around them, as in both Trux and Serval teams, a studious environment demonstrating what discipline and tenacity are.

I desire to express my gratitude to the members of my PhD defence committee, including Prof. Olivier Barais, Ass. Prof. Pierre Graux, my supervisor Ass. Prof. Jacques Klein, Prof. Yves Le Traon, and Ass. Prof. Tegawendé Francois d'Assises Bissyandé. It is my great honour that they accepted to join my defence committee, and I highly esteem the opportunity to have my PhD thesis examined by such reverential members of the scientific community .

Finally, I want to thank the people, from outside the scientific community, who supported me all along this enterprise. May they be family, may they be friends; their companionship and their care strengthened the perseverance in achieving this work, and gave this experience its worth.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter we introduce the context of this PhD work. In particular, we will discuss software vulnerabilities and their impact on human activities. We also present the challenges these vulnerabilities bring to the community. Finally we describe our contributions.

Contents

1.1 Motivation

Our societies are living through succeeding digital revolutions. The developed economies have shifted to services through the mean of digital solutions. An increasing number of human activities rely on lines of code.

It is through the online version of articles that we get the news, when we are not only relying on uploaded videos, streaming services or social networks. During the emergence of the 2020 COVID-19 pandemic, this dependence was highlighted for countries in which a lockdown was set: to communicate with our relatives and coworkers we almost exclusively resorted to social networks, instant messaging and VideoTelephony services. Either these services were known for not meeting security standards [2, 3, 4] or not so private[5]. Our democracies thus heavily rely on the Confidentiality, the Availability and the Integrity of these services.

The explosion of computer performance has revealed crucial to economics and research. For instance, it enables the modelling of complex ecosystems[6], the understanding of the root causes of late tendencies[1] and provides solutions to how humankind may still evade the worst consequences, the model concludes, we are heading to[7].

It is notably used for what is among humankind's most significant successes: space exploration. We sent humans into space to land on the Moon with no more than 72kB of Read-Only Memory and 4kB of Random-Access Memory. Sixty years later, we are sending rovers to visit other planets[8] and flying a helicopter drone over Mars, given the three to twenty minutes delay for either instructions to reach Mars from Earth or for information to come back.

Human lives daily depend on such kind of decisive computation, and the lives of billions will further depend on complex computations.

However, sometimes the system encounters a defect. A defect that could be more or less expected, as it happened with the notorious Error 1202 that the Apollo XI mission faced just seven minutes before landing on the Moon[9]. As the lander was getting closer to the surface, radar computation to rendezvous with the orbital module was allocated most of the computation cycles in case the mission shall abort. When the crew attempted to add tasks so to check the situation, new computations overloaded the Apollo Guidance Computer Resulting in several system reboots, as the Moon was getting closer and the landing site overshot[10]. The root cause of the Error was unbeknownst to the lander's crew and the team on Earth. They had yet to understand the gravity and weight the hazards: eventually making the call not to abort the landing. Neil Armstrong then took the command in semi-automatic and landed the Lunar module. The rest is History.

In computer science, to these defects preventing the execution of a program from going as planned, we usually give the name *bugs*. The term is usually attributed to Grace Hopper, for finding a moth between two relays of Harvard's Mark II in 1947. However, the use of the term *bug* to mean *defect* can be traced back further to the second half of the 19th century [11]. Evidence was found related to Thomas Edison's communication and reports on the phonograph in 1889 and lighting in 1878. Implying the use of the term was already well installed. For instance, the term is also present in Funk & Wagnalls's 1895 dictionary.

There is a specific category of bugs that will further be of interest in this thesis called vulnerabilities. A vulnerability, as given by Robert Shirey's 2000 Internet Security

[1] p4: "*it is unequivocal that human influence has warmed the atmosphere, ocean and land.*"

Glossary[12], is:

> *A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.*

This definition involves in itself several other tenets that need an explanation; as it stands: *security policy* and *exploitation*.

For security policy, Robert Shirey's glossary provides:

> *A set of rules and practices that specify or regulate how a system or organization provides security services to protect sensitive and critical system resources.*

Hence, any system has to define the list of resources it provides or connects to; it also has to define a list of actors (desired and undesired) that may interact with the system. Finally, it has to define a set of statements ruling how each actor may or shall not interact with the resources.

Another central principle in the definition of vulnerability is the notion of *exploitation*. What will make the flaw a security issue, called a vulnerability, is that it is exploitable to violate the rules established by the security policy. We can consider this defect a vulnerability if it is detectable and exploitable to extend specific actors' range of action on the sensitive or critical resources.

These vulnerabilities can be introduced during different steps of software development. For instance, the program can present flaws early in the design phase. For the Wired Equivalent Privacy (known as WEP), an algorithm for secure wireless networking, several core vulnerabilities produced the need for its complete substitution in 2003[13], just four years after its introduction, by the Wi-Fi Protected Access (WPA) [2].

In other cases, vulnerabilities emerge in relation to how is the code written. For instance, a vulnerability identified by the tag CVE-2018-5703 and found in the Linux kernel, as described in [15], provides the possibility for a type confusion. In optimal, and provoked circumstances, it enables to overwrite addresses and eventually redirects the flow of execution. As some of these vulnerabilities can be regrouped under categories, or weaknesses, it can appear that some implementation parameters, such as the code language, can be specifically prone to several of these weaknesses. For instance, low-level languages, such as C or C++, are more susceptible to memory-related vulnerabilities. In comparison to C and C++, Java uses a garbage collector that manages the memory and significantly reduces the possibilities for objects to interact with each other. Another coding language, Rust, has been written with security in mind while still providing the speed and performance predictability of C and C++.

Finally, operations might not be completely thought through as a variable may be declared with a type that requires its value to stay within a range. Yet code operations enabling the value to reach the maximum and collapse to a small or negative value are

[2]These vulnerabilities included a 24-bit space for vector initialisation. Given the construction of both the key and the exchanged packets, only a few hours of eavesdropping were required to decipher these packets [14]. The algorithm did not efffectively protected the integrity of sources and destinations, nor was the standing integrity protection genuinely preventing packet changes. As a substitution, first was implemented WPA, using TKIP, Temporal Key Integrity Protocol, in 2003. It used a 64-bit key to generate 128-bit keys per packets, and better protected the integrity of both packets and source/destination. A more withstanding answer was implemented as a standard in 2004 with WPA2. WPA2 standardised the use of the Advanced Encrypted Standard (AES). Moreover, WPA2 needed to be superseded only in 2018, by WPA3, while TKIP is obsolete since 2009.

not checked. These operations are thus waiting to be found and exploited by those who will find them.

Given how entangled with code our societies are, when there is the possibility for exploitation, the outcome can be dramatic.

- Our democracies and the balance of power can be impacted by the exploitation of vulnerabilities, as demonstrated since 2016 by the Canadian Citizen Lab multi-disciplinary department. They revealed how the market of exploit programs targeting yet-unknown-to-vendor (or 0-day) vulnerabilities ends up targeting country leaders [16], opposition groups [17, 18] and minorities within a country [19]. The specificity, in comparison with the usual payload sent through emails, is that these vulnerabilities may involve a zero-click exploitation: thus do not require interaction from the receiver. Owning a device like a smartphone, being of interest to the buyer of such exploitation programs and the buyer knowing how to send a message to one's device (as simply as the telephone number) is all it requires [3].
- Our economic system also revealed vulnerable in June 2017, when the NotPetya ransomware, seemingly first aimed at Ukraine, spread to multinational companies. With a backdoor set in a program helping to return tax specifically for Ukraine, the believed-to-be-NSA-developed Eternal Blue exploit [20] can redirect the program to execute the encryption of all partitions while the system refuses to boot [4]. Any company paying taxes in Ukraine was likely to be compromised, and it is estimated that the attack directly cost over ten billion dollars[21]. It further leads to years of court cases for companies trying to be compensated by insurance. Only in 2022 did Merck get compensated up to 1.4 billion dollars[22].

While so much is at stake, their are several reasons explaining why these vulnerabilities still appear.

- First, allocating too many resources to prevent any vulnerability might not be the most rational answer. Scientific research in economics [23] argues that vendors have little incentive to produce more secure programs. Resources spent on security stand in the way of integrating features and gaining a market advantage. A quote could summarise the general rule: "*Ship it Tuesday, and get it right by version 3*" [23].
- A second element, not among the easiest to measure, concerns the complexity of exploitation. Some exploits might be easy to develop, and thus their prevention becomes necessary in order to protect customers and businesses. In contrast, other vulnerabilities may be considered only exploitable by a restricted number of highly skilled experts, not widespread, or not providing critical resources. In this context, provided the actual risks to end-users and other external incentives

[3] A more precise explanation of how an external library rendering PDF on iOS can be targeted to trigger an integer overflow is provided by Google's Project Zero's team at `https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html`. It eventually results in a complete and parallel logic computer architecture to be built and to operate freely on the device.

[4] NotPetya involves the setting of a backdoor, the exploit of a vulnerability and a payload. The backdoor was set in an ukranian tax return software called M.E.Doc. Any server updating the software on the eve of Ukraine's Constitution day would install this back-door. It made possible for the further installation and execution of believed-to-be NSA-developed exploit triggering Microsoft Server Message Block, responsible for file sharing [20]. A crafted package overflows a structure and is both capable of overwriting a bit set to prevent the data to be executed as instructions and to redirect the flow of executions to attached code. The payload further targets Microsoft Master Boot Record (**MBR**): bootloader enabling to list all partition available to a device and selecting the one to boot from. It overwrites it while encrypting partitions: locking data out of reach of users unless a key is provided.

(e.g. the market), it might not be in the vendor's interest to allocate too much or any resources at all to tackle this specific issue.

- Last but not least, a consideration that makes it necessary for an economic actor to stop the development of resources allocated to security is that vulnerable code will always slip through. The first reason is that human knowledge cannot cover all existing exploitation techniques, and new techniques might be discovered daily. Hence what might be considered secure code one day, might not be secure anymore the next day. Second reason is that safe practices and secure reviews of the code are skills that need time to develop, need platforms for training purposes, and that the range of possibilities of the code to be exploited is so widespread that it is not possible to be an expert to cover them all. For instance, it would include physical attacks (electro-magnetic or LASER perturbation), to develop a fuzzer searching for vulnerabilities, through finding malware using Machine Learning in Android application[24], or static analysis of the code to reveal atypical communication [25] and so forth. The outcome is that it is complex for developers to assess correctly when a bug is exploitable, needing much more attention and care to fix the issue.

There is therefore a limit to what an actor can do facing the issue of vulnerabilities.

1.2 Challenges

However, if given these limits, there is no perfect solution; we are not doomed to inaction . Researchers can aim to improve the situation by targeting the following challenges:

- The first challenge is to prevent the introduction of vulnerabilities in the code. It takes trained developers so that a vulnerability is either spotted early in the process and corrected quickly, or even never written at all. Never replicating the past implies studying know threats and constant diligence toward newest ones. It further provides to update security policies accordingly. Understanding the severity of a vulnerability is crucial to measure how much resources shall be allocated to tackle the exposure.
- Detecting software vulnerabilities as early as possible becomes an essential endeavour for software engineering and security research communities [26, 27, 28, 29]. Typically, software vulnerabilities are tracked during code reviews, often with the help of analysis tools that narrow the focus scope by flagging potentially dangerous code. The best conditions are when vulnerable code is flagged before release.
- Another dimension through which fighting exposure is possible does not necessarily address the presence of vulnerable code. It instead focuses on preventing its exploitability. For instance, the system might be capable of detecting an altered run of its programs and halting them to prevent further harm. It could also be made resilient to particular vulnerabilities and still provide the desired feature.

1.3 Contributions

We contribute, in this thesis, to address to the aforementioned challenges as follows:

- **[Vulnerability Understanding]** In light of the Android Open Source Project, we provide a study of the vulnerabilities that affected the system until June 2020. This study heavily benefits from the transparent approach provided by Google regarding Android, specifically since 2015. A clear methodology for

categorisation, based on the fixing patch and an updated set of weaknesses is provided. It generates a more accurate data set, linking the vulnerable code to the vulnerability type. Such a set can reveal handy to train developers, learn to patch given a specific weakness, or help the testing and training of detection algorithms.

- [**Vulnerability Detection**] To detect vulnerabilities in the early stages of the development, we undertake the replication of the machine-learning-based VCCFinder [30].
  While many static and dynamic approaches have focused on regularly analysing the software in its entirety, a recent research direction has focused on the analysis of changes that are applied to the code. VCCFinder is a seminal approach in the literature that builds on machine learning to detect whether an incoming commit will introduce some vulnerabilities automatically. Given the influence of VCCFinder in the literature, we investigate into its performance as a state-of-the-art system. The insights of our failure to replicate the results reported in the original publication informed the design of a new approach to identify vulnerability-contributing commits based on a semi-supervised learning technique with an alternate feature set. We provide all artefacts and a clear description of this approach as a **new reproducible baseline** for advancing research on machine learning-based identification of vulnerability-introducing commits.
- [**Vulnerability Exploitation Prevention**] We finally provide an investigation into the Android Open Source Project (AOSP) implementation of defence mechanisms over 10 years. The system is an authentic opportunity for several reasons. First of which regards how recent Android is. As a project for an Operating System of the new millennium, it provides insights into a complex modern system. Further, the code of AOSP is available online, providing the community with the opportunity to study it properly. It is a worldwide adopted system, centralising numerous sensitive features and resources into one embedded device. AOSP is thus under pressure to provide this great deal of customers with an adequate level of security. We provide a chronology of the implementation of defence mechanisms, which we complete with the in-device binaries analysis. We further discuss how these implementations have sometimes answered comments from the community. Eventually, we confront the defence mechanism implementation with the number of related vulnerabilities that the Android development team advertises to have needed to patch.

2 State of the Art & Background

In this chapter, we aim at providing both an overview of the state-of-the-art works and the relevant background related to this PhD thesis. We first define terms related to vulnerabilities and introduce their life-cycle. We also describe studies providing insight on the prevalence of vulnerabilities in programs. Second, we report related works focusing on vulnerability detection. Third, we provide an historicity of the development of Defense Mechanism. Eventually, we provide a background on the Android Operating System.

Contents

2.1 Vulnerabilities

This section enables to remind what vulnerabilities are and provides with the terminology that is further of use in this thesis. Among theses aspects, we precise how are vulnerabilities referenced by the community and the challenges these listings includes. We also precise the different steps in a lifetime of a vulnerability. This description provides how the research community highlights crucial time frames the community and developers needs to trigger. We eventually provide with a few analyses stating the exposure of systems and programs through the use of vulnerabilities.

2.1.1 Definitions

In the introduction of this thesis, we introduced the following definition for a vulnerability [12]:

> *A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy.*

Each vulnerability, during the disclosure process, is attributed with a unique identifier: Common Vulnerabilities and Exposures (or **CVE**). These tags are in the form `CVE-XXXX-YYYY`, with `XXXX` as the year, and `YYYY` a yearly incremental counter of registered CVEs. Alongside, several other information will be registered as a description of the vulnerability, the affected software and version, how confidentiality, integrity and availability can be impacted by this vulnerability, a categorisation (or CWE), potentially a link to the patch and to external resources.

The impact of vulnerabilities on a system is usually measured on the base of three properties: Confidentiality, Integrity and Availability, often referred as CIA. A vulnerability is also usually characterised by its severity, which is ranked through the Common Vulnerability Scoring System (or **CVSS**). Vulnerabilities can be attributed a type, or weakness, through the Common Weakness Exposure (**CWE**) tagging system.

These characteristics can be defined as follows:

Confidentiality: [12]

> *The property that information is not made available or disclosed to unauthorized individuals, entities, or processes [i.e., to any unauthorized system entity].*

Integrity: [12]

> *The property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner.*

Availability: [12]

> *The property of a system or a system resource being accessible and usable upon demand by an authorized system entity, according to performance specifications for the system*

Common Vulnerability Scoring System:

The **CVSS** is computed depending on several *criteria* [31]. Among others: the complexity of the attack, its impact on Confidentiality, Integrity and Availability of the vulnerable software, the proven existence (or not) of in-the-wild exploit(s) for this vulnerability, ... Registered CVEs might be referenced with either one or two versions for the severity computation. Since 2015, version 3.0 has superseeded. However, not all

vulnerabilities are given both scores and the conversion is not trivial [32]. This score is especially relevant for teams using a vulnerable version of the software in order to measure their exposure and prioritise adressing the issue with their usual tasks. A mark, over 10, is attributed and thresholds help understand the severity of the exposure. For version 3.0, a severity below to 4.0 is considered Low; below 7.0, it is considered Medium; until 9.0, it is High; and Critical up to 10.

Common Weakness Exposure: There exists a taxonomy enabling to categorise CVEs into 600 groups, or CWEs. Maintained by Mitre organisation[1], CWEs are hierarchically organised in trees and any CWE can be reached from three main domains (or roots) at the base of their internal mapping[2] (provided in Section 3.2.2.1).

In this trees, two types of links relate CWEs to each other: generalisation and potential causality.

- The `ParentOf` field links a coarser vulnerability category to a finer one. For example, `CWE-121`: Stack-based Overflow is a sub-type of `CWE-787`: Out-of-bounds Write. Thus CWE-787 is `ParentOf` CWE-121, as a more general category than CWE-121.
- The `canFollow` field relates vulnerabilities on the *causality* to *consequence* model (i.e., one weakness may induces the other elsewhere in the code). For instance, an Expired Pointer Dereference (CWE-825) may be followed by an Out-of-Bounds Write (CWE-787).

These connections, or relationships, enable to provide finer classes for comparing two differing analyses than only relying on exact matches (as conducted in Section 3.1).

2.1.2 Data sources & reliability

Software vulnerabilities databases have many applications in research fields such as vulnerabilities prediction, and Automatic Program Repair. In those fields, doubts have been cast as to whether the vulnerability prediction results obtained on academic datasets could also be obtained in real-world settings [33]. A literature survey lists the many issues that could affect Software Vulnerabilities databases [34], and that could hinder the generalisability of experimental results obtained on such databases. Among these challenges is mentionned the intensity of the effort required by manual labelling, the expertise required by such a task. Are also addressed the causes and consequences of erroneous labelling. For the causes: the subjectivity of the categorisation, and inaccuracy of tools for automatic labelling. These heavily impacts analysis based on futur exploitations of this data.

Data preparation is thus an essential step for Software Engineering experiments. We present here prior works that discuss the quality and availability of relevant data sources.

2.1.2.1 Mitre and NVD

The Mitre Corporation operates and maintains a reference of disclosed vulnerabilities[3]. All CVEs are also recorded by the NIST in the US National Vulnerability Database (or NVD), where CVEs are associated with additional information.

The quality of registered CVE data on reference websites such as CVE Mitre, the NIST's NVD and several bug-trackers is already investigated [35]. The information

[1] `https://cwe.mitre.org/`
[2] `https://cwe.mitre.org/data/index.html`
[3] `https://cve.mitre.org/`

provided is, overall, enough to reproduce the exploitation of the vulnerability in less than one case out of two. For some bug-trackers, only 4% of the vulnerability reports provide sufficient information.

Another study showed that the information available in NVD can be incoherent and incomplete [32]. The authors of this study propose approaches to fix several of the issues they identified. For instance, they devise a method to obtain a relevant CWE tag from the CVE description when it is missing in the web page.

2.1.2.2 Dataset labelling

If Manual labelling is eventually the most reliable labelling approach [34] it is also a non-trivial and labour-intensive task [36]. A central bias of manual analyses is the specification of the risk model. In the absence of a clear risk model, results may be not reproducible, and may be usable only in specific cases (as the NIST states[4]). Data labelling is also affected by the issue of subjectivity [37].

It has been shown that errors—or even noise—in the data used to train AI approaches can greatly impair vulnerability detection performance, in particular in real-world settings [33]. The NIST advises analyses to clearly state the chosen risk model. Otherwise, the very same vulnerability can be attributed a different category depending on whether the analysis focuses on (a) the source threat, (b) the assets that are impacted in the end of the exploit, or (c) the moment (the lines of code) at which the behaviour would deviate from the expected one.

Manual labelling can be prohibitively expensive. Thus several works have presented approaches to automatically label code with either a vulnerable flag or a non-vulnerable flag. For instance, a deep learning classifier on Abstract Syntax Trees can enable the comparison on how much effort can be saved in comparison with usual methods relying on code metrics to appropriately label vulnerable functions [38]. Bi-directional Neural Networks have also been used to attempt to transfer learning from unlabelled to labelled projects (regarding vulnerabilities) [36].

2.1.3 Vulnerability Lifecycle

A vulnerability, from its introduction until the delivery of the patch to all instances, goes through several phases that we further detail.

First, there is the (i) introduction of the vulnerability, its birth. Afar from what the name may suggest, introduction of a vulnerability does not mean that it only happens with the addition of new lines of code. Second stage is the (ii) discovery of a vulnerability. This step can occur a long period of time after the actual birth of the vulnerability and can happen by accident. After the discovery, a potential resource intensive step is the (iii) writing of a Proof-of-Concept (or PoC) that would trigger the defect in such condition so to gain advantage in regard with the resource the target program holds. Once it works the next step depends on the incentive of the discoverer(s) and the people with the knowledge of this vulnerability: either they use it for their own benefit or (iv) disclose it to the developer of the vulnerable program. At the moment the vulnerability is reported, it can be (v) fixed by the modification of the code. It is the correction phase. Eventually, the vulnerability meet its death once the patch is (vi) delivered to all instances using the originally vulnerable version of the code.

[4]See page 6-7 and 15 of `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf`

2.1.3.1 Vulnerability Introduction

As discussed in the introduction of this PhD work, there is a vast common ground between bugs and vulnerabilities. Thus there are as many good reasons for a vulnerability to be accidentally introduced as there are for a bug.

When it comes to determine the actual cause of the introduction of a vulnerability, an example demonstrates the precocious approach taken regarding the labeling of such commit. It is conservatively called Vulnerability Contributing-Commit (or VCC) [39]. This name underlines that it is not necessarily the changes made that introduces themselves a vulnerability but potentially more how modified lines integrates in the whole program architecture and behaviour.

For instance, the implementation phase of a feature is confronted to deadlines. These deadlines might induce prioritisation and the writing of functional yet temporary code. This code is a liability as it has not been properly tested and requires more time to be spent to be properly rewritten [40].

This leads to another human related liability that is the skill to develop proper test cases for ones' program. Thus several fields of researchs dedicate to increase the number of tests and the location in which these tests seek for corner cases [41].

There resides plenty of other reasons for a fault or a bug to appear. Plenty for each step of software development (specification, design, coding skills, version control), either they are human-related or related to a third party so far trusted.

The code-review process itself can let slip vulnerabilities through. It persists as a sain and beneficial habit though. For instance, an investigation over Chromium OS covers 516 defects discovered during the the code review process, and 374 that slipped through it [42]. Among these vulnerabilities that slipped through, the type was usually vulnerabilities that reveal during the execution such as input neutralisation, datatype conversion, occurring in a complex software as access control, or resulting in an information exposure. The favoured condition for such security defect to evade the review where also related to the complexity of the program, measured in the depth of sub-folder the file to review is located in. Other conditions were also human factors as the number of prior reviews was decreasing the accuracy of the analysis of the reviewer, altogether the number of commits this reviewer has been responsible for in the code. In opposition, the more time was spent on the review, the more the detection rate was increased. Two other elements are worthy to mention so to keep reviewer alerted to risk and efficient in detecting security defects are the experience one reviewer might have reviewing a specific file, and in the case the review concerns a bug fix.

As earlier mentioned, if there are such a thing as innocently introduced vulnerabilities, we may be entitled to think about the contradiction to innocently introduced. However this aspect brings us closer to what could be a malware than a vulnerability. In this case, the product is purposely engineered so to offer vulnerable ground to actors aware such vulnerability is available to trigger. Such a categorisation is nonetheless speculative as it, first, requires to interpret why such a vulnerability in the code was never corrected, and, secondly, requires to agree on the definition of what a malware is. Contraction between *malicious* and *software*, Robert Shirey's 2000 Internet Security Glossary [12] associates this term with malicious logic. *Hardware, software, or firmware that is intentionally included or inserted in a system for a harmful purpose.* The latter introduces another controversial debate attempting to answer what is an *harmful purpose*. For instance, citing examples of harmful purpose Android applications, it can go from "*simple user tracking and disclosure of personal information to advanced fraud and premium-rate*

SMS services subscription, or even unwarranted involvement in botnets" [43]. And the attack surface can widen as much as the software gets more complex. For instance, Android Applications use libraries that can themselves carry a payload and/or be modified by the application using them so to appear as being the source of the harmful behaviour [44]. However, Malicious Software or Malicious Logic is a contiguous subject to the one of vulnerabilities but is not the principal subject of this thesis.

2.1.3.2 Vulnerability Discovery

The listing of techniques used and employed so to detect vulnerabilities in a program are detailed further below in Section 2.2.

Regardless, we can detail that discovery intervenes when either an experienced look, specific testing and/or programs developed with this specific purpose analyse the code and recognise exploitable defects. It can however of course happen by accident when attempting to use a program in specific conditions as for many scientific discoveries.

An unaccountable number vulnerabilities exist but never are discovered. We have nonetheless the knowledge to detect them and patch them. In some cases, we may even have the tools to, for instance, statically analyse the code. The issue resides in the fact that documentation may be so emaciated and/or interaction with these tools not so intuitive that these tools are never used in the end [45].

2.1.3.3 Scripting of the exploit

Before reporting and/or engaging in any kind of disclosure of the vulnerability, an important step is to assess the range of consequences this defect may be the root cause of. This involves developing an exploit that will trigger the defect in the target program and then, depending on the specificity of this defect, attempt to gain privileges regarding the resources this program manages.

In this exploit resides the proof of the damage one program may be subject of, and thus a key element into convincing the team the program originates from to take action. It requires skilled individuals into writing these exploits as they have to understand the extent of resources newly available through the defect and know already the most efficient methods so to capture them.

However several vulnerability exploits do not reach their full potential. For instance, the exploit associated with CVE-2018-5703 did not see that, not only could the flow of execution be redirected to a value hold by a variable, but also the attacker, in this scenario, was in capacity to modify this variable through an type confusion. The cause for such an understatement of the severity of a vulnerability may reside in the fact that this step heavily relies on human skills. So to tackle this issue, several program, such as Koobe [15], written for Linux kernel out-of-bounds writes, attempt to explore further the capabilities of a vulnerability from an already existing exploit (sometimes called Proof-of-Concept). For Koobe, this "*exploration*" is made using Google's developed fuzzing program Syzkaller [46] and symbolic execution tracing program S2E [47] (instrumentation involves angr [48]). Authors of Koobe aim alternatively for code coverage and exploration of capabilities at some trigger point through inputs mutation. Slake [49] is another program attempting to generate exploits in the Linux kernel. This evaluation improvement of the severity of vulnerabilities results that more adequate resources can be delegated to tackle the issue.

2.1.3.4 Vulnerability Disclosure

We now address the common knowledge such as how are vulnerabilities disclosed once they are discovered. We also address how to maximise the effects of such a disclosure if, as an analyst, we want to provide the team responsible for maintaining the code with the most valuable information. Then it is up to the developers to assess if the vulnerability needs to be patched and how much resources to allocate to the resolution.

2.1.3.4.1 Types of disclosure Once a vulnerability is discovered, several option are offered to discoverer and/or its hierarchical superiors.

- It can be decided to not report it. This behaviour is named **Non-disclosure** and might originate from actors interested in taking a direct benefit in the exploitation of the vulnerability. Either by exploiting it themselves or selling it. Both of these behaviour are reprehensible by the Convention on Cybercrime (ETS No. 185) [50]. This document requires from all signing countries to implement laws repressing: "**illegal access**", "**illegal interception**", "**data interference**" (among other forgery and copyright protections) [...] " *when committed intentionally, [...] without right*" and generating "*[o]ffences against the confidentiality, integrity and availability of computer data and systems*".
  It may however be the approach of intelligence services up to a defined point. In the United States, for agencies to retain knowledge of discovered vulnerabilities, they shall plea their case in front of a Vulnerability Equities Process(VEP) [51]. Decisive elements regarding the eventual decision are the *"assessment of the usefulness to the communities"* receiving information about the vulnerability, and knowledge regarding if " *the vulnerability is currently being exploited on USG*[overnement] *or critical infrastructures...*" If decision is made to spread information, agencies publish advisory bulletins (as [52]) to stress attention on specific issues.
- In opposition to the first type, a complete transparent approach is sometimes carried out. Claimed benefits of **Full-Disclosure** are to force the maintenance team to take urgent action to tackle the issue and provide the discoverer(s) with its due recognition. It can sometime be called in the literature **instant disclosure** .
- Another approach, requires from the originators to contact the maintenance team and provide, confidentially, all information they hold regarding the vulnerability. The maintenance team can weight the severity of the exposure and allocate due resources to the issue. Originators and maintenance team agree on a schedule for a safe disclosure of the vulnerability greeting credit to the originators of the vulnerability.
- In case the discussion is complicated between both parts, a variation to the late **Responsible Disclosure** can be adopted. It requires the involvement of a Coordinator that will weight the risks and due benefit for both the originators and the maintenance team. This coordinator will be responsible of setting the calendar. It is the **Coordinated Disclosure**.

2.1.3.4.2 Usefulness of Disclosure We have considered so far that vulnerabilities are eventually disclosed, patched and with the patch delivered to all using instances. However it might not always be in the interest of the publisher and/or the users for vulnerabilities to be disclosed, as we will see in this section. We first discuss the modelling of vulnerabilities disclosures in a target program's lifecycle. Then we discuss work advocating for disclosure of vulnerabilities, before other works doubting it is

beneficial or even finding these disclosures to induce more cost. We finish with a section presenting articles providing advice to reach, or get as close as possible to, optimal disclosure conditions.

- **Disclosure Evolution**
  Some studies have attempted to estimate the rate at which vulnerabilities will be discovered over the lifetime of a program but it often results in the absence of clear trends over the software they analysed [53] . Only RedHat Linux6.2 provides a unique and distinguishable trend. It nonetheless persists complicated to extrapolate from these trends as the study covers a reduced period of time of 4 years. Investigations over several vulnerability discovery processes lead to discuss their applicability and their limitations [54]. It results that their model (AML [55]) generally outperforms Anderson Thermodynamics' one [56].
- **Positive Influence of Disclosure**
  The trade-off relies heavily on the vendor's reaction and internal policy when made aware of a vulnerability. It results that the disclosure multiplies by two point five times the probability of instant patch release by the vendor [57]. Another increasing factor is if Computer Emergency Response Team (US-CERT) is involved in a coordinated disclosure with, this time, the instantaneous probability of patch disclosure rising to three point nine times. However, if these disclosure make Open Source Software more responsive than average, authors could not measure eagerness from large vendors relatively to smaller ones. Covering the patch deployment process of 1593 vulnerabilities [58] , researchers note that for 77% of the vulnerabilities, the patching happens in the seven days before or after the vulnerability disclosure. 92% of these were patched within 30 days after disclosure. It then depends on the capacity to dispatch the correction to instances, and to users' behaviour to seek updates and accept them. Security analysts make almost three times less days to have 50% their devices patched than average user (13 days versus 36 days respectively).
- **Negative or undecided influence of disclosure**
  Some studies conclude that there are no evidence to consider the system more secure with time going [59] .
  Other studies [53] go further by stating that disclosure is affordable for vendor if the cost of Discovery by White Hats (WHD) is less than the probability of rediscovery by Black Hats time the sum of the costs of Black Hat discovery and White Hats discovery. In other words, the cost of malicious intrusion has to outweigh the consequent cost of public disclosure. Before disclosure, the number of actual attacks is increasing but remains low. Public disclosures induce the multiplication of exploits available online by 5 [60], hence generating a cost to users and therefore vendors. It is further confirmed that the most critical period is after the disclosure and before all devices could be updated [61]. Another publication [60] provides that the number of downloads of available exploits multiplies by a hundred thousand after disclosure, while the number of variants, potentially overcoming the patch, is multiplied by between one hundred eighty three to eighty three thousand. Earlier mentioned article [53] concludes that in situations where public disclosure costs less it has yet to compensate the required defensive effort in all situations, which is even further less guaranteed.
- **Conditions for optimal disclosure**
  Better than attempting to assess if the balance is negative or positive, assessing

conditions under which the policy of disclosure is optimal could prove a better use of resources [62]. Overall, full-disclosure (or instant disclosure) triggers a rise of the cost to users of three hundred percent to four hundred percent and is thus to be avoided if one cares about users. One factor maximising the users' safety is if a respected third party is involved in the disclosure and plans it (coordinated disclosure). It forces the vendors to take action in adequate time. Authors also find that giving too much time to the vendor does not guarantee an improvement in the quality of the patch. Authors estimate that if the vulnerability is considered to involve high customer loss then disclosure has for consequence to reduce the time before patch release from 17 to 12 days. In less impacting vulnerabilities, disclosure reduces the time before patch from 60 days to 48 days. However if the vendors' customers are considered mostly smart users, and only if *mostly*, the vendor may rely on them to provide a solution and to share it among other customers, without having to patch it themselves. The time before disclosure can be drastically reduced in those cases.

There is a general agreement that the daily cost after public release is more important than the cost when the vulnerability is only known by a few, no matter the majority of these happy few have malicious intent [63] [62]. Authors argue that time-driven release and time-driven updates is the unique equilibrium outcome of the game. Synchronicity is then permitted when the Firm or end-users tolerance is lower than twice the tolerance of the vendor. Vendor have to share the cost to its end-users (i.e., consider their loss its loss) but it has to share also liability (i.e., cannot have all the blame). It is not, however, a solution the authority responsible of disclosure will enforce if the market incentive is to minimise the cost of the vendor.

2.1.3.5 Fixing

The correction phase of a vulnerability can be done by the maintenance team or can be suggested by the originator itself. It then only takes from the maintenance team to integrate the provided solution in the code, shall the team agree the fix is suitable.

Still crucial, this step benefits from a consistent characterisation of the vulnerability and the complete exploration of its capabilities (i.e. its severity). A healthy and beneficial approach is for the fixing to be test-driven. Confronted to the specific triggering conditions, patch validation occurs when the test passes.

The same way it is done for more generic bugs, a field of research targets the capability of automating the patching process [64]. Generally Automating Program Repair (APR) will base on two versions of the code, one vulnerable and the other fixed. Pattern [65], properties [66] or *diff* are extracted and applied in case one vulnerability is recognised [67]. Nonetheless some techniques manage to augment the available set of patching editions by clustering looking-alike patches to produce, for one code, several variations of hopefully-fixed code. Then these new attempted patches are applied to vulnerable versions until the vulnerability can be evaluated fixed [68].

It opens to more complex techniques as Generative Adversarial Networks (GAN) in which two neural networks are competing, one attempting to produce a fixing patch, the second judging if the first presents characteristic of an actual patch more than generated code. This zero-sum game presents the advantage of not requiring paired data (vulnerable-to-patched codes) while providing the same results in optimal conditions [69].

Another approach attempts to compensate the lack of security related data by

using the proximity between bugs and vulnerabilities [70]. A Neural Network is pre-trained on large data sets of bug fixing data and is eventually more precisely tuned on vulnerability fixing data sets for C code. This transfer learning method enables an improvment from 18.2% of the resulting patch to actually fix the vulnerability to 21.9% for BigVul [71]dataset, and from 15.98% to 22.7% on CVEFixes [72] dataset.

2.1.3.6 Patch delivery

An extensive Systematic Literature Review(SLR) [73] provides that this step is subdivised in 5 sequences: 1. Info retrevial, during which organisation are made aware of the availability of a security patch, or keep their attention toward the publication of such a patch, 2. Vulnerability Scanning, during which organisation assess their exposure to the vulnerability the patch aims to correct, and prioritise the patch implementation, 3. Patch Testing, i.e. integration, 4. Patch deployment, 5. Patch Verification, e.g. supervision of the patch.

Main challenges concerns the lack of centralised resource for patch advisory publication, and the lack of details and/or reliability regarding the provided data, as also seen in Section 2.1.2. A consequence is that concerned actors struggle to get the information that they are exposed when they are. The deployment of security related patchs is confronted to requirements regarding the availability of features to end-users. Additionally, policies usually delay the update process. A mitigation is to hire skills and expertise, that is yet another challenge affecting the whole chain of events. Authors observe that most of the effort focus on the patch deployment, mainly through automating the deployment to using entities, and to develop the features of updating the software without needing to interrupt the availability of the software's features.

2.1.4 Existing study about the evolution of vulnerabilities in Programs and Operating Systems

The number of vulnerabilities affecting a system, and/or their evolution can indicate the exposure propensity of this system. Several studies have attempted to do so for several or for one system. Retracing these evolution, and specificity can provide lessons, signal specific exposure, and prevent the introduction of newest vulnerabilities.

2.1.4.1 Cross-Platform

In an empirical analysis from 2012 [74], focusing on web injection vulnerabilities, authors did not observe a significant increase in the complexity of attacks. They conclude it suggests developers are not learning from known errors. Another cross platform analysis [75] considers 4000 bug fixes for 3000 vulnerabilities over 682 Open Source Software from the NVD and over 11 years. Authors are able to compute that 50% of vulnerabilities have a lifespan that exceeds 14 months. Only 6.5% of studied vulnerabilities have a lifetime that is less than 10 days, with an average of 5 years. They cannot find either a correlation between the severity of a vulnerability and its lifespan.

2.1.4.2 Singular System

A similar conclusion is reached specifically concerning OpenBSD vulnerabilities [59]. It considers version 2.3 as the *foundation version* and computes that 62% of vulnerabilities reported between the release of the foundation version and the end of the study affect *fundational* lines of code. This shall be put in perspective with the fact that 61% of the lines of code of OpenBSD by the end of the study are from this foundational

release. No significant trend can be observed for, independantly, Windows NT, Solaris 2.5, FreeBSD 4.0 and RedHat Linux6.2 [53] .

The analysis of the Wheezy version of Debian denotes no maturity over time regarding security. The number of CVEs do not reduce and nor do any CWE disappear from the pool of newly disclosed vulnerabilities. The rate of vulnerability discovery do not compete with the increasing number of feature and their complexity: "*only chopping the tip of the iceberg*" [76].

2.1.4.3 Android

The first work to analyse Android related vulnerabilities is Jimenez et al [77]. Authors analyse, in 2015, 43 vulnerabilities disclosed from 2008 to 2014. The relatively small number self explains as Google hadn't yet the same transparency policy regarding their communication on security. Still, *Jimenez et al*, concluded that most of the vulnerabilities, by then where due to the code implementation (70%) rather than design or testing. That the most touched components were related to web browsing and cryptography.

Regarding the vulnerabilities of the AOSP project, the last version of the work of Mazuera et al. [78] gathers 472 vulnerabilities from 2008 until august 2017. They highly benefited from the monthly released Android security bulletin[5] since august 2015. Their data is available at **`https://ml-papers.gitlab.io/android.vulnerabilities-2017/appendix/index.html`**. If, we mention only 472 from the 1235 advertised vulnerabilities it is because we only focused on the vulnerabilities of the Android Open Source Project, for which the code is available at, and patches redirect to, `https://android.googlesource.com/`. The advertised one thousand thirty five include vulnerabilities from Qualcomm, NVIDIA graphic cards, Pixel devices or regarding the upstream Linux kernel. Elements for which vulnerability patches are not necessarily available.

They led a manual analysis regarding the vulnerabilities causality that made them change in 81,4% of the CWEs from the one provided by the website CVEDetails. We specify in Section 3.1 motivation to reproduce such a task further. Most of the vulnerabilities, to them, were related with permission CWE-284: Improper Access Control or CWE-275: Permission Issues.

[79] conducts an temporal extension that however considers CVEs' CWE to be the one provided by the NVD database up to 2019. If using this CWE is presented as not optimal by [78] at a 81.4% level, they further use fixing patches to find vulnerability patterns. They are nonetheless capable of assessing the complexity of vulnerable code. Also considering the Linux kernel, they conclude that it is the most affected area, but then comes Native Libraries and the application network. They find that 80% of vulnerabilities are related to less than two files to be changed. From 940 code fragments, they do a pair-wise comparison. Eventually they are capable of finding sixteen vulnerability pattern clusters, from address leakage to use-after-free, passing by missing Android Permission/UID checks. They claim six are yet uncovered in litterature: namely kernel address leakage, misretrieving android service, inconsistent android parcelable serialisation, incomplete C++ destruction, missing parameter and forgetting to set certain variables.

[80] also study the patchs but also keeps the NVD's CWE categorisation. Working on vulnerabilities up to January 2019 (indifferently AOSP and others elements such

[5] `https://source.android.com/security/bulletin/`

as Qualcomm), they observe general inconsistencies between available data sources regarding android vulnerabilities. They observed, on their specific pool of data, no reduction of severity of vulnerabilities. They characterise that a spike of vulnerabilities has been seen over API-level 24 (oct 2015). Also that there is usually an average one month window during which the patch is available but no device from considered vendors is patched when it comes to CVEs affecting the Linux Kernel and Qualcomm.

2.2 Vulnerability Detection

In this section, we list different approaches enabling to detect vulnerabilities in the code from lower to higher levels. We start from static analyses to symbolic and dynamic ones; before explaining several machine learning approaches. We eventually focus on machine learning algorithms tackling the detection of vulnerabilities at the commit level.

2.2.1 Static Analysis for Vulnerability Detection

The first approach we will discuss is Static Analysis which takes advantage of the source code of a program and unravels different possibilities and techniques to detect patterns leading to vulnerable faults, adding constraints to be verified and producing taint analysis. Beyond the availability of the source code, a second characteristic of static analysis is, as the name suggests, that the code is not run. The following listing heavily benefits from a literature review of 2017 that compares several open-source, security-oriented, Static Analysers for C and C++ [81].

A first group of static analysers evaluate constraints on statements to verify they do not breach code-security rules. First released in May 2001, **Flawfinder** performs static analysis of C and C++ programs and detects calls to a manually curated list of sensitive APIs [82]. Examples of such APIs widely recognised as sensitive are `strcpy`, `random` or `syslog`. **Splint-C** [29] is another static security testing tool, which performs lightweight analyses of ANSI C code and augments the code with annotations that set constraints on each C statement. It notably reveals the risks of buffer overflows, and alteration of the flow of instructions around loops and `if`s. Splint does not pretend to be complete nor sound but a good first pass at a very small cost. It was evaluated on BIND and wu-ftpd and uncovered a few buffer overflows, both known and by-then-unknown. **Frama-C** [83]'s static analysis (the tool can also produce concolic and dynamic analyses) will verify the values of the bounds of variables, pointers, among others, and verify if they stay inside safe bounds deducted from the code. **Cppcheck**[6], which specialises in finding undefined behaviours, and that strives to produce very few False Positives. **Uno** [84], that offers an approach aiming at detecting a limited number of errors, but with high precision. **Sparse**, that was developed by [85] specifically for the Linux kernel and thus can detect low-level errors in (among other things) bitfields operations or endianness.

Some other static analysers will rely on abstracting the code to detect known vulnerable patterns for the chosen level of abstraction. **Find-Sec-Bugs**[7] targets Web applications written in Java, and searches for potential vulnerabilities by matching high-level patterns that model problematic code pieces. Find-Sec-Bugs was made available to developers through a convenient IDE plugin. **Oclint**[8], performs analyses of Abstract

[6] `http://cppcheck.sourceforge.net`
[7] `https://find-sec-bugs.github.io`
[8] `http://oclint.org`

Syntax Trees. These code representations are then confronted to a library of patterns for dangerous code constructs. **Flint++**[9], can detect and warn developers about dangerous coding practices it detects. Flint adapts better Facebook's Flint program for C++. Flint, itself named after the Unix static analyser lint. **git-vuln-finder**[10] is another solution basing its analysis searching for patterns only in the commit message. Finally, another approach introduces a Query Language **PQL**(Program Query Language) of their own to search enable analysts to query for patterns of dangerous use in the code [86]. For instance, PQL has proven capable of detecting non-encrypted password hard-disk writing or possibilities left for a SQL injection among the 206 errors from 6 large open-source Java applications. The pattern research has also been successfully exploited by Matching Vulnerabity Patterns (**MVP**) tool [87]. The approach leverages ASTs and Program Dependence Graphs to increase the granularity to the function level of programs as extensive as the Linux kernel. Each function is normalised eliminate the influence of variable names, comments and other tabulations. They are then stored into a set of syntactic hashes (for each statement) and semantic (interaction between statements). In parallel, a library of twenty-five thousands vulnerability-related patches also follows the same abstraction process and computation in sets of hashes. Eventually, a set of rules, so as analysed functions have to look like the vulnerable version of the patch, without being too close to the patched version, determine the detection of a clone of a known vulnerability. Yet slower than relative tools, like **VUDDY** [88] and **ReDebug** [89], MVP reveals several new vulnerabilities in the same code. Out of the 97 vulnerabilities reported, 23 were already confirmed and attributed a CVE by the time of the publication.

Data-flow analyses also reveal beneficial when it comes to unravel vulnerabilities. Facebook's **Infer**[11], catches memory safety errors by building formal proofs of programs, and then interpreting failures of proof as bugs. Taint analysis allows to follow the path data travels inside a program to the sinks and deduce to whom are resources available. This process allows to uncover vulnerabilities that would not be detectable by analysing one function, one class, or one package at a time. Such approaches were proposed by **FlowDroid** [90] for Android applications in order to locate insecure use of data caused by the interactions of several software components. Yamaguchi et al. [91] demonstrated an approach that combines Abstract Syntax Trees (AST), Program Dependence Graphs (PDG), and Control Flow Graph (CFG). They were able to discover 18 new vulnerabilities in the Linux kernel. It inherits from **Deckard** [92] which vectorises ASTs and computes a hash for this vector. If the distance with a vulnerable code's vector is beneath a certain threshold, then both hash will have the same value. This method is called *Locally-Sensitive Hashing*. **BUGRAM** is a recent implementation that generates n-gram sequences and considers the least likely ones as a bug [93] . BUGRAM was run on 16 Java projects and found 14 confirmed bugs that other state-of-the-art tools were not able to find. [28] presented a framework available as an Eclipse plug-in to perform various static analyses (as points-to and taint-analyses). Their approach managed to find 29 security errors, two of which in widely used Java software: hibernate and the J2EE implementation. **Clang**'s analyser[12] produces Inter-Procedural Data-Flow Graphs that enable to find bugs such as memory

[9] `https://github.com/JossWhittle/FlintPlusPlus`
[10] `https://github.com/cve-search/git-vuln-finder`
[11] `https://fbinfer.com`
[12] `https://clang-analyzer.llvm.org/`

leaks, 'use after free' errors, and dangerous (though valid) type casting in C and C++ code. **CHUCKY**[94] identifies anomalous or missing checks on C programs. ANTLR based instrumentation enables a taint analysis that computes, in one target program, usual conditions applied to one function. When a similar call does not implement usual checks for this call (computed as a distance to usual behaviour), a flag can be raised. As the program is based on similar functions, the precision increases up to 96% when the set of neighbours reaches 20 functions, while 50% for 5 neighbours.

Overall, the literature review from 2017 [81] compares several of these approaches[13] both quantitatively and qualitatively: Frama-C appears as the most precise approach, Oclint as the tool uncovering most dangerous behaviours, and Cppcheck as presenting a very low false-positive rate.

A general study of static analyser has concluded that the porting to application of vulnerability detection heavily suffers from the lack of documentation of published and available analysers and an ergonomy not thoughts to be user-friendly [45]. It results that several vulnerabilities for which we have the knowledge on how to detect them, and we even have the tools to detect them, are not discovered because the time cost of learning to use the tool is too high.

2.2.2 Vulnerability Detection with Symbolic execution

Symbolic execution is a specific way to execute the code to determine the conditions on input variables that enable exploration paths in the code. Variables are initialised capable of being anything, and constraints are iteratively added to cover the code until a contradiction or an error condition is met. It can stand in opposition to using hard-coded-like asserting tests or being a preliminary step before getting a set of issue-provoking test-suit. Symbolic execution methods were notably experimented in 2008 by the tool **KLEE** which achieves around 90% coverage in its evaluation and found 56 new bugs, including ten in 89 COREUTILS utilities [27]. A good review of the use of Symbolic execution for software security was published in 2013 [95]. Are explained, for instance, different evolutions into Concolic testing, which executes the program alongside the path analysis and computes the next inputs to test to increase coverage based on stored conditions. Concolic stands as the mix of **conc**rete and symb**olic** execution. With Execution generated Testing, these evolutions enable a gain in precision in the generation of tests while potentially impacting the completeness of the generated test suit.

A reference regarding concolic execution is **DART** [96] (for Directed Automated Random Testing) which in 2005 describes the techniques with randomly initialised inputs.

MACE [97] uses model-inference to direct concolic execution. This approach improves the exploration of the state-space of programs, thus allowing to find more vulnerabilities than tools with less coverage.

More recently, the CIL—a C intermediate language—library has been leveraged to statically analyze the source code, allowing backward tracing of the sensitive variables [98] . Then, the instrumented program is passed to a concolic testing engine to verify and report the existence of vulnerabilities. Their approach focuses on buffer overflows and was reportedly unable to deal with nested structures in C code, function pointers and pointer's pointer.

[13] Frama-C, Clang Analyzer, Oclint, CppCheck, Infer, Uno, Sparse, Flint++

2.2.3 Vulnerability Detection with Dynamic analysis

Another essential technique for software security is Dynamic Analysis, where programs under test are effectively run and monitored.

First, Frama-C, an earlier mentioned tool, is capable of checking specified annotations in C code at runtime [99]. These annotations are of the type that Frama-C's static analyser's program is capable of generating.

Fuzzing, which automatically generates inputs and tests a program on them, has rapidly come to play a major role in software vulnerability detection. Fundamentally, a fuzzer is an infinite loop which mutates input seeds and launches the target program on the mutated seed. If the target crashes, a bug is detected. Manual analysis will tell if the bugs are vulnerabilities or not. **AFL** is a popular fuzzer for C/C++ programs [100]. Recent works [26, 101] use it as a reference. AFL instruments the target program to keep track of the coverage. If a mutated seed increases the coverage, the seed is mutated further. **FuzzIL** is a fuzzer for Javascript VM [102]. Like AFL, it uses coverage to rank seeds. **JQF** [103] or **Kelinci** [104] are coverage-guided fuzzers to test Java programs. AFL can be also found online to be implemented to fuzz on Android [105].

Google also implements their own fuzzer OSS-Fuzz [106]. Specifically dedicated to the detection of bugs in Open Source projects, OSS-Fuzz has discovered over five hundred thousands bugs as of June 2021. Developers of such projects can apply to the service, but to be accepted they must "*have a significant user base and/or be critical to the global IT infrastructure.*"[14].

Some of these fuzzers do not leverage coverage intel while still heavily benefit from target analysis. It is the case, for instance, when fuzzing an extensive target for which all the entrance points might not be documented and/or could be exploited in violation of the security policy. It is what **FANS** [107][15] does for the Android Open Source Project native services. Before fuzzing, the tool detects and extracts all the Interfaces enabling access to a service available from Android Interface Description Language (AIDL) files. The restrictions over variables are deducted on these calls so that the call is successful and the service runs. Then only will the fuzzer engine produce inputs given the set of restrictions and dependencies. Eventually, at the time of the publication, FANS flagged thirty vulnerabilities, twenty of which are confirmed by Google. Over the thirty, five concern libraries and three the Linux kernel. FANS, however, has yet several challenges to overcome as it does not use coverage information inside the services (hence no knowledge about untested parts of the services), runs as root, and only explores the normal domain of permissions of Android (thus does not cover risky behaviours Elevating the Privileges of the originating process).

As mentioned in Section 2.1.3.3 with Koobe [15], symbolic analysis and fuzzing can be combined to reveal not only a vulnerability but the actual severity of the vulnerability. In the case of CVE-2018-5703, the exploit was redirecting out of the mapping of the memory layout while the redirection was possible anywhere the attacker desires. It also matters as a wrongly assumed severity will impact the prioritising decision the team with an exposed program will take regarding the approach to follow to tackle the issue.

Another dynamic analysis method takes is collecting the different actions an exploitation undertakes and the resources this exploitation has to leverage to succeed. So-called **Attack-Trees** [108] are created, presenting, hierarchically, these goals, sub-

[14] https://google.github.io/oss-fuzz/getting-started/accepting-new-projects/

[15] `https://github.com/iromise/fans`

goals, and actions as nodes. They represent actions that have to be taken or sub-goals to be achieved in parallel and/or on independent paths until the root node is reached. This root node is the eventual goal of an attacker, implying successful exploitation. A system aware that certain sub-goals are achieved by an actor or a process, in a suspicious sequence, may, unknowingly to how precisely done, consider itself under threat and take planned actions. To diminish the level of false positives such a detection system might generate while not restricting the execution of harmless features, several metrics are suggested to produce **Enhanced Attack Trees** [109]. These metrics can be the probability of an attack given a sub-goal is achieved, or for actions to be only remembered for a specific duration, or to keep track of the attack levels already achieved once yet another sub-goal is accomplished. Alongside the challenge raised by the quality and quantity of attack-trees libraries, these techniques also require the implementation of an entrusted system-specific automaton capable of tracking the actions taken and sub-goals reached for each process. This automaton then reports the attacks with a certain confidence level (e.g. a probability). Eventually, an adequate countermeasure is triggered.

2.2.4 Machine Learning for Vulnerability Detection

The following subsection heavily benefits from the thorough Literature Review from Ghaffarian et al. [110].

By 2005, a first work proposed to extract features from code and metadata as premises of software vulnerability detection. In this work, *Sliwerski et al.* [111] famously stated that changes made on Fridays to the Mozilla and Eclipse projects were more likely to introduce problems than the changes made on other days.

Since then, a large body of work has been proposed relying on different learning techniques, targeting diverse programming languages and software systems:

In 2007, **Vulture** [112] gathers frequent patterns of imports and function calls. The data set includes a ground truth of vulnerable code with a training set twice as large as the test set. The application of a Support Vector Machine is reiterated forty times. The tool nonetheless ensures that the proportion of vulnerable code in both sets is the same. Eventually, Vulture managed to obtain a 70% precision on the Mozilla project, detecting vulnerabilities and pinpointing their location.

Another idea explored was to discover neglected conditions [113]. This approach begins with static analysis of the code to produce Enhanced Procedure Dependence Graphs. Heuristic Maximal Frequent Subgraph Mining algorithm (HMSM) identifies recurring patterns in relation with their frequency and provides them for manual review to the user. The evaluation finds 3800 violations of well-known patterns over four globally used programs.

Code complexity is among the most scrutinised feature for vulnerability prediction. In 2014, a measure of code complexity dating from 1976 [114] was implemented in order to detect the presence of vulnerabilities in Windows Vista [115]. The process applies a binary logistic regression to the system by considering, for features, the evolution of the considered binary, static code complexity, dependencies and metadata related to the developers involved in the binary modifications. With a threshold of 0.5, authors reach a precision of 0.64 for, nonetheless, a low recall on a ten-fold cross-validation. Other papers focused on the code complexity. *Shin et al.* [116] tried to focus on the correlation between code complexity features and the presence of vulnerabilities. The overall performance was only relatively conclusive as results demonstrated a correlation only in the case of

Firefox but not for the Wireshark project. Though, another paper [117] replicated this study with much more success using the same Bayesian Networks but only focusing on Firefox. Authors were also provided with more complete information through the allocated Common Weakness Enumeration of vulnerabilities (i.e., the vulnerability type). They even reached greater success changing either for IBK algorithm or Random Tree by Random Committee, reaching a Recall of 92% and a Precision of 98% for the latter case; yet still only on Mozilla.

As bugs may reveal actually to be vulnerabilities only years after their discovery, a branch of exploration for vulnerability discovery offered to mine bug databases [118]. The process starts through text mining and static analysis of bugs into binary vectors. These vectors can be later reduced by removing 10% of the features(i.e. dimensions) the less present in data set entries.

The idea is implemented only in another work, applied to the Linux Kernel for data between 2006 and 2011 [119]. A comparison between Naive Bayes, Naive Bayes Multinomial and Decision Trees for ten-fold cross-validation with 73 (so-called) *hidden impact bugs* and 6000 bugs produces a high recall for the two first algorithms but the best precision of Decision Tree.

In 2014, another approach [120] followed a vectorisation process based on frequency per file of words used in the source code [121]. The resulting evaluation of twenty Android Applications with both Naive Bayes and Random Forest was split into three steps. First, a ten-fold cross-validation for which both algorithms are neck to neck. Then a prediction by training on early data and testing on the oldest shows Random Forest having an advantage. However, the third evaluation, cross-project, by applying a model train on one application to others, presents Naive Bayes as the best performing algorithm.

DEKANT was proposed in 2016 to generate a model out of sliced pieces of PHP applications and WordPress plugins [122]. Standing for hidDEn marKov model diAgNosing vulnerabiliTies, the tool translates slices into Intermediate Slice Language for in-the-wild plugins and web-apps. It overall found sixteen zero-days vulnerabilities by the time of the publication.

Researchers have explored various code representations for learning vulnerability properties. [123] used machine learning on Control Flow Graphs. VALD (for Vector of Locally Aggregated Descriptors) [124] tokenises these CFGs. A Locally Sensitive Hashing [125] algorithm provides proximity on the hypothesis that close results are likely categorised the same. Their tool, **Genius**, identified 38 potentially vulnerable firmware, 23 manually confirmed. Similarly, Abstract Syntax Trees (AST) have been used as a feature and tokenised [126]. The resulting vectors are inputs of a deep learning classifier (Bi-LSTM for Long Short-Term Memory) to obtain a model of vulnerabilities. Evaluated on 457 vulnerabilities and over 30.000 non vulnerable portion of code, the result is that 80% of the top-10 ASTs flags are in fact vulnerabilities. A proportion dropping to 45% for the top-20 and to 30% for the top-50. However, for the authors' test-set, results using ASTs performs better than code metrics. Using bi-LSTM on ASTs has been re-experimented by another paper[127] but on different or enhanced data sets. This leads to better results specifically for data sets focused on buffer errors and, independently, on management error types. For both, the F1-score (combination of precision and recall) overtakes 0.8. However, the solution still performs poorly on other data sets included in the experiments.

An article [128] attempts to compare supervised algorithm to unsupervised. In

order to do so, authors investigate what features to consider for vulnerability detection. They conclude that the features do not affect the classification performance significantly. The supervised algorithms were only significantly performing better than unsupervised relatively in conditions where the quality of the labels in the data set was superior. Still, generally speaking about the field of Machine Learning detection, one of the key findings reported by the authors of the above-mentioned 2017 literature review is that the field of vulnerability prediction models was not yet mature [110].

2.2.5 Vulnerability Detection at Commit Level

A few articles try to address the issue of automated detection of vulnerabilities at the specific granularity of the commit level.

A first article [129] considers the relevance of detecting the commit that introduced a bug on the basis of the extracted change log message, terms used in the source code and both directory and file names. After tokenisation through a bag-of-words technique, a Support Vector Machine is applied to 12 well-known software projects (including Apache HTTP, Bugzilla, Eclipse, PostgreSQL, ...). *Change*s introducing a bug are manually confirmed, and the precision goes from 0.4 (for PostgreSQL) to 0.86 (for Bugzilla), while the recall goes from 0.43 (PostgreSQL) to 0.86 (for Bugzilla).

A study reveals why this scale of the commit level is relevant by analysing 68 vulnerabilities from the Apache HTTPD project [130]. Authors manually `git bisect` up to 124 Vulnerability Contributing Commits of these 68 vulnerabilities. The analysis reveals that these VCCs are usually larger than harmless commits. Also, new committers are more likely to introduce vulnerabilities and developers that have introduced a vulnerability already are more likely to introduce another one. Finally, their analysis reveals that 25% of vulnerabilities stay less than one year, and 6% may stay present for more than a decade.

A 2015 tool, named VCCFinder [30] applies the same principle on a self-made data set. If the tool is unavailable, several inviting details are provided, as they name their tokenisation tool or provide the SVM's tunning parameters. Another attractive characteristic is the discussion over the threshold of the resulting model. A team of analysts could tune this threshold to a rate of false positives they can afford. Another particularity regards the evaluation as they split their data-set from train-set to test-set temporally. It prevents training on more advanced coding and mimics conditions of in-the-wild detection.

Other works have directly mentioned and inherited from **VCCFinder**. A 5 pages technical report [131] aims at decreasing the number of false-positive results yielded by VCCFinder. To that end, the author proposes to separate additions from deletions in the commits to extract code-related features. The results presented in this technical paper may yet, but also only, slightly improve VCCFinder's performances.

VulPecker [132], might not properly use machine learning to detect vulnerabilities. However, it uses, per vulnerability type, machine learning to associate the most efficient code similarity algorithm to detect this specific vulnerability signature, and deduce the most significant features in this signature. VulPecker can choose between eighteen algorithms, including CP-Miner and variants [133]. A code is considered vulnerable if its signature is computed closer to the unpatched version of a vulnerability than it is close to the fixed version. This strategy provides the best of all eighteen worlds and brings, overall, the best results for both precision and recall in synchronicity. Eventually, Vulpecker detected forty 0-day vulnerabilities, eighteen of which were yet unpatched.

Still on the base of the work reported by VCCFinder, other algorithms and algorithm combinations have been experimented in the following years [134]. Albeit mentioning that VCCFinder uses LinearSVM, they only consider information from the commit message, gathered using regular expressions, and from bug reports. Contrary to VCCFinder, no information is taken from the patch code itself. Another difference with VCCFinder is that not only one algorithm is applied but six different algorithms(linear SVM, Adaboost, Gradient boosting, Random Forrest, Gaussian Naive Bayes, k-nearest neighbours). These algorithms are assessed on a k-fold detection during which the data set is split into k parts. Each slice is consecutively used as the test set, while the k-1 resting parts serve as training set. For each slice, the result for each commit of each of the six classifiers is fed to a logistic regression that concludes whether the commit is classified as contributing to the introduction of a vulnerability. For their specific data set, their **stacking** approach demonstrated an improvement of the precision of 0.12 (from 0.22 to 0.34) for a recall fixed at 0.76 compared to a method analogous to VCCFinder.

In contrast to these works, an approach [135] balances text and structural features tested the PhP code of phpAdmin and Moodle. Authors consider the issue as a *change classification* problem and develop a semi-supervised detection algorithm using both contextual features and structural (through the ASTs of the code). While the authors do not provide further details regarding the algorithm than mentioning their inspiration [136], one can retrieve from this reference that it is a mix of Expectation-Maximization (EM) and a naive Bayes classifier. EM is supposed to apply the naive Bayes classifier to unlabeled data and to steadily complement the training set with unlabeled data. This process continues until the parameters (or weights) of the naive Bayes classifier are not altered anymore by the the training set modification. Another difference with VCCFinder is the use of a specific data set, derived from a hand-crafted security-oriented data set [120] for PhP web applications. Authors conclude their approach, with their set of features improves performances from the Bag-of-Words technique, and that unsupervised classification has the potential to surpass the performances of a fully supervised classification regarding the problem of vulnerability detection.

VulDigger [137] focuses on the Mozilla Firefox project for automatically detecting vulnerability-contributing changes. Heavily comparing its approach to VCCFinder, the article, however, selects Random Forrest for classifier and also focuses on detecting vulnerabilities in the code of Mozilla Firefox. Another particularity is the capacity to whitelist significant code changes to detect vulnerabilities. This selection uses a Random Forrest regression to select (in the context of the authors) only 20% of the effort the analysis of all commits would require, and yet reaches a recall of 0.31. Nonetheless, author compare their results with the static analyser Flawfinder [82], as VCCFinder [30] does, but not with VCCFinder itself.

Given the split of features between patch related and related to the versioning, Sabetta et al. [138] trained two classifiers for the same data set of commits: each for a different type of feature already mentioned. The authors selected Support Vector Machine as it performs best according to the authors. The authors provide a list but they do not precise in detail the combination that could have been made for both classifiers. Both classifiers are tuned for high precision. To compensate for a low recall, a voting system enables that when any of the classifiers flags a commit, the commit is selected for review. The evaluation implements a ten-fold cross-validation on a data

set internal to the SAP company. The combination of classifiers, in comparison with both classifiers taken independently, increases both the precision at a fixed recall, and the recall for a fixed precision. Among both, the classifier tokenising the code through a Bag-of-Word technique performs slightly better than the classifier extracting from the log message. Another teaching of their approach is that their model outperforms another above-mentioned work [134].

Another approach regarding commits, given the heavy imbalance between VCCs and regular commits, is to impose prior filtering using per-language-list-of-terms as qualification [139]. In a 2019-master thesis [139] presenting this approach, up to 92% of commits can be discarded for the Linux kernel, and between 79% to 84% for other C-projects. Hence, it overall vastly reduces the effort to analyse the resulting commits. The authors also conduct a result comparison between a K-fold stacking [134] detection of vulnerabilities based on the commit message, with a Long Short-Term Memory classifier, and their own Neural Network classifier. On projects like FFmpeg, Qemu and Wireshark (among others), the choice of Neural Network on word2vec tokenisation performs the best. Neural Networks are actually a recommendation from Sabetta et al. [138].

Finally, even if they do not propose an ML-based approach to detect vulnerability at the commit level, [39] address the issue of reliability in labelled data taking VCCFinder as an example. They simplified the version of the project scrapper available online for VCCFinder, re-adapted the code to make it work regarding their focus and manually analysed the commits considered as VCCs. They conclude that only 58% of the commits that would be considered as ground truth, if they relied on VCCFinder's technique, are actually contributing to a vulnerability. This is an issue we did not have to address, in our later-explained attempt to replicate VCCFinder, since we attempted to replicate the performances presented in VCCFinder original paper using data provided by the authors. We did not have to check the validity of the ground truth construction method. The issue raised by [39] underlines an important problem for the field that had already been mentioned by [128] and mentioned in Section 2.1.2.

2.3 Defence Mechanisms

2.3.1 A little history of Defence Mechanisms development

One of the contributions of this PhD thesis is an investigation into the Android Open Source Project (AOSP) implementation of Defence Mechanisms over 10 years. Before detailing this contribution in Chapter 5, we introduce and define here the main defence mechanisms that can be found in AOSP. In particular, we will detail these defence mechanisms based on the following categories:

- a corruption of the memory,
- what to encrypt and how to encrypt it,
- authentification,
- access control,
- other categories of improvement.

We will present several of the main mechanisms that are implemented. We also define what behaviour they try to prevent, present a short historicity and present how they coarsely work.

2.3.1.1 Memory Corruption Prevention

Vulnerabilities related to Memory Corruption are among the most numerous and more impacting vulnerabilities. According to Google[16], memory safety-related vulnerabilities represent 86% of all vulnerabilities Android ever faced.

We will here shortly describe the history of Address-Space Layout Randomisation, Control-Flow Integrity and Fortify_Source.

ASLR: **A**ddress-**S**pace **L**ayout **R**andomisation is a technique that "*uses randomness to conceal regions of virtual memory address space of processes*" [140].

Instead of always allocating the same elements (such as the stack, the heap or libraries) to the same location in memory, ASLR distributes at a random address. Thus, exploits depending on the knowledge of the memory layout with hard-coded addresses, are unlikely to be successful.

The idea of randomising address layouts was introduced in 1997 [141], all main Operating Systems since adopted ASLR: OpenBSD in 2001, first as a feature and then as a default [142], Linux in 2005, Windows in 2006 [143]. ASLR—and attacks against ASLR—has a complex history [144]. It is ever since a cat and mouse game, still ongoing nowadays: each new attack calling for a new protection strategy. Indeed, ASLR can, in some cases, be bypassed, for example, by guessing the memory layout by repetitive attempts[145, 146]. Furthermore, a lack of entropy may lead to poor randomisation, and several specific methods can uncover parts of the randomised layout [147]. One main weakness is that if an attacker is in capacity of leaking the mapping between randomisation rounds, this very same attacker is capable of rewriting its attack accordingly to the new mapping and/or bypassing it [146]. A corollary to ASLR is **P**osition **I**ndependent **E**xecutable, which refers to the executable file's ability to be executed in a context (e.g. Operating System) enforcing ASLR.

CFI: **C**ontrol **F**low **I**ntegrity was introduced in 2005 [148]. Put simply, CFI aims to ensure at run time that a branching instruction can only jump (or call forward, [149]) or return (backwards, [150]) to a small set of addresses, that is known at compilation time. Indeed, in memory corruption attacks, it is common for the attacker to inject code to be executed into the target program. If the injected bytes are located in not executable sections of the memory, the attacker can instead input addresses of code interesting to the attacker in executable sections of the memory(called gadgets). However, for any code to be executed, the attacker has to trick the program to jump into the injected code or to gadgets. Consequently, CFI prevents attempts to hijack the flow of execution. However, CFI cannot always be 100% complete, i.e., the list of addresses that can legitimately be called may be an over-approximation of the actual list, due to imprecision in static analysis [151]. The CFI implementation presented in [151] has an overhead of 16% on average for the SPEC2000 benchmark, because of the necessary bookkeeping necessary to check whether code jumps are known and thus should be allowed to proceed.

Some mechanics such as Shadow Call-Stack can further harden CFI. This technique involves the creation of a copy of call-site addresses (i.e., addresses the Instruction Pointer headed to when hitting a jump). This copy is specifically not writable by the attacker (hence *shadow*). When a program hits a return instruction: if the addresses in the stack and in the copy do not match, the system is made aware of unusual behaviour. The program can then adopt a scheduled countermeasure to prevent the program from

[16] `https://source.android.com/devices/tech/debug/cfi`

further corrupting the system, usually terminating[152]. Shadow Call-Stack increases the overhead of CFI, unless it is backed by the architecture as **C**ontrol-flow **E**nforcement **T**echnology [153] now does.

FORTIFY_SOURCE: Implemented in GCC (**G**NU **C**ompiler **C**ollection) in 2004[17], "*FORTIFY_SOURCE works by computing the number of bytes that are going to be copied from a source to the destination. In case an attacker tries to copy more bytes to overflow a buffer, the execution of the program's execution is stopped*" [154]. It provides customisable levels of protection that may detect buffer overflows in some cases at compile-time, in others at runtime. Such behaviour is provided by changing several instructions such as `memcpy`, `memset` or `strcpy`, to specific substitutes capable of comparing size between input and output buffers. The capacity to detect overflows depends on the code syntax, and the level of checks asked at compilation time. It is not, however, foolproof but it can significantly prevent avoidable vulnerabilities from being released unnoticed. For instance, objects for which size is dynamically allocated at runtime are not covered by certain protections[18].

The Android development moved from GCC to LLVM's clang as default C compiler in March 2016 with the Android NDK r11. Consequently, the team also developed FORTIFY_SOURCE's features for clang in the following year [19] for full support since OREO (API 26). The original GCC macro still improves with oncoming update of the third level of hardening. This level improves the support of GCC's checks around objects with non-constant bounds[20].

2.3.1.2 Cryptography

The improvements regarding encryption are of two kinds: what to encrypt and how to encrypt it. The *what* is either related to data stored or transferred. The *how* either regards the protocol resorted to encrypt this same data, but also to what extent (e.g., encrypted as a block, by folder, by file,...)

TLS: **T**ransport **L**ayer **S**ecurity is the cryptographic protocol that most of nowadays peer exchange employ. It enables a client and a server to determine what algorithm they can use to safely exchange first encryption keys, then their data, away from eavesdroppers. It is not itself the original secure protocol the internet was based on, as it inherits from SSL's 1996 v3. Due to the unravelling of major weaknesses, SSL has been widely abandoned since 2014. Only 256 requests were enough to decrypt one byte of a cipher. Ever since TLS 1.2 and now 1.3 (defined in 2018) are considered standard protocol for communication.

Certificate pinning Authentication of trustable servers, to start a secure channel and transfer sensitive data with, relies on the involvement of **T**hird **P**arty **A**uthorities. The role of this authority is, as a intermediary of trust, to exchange keys and to provide certificate of servers' authenticity. A Man-In-The-Middle attack can however happen if a fourth party manages to install on the system as a TPA and signs its own certificates. Certificate pinning prevents this altered behaviour by pinning its Certificate Authority and asking verification of the issued certificate.

[17] `https://gcc.gnu.org/legacy-ml/gcc-patches/2004-09/msg02055.html`

[18] Example at: `https://p0lycarp.github.io/2019/fortify/`

[19] `https://android-developers.googleblog.com/2017/04/fortify-in-android.html`

[20] `http://patchwork.sourceware.org/project/glibc/patch/20211217040753.4176265-1-siddhesh@sourceware.org/`

2.3.1.3 Authentification

Authenticating mobile device users mostly rely on passwords, and still will in the foreseeable future [155]. This legacy method is weakened by the tendency of users to reuse passwords or to use passphrases from which the hash is easily guessable for [156][157].

Another drawback of passwords is that they are secrets that can be shared or eavesdropped [158]: they cannot guarantee that it is the owner who tries to log in the mobile device.

Unlocking patterns have also proved not to be foolproof with some study being capable of guessing 95% in less than the five attempts before the device locks [159].

Biometry: Access restriction to an exclusive list of users has since been made technically possible and more affordable by relying on user's physical attributes. For example, user's fingerprints can be used for authentication. Current techniques are not perfect and are under improvement to prevent spoofing [160] and for a more affordable price [161]. Other methods are also used as face and voice recognition.

SmartLock: However, authentication frequency happens to be sometimes considered more of a burden. Security enhancing efforts relying on users "*are not for free*" (i.e., it costs time which costs money) [162]. Consequently, certain people become reluctant to set any unlocking method for the sake of convenience. One solution consists in relying on an "automatic unlock" which unlocks the device only in situations of trust. These situations of trust can be based on the current location of the device or its proximity to a paired-with object. If at first glance, this might be a measure reducing the security standard, it globally increases the minimum security level. The only cases it reduces it are scenarii the user itself sets as safe.

2.3.1.4 Access Control

AOSP enforces 3 levels of resource access-control. A first mechanism enforces Unix permissions for processes following Discretionary Access Control(DAC). It differs from usual User-ID attribution as each process is provided with a different UID, rather than being granted the user's UID and permissions. Since API-level 18 **S**ecurity **E**nhanced **Linux** further protects components with a Mandatory Access Control [163, 164].

Last but not least, Applications follow another DAC process where permissions they hold are written in a Manifest.xml file. This means that, upon request, the system checks that the supplicant (e.g., API call, Content Provider or Intent) holds the appropriate permissions. Early versions of this permission system had a scarce, incomplete and sometimes inaccurate documentation [165]. As a result, it was troublesome for developers to know what permission they exactly require, and troublesome for users: a majority of users seemed not to understand the implications of the access they grant which reduces the attention granted to permissions warnings[166]. As described further in section 5.3.2, the Android Permission System has been thoroughly reviewed over years and versions. Improvements include the addition of different levels of granularity (read or write, at runtime only), better documentation, and improved display permission information to users.

2.3.1.5 Other

Updates deployment The developing team also provided a profound shuffling of the AOSP architecture to deliver corrective patches faster.

The scientific community accepts that the most costly period of a vulnerability's lifetime for both the vendor and users is between the public disclosure of a vulnerability and the on-device patch [61]. Called Project Treble, and unravelled with API-level 26, the vendor's low-level software core and AOSP's higher-level code are now better compartmentalised[21]. As a result, AOSP updates do not require the vendor's implementation and AOSP's to be recompiled as a whole. Updates can only be addressed to AOSP's framework as soon as these AOSP updates are released. These changes precisely match with security-related patches based on monthly Android Bulletins.

Rollback protection In our context, a version rollback attack is a technique where the attacker installs an older version of a system containing known vulnerabilities enabling to gain leverage over the data. Consequently, the attacker could exploit the vulnerabilities to elevate his privileges or gain control of the resources [167][168]. To prevent such attacks, tamper-resistant storage are installed in the file-system metadata and checked. These tamper-resistant storage hold the latest-most-advanced version tag and will prevent the device from booting if the version attempting to boot is older than the tag of the stored version. [22]

2.3.2 Security Enforcement

2.3.2.1 Low adoption

In an economical analysis of security in software market [23], the authors argue that vendors could produce more secure systems but have little incentive to do so as security could stand in the way of integrating features. To quote the original paper, the rule is rather: *Ship it Tuesday, and get it right by version 3.* Also, measuring security is a relevant and open question regarding one's software, but authors argue, there is little ground for buyers to trust vendors' claims straight as they are made. The authors then explores possibility of designing a vulnerability market or relying on insurers to assess these security markets. They have to abandon the latter has the cost of such insurance would be unaffordable for most if not all companies taking Microsoft as an example, given how widespread it is.

2.3.2.2 Patch delivery

Patch mechanisms shall be consolidated and automated with centralised software delivery to reduce the costliest phase between disclosure and on-device patching [58]. This suggestion is supported by the fact that relying on users has a cost (i.e., negative impact) that depends both on users' awareness and on the weight of the update procedure to these users. [162] clarifies this cost over the years in comparison to the real-world harm caused to users not following a secure practice. Hence, the difference explains why users reject or adopt some practices. Authors advise vendors to know their users' profile. This would enable these vendors to target their at-risk population and prioritise the security advice they communicate.

2.3.2.3 Android

Android Security Model

[21] https://android-developers.googleblog.com/2017/05/here-comes-treble-modular-base-for.html

[22] https://android.googlesource.com/platform/external/avb/+/master/README.md#Rollback-Protection

A review of the Android's Security was published in 2021 [164]. Their descriptions of the threat model and android security ecosystem cover up to Android API-level 28 (9.x). Nonetheless, explanation focuses on describing the different access control systems and their timeline. They also mention Treble's impact change for updates and briefly name memory corruption defence mechanism.

Vendors related security FUM security metrics (FUM for Free Update Mean) for affecting a security score to devices and to vendors were proposed in 2015 [169]. It combines the proportion of devices **F**ree of vulnerabilities over time, the proportion of devices running the latest **U**pdate of Android shipped by that device manufacturer, and the **M**ean number of not-fixed critical vulnerabilities that any device shipped by the manufacturer is affected by, from a list of 11 critical vulnerabilities. Intermediate relevance results are that 87.7% of devices are affected by at least one of these 11 critical vulnerabilities. Their study covers from July 2011 to July 2015 and concludes that the average security score is 2.83 over 10. Google's Nexus is the best performing device with 5.17/10, while LG is the best overall manufacturer with 3.97/10. Authors emphasise that the main issue is the update bottleneck: a patch may be released, but there are consequent delays before they are integrated by vendors in the next update and then downloaded by users. [169] serves as an accurate motivation for Google to change Android's architecture so as not to rely on vendors to update AOSP as it used to be before Treble.

[80] also noted that after Android adopted their Bulletins, some vendors such as Samsung and LG did the same in order to adopt the same benefit.

2.4 Android

In the following section we provide background information about the Android Operating System. We aim to provide all resources for the reader to accordingly understand following chapters of the thesis.

2.4.1 Context

Android is a recent Operating System based on the Linux kernel. In early 2007, the first smartphones emerged in an ideal technical and economic context. The update of the Third Generation of wireless mobile telecommunications technology by the International Telecommunication Union enabled, by 2006, the rate of data transfer to increase from 384 kbit/s (UMTS, 1999 Release) to 14 Mbit/s. This range of change enables access to many resources accessible through the internet, specifically data-hungry features such as online video feeds (e.g., video calls but now further with streaming). By now, and with 5G, specifications aim to overtake the Gigabit range per second.

In addition, the price drop of high-density, rechargeable batteries made possible a world of general-public embedded systems. Li-ion batteries were developed in the 1980's to start being sold in the 1990's [170]. Their development granted its inventors the 2019 Nobel Prize in Chemistry. Their price dropped significantly from over 6.000 euro/kWh in 1992 to around 600euro/kWh in 2006 [171]. The price kept dropping to 200euro/kWh in 2016 while the usage have expanded. The broad development of Electric Vehicles (EVs) has made the yearly produced storage capacity evolve from 20GWh produced in 2010, to 767 GWh in 2020 and growing[172].

A third main technological advance that boosted smartphones is capacitive touch-screen. In 1998, Wayne Westerman [173] published its doctoral dissertation based the contour of its gesture-recognition company Fingerworks. Concurrently, Zytronic, a

UK-based company, shifts its production to interactive touch sensor products, including projected-capacitive touchscreen (or p-cap) [174]. Touchscreen they will develop up to 2002, and their first commercialisation [175]. These capacitive touchscreens present the feature of being interacted with a finger, in comparison with resistive touchscreens requiring specific stylus. In 2005, Apple Inc would buy Fingerworks and use the patents to present, in January 2007, their smartphone with multi-touch capacitive touchscreen: the iPhone [176].

In this context, other vendors and usual leaders in the market of cell phones decided to unite around Google to form the Open Handset Alliance in November 2007. "*Mobile operators, handset manufacturers, semiconductor companies, software companies, and commercialization companies*" aim to develop open standards and make Android (a company acquired by Google in 2005) an open and free platform [177] with a first release in September 2008 [178]. This means anyone can access and download the source code[23]. It can be modified, compiled and flashed into one's device.

Adaptable to most handsets, open-source, fully operational out-of-the-box yet customisable by vendors, Android is the most installed handset OS globally. One characteristic is, however, that companies in the Open Handset Alliance cannot install on their device a fork of the original Android and aim to sell it if they want Google's specific features (Google APIs) to keep being provided to one of these companies' customers [179]. Only in specific countries does Apple's iOS compete, with the difference that iOS comes with the sale of the handset. Thus sales are balanced in the United States, in Japan and in the UK [180]. In Luxembourg, iOS is losing its advantage gradually. Yet, overall it is globally yet 87% of the handsets that are powered by Google's dedicated OS.

Smartphones were developed to interact with always more sensors and provide further more features. More accurate than cell towers triangulation, Global Positioning System (GPS) (or concurrent systems) enables, for the least, live navigation. Cameras make smartphones compete with dedicated hardware, and the OS shall adapt to specific device capacities such as slow-motion or timelapse; accelerometers enable tracking the carrier's activity and monitoring its health. Hence, each hardware Android is installed on provides a different configuration. Consequently, the OS needs to be flexible to provide all the features given the hardware it is provided with.

However, the requirements on the Android OS apply not only to the hardware it is installed on but also the hardware and network it is required to operate with. For instance, the connectivity through Bluetooth shall provide support to headphones, connected watches and other health trackers. The system is also required to interact with the cellular network, often different versions as a fallback, the WiFi and more recently Near Field Contact operations. The latter shall be provided a specific attention as the feature is specifically used for contactless payment [181, 182].

Hence, as many sensors and connection capabilities handsets are to be provided with, as many interfaces needs to be developed by the OS either for the use of the OS itself or applications the user intents to use. As such, we provide below how the Android Open Source Project provides such features and interfaces.

2.4.2 Android Architecture

Presentation The Android Open Source-Project is hierarchically organised so to deliver these functionalities, from third party-applications to hardware through the

[23] `https://android.googlesource.com/`

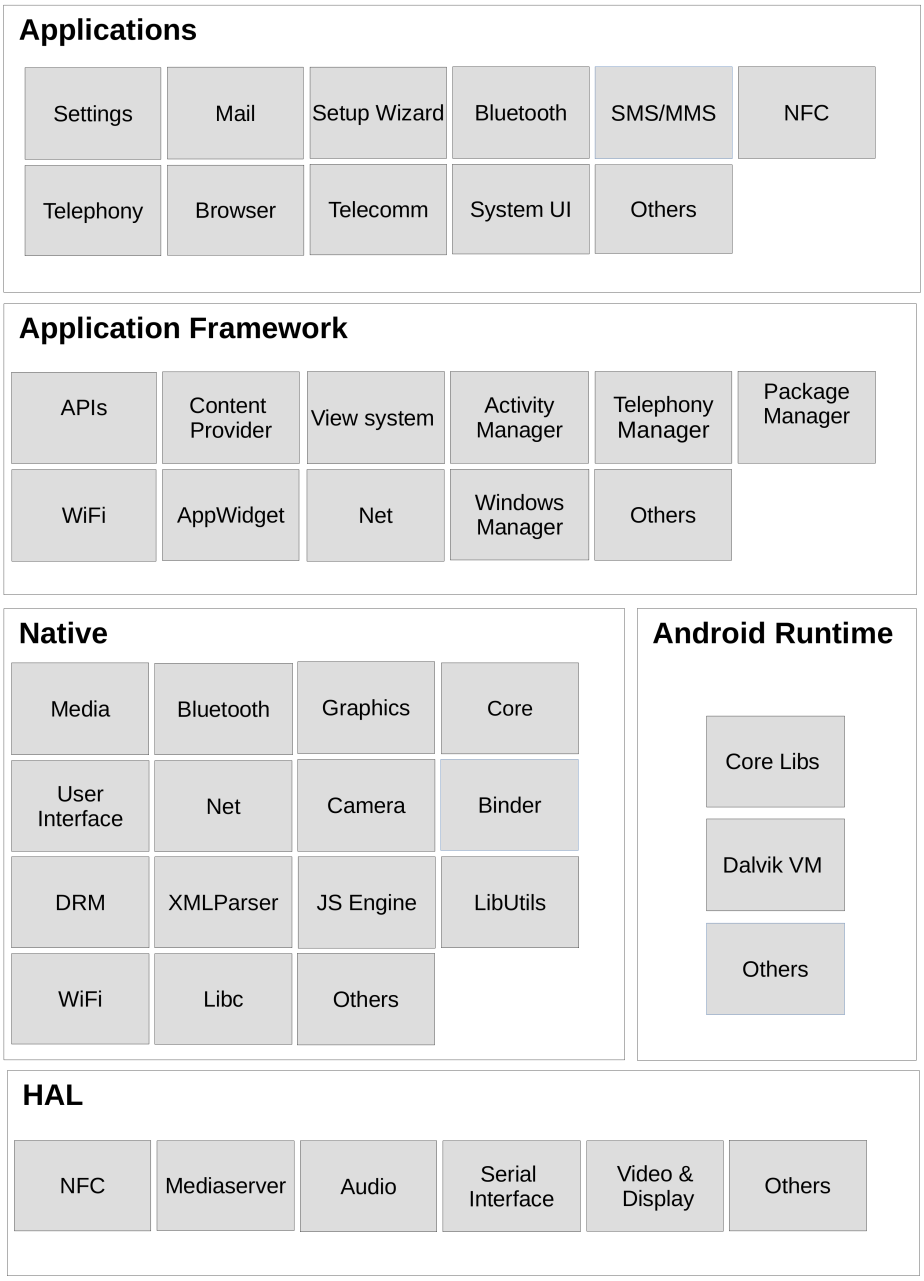


Figure 2.1: Android Open Source Project Platform Architecture

libraries and a specific implementation of a Java Virtual Machine, as presented in Figure 2.1.

- **Applications Layer** This layer holds all the applications, either from the system or from third-parties. System applications are related to standard activities expected from the OS and the device, such as calling, texting, emails access and web browsing. The user can, nonetheless, decide to install other applications for specific features and different experiences as, for instance, accessing social networks, banking services, video-streaming or playing games. Users can install these applications through several Applications Markets. Google provides the Play Store, that references all applications from third parties and manages their updates. To be referenced on the Play Store, applications must respect certain

rules and behaviours [183]. For instance, applications must be signed with a specific key attributed to its author(s). This signature protects the integrity of the code uploaded to the store and also makes the developer(s) accountable for the behaviour of it/their application. The signature schemes have evolved over time [184] but the process is now fully integrated into Application development through Android Studio [185]. It does not, however, prevent users from installing applications from other application markets, either these are proprietary [186], or referencing only free and open source applications [187]. The user can even download applications from the browser. In these cases, the user has to state that he/she authorises these sources to install an application on the device. Research attempting to treating with Android Applications in the wild can benefit from applications data sets, crawling several of these application stores and retrieving as many applications and as many versions of these applications they can through the AndroZoo repository [188].
These applications can be written in Java, kotlin and include compiled native code for acceleration. They can also include libraries developed and made available by third parties. Before upload, applications are compressed into a `.apk` format that includes all the code, libraries, and specification files as the Manifest. The Manifest will declare the API-level of the Android OS this application is built for. As Android evolves, features appear, new sensors might become standard; therefore, older handsets, not up to date, might not be capable of providing the features to either the OS or the application. The Manifest will also declare the list of permission it requires, either for the whole application or part of it. The granularity with which the user can grant these permissions is a point we will see later in this thesis.
In AOSP's source code [24], applications can be found under the path: `/platform/packages/apps/`.

- **Application Framework** This layer provides all the Application Programming Interfaces (APIs) that the Android OS provides to applications. This layer is sometimes referred to as the Java API Framework, as the code is written in Java. As clearly exploited by Guan et al. [189] and Chizpurfle [190], a service boots as, on the one side, a server to which, on the other side, applications will register. Applications and other components claim, to the Service Manager, for an interface that enables them to further request the service to perform specific work. At a lower level, these transactions are performed by the Binder Inter Process Communication (IPC). At this level, services provided are in the range of providing content such as media, from the device storage to the components, managing the activities lifecycle (which are the name of applications '*pages*'), or providing an in-app web browser (called WebView). One specificity of the work of Liu et al. [107] is that they automatically fuzz the android service so as to infer also the dependencies and the conditions for a services interface to be provided to an application component.
  In source code of AOSP [25], APIs can be found under the paths: `/platform/frameworks/base`, `/platform/packages/providers/`, and `/platform/packages/services`.
- **Android Runtime (ART)** Android Runtime's layer deals with the implemen-

[24] `https://android.googlesource.com/`
[25] `https://android.googlesource.com/`

tation of the applications in a specific, sandboxed runtimes environment. ART executes the `.dex` file present in the `.apk` compressed file. Dex stands for Dalvik EXecutable, with Dalvik the name of the former Just-In-Time virtual machine fulfilling the execution of applications for Android. ART has been implemented since Android 5.0 - Lollipop. ART provides with Ahead-Of-Time (AOT) compilation, and `.dex` files are converted into Executable and Linkable Format files while providing retro-compatible with older formats. A specificity in the creation of Java process is that they are forked from the same process called `Zygote` and are provided with the resources any specific application mightneed or request further. Android runs with OpenJDK's Java 8 language APIs[26].
In the source code, the art related code can be found under the paths: `/platform/dalvik`, `/platform/art`, `/platform/libcore`.

- **Native Libraries** This layer holds many libraries written in C and C++ that the system exposes to applications through specific services and interfaces. During the development of their application, developers require to use the Android Native Development Kit[27] (NDK). These libraries can be related to support Bluetooth, User Interface, produce or use 2D and 3D modelling in-app. These are, more precisely, Native Services for which FANS [107] extracts interfaces and dependencies. Also provided to applications since Android 8.1 (API-level 27) are Neural Network API support[28].
  In the source code, external native libraries can be found under the paths: `platform/external/`.
- **Hybrid Abstraction Layer (HAL)** This layer provides applications and APIs with interfaces to the vendor's hardware implementation. This layer is split into modules depending on the resource from which the system of applications might request a service. For instance, there is access to the hardware responsible for NFC connection, or access to storage such as an SD card. There is also the possibility to feed the hardware information for sounds (voice and/or music) to be displayed. These interfaces were described in HIDL format until Android 10 and are now supported in AIDL format (Android Interface Definition Language).
  In the source code, these modules are located at `/platform/hardware/`
- **Kernel** The Android kernel is based on Long-Time Support versions from the Linux kernel [191]. On top are installed a "*patches of interest to the Android Community*" [192]. The addition produces the Android Common Kernel (ACK). These patches of interest contain specific features required from android but not available or still under development upstream. As we will detail later: this layer has been reshuffled for versions above 5.4 (available since Nov. 25 2019 [29]) under the name General Kernel Image (GKI) [193]. One part of the kernel is common for all devices using the same kernel branch. Device-specific elements are better isolated into vendor modules. These kernel accesses these modules through the Kernel Module Interface(KMI).
  Elements of the kernel part of AOSP are available in the `/kernel/common/` folder.

[26] `https://developer.android.com/studio/write/java8-support`
[27] `https://developer.android.com/ndk`
[28] `https://developer.android.com/ndk/guides/neuralnetworks`
[29] See: `https://mirrors.edge.kernel.org/pub/linux/kernel/v5.x/`

2.4.3 Android Specificities

Android presents other specificities worth mentioning to understand the following thesis and that we will further describe below.

- **Different versions**
  Since the first version, in October 2008 [178], android has known different, almost always yearly, updates. We provide in Figure 2.2 the different, sometimes confusing, namings of Android and their corresponding dates as a reference.

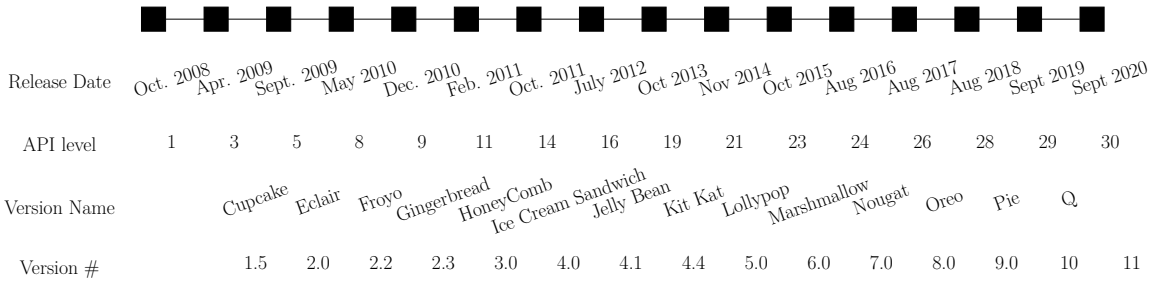


Figure 2.2: Android different versions

  We can also contextualise a few features with their introduction
  - Bluetooth is one of the first features available. If present in Android 1.0 for file sharing, notably, stéréo support will be available in Android 1.5, six months later [194].
  - The capacity to provide a web browser *inside* applications is also among the first features available [194].
  - The capability of the device to automatically rotate given the orientation of the screen dates from the same time.
  - Near Field Communication (NFC) started to be supported with Android 2.3 in Decmber 2010.
  - Support for Multi-core processor was implemented by Android 3.0 (API-level 11)
  - Face unlocking started to be possible by Octobre 2011 and Android 4.0 (API-level 14)
  - The Dalvik virtual Machine was substituted by Android Runtime Component in Android 4.4 (API-level 19)
  - Android implemented a service to mirror the on-device GPS and routing information on compatible cars starting March 2015.
  - Google Assistant has been introduced in Pixel phones in 2016 has an evolution from Google Now. It was made available to all devices in February 2017.
  - The display of several application in split-screen mode was implemented in Android 7.0 (API-level 24)
  - The Picture-in-Picture feature to display a video over other activities ,so to enable to complete 2 tasks at once, is available since Android 8.0 (API-level 26).
- **Applications** An application is downloaded from the playstore as an `apk` compressed file. Once expanded, the `classes.dex` file, folders with the resources (pictures, icons), build-properties file and the Manifest file can be accessed. This file shall declare the following elements [195]:
  - **Application package name**

  - **App components** as the activities (usually the windows the user can interact with), the services ("[used for] *longer-running operation while not interacting with the user or to supply functionality for other applications to use.*" [196]), the broadcast receivers ("*Android apps can send or receive broadcast messages from the Android system and other Android apps*" [197]), and the content providers (providing content to applications).
  - **Permissions** Either from the whole app, or for its components one application shall define the permissions it requests, either always or at runtime (in newest versions). One application can also define the permissions any other application shall hold so to use content, and features this application may provide.
  - **Features**, either software or hardware, that the app requires to be properly installed and not crash.
- **Permission System** Android develops a permission-based security model implying that any process, to be authorised to access a specific resource or execute a protected feature, shall be namely granted the right to do so [198]. As above mentioned, these permissions have to be listed in the Android `Manifest.xml` file, at the root of the apk file. Researchers deployed systematic tools to gather the complete list of permissions [199].

2.4.4 A few specific threats to Android

In the following subsection, we will briefly list a few proven threats to the Android OS. Not necessarily core to the following thesis, these threats enable to highlight the sources of threats to the system and entrance points to be monitored and to which specific attention must be paid to prevent abusing the system.

- Google's Play Store proposes several verifications on the code of applications [200]. For half a million apps daily, it includes undisclosed static, dynamic analysis, and machine learning [201]. Android developers have evoked to use logistic regression to reverse engineer malware and then train *"best models"* (as deep learning) with improved features engineering process [202]. Certain anomalies may still evade the detectors, resulting in either Malware, or other PHA (Potentially Harmful Applications), to be installed on the device. Thus a whole pan of research has developed so to find these applications, either properly malicious or other dissatisfactory behaviour (such as leaking data [203] or displaying advertisement [204]). New techniques developed can involve:
  - static analysis [205]: detecting logic bombs [206], finding leaking components [207], analysing the bytecode [208]
  - dynamic analysis:
  - or using machine learning [209]: among the most famous is DREBIN [210], that has since been investigated to explain the most significant features leading to the decision [24],
  - among other attempts to understand labelisation as malware and characterising [211]
- Specific applications may be copied and republished on application markets with an altered, and potentially malicious, payload. As applications include libraries, sometimes left aside for investigation, these libraries are particularly interesting for hiding payloads. Generally speaking, this practice is called piggybacking [212].
- Android can to use reflection and make native calls. This, conjugated with the

ability to access other applications systeme and services, make it possible to access resources and native code of other applications [213].

- The permission-based-security-model implemented may results in so-called *permission gaps* [199]. These occur when applications ask, in their Manifest, for more permissions than they need given the feature they aim to provide. This is further worrying given the possibility for applications to use other applications' code mentioned above.
- For Android 4.4 and below, **MADAM** [214] gets triggered by seven different types of malware given fourteen features. These features include application metadata, user actions, API calls, SMS sending, and system calls. On Application installation, seven malicious behavioural patterns have been set. Moreover during the use, the on-device system is trained on two K-nearest neighbours algorithms. These are trained genuine, in-the-wild application activity and confronted with artificially generated malicious behaviours. One classifier gets triggered by short-term suspicious behaviour, and the other by long-term suspicious behaviour. MADAM provides an accuracy measure of 96.9% based on three data sets, including malware: Genome, Contagio-Mobile and VirusShare. Authors argue False-Positives impact rather than usability than security and provide to integrate User Validated False Positives in the training set for improvement. The ground truth is established on a majority vote of VirusTotal classification.

3 Analysis of AOSP's Disclosed Vulnerabilities

In this chapter, we analyse over ten years of Android Open Source Project vulnerabilities. We first provide our methodology to gather the data using Data7. We then present our analysis approach, aiming to make the analysis explainable and the resulting data set updatable. In a second section, we discuss our analysis results from the analysis of about one thousand patched vulnerabilities we analysed on different aspects. We provide temporal descriptions of the evolution, as a discretisation per components of AOSP and per weakness type. We further provide a focus on the evolution of Memory Corruption vulnerabilities.

Contents

3.1 Motivation

Android is a leading operating system for mobile. Its particularity in the market is that its source code is made available via the Android Open Source Project (AOSP), an open-source initiative led by Google. Regarding AOSP security, valuable information can be found on the MITRE website about patched vulnerabilities. In addition, since 2015, Google regularly releases security bulletins documenting the recently fixed vulnerabilities. The scale and the open-source nature of AOSP, combined with this security information, offer the research community a rare opportunity to study the evolution of the security posture of a large and evolving software system spanning across its lifetime, i.e., in the case of AOSP, over more than 10 years.

The literature already includes several longitudinal studies on the security of software systems. For example, researchers applied security scanners on open-source systems to investigate the evolution of vulnerabilities over time (Debian [76], OpenBSD [59]). While findings of such studies offer relevant insights, they bear some threats to validity due to the inescapable false positives in static analysis. To overcome this limitation, other research works consider vulnerabilities reported and acknowledged in public vulnerability databases such as MITRE [1] or NVD [2]. In such databases, a *vulnerability* is generally defined as a software defect that may impact the security of the software, e.g., that can be *exploited* by an attacker [215].

Each known vulnerability is attributed a Common Vulnerabilities and Exposures (CVE) identifier. Additionally, CVEs are associated with a vulnerability type known as Common Weakness Enumeration (CWE). In a recent study [32], reported vulnerabilities metadata was proven significantly erroneous: among 10 000 CVE that were analysed, they had to correct information of the CWE field for up to 31% of the dataset. The heterogeneity of the data, specifically regarding Android vulnerabilities [80].

In prior studies related to Android vulnerabilities [216, 78] manual update of the information collected from CVE datasets have been undertaken. Due to multiple recurrences throughout our manuscript of these two related work, we will refer to them in the rest of the chapter as LV17 and MAZ19 respectively

LV17 and MAZ19 provide a substantial *manual* categorisation of Android vulnerabilities. Their dataset includes samples until June 2017. For each CVE, they manually associate a CWE. Nevertheless, the authors provide no information regarding the chosen risk model, i.e., the perspective/mindset under which they analyse vulnerabilities. Yet, information about this perspective is necessary to ensure a consistent *analysis approach* as recognised by the NIST[3]. To illustrate potential inconsistencies depending on the perspective/mindset, let us consider two examples of CVEs.

Example 1: CVE-2015-6599 is a vulnerability released in the October 2015 android bulletin[4]. The patch that provides a fix for this vulnerability is available[5] with the following commit message: "*libstagefright: check overflow before memory allocation in OMXCodec.cpp*". On the NVD website, this CVE is registered with the type "*CWE-119: Improper Restriction of Operations within the Bounds of a Memory*

[1] `https://cve.mitre.org/` or `https://www.cve.org/`

[2] `https://nvd.nist.gov/`

[3] page 15: `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf`

[4] `https://source.android.com/security/bulletin/2015-10-01`

[5] `https://aofndroid.googlesource.com/platform/frameworks/av/+/af7e33f6043c0be1c0310d675884e3b263ca2438`

Buffer"[6]

```
     CODEC_LOGV("allocating %lu buffers of size %lu on %s port",
             def.nBufferCountActual, def.nBufferSize,
             portIndex == kPortIndexInput ? "input" : "output");

+    if (def.nBufferSize != 0 && def.nBufferCountActual > SIZE_MAX / def.nBufferSize) {
+        return BAD_VALUE;
+    }
     size_t totalSize = def.nBufferCountActual * def.nBufferSize;
     mDealer[portIndex] = new MemoryDealer(totalSize, "OMXCodec");
```

Listing 1: CVE-2015-6599 Patch Code

In LV17/MAZ19, the authors do not consider the original *CWE-119* and instead manually associate two CWEs to this vulnerability: (1) A generic *CWE-264: Permissions, privileges, and access control*[7] which is a category unused since 2016[8]; and (2) a more specific *CWE-265: Privilege/Sandbox Issues*, a category that has since been renamed as *Privileges Issues*.

With this example, we first notice that LV17/MAZ19 refers to a CWE that is now obsolete, a situation that calls for an update of previous studies. Moreover, and more importantly, their CWE manual categorisation is focused on the *consequence* rather than the *cause* of the vulnerability. Indeed, "Privilege issues" refers to the *impact* of the exploitation of this vulnerability. In some cases, however, the analysts who must fix the vulnerability are looking to understand the root cause of the vulnerability. Hence, the patch code presented in Listing 1, would lead to a different categorisation related to the integer overflow that is fixed. The assigned CWE would then be *CWE-120: Buffer Copy Without Checking Size of the Input*.

Example 2: CVE-2016-2448 has been disclosed in the May 2016 Android Bulletin[9], and associated with *CWE-264: Permissions, Privileges, and Access Controls* on the NVD database[10]. The patch is publicly available[11] and the commit message is: "*NuPlayerStreamListener: NULL and bounds check before memcpy*".

In LV17/MAZ19, the authors decided on *CWE-120: Buffer Copy Without Checking Size of the Input*, which can only be attributed when considering the *cause* of the vulnerability and by analysing the fixing patch (Listing 2).

As is visible from these two examples, a given group of authors may very well, in the same study, attribute CWE sometimes based on the *cause*, and sometimes on the *consequence* of a vulnerability. Such inconsistent labelling will necessarily lead to issues with the systematisation of knowledge, and to a lesser extent with reproducibility. Furthermore, the security community has concluded that using different perspectives/mindsets to analyse a vulnerability can lead to diverging conclusions [217].

Unfortunately, the authors of LV17/MAZ19 do not precise their analysis approach, and we have found evidence for alternative uses of different approaches. In order to overcome these limitations and shortcomings, we propose to come up with our own

[6] https://nvd.nist.gov/vuln/detail/CVE-2015-6599

[7] As reported in the Top Level Type field at https://ml-papers.gitlab.io/android.vulnerabilities-2017/appendix/EMSE2018/vulnerabilitiesList.html

[8] https://cwe.mitre.org/data/definitions/264.html

[9] https://source.android.com/security/bulletin/2016-05-01

[10] https://nvd.nist.gov/vuln/detail/CVE-2016-2448

[11] https://android.googlesource.com/platform/frameworks/av/+/a2d1d85726aa2a3126e9c331a8e00a8c319c9e2b

```
@@ -144,8 +144,17 @@
         copy = size;
     }

+    if (entry->mIndex >= mBuffers.size()) {
+        return ERROR_MALFORMED;
+    }
+
+    sp<IMemory> mem = mBuffers.editItemAt(entry->mIndex);
+    if (mem == NULL || mem->size() < copy || mem->size() - copy < entry->mOffset) {
+        return ERROR_MALFORMED;
+    }
+
     memcpy(data,
-           (const uint8_t *)mBuffers.editItemAt(entry->mIndex)->pointer()
+           (const uint8_t *)mem->pointer()
             + entry->mOffset,
            copy);
```

Listing 2: CVE-2016-2448 Patch Code

Table 3.1: Disagreement Table

	Exact Match	Close Match	Following Nodes	Deprecated	7-PK	CWE-391	Clear Dissociate	TxUnclear	In Common
#	241	89	1	17	5	30	78	9	470
%	51.27	18.93	0.21	3.62	6.32	1.06	16.60	1.91	100

systematic and fully disclosed methodology to attribute CWEs to vulnerabilities. We applied this methodology to all the vulnerabilities in our dataset, even for the 483 vulnerabilities that were also attributed CWEs in LV16/MAZ19.

Our methodology for mapping AOSP's CVEs to CWE is presented in Section 3.2.2.1. We map 987 CVEs, 487 in common with MAZ19's study. The rest is an AOSP-focused temporal update. Eventually, we found that our categorisation is matching that in LV17/MAZ19 for about 70% of the 487 shared CVEs (see Table.3.1). In 52% of CVE cases, the categorisations perfectly match. For the rest, the agreement is partial due, for example, to mapping to CWE types that are no longer supported. We disagree over 16.6% of vulnerabilities in common, which differences in analysis approaches can partially explain.

In this chapter:

We investigate the evolution of the publicly disclosed vulnerabilities that affected the Android Open Source Project. We manually analysed the 987 vulnerabilities that were publicly released until June 2020. Such an investigation has been started by peers until June 2017, that we reactualise up to June 2020 with most recent data and up-to-date CWE list.

We aim to improve reproducibility and enable potential future exploitation of our analysis of AOSP vulnerabilities by differentiating from the existing related work by providing a precise analysis approach. We also express clearly the set of categories we use for our categorisation, following the NIST's *Guide for Conducting Risk Assessments* [217] *vulnerability oriented*.

We evaluate the trend of these disclosed vulnerabilities and confront them with the effort produced by developers to harden the host system.

The main contributions of our work are as follows:

- We provide a precise methodology for a manual analysis of 987 AOSP vulnerabilities based on their related patch fixes.
- We provide several analyses for the evolution of these vulnerabilities over time

Table 3.2: Number of Vulnerabilities (i.e., CVEs) considered in our study

NIST only	NIST + Android Bulletins	Final Set
37	950	987

and across AOSP. These analyses also cover the evolution of the type of these vulnerabilities.

The link to CVE-to-fix could be useful to recent promising approaches [87] that leverages fixing patches to find known vulnerabilities in the code. Their tool MVP improves from previous state-of-the-art ReDeBug [89] and VUDDY [88]. The precision in the categorisation could fulfill the teaching from *Goseva et al.* [128] that supervised detection of vulnerabilities only outstands unsupervised methods with the quality of the training data labelling.

3.2 Vulnerability Analysis Methodology

In this section we first explain how we gathered data for our study. Then we detail the systematic approach we followed to manually categorise vulnerabilities.

3.2.1 Data Retrieval of Android Disclosed Vulnerabilities

To collect data related to Android vulnerabilities, we consider the Common Vulnerabilities and Exposures (CVEs). As defined by the MITRE Corporation, a CVE is identified by a string *CVE-Y-X* where Y is the year when the vulnerability has been discovered and reported to MITRE. X is an incrementing id number over the year. All the CVEs are present in a list maintained by MITRE, which is used to populate a database named NVD maintained by NIST[12]. In the remainder of this work, we will refer only publicly disclosed vulnerabilities and thus have a CVE number. Therefore, in practice, we will use *vulnerability* and *CVE* interchangeably.

In practice, we collected data related to Android vulnerabilities from the NIST NVD database, the primary resource we rely on for vulnerabilities disclosed from 2007 to 2015. Starting from 2015, we complement the information provided by NIST thanks to the *Android Bulletins*. These bulletins are Google's monthly reports providing information about the last patched vulnerabilities in AOSP. A significant strength of the Android bulletins is that they provide key information regarding a vulnerability: the link to the patch that fixes it. This patch is central for our study since we use it to categorise its associated vulnerability (cf. Section 3.2.2). In particular, the changes that have been applied to transform a vulnerable piece of code into a non-vulnerable one are helpful to categorise the vulnerability. Note that the patch fixing a vulnerability is sometimes also available (directly or indirectly) from the NVD database information. However, this information is often not directly available and requires extra effort by navigating related web pages and blog entries.

Table 3.2 reports the total number of vulnerabilities that we consider in our study. Overall, we collected 1007 vulnerabilities related to the Android framework.

3.2.1.1 Data7

To gather all the vulnerability information regarding the Android Open Source Project (AOSP), we relied on a customised version of Data7 [218]. This tool has been

[12] `https://cve.mitre.org/about/cve_and_nvd_relationship.html`

designed to collect and combine several sources of information related to vulnerabilities. In particular, given vulnerability reports such as NIST's data feeds, Data7 automatically outputs a structured dataset that connects vulnerabilities to the patches that fix these vulnerabilities. Note that collecting these patches is not trivial. Indeed, dedicated scrappers need to be developed to follow the url links provided in the vulnerability reports.

For each vulnerability of a given software project, the output dataset contains the following information:

- CVE number
- description
- CWE number (if applicable)
- time of creation
- time of last modification
- CVSS severity score
- bug ids (if existing)
- list of impacted versions of the software project
- list of commits that fixed the vulnerability which contains:
  - hash
  - timestamp
  - message
  - fixes (files in their states before and after fix)

In order to apply Data7 on our target sources of information (i.e., NIST and the Android bulletins), we extended Data7 on two fronts. First, we updated the input parser of Data7 to not only consider *xml* files as input but also *json files*, json being the updated format of the data provided by NIST[13]. Second, we implemented a new module specific to the Android bulletins in order to scrap and gather the patches fixing vulnerabilities.

3.2.1.2 Android Bulletins and NIST: Vulnerabilities disclosed from 2015

Since the disclosure of CVE-2015-1538 in August 2015[14], Google releases monthly security bulletins called *Android Bulletins*. A bulletin released in a month n provides the list of CVEs affecting Android that were patched during month $n-1$. The next update provides the users with these fixes. Not all vulnerabilities present in the Android bulletins are of interest for our study because they are not all directly related to AOSP. Indeed, vulnerabilities affecting, for instance, the upstream Linux kernel, hardware exploitation modules (Qualcomm, Mediatek, lgmtk) or specific Google devices (Nexus and Pixels) are included. To only keep AOSP-related vulnerabilities, we rely on the "location" of where the patch correcting a vulnerability has been applied. Concretely, we considered in our study all available vulnerabilities that have been patched in the Android repository `https://android.googlesource.com/`. Patches that, for instance, affect the upstream Kernel, i.e., redirecting to the Linux kernel repository `git.kernel.org` have been discarded.

As shown in Table 3.2, overall we collected 950 CVEs from the Android Bulletins for which we could also obtain the matching correcting patch.

[13] `https://nvd.nist.gov/General/News/XML-Vulnerability-Feed-Retirement`
[14] `https://source.android.com/security/bulletin/2015-08-01`

3.2.1.3 Completion for vulnerabilities disclosed from 2007 to 2015

Before the introduction of the Android bulletins, i.e., from 2007 to 2015, the NIST NVD database and the associated feeds were the main sources of information regarding vulnerabilities related to AOSP. We downloaded and processed the `json` file associated with each year from 2007 to 2015 present on this NIST web page[15]. To only retrieve vulnerabilities related to Android, we first check for each vulnerability if the field[16] providing the list of affected systems contains the string `android`. Overall, this methodology enables us to gather 57 vulnerabilities.

This list of 57 vulnerabilities required additional manual filtering. Indeed, among those 57 vulnerabilities, 5 are either related to a third-party application (e.g., Symantec application) or to specific smartphone suppliers (HTC, Motorola and LG).

Eventually, we were capable of retrieving the fixing commit for 37 vulnerabilities out of these 57 CVEs.

3.2.2 Categorisation of Vulnerabilities

In the case of software vulnerabilities, one critical piece of information is the categorisation into **C**ommon **W**eakness **E**numeration (or *CWE*). CWEs constitute a catalogue of over 600 categories of issues maintained by MITRE, that can be related to security, allowing a high-level tree-structured categorisation of vulnerabilities. MITRE advertises that three roots at the base of their internal hierarchical representation permits to reach all CWEs.

As highlighted in [37], categorisations of CVEs by CWEs can be seen as subjective. It has been shown that errors—or even noise—in the data used to train AI approaches can greatly impair vulnerability detection performance, particularly in real-world settings [33]. The NIST advises analyses to clearly state the chosen risk model (see Section 3.2.2.1). Otherwise, the very same vulnerability can be attributed to a different category depending on whether the analysis focuses on (a) the source threat, (b) the assets that are impacted at the end of the exploit, or (c) the moment (the lines of code) at which the behaviour would deviate from the expected one.

We thus have to retrieve those CWE with the entangled relationships, before stating a clear reproducible risk model.

3.2.2.1 Categorisation Process

As explained in Section 3.2.1, we collected over 987 AOSP CVEs. The Mitre CVE webpage also provides a description of the vulnerability and links to external resources. Sometimes, these resources (e.g., Proofs-of-Concept, exploits, issue trackers entries, academic presentations or forum discussions) may be unstructured and of varying quality and depth, and thus may not always allow us to understand the vulnerability appropriately nor to bring enough confidence regarding our analysis [80].

Hence, we focus our analysis on the patches the Android Bulletin provide.

1. We first need to describe our analysis approach. Such a categorisation depends on the exact question an analyst tries to answer when attempting to analyse a vulnerability. The *analysis approach* is an essential point, as stated by the NIST[17]:

[15] `https://nvd.nist.gov/vuln/data-feeds#JSON_FEED`

[16] This field is `cpe23Uri`, where cpe stands for **C**ommon **P**latform **E**numeration

[17] See page 6-7 and 15 of `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf`

> "By making explicit the risk model, the assessment approach, and the analysis approach employed, [...] organizations can increase the *reproducibility* and *repeatability* of risk assessment".

In the same document, the NIST lists three approaches:

(a) **threat oriented**, focused on the development of *threat scenarios* and the impact identified based on *adversary intents*

(b) **impact oriented**, focused on the impacts on critical assets and identifying *threat events that could seek those impacts or consequences*

(c) **vulnerability oriented**, that starts with a set of predisposing conditions or exploitable weaknesses. It identifies threat events that could exercise those vulnerabilities.

To achieve the goals of repeatability and reproducibility, we state that our analysis follows the vulnerability oriented approach.

Furthermore, from a predisposing condition in the code to its exploitation, there can be a chain of events, or *threat events*. Each of these events, when taken individually, may lead to a different categorisation. We must then also state that we consider the code before the patch, at the location of the patch, as the *predisposing condition* toward the exploitation of a weakness.

We formulate the question summarising our analysis approach as follows:
How does this patch prevent the unwanted behaviour from happening again?

An concrete example of how this matters is provided further below, in Section 3.2.2.2.

2. To choose the right CWE, we use Mitre's website dedicated to CWEs[18]. We expand the three *CWE domains* that are:

```
CWE-699:  Software Development
CWE-1000:  Research Concepts
CWE-1194:  Hardware Design
```

This hierarchy (we might refer to as CWE trees) goes from the top to the bottom as from the broader to the more specific weaknesses. For each CWE tree, we select the more general categories that could match the kind of vulnerability the patch aims to prevent. We confirm with the description. For each of the selected CWEs, we try to find more specific CWE branches in the hierarchy.

3. We do this manually for each branch of exploration until we cannot go any deeper, i.e., until available sub-types do not match anymore with the vulnerability we are analysing. We keep all these general CWEs, before more specific ones can be found no more, as potential candidates.

[18] `https://cwe.mitre.org/`

4. We have then a list of potential weaknesses that we confront. CWEs's descriptions are necessary as they provide a stricter frame than the title of the CWE sometimes does. Focusing on the behaviour the patch prevents, one CWE is selected and attributed to the considered CVE.

For each CVE, we provided a CWE before looking at the attributed category by MAZ19. It prevents our analysis from being biased, at the cost of a potential increase in the number of differences. Afterwards, and in case of a mismatch, we made sure to re-analyse concerned vulnerabilities to decide if we stand with our conclusion or if we are eventually more convinced by the related work's analysis.

Eventually, we were able to provide a CWE category following our categorisation process to all but four vulnerabilities.

3.2.2.2 Example of categorisation

We provide in Listing 3 the patch that fixes the vulnerability CVE-2017-0382.

diff –git a/libs/binder/Parcel.cpp b/libs/binder/Parcel.cpp
index e88ae29..19ce3eb 100644

```
@@ -548,7 +548,7 @@
        // grow objects
        if (mObjectsCapacity < mObjectsSize + numObjects) {
            size_t newSize = ((mObjectsSize + numObjects)*3)/2;
-           if (newSize < mObjectsSize) return NO_MEMORY;   // overflow
+           if (newSize*sizeof(binder_size_t) < mObjectsSize) return NO_MEMORY;   // overflow
            binder_size_t *objects =
                (binder_size_t*)realloc(mObjects, newSize*sizeof(binder_size_t));
            if (objects == (binder_size_t*)0) {
```

Listing 3: CVE-2017-0382 Patch Code

Analysing the code before the patch, we deduce that an Integer Overflow may happen in lines 552-553 with the expression `newSize*sizeof(binder_size_t)`. Then the new buffer would have a size smaller than the required space. When the elements are copied, they overflow the too-small space allocated in memory. However, the patch focuses on another line. On line 551, the original code attempts already to check if the calculated `newSize` suffers an integer overflow by comparing with `mObjectSize`. This verification forgets that the computation on line 553 (`newSize*sizeof(binder_size_t)`) is another threat for the value to overflow. The patch corrects by calculating the appropriate size of the buffer.

To summarise, the miscomputation of the size of the buffer (`CWE-131`) paves the way to an integer overflow (`CWE-190`). The latter enables a buffer overflow (`CWE-680: Integer Overflow to Buffer Overflow`). As the patch focuses on correcting actual buffer size calculation, relatively closer to the design level of the code, we conclude to `CWE-131: Incorrect Calculation of Buffer Size`.

Note that if we were focusing on the impact on the assets, we would conclude to a `CWE-680: Integer Overflow to Buffer Overflow`, or just to a `CWE-787: Out-of-bounds Write` (which are both sub-types of `CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer`). Also, if the pre-patch version was not so obviously verifying an integer overflow on a value that is not the eventual size of the buffer, and if we focused on the *adversary intents* (see Section 3.2.2.1), we would conclude to Integer Overflow alone or Integer Overflow to Buffer Overflow.

3.3 Analysis of 10 years of AOSP Vulnerabilities

In this section, we discuss AOSP vulnerabilities over ten years. We pursue several objectives, the first of which is to present the results of our manual categorisation and the profile of the resulting dataset.

These observations may lead to insights regarding the security profile of AOSP. A similar study was carried out regarding the Long Term-Support (LTS) Debian Wheezy [76]. They specifically focus on whether the version achieved *maturity*. They hope the duration of the support might enable system hardening.

We present a temporal evolution of the AOSP vulnerabilities in Section 3.3.1. In Section 3.3.2, we use Mitre's data to discuss the evolution of the number of AOSP versions impacted by each vulnerability. Then, we identify how vulnerabilities are distributed among AOSP's software packages/modules in Section 3.3.3. We also observe the temporal evolution of vulnerabilities' severity affecting AOSP in Section 3.3.4. In Section 3.3.5, we classify the vulnerabilities by type to understand which affects the AOSP system the most and what conclusion we can draw from the temporal evolution of those vulnerability types. Eventually, in Section 3.3.6, we focus on vulnerabilities affecting the memory, i.e., that rely on corrupting allocated memory blocks. The results of our manual categorisation are accessible at `https://github.com/TimUniLu/TimUniLu-Android_Vulnerability_Analysis`.

3.3.1 General Evolution of Vulnerabilities

In the following section, we discuss the temporal evolution of android vulnerabilities to investigate the following research question:

RQ: *How has evolved the disclosure of AOSP vulnerabilities ?*

3.3.1.1 Description

Figure 3.1 presents the temporal evolution of publicly disclosed vulnerabilities we analyse. The date of public disclosure considered for each vulnerability is the earliest we could find between the date of the bulletin or the registration date of the CVE on the CVE Mitre website, as other works assumed [32]. The graduations of the `X-axis` represent a three-month period each. This 90-day[19] period is generally accepted as the standard for a responsible disclosure. *Google Project Zero* itself applies this 90-day period regarding the vulnerabilities they might find[20].

In Figure 3.1, we observe that there are two clear time-spans. The first time-span, until 2015's first quarter included, shows a very low number of vulnerabilities related to AOSP being released: 43 over seven years.

In the second quarter of 2015, an impressive peak appears: Almost ten times more vulnerabilities are disclosed this quarter than in all previous quarters. This peak is not a random fluctuation. Indeed, at the end of this quarter, Google published the first instalment of its "Android Security Bulletin", that described the recently patched vulnerabilities. As discussed further below, it also corresponds with the disclosure of several CVEs affecting the media stack of Android, through what is known under the name **libstagefright**. Then the number of vulnerabilities disclosed keeps, overall, increasing until the second part of 2017. It peaks at 70 vulnerabilities for the second

[19] `https://www.google.com/about/appsecurity/`

[20] `https://googleprojectzero.blogspot.com/p/vulnerability-disclosure-faq.html`

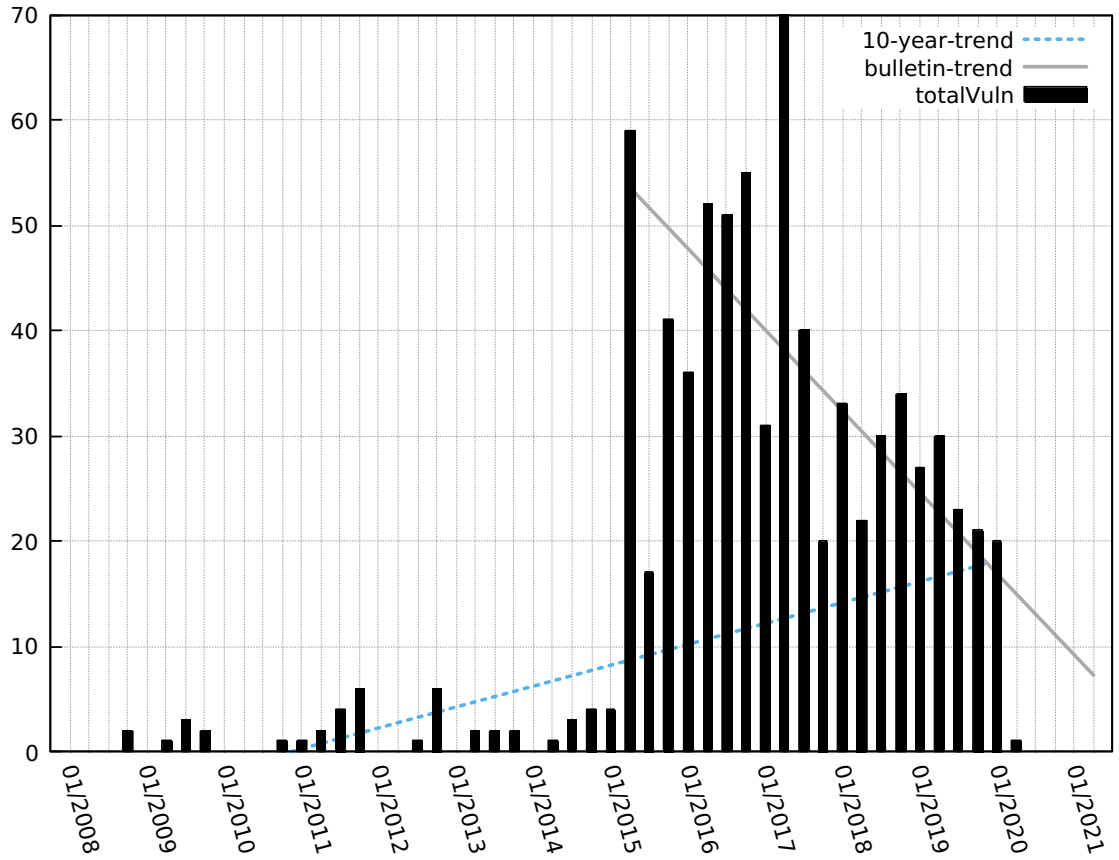


Figure 3.1: Evolution of Android Vulnerabilities by quarter over the last 10 years (date of release is the earliest we could find between the date of the bulletin and the registration date of the CVE on the CVE Mitre website)

quarter of 2017 and from then gradually decreases to 40 CVEs per quarter. Eventually, the number of AOSP vulnerabilities released stabilises around 20 per quarter by 2020.

Two trends were added to Figure 3.1: one over the whole period (since 2008), and another one since the apparition of Android Bulletin. They are both Mayer approximations, meaning the data is split into two equal sets over the study period. An average value is computed for each period of these sets.

One may arguably wonder if the last slow descent and, at least, stabilisation is not an illusion. Vulnerabilities could have already been discovered but not have a reserved CVE yet.

3.3.1.2 Analysis

As mentioned above, before 2015, and thus before the first Android bulletin, information about AOSP vulnerabilities is scarce. Neither was clearly defined what a vulnerability affecting Android was. For example, CVE-2011-3975 and CVE-2012-3979 are both registered as Android-specific while they respectively affect only a few HTC devices, in the first case, and affect the Mozilla Firefox application, in the second case. Thus it might be hard to infer any trend or specificity further than *Jimenez et al.* [77] (see Section 2.1.4.3).

We may at first dismiss any significance about the first peak of 2015. It could only represent the vulnerabilities that have been pilling up before the first released bulletin rather than an actual trend. If so, it may only enlighten that action was required

regarding the number of discovered vulnerabilities.

The increasing number of vulnerabilities may come from an increase of focus on vulnerabilities of the Android OS as Google advertises about it through the monthly bulletin. Also, the libstagefright dread may have put Android under the spotlight. Google may then have had no choice but to go public and seek the help of analysts.

The decrease in absolute number that we can see in Figure 3.1 is a behaviour we can observe for AOSP only. Indeed, other studies such as [76] focusing on the analysis of vulnerabilities in LTS Debian Wheezy do not observe a similar trend. In their case, the semester per semester trend kept augmenting either generally or per package per semester. The authors conclude that Debian Wheezy had not achieved the point of maturity by the time it stopped being supported. The same analysis goes for Windows NT 4.0, Solaris 2.5.1 or FreeBSD 4.0 with data over four to seven years [53]. Only Red Hat 6.2 presents a kind of a clear decrease that, however, is based on a too short time span to be conclusive.

3.3.1.3 Conclusion

Since at least 2017 up to June 2020, the trend is to a reduction of the number of CVEs that needs to be patched in AOSP. So in contrast with Debian Wheezy [76], it is an encouraging sign. It is however difficult to be categorical as there is still a high number of vulnerabilities overall.

Also, Android evolved over the years: both with new features and certainly also with developers learning techniques to prevent the exploitation of vulnerabilities that occurred in AOSP. We can hope they were then implemented. It results that later vulnerabilities might be very different from the earliest ones, potentially more complex.

This temporal study may not be enough to characterise the maturity of the Android OS in regards to security. It may however be enough to later wonder about the roots of that late evolution. Do the team of developers has used the accumulated knowledge to better tackle or better prevent vulnerabilities and their exploitation?

Yet, an encouraging sign regarding the efficiency to tackle vulnerabilities would be if their life duration shortens.

3.3.2 CVE Lifespan

RQ: ***For how long vulnerabilities tend to last in Android-OSP versions ?***

3.3.2.1 Description

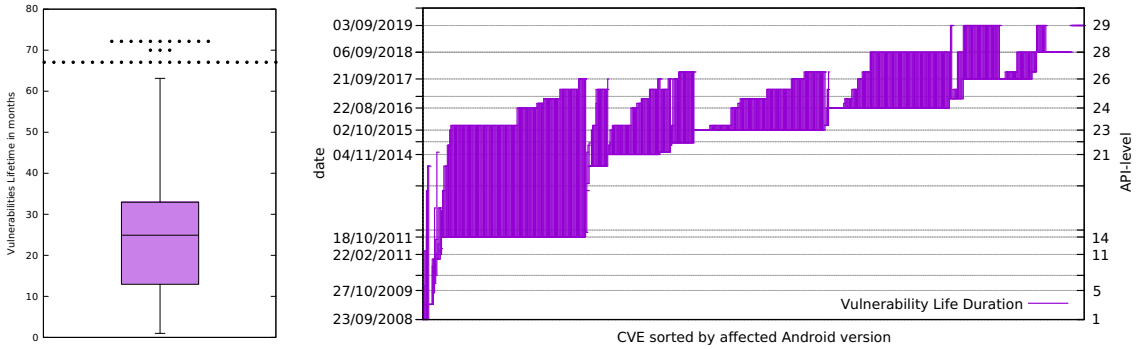
To answer this question, we use the data given by the NVD database regarding our set of CVEs.

We count, for each CVE, how many months separates two affected versions: from when it appears to when it is patched. We count as *version* any version appearing in the NVD JSON file and order them.

The results are provided in Figure 3.2a and Figure 3.2b.

Regarding the figures, we count the months separating the earliest version provided in the NVD database until the last provided. We assume all those in-between to be affected. The second (i.e., right) `Y`-axis of Figure 3.2b indicates the official API-level of major Android versions (in X.0).

Figure 3.2a presents that the median lifespan of a CVE is 25 months, with a first quartile at 12 months and the third quartile at 32 months.



(a) Lifespan (in month between affected versions) of vulnerabilities according to NVD database

(b) Lifespan of sorted vulnerabilities identified by CVE over Android version

Figure 3.2: Lifespan of AOSP related vulnerabilities

Also visible on this figure are several outliers above the higher whisker. CVEs present for more than five years gather vulnerabilities that were introduced very early in AOSP (API level 1 to 8, between 2008 and 2010, or Android version 1.0 or 2.2 for the latest) but were only corrected in API level 19 (in 2014, i.e., version 4.4). In Figure 3.2b, they are on the far left of the representation.

Another set of vulnerabilities present for five years or more was introduced as early as October 2011, but only corrected in late 2016. A few vulnerabilities even survived until as late as August 2017. They are relatively easy to identify in Figure 3.2b as among the last vulnerabilities that starting to affect API-level 14.

We can observe, from Figure 3.2b, that the life-span of CVEs shortens starting API level 21. API-level 27 and 28 were the last versions to be affected by several vulnerabilities.

3.3.2.2 Analysis

These figures highlight that the outliers from Figure 3.2a are mainly introduced around the first versions of Android. From API-level 21 on, three years is usually the maximum length for the lifespan of a vulnerability affecting AOSP.

To provide a point of comparison, in another study [75], the authors conclude that for 50% of the 80k CVEs, over the NVD global data feeds up to 2016, have a lifespan that exceeds 14.4 months. They also find the average lifespan of CVEs to be of 5 years. AOSP appears more responsive with an average of 25 months.

If Android Bulletins started to be published at the same period of time, it may not entirely explain this characteristic as vulnerabilities would only appear fixed, maximum, in the next version. Thus not adding such a short cap we observe. It may highlight that AOSP developers have taken other actions than just the Android Bulletins.

3.3.2.3 Conclusion

The response time is shorter than in other studies analysing vulnerabilities in a system, with the lifespan of the CVEs we analyse rarely (if not never) reaches extreme values over 3 years since 2014-2015.

As we are only analysing publicly released vulnerabilities, it weakens slightly our

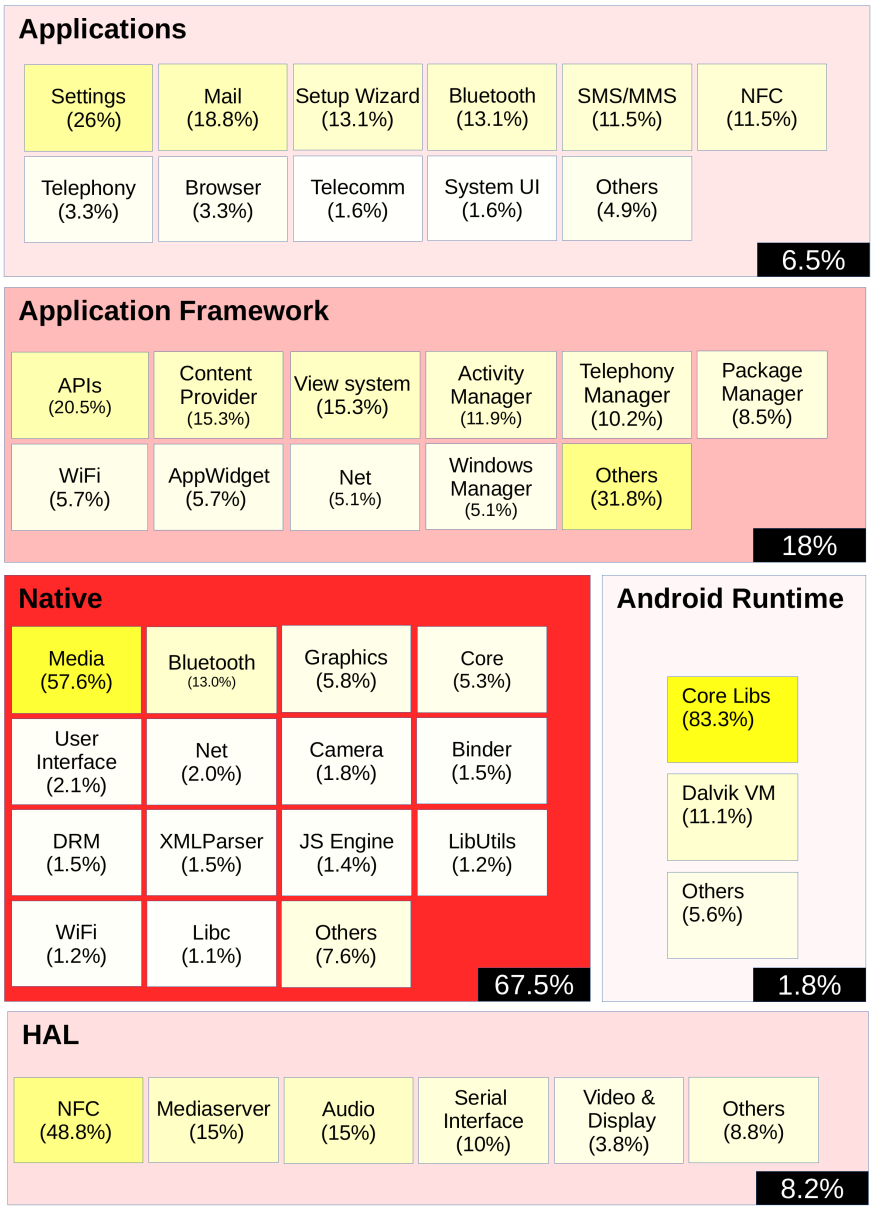
conclusion that vulnerability lifetime reduces. Other vulnerabilities may lie uncovered from a long time on. These could be a vulnerability type analysts have not focused on yet.

3.3.3 Vulnerability Location

In this section, we investigate the distribution of vulnerabilities accross the layers and the modules of AOSP.

RQ: ***Have some AOSP layers or modules been more exposed than others?***

To answer this question, we first provide a timeless analysis by describing Figure 3.3. This figure represents the percentage of vulnerabilities affecting each layer, and each module. Then we consider the temporal aspect for each layer through Figure 3.4.



Note: Sum of percentages may be above 100% as the list of patched files may redirect to several layers and/or modules. However layers and modules are counted only once per CVE.

Figure 3.3: Vulnerability Location in the AOSP Framework

3.3.3.1 Global Description Across AOSP

To produce the Figure 3.3, we used the usual AOSP's Framework representation and design from other works [216, 78] with our own data and conclusions. To achieve this, we extract all the paths of all the files that are modified by all fixing commits into a list. From this list, combined with the AOSP repository and its documentation, we extracted paths that we could connect to a specific layer of the AOSP Framework. For example, the directory `platform/hardware/libhardware` is used to host the code of the Hardware Abstraction Layer (or HAL) as suggested in the documentation[21]. For instance, it helps to conclude that CVE-2016-3760 has been corrected, at least partially, by modifying the Bluetooth module of the HAL. The resulting list of such rules manages to sort the 4400 file paths of the 978 CVEs. When several files are modified, we may count several layers as concerned by this vulnerability. However a layer is only counted once per CVE. Results of this automatic method may thus also differ from works of reference [216, 78] as authors organised a vote for one affected area per CVE, while here we infer automatically from the moment we understand one path is related to one module of one layer. Our list of rules could be improved and completed further as new vulnerabilities are added to the dataset.

Figure 3.3 is readable as follows: for each **layer** (from Application to HAL) the more often CVEs affects a layer, the darker red this layer is coloured. Inside each layer, the darker yellow the **module**, the more it is affected relatively to the layer's individual count. Thus, globally, the `Media` module of the Native layer is more affected than the `Core Libraries` of the Android Runtime layer.

It appears the most affected layer is the Native layer, with 661 unique CVEs related to it. In particular, the Media module is related to 382 different CVEs. This module contains all the codecs for video display. The Bluetooth module of the Native layer is the second most affected module with 86 CVEs. The most common root CWE of the Native layer is CWE-664: Improper Control of a Resource Through its Lifetime with 54%, followed by CWE-682:Incorrect Calculation, with 11%, and CWE-707: Improper Neutralization, at 10%.

The second most affected layer is the Application Framework (sometime called Java API Framework), with 176 CVEs. In this layer, the most affected module is `Content Provider`.

The `others` category is the sum of different but clearly identified modules like the Location Manager, the Bluetooth Service or the Keystore.

In the Application Framework, the most common root weakness is CWE-284: Improper Access Control, with 40% of CVEs related to it. Then comes CWE-664: Improper Control of a Resource Through its Lifetime with 26%.

Third comes the Hardware Abstraction Layer, with 80 vulnerabilities. The NFC module is the most affected with 39 of the related vulnerabilities pointing to it. Then comes the Mediaserver and Audio modules, with 15% each. The two third of related vulnerabilities are CWE-664: Improper Control of a Resource Through its Lifetime.

Ranked fourth by number of vulnerabilities, the Application layer is related to 61 vulnerabilities. The Settings application is the most affected with 26%. Then comes main applications related to emailing (almost 19%). The most common weakness in the Application layer is Improper Access Control, with 45% of the CVEs related to CWE-284.

[21] `https://source.android.com/devices/architecture/hal`

Finally, the Android Runtime has been affected by 18 vulnerabilities. The most present CWE in the Android runtime is CWE-664: Improper Control of a Resource Through its Lifetime, followed by CWE-707: Improper Neutralization.

3.3.3.2 Global Analysis Across AOSP

The most representative types of vulnerability follows expectations. Native layers, using C and C++, offer more ground to memory corruption (that we can find in Improper Control of a Resource Through its Lifetime). Specifically the `Mediaserver` libraries that are impacted by several vulnerabilities in which the input was not appropriately interpreted or neutralised.

Higher Layers are more prone to permissions related issues with Improper Access Control vulnerabilities. Given how the user and the system grants applications permissions to access resources and services it is logical result.

However, these results do not enable to answer how vulnerabilities are acknowledged and fixed in relation with their location. Are some module more affected at some point? Or are they affected in waves? With the focus being put on some modules/layers at some point in the timeline?

RQ: ***How has each layer been impacted over time?***

3.3.3.3 Temporal Description of AOSP Layers' Exposure

To answer this question we propose Figure 3.4 that displays, for each layer, the number of CVEs disclosed per semester. We find the trends we mentioned earlier: the Native layer is the most impacted and overall there is a decrease of CVEs over time. The Native layer has seen a peak in the first semester of 2016 and is since losing, on average, 20 vulnerabilities every year. The peaks of the Native layer in 2016 and 2017 are largely related to the media server module. The Native Bluetooth module was the mostly impacted in 2017 and 2018 with 26 and 37 CVEs.

The Application Framework noticeably reached a peak in 2016 related to the ContentProvider, the ActivityManager and the TelephonyManager modules. The 2019 peak cannot be linked to a specific module given that almost all the layer's module, presented in Figure 3.3, had more related vulnerability disclosed.

In the HAL layer, the 2019 peak is mostly related to NFC vulnerabilities.

3.3.3.4 Temporal Analysis of AOSP Layers' Exposure

Globally, more vulnerabilities seem to be published in first semesters. This time-lapse could match with a few months after a new release. The number of vulnerability in the HAL does not seem to demonstrate a global fall. It may be related to the fact that figures are already low, or that the focus has been put on finding the HAL's NFC module's weaknesses with relation to the increased number of authorisations for Google Pay to be used.

3.3.3.5 Conclusion

Most of the vulnerabilities affected the native layer and media related modules. Overall, vulnerabilities affect the ASOP framework according to both the usual vulnerabilities of a language (memory corruption in the Native layer written in C/C++), and the use of the language (permissions centered for the higher levels written in Java). Temporally, we can observe the global decrease of vulnerabilities; we can also notice peaks that correspond to enquiries in specific modules for a semester or a year: Media server libraries in 2016, Bluetooth on 2018 and the NFC in 2019.

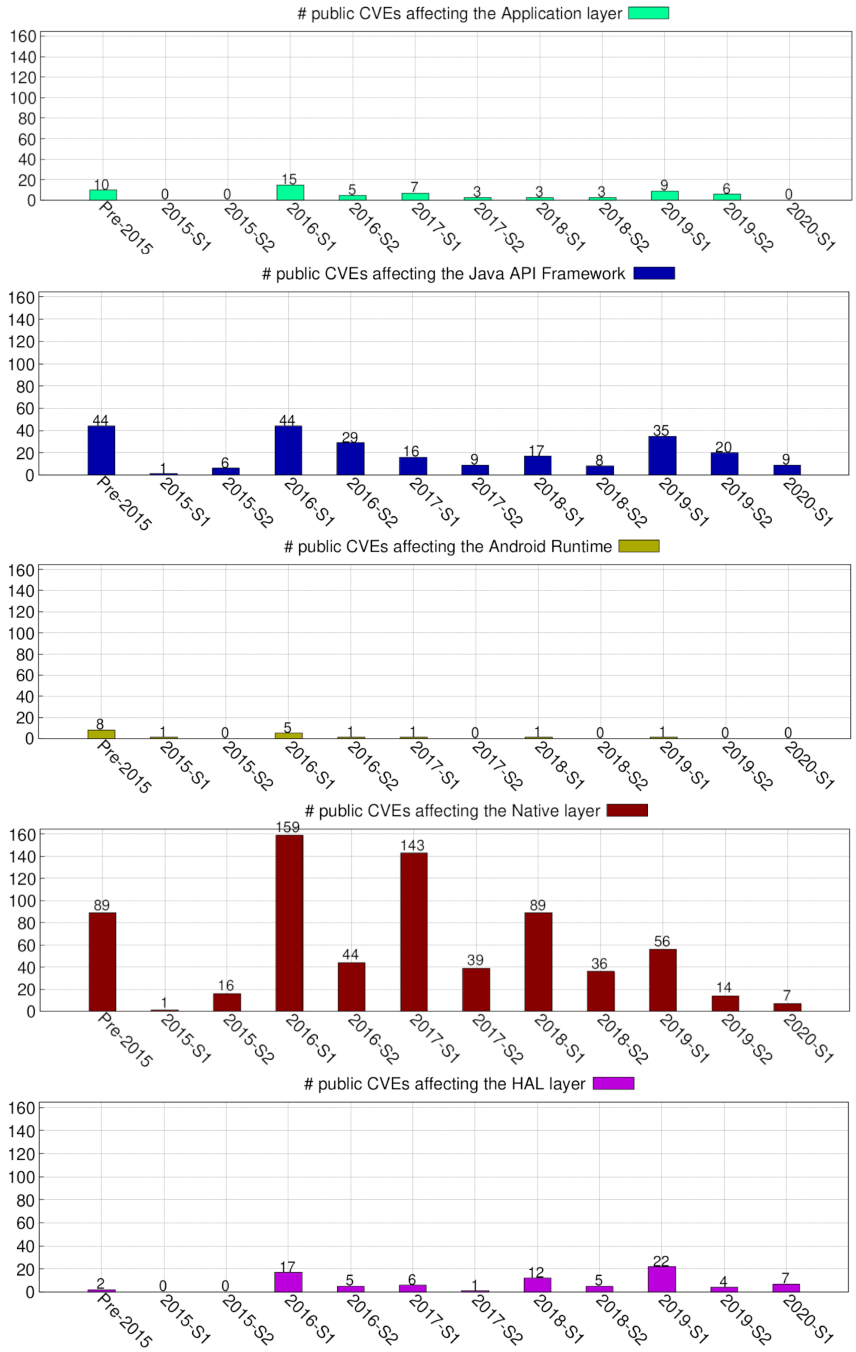


Figure 3.4: Temporal evolution of vulnerability emergence in AOSP Framework

3.3.4 Vulnerabilities Severity

Another aspect for which CVE Mitre's data could help reveal the impact of Android developers' actions towards vulnerabilities are regarding the severity of the new vulnerabilities.

RQ: ***Can we assess a diminution of the severity of new vulnerabilities as a sign of maturity?***

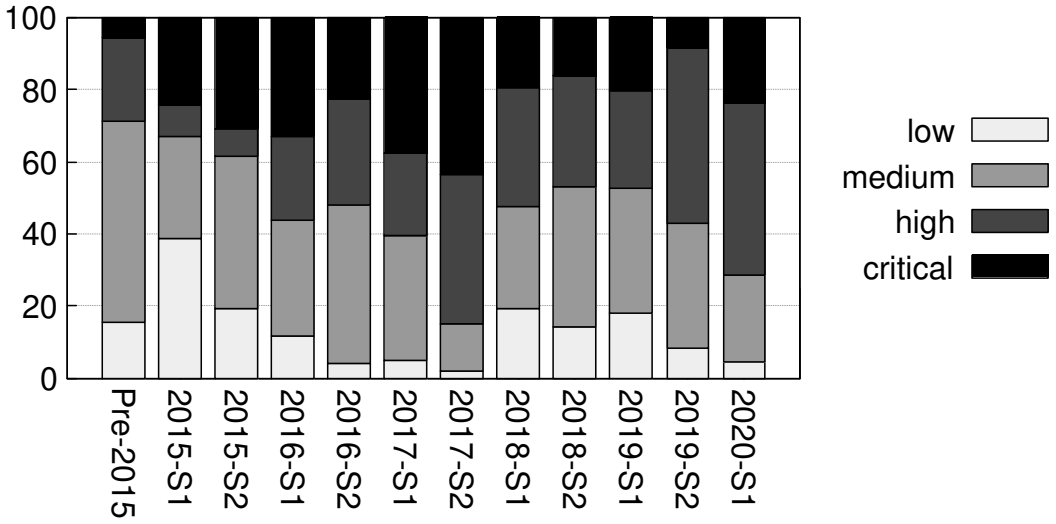


Figure 3.5: Semester per semester evolution of the relative weight of the CVSS score categories

3.3.4.1 Description

Figure 3.5 presents the distribution of the Severity Score (called CVSS) associated with each CVE in the NVD database. 68 of the CVEs do not include a CVSS score in their Mitre Webpage, and are thus excluded from the analysis.

If we were to strictly stick to an analysis semester per semester, some of the semesters of the pre-2015 period would contain no vulnerability, or very few in comparison with late semesters. Therefore, we gathered the vulnerabilities from before 2015 in one timespan: the *pre-2015*. As this category gathers 115 CVEs, while *2015-S1* and *2015-S2* hold respectively 105 and 168 CVEs, it enables a more accurate comparative analysis than if we were to give the same importance to periods with hundreds of vulnerabilities as to periods with just one or two vulnerabilities.

Vulnerabilities have been grouped together depending on their CVSS score following the Severity Ratings provided by the NIST [22] as follows:

- Critical impact for CVEs with a CVSS of 9 or higher;
- High impact vulnerabilities are CVEs attributed a CVSS between 7 and 9 (excluded);
- Medium impact vulnerabilities are CVEs with a score between 4 (included) and 7 (excluded);
- Low impact are those with a score strictly below 4.

A first trend goes from the beginning to the end of 2017. In this period, the rate of High impact vulnerabilities keeps increasing, reaching 84.8% of all the vulnerabilities released during the second semester of 2017. From then, the share of High impact vulnerabilities falls to 50%. It stays at this range between 46% and 57% for two years before increasing again in 2020, reaching 71.4%. In the meantime, Low impact vulnerabilities only reach 20% once (in the first semester of 2015) and their share is decreasing in the last semesters of the study.

[22] `https://nvd.nist.gov/vuln-metrics/cvss`

3.3.4.2 Analysis

The trends evolve back of forth several times. However, there are a few observations we can make. For example, high impact vulnerabilities represent at least 47% since 2016. They had two surges in 2017 and 2020. Low impact vulnerabilities, in the meantime, rarely represent more than 20% of a semester's CVEs. They have been decreasing for two years between 2018 and 2020. With this unstable evolution, definitive future trends stay unpredictable. The fact that there is no increase of low impact vulnerabilities and that they keep representing a small share of CVEs affecting AOSP is nonetheless sufficient to deny signs of maturity regarding AOSP vulnerabilities through the CVSS at the time of the study.

In the Debian analogue study [76] about Debian Wheezy, high impact vulnerabilities kept representing an important and steady proportion. The absolute number of low impact vulnerabilities was also diminishing. Regarding this trend of low impact vulnerabilities, original authors conclude that maybe only higher impact vulnerabilities are patch-worthy. Less impacting vulnerabilities might wait the next major version to be corrected. In our study, we consider several versions of Android rather than just one. Regarding Debian [76], low impact vulnerabilities could not observed as patching and disclosure happen only by the release of the next major versions of a target program. In our study, they are normally part of our figures, as long as they are released as vulnerabilities. It is nonetheless possible that these low impact vulnerabilities are considered as bugs, and internally dealt with.

3.3.4.3 Conclusion

We can conclude that the overall CVSS score distribution has not significantly evolved. Only a slight comparative increase of concerning vulnerabilities can be noted. Thus, over the new versions, AOSP still features as significant vulnerabilities as it used to over the analysis of MAZ19. The observable trend is a clue in the opposite direction from which a mature or maturing system would go.

3.3.5 CWE Types of vulnerability

One could wonder then if the time passing enabled to tackle one type of vulnerability. Or if one new type has appeared,evolved or skyrocketed over time.

RQ: ***Did any weaknesses emerge or disappear over the years?***

We provide with semester per semester description of the proportion of vulnerability types affecting AOSP. We then connect these weaknesses with the absolute number of vulnerabilities disclosed every semester to provide another perspective. We eventually conclude these two parts together.

3.3.5.1 Comparative Analysis of AOSP's Weaknesses

a. Description

In Figure 3.6, we represent, per year of registration of the CVE, the evolution of the six most numerous root CWEs to the Android OS given own manual categorisation. The root CWEs are the CWEs of the first level below the three CWE trees described in Section 3.2.2.1. These root CWEs are:

- CWE-664: Improper Control of Resource Through its Lifetime
- CWE-707: Improper Neutralization
- CWE-1218: Memory Buffer Errors
- CWE-284: Improper Access Control
- CWE-703: Improper Check or Handling of Exceptional Conditions
- CWE-682: Incorrect Calculation

As for some semesters before 2015 there are no vulnerabilities in our study set, and they are very few when any is present, we gathered this whole period as one in the *pre-2015* category. It enables a more accurate comparative analysis as this category gathers 115 CVEs, while *2015-S1* and *2015-S2* hold, respectively 105 and 168. Otherwise, we would analyse based only on a few or no vulnerability per semester.

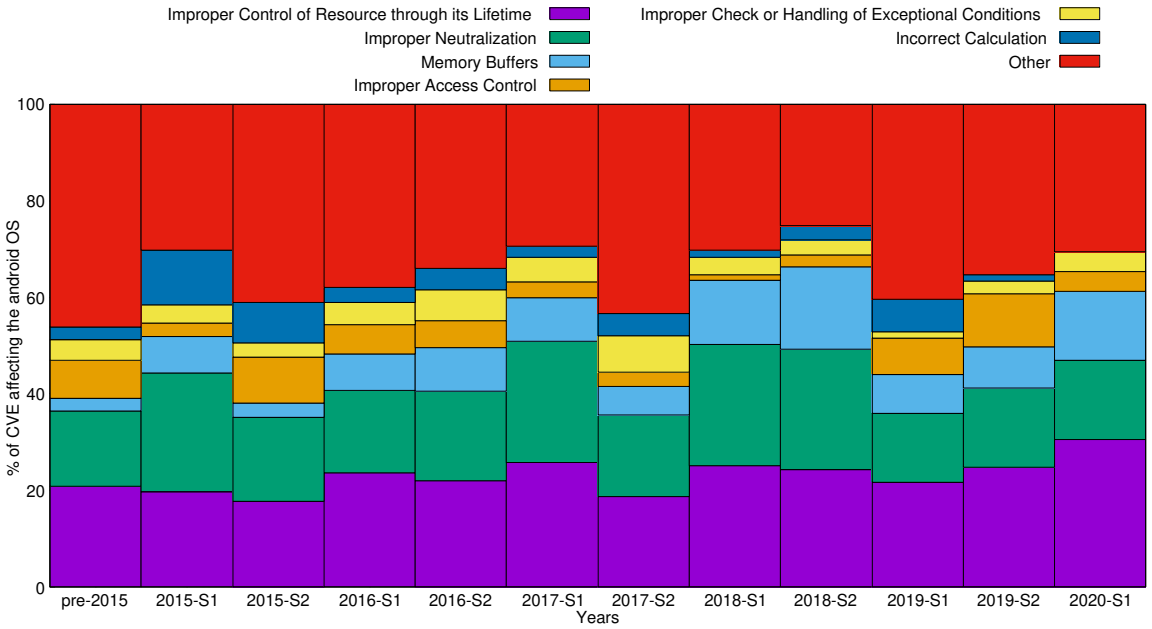


Figure 3.6: Comparative evolution of the root CWEs affecting AOSP over the years

From Figure 3.6, we can see there can be a substantial variability from one semester to the other. In order to reduce our sensitivity to those jumps, we average the percentages over a temporal window of four temporal slots. In other words, the first one covers *pre-2015* until *2016_S1*, the second slot from *2015_S1* to *2016_S2*, so long and so forth. They are presented in Table 3.3.

Table 3.3: Averages over 4 periods of CWE presence

Time Period / CWEs	CWE-664	CWE-707	CWE-1218	CWE-284	CWE-703	CWE-682	Others
Pre-2015 to 2016-S1	20.55	18.61	5.20	6.55	3.94	6.32	38.58
2015-S1 to 2016-S2	20.83	19.34	6.80	5.98	4.48	6.75	35.58
2015-S2 to 2017-S1	22.32	19.48	7.17	6.11	4.79	4.50	35.39
2016-S1 to 2017-S2	22.53	19.41	7.88	4.46	5.95	3.59	35.94
2016-S2 to 2018-S1	22.89	21.44	9.30	3.22	5.68	3.22	34.26
2017-S1 to 2018-S2	23.50	22.99	11.32	2.43	4.87	2.85	32.04
2017-S2 to 2019-S1	22.50	20.28	11.06	3.48	3.94	3.94	34.79
2018-S1 to 2019-S2	24.06	20.12	11.73	5.54	2.70	3.09	32.80
2018-S2 to 2020-S1	25.41	17.92	11.97	6.30	2.82	2.71	32.89

1. We can relate that vulnerabilities categorised as *CWE-664: Improper Control of Resource Through its Lifetime* have generally and steadily increased from 20.5% to 25.4%;

2. If CVEs related to *CWE-707: Improper Neutralisations* have, at first, increased from 18.6% to 23%, they thereafter gradually dropped back into representing

barely 18%. The peak occurs when the first semester of 2018 is the last considered slot.

3. *CWE-1218: Memory Buffers Errors* represented only 5% but have since reached 11% and maintained to this level.

4. Regarding *CWE-284: Improper Access Controls*, they have dropped from 6% to 2.7% in the 2017-2018 period. Since, they bounced back to over 6% .

5. *CWE-703: Improper Check or Handling of Exceptional Conditions*, likewise Improper Neutralization, rose to drop in the end. Figures are relatively low, but from 4% they reached 6% during 2016-2017. The have since decreased to only 2.8% by mid-2020.

6. *CWE-682: Incorrect Calculation* have steadily dropped from 6% to 2%.

7. The group of Other CWEs have recessed if we group the four first and the four last semesters: from 39% to 32.5%.

b. Analysis

None of the above Weakness Enumerations has totally disappeared. Only `Incorrect Calculation` may be on the verge of it, but it is still a trend to be confirmed. Observation of this figure may also go opposite to signs of maturity. For instance two of those main weakness types are increasing. It is not a tremendous rise perhaps, in the range of 5% to 6% each, but a rise that justifies wondering about it further. Those two categories: Memory Buffers and Improper Control of Resource Through its Lifetime, also bring the focus over a specific type of weakness: Memory Corruption, which could regroup several, but not necessarily all, of their sub-categories. A study covering over 10.7K vulnerabilities from the NVD [32], that these vulnerabilities are among the most reported, if not the most affecting software. In their study, Buffer Overflows are ranked first and Use-after-Free ranked sixth in the top 10 most common vulnerabilities. Another study [75] ranks Buffer Overflows first with an average lifespan of 25 months. To continue the comparison with Debian Wheezy [76]: the root CWEs authors studied and that we share are `CWE-664`, `CWE-707` and `CWE-682`, by order of importance in Debian Wheezy's release. Oddly, they consider CWE-118 to be one of those root CWE while it is, when we access it, a *ChildOf* CWE-664 [23]. This category is thus, for us, integrated to `CWE-664`. Even though the Debian Wheezy studyis using the CWE attributed on the NVD website, they seem to have a substantially higher proportion of `CWE-693: Protection Mechanism Failure` and `CWE-118: Incorrect Access of Indexable Resource ('Range Error')` than we do.

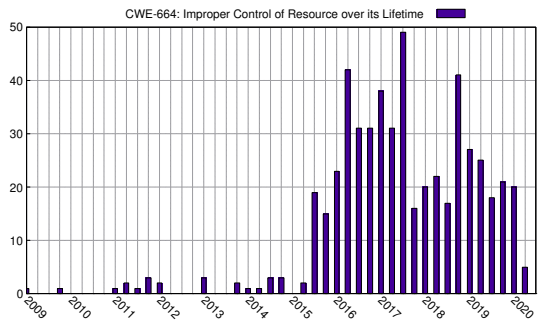
c. Conclusion

Thus, the analysis that several vulnerability types are relatively more persisting than disappearing, nor diminishing, prevails. Only Incorrect Calculations weaknesses show a significant decrease, though from an already low point. We cannot conclude to an internal maturity of Android about any of the main weaknesses, identically as for Debian's Wheezy LTS [76]. This analysis however requires an absolute context to be confirmed.

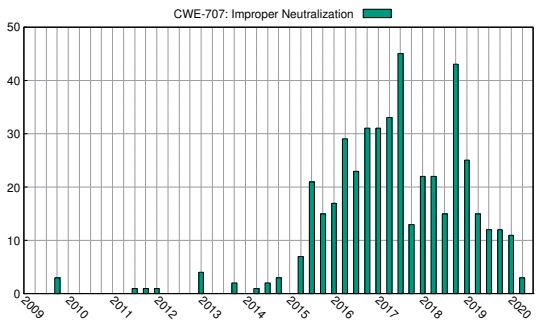
[23] `https://web.archive.org/web/20210415124435/http://cwe.mitre.org/data/definitions/118.html`

In the meantime, we can wonder why certain types, at least, of vulnerabilities are not more significantly diminishing. Is it related to the coding standard? Or is the introduction of vulnerabilities unpreventable? In this case, can we however prevent their exploitation?

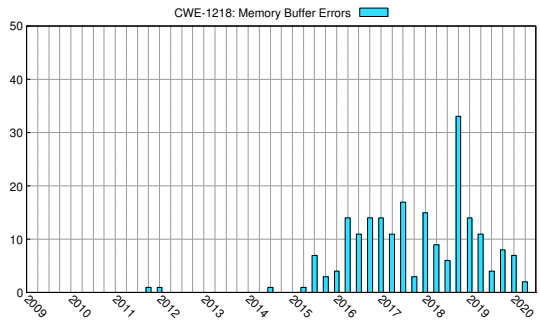
3.3.5.2 Global Analysis of AOSP's weaknesses



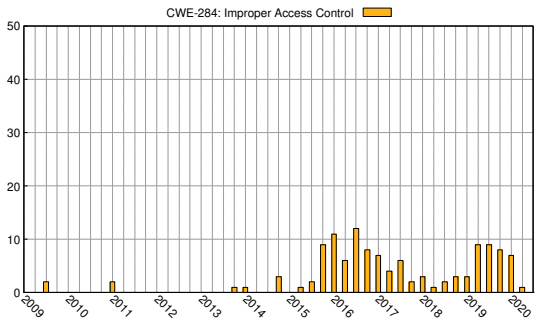
(a) Evolution of the CVEs analysed under CWE-664



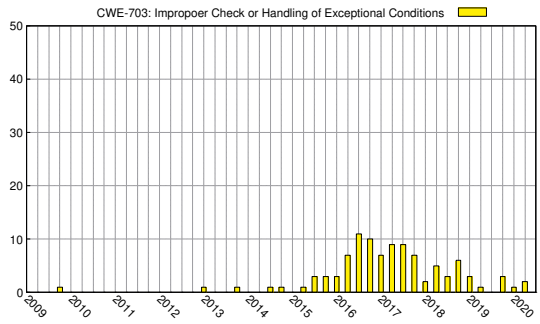
(b) Evolution of the CVEs analysed under CWE-707



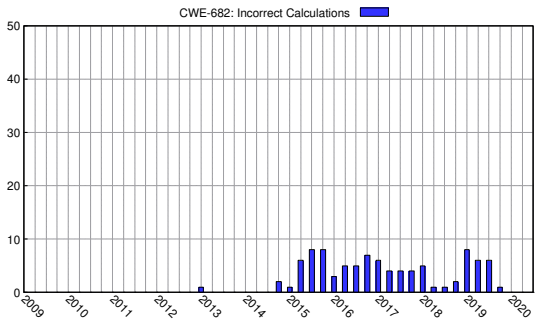
(c) Evolution of the CVEs analysed under CWE-1218



(d) Evolution of the CVEs analysed under CWE-284



(e) Evolution of the CVEs analysed under CWE-703



(f) Evolution of the CVEs analysed under CWE-682

Figure 3.7: Absolute Evolution of the number of CVEs by CWE affecting AOSP the most

a. Description

In Figure 3.7, we represent, by quarter (like we did for Figure 3.1), the number of vulnerabilities per main CWE categories.

Most of them seem, in their range, to follow the global trend of CVEs seen in Figure 3.1.

1. Figure 3.7a: The weaknesses related to improper control of resource over its lifetime seem to overly follow the global trend of vulnerabilities with the same 2 phases: an increase from 2015 to 2017, a low gap for a year and a decrease ever since from a late peak in 2018.

2. Figure 3.7b: The trend of *Improper Neutralization* is overly the same

3. Figure 3.7c: Regarding *Memory Buffers Errors*, the peak arrived later, in the last three months of 2019, and ever since figure dropped.

4. Figure 3.7d: Improper Access Control had almost disappeared in 2018 but have since regained importance.

5. Figure 3.7e: Improper Handling of Exceptional Conditions show, in absolute number, a constant decrease since 2016.

6. Figure 3.7f: Regarding incorrect calculation, the decrease is no more obvious as it seemed during the comparative analysis. This type disappeared for nine months in 2018, regained its previous level for another nine months in 2019 and back again disappeared in the last three quarters with almost no new related CVE.

b. Analysis

CWE-664: Improper Control of Resource over its Lifetime

In the comparative analysis, this category seemed to increase, though with the absolute analysis we can see less and less of this vulnerability type. Thus, rather than an increase of this type, it is more likely a resistance of this kind of vulnerability relatively to the global decrease.

CWE-707: Improper Neutralization

This weakness follows the same trend both relatively and absolutely since 2015: in a first time an increase, then decreasing. The relative decrease synchronised with the global reduction induces an absolute collapse of this kind of vulnerability. It is a trend yet to be confirmed as it remains the second most numerous family of weaknesses.

CWE-1218: Memory Buffers Errors

Here again the rise and stagnation is almost compensated by the general drop of vulnerabilities. This type relatively resists but generally decreases.

CWE-284: Improper Access Control

This types performs better, in absolute number, as it has lately regained its 2016 absolute level after a year and a half of absence.

CWE-703: Improper Check or Handling of Exceptional Conditions

Meeting their peak in 2016, they have diminished ever since: justifying relatively good hopes that the developing team might have the processes and/or the tools to reduce the attack surface to that type of weakness.

CWE-682: Incorrect Calculation

As for *Improper Access Control*, this category has recently known periods where the number of vulnerabilities related is almost nil. Before and after this period, the figures stay at almost the same level. It could then merely be that analysts were not focused on this type of vulnerability for a few months. It could be erroneous to consider that this category as disappeared once and for all based on the last time slots. It had disappeared already, and came back nonetheless.

c. Conclusion

None of the most present CWE-Types are at their absolute highest by June 2020, and nor is the absolute trend to an increase. It is an encouraging sign of maturity first absolutely, and then it also lowers the concerns from the comparative analysis. However, it is not possible to be positive regarding one category of weaknesses being dealt with for good. There are signs of hope to be confirmed: we do not know if we are heading toward a stabilisation or a clear decrease. For example, Incorrect Calculations show both comparative and absolute weakening. Nevertheless to be confirmed as this category reappeared in 2019. It is an analysis that we might extend to all CWE-Types given that it happens for two of them (CWE-682 and CWE-284).

Others are persisting, such as *Improper Control of a Resource over its Lifetime* and *Memory Buffer Errors*. Moreover, as noted above both have sub-categories that can be mapped together under *Memory Corruptions* (through the management of pointers in native as a foremost example for Improper Control of Resources during their Lifetime).

3.3.6 Memory Corruption

In 3.3.5, we mention a perpendicular category of weaknesses called Memory Corruption. It has been stated this CWE type to produce most impact on systems [32], and Google claimed in 2016, that 86% of vulnerability that had affected the whole Android system (i.e., beyond just AOSP) were related to Memory[24].

RQ: ***How have evolved memory corruptions-related CVEs over the years in AOSP ?***

Description After analysing all the External Views from Mitre and not finding one that sufficiently regrouped CWE that gathered all and exclusively Memory Corruption, we decided to regroup all the CWEs unambiguously and *stricto sensu* related to Memory Corruption. The list, based on the three main mappings described in the Section 3.2.2.1, is provided in Appendix 6.2.3.

Comparative Description What Figure 3.8 shows is that Memory Corruption has been an increasing issue until mid-2018, when it reached its peak. If again we compute the average over a period of 4 semester, the share starts from 16.8%, peaks at 23% in 2018 and lowers to 20% in June '20.

Description in Absolute Figures Judging from Figure 3.9, we can observe that it is a trend that confirms in absolute figures. Before the bulletins, 14 Memory Corruption related vulnerabilities were released. It reached above 60 in both 2017 and 2018 and seems, at least until June 2020, to slower ever since.

Description of Severity Score In Figure 3.10, we provide with the severity of the CVEs related to Memory Corruption per semester. The year 2017 is also a period

[24]Corruption`https://source.android.com/devices/tech/debug/cfi`

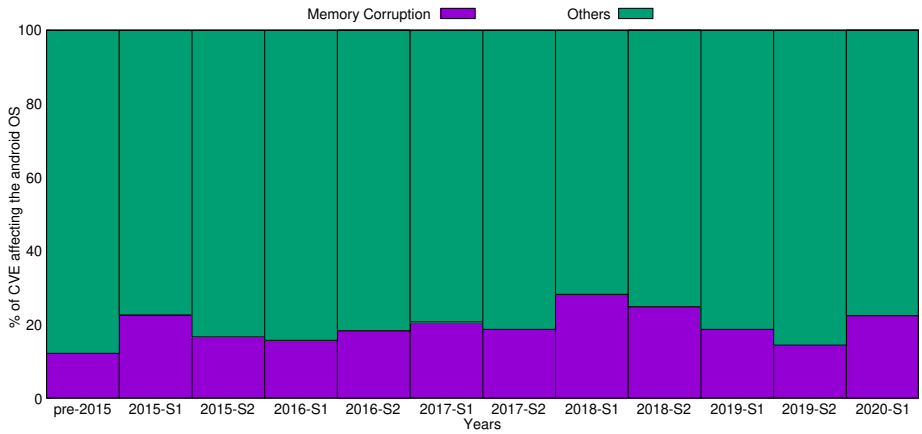


Figure 3.8: Comparative Evolution of Memory Corruption CWEs affecting Android over the years

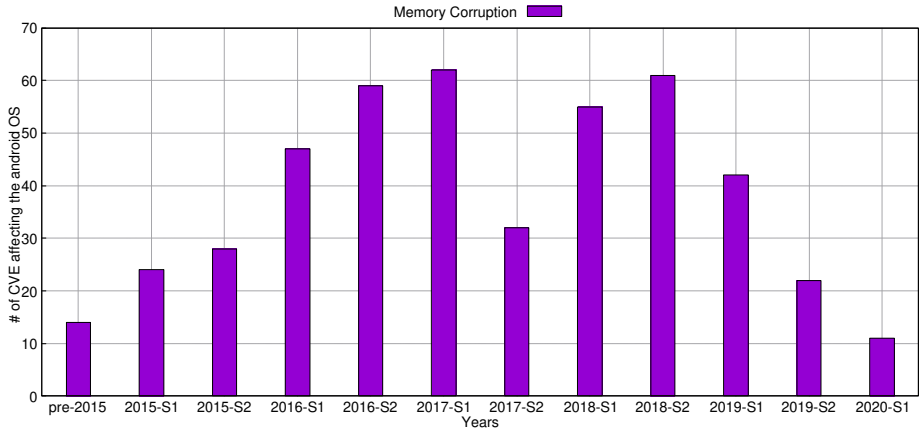


Figure 3.9: Absolute Evolution of Memory Corruption CWEs affecting Android over the years

of time where Memory Corruption related vulnerabilities were scored as having the biggest impacts to the Android ecosystem. It is, respectively, 72.3% and 94.1% of the vulnerabilities that were scored *high impact* (rated 7 or above).

It can also be noted that Figure 3.10 does not present all the CVEs discussed above as several of them are lacking an attributed CVSS score (as discussed earlier). The most obvious example being the **June 2020** period with none of the eleven CVEs being attributed a CVSS. Another reminder of this lack of completeness of NVD data is the first semester of 2018 in which 30 vulnerabilities have no CVSS.

Analysis of Memory Corruption An increase, both comparatively and absolute, can be noticed regarding Memory Corruption related CVEs affecting AOSP around the years 2017 and 2018. Then absolute figures reduce until June 2020. Only the comparative figure for the first semester of 2020 increases back over 20%. It is however a semester with few (only eleven CVEs) and partial (no CVSS) data. This category remains nonetheless concerning as it often contains CVEs with high or critical impact. Some semesters, more than 80% of Memory Corruption related CVEs have at least a high impact. It is often the case in the last semesters of the study.

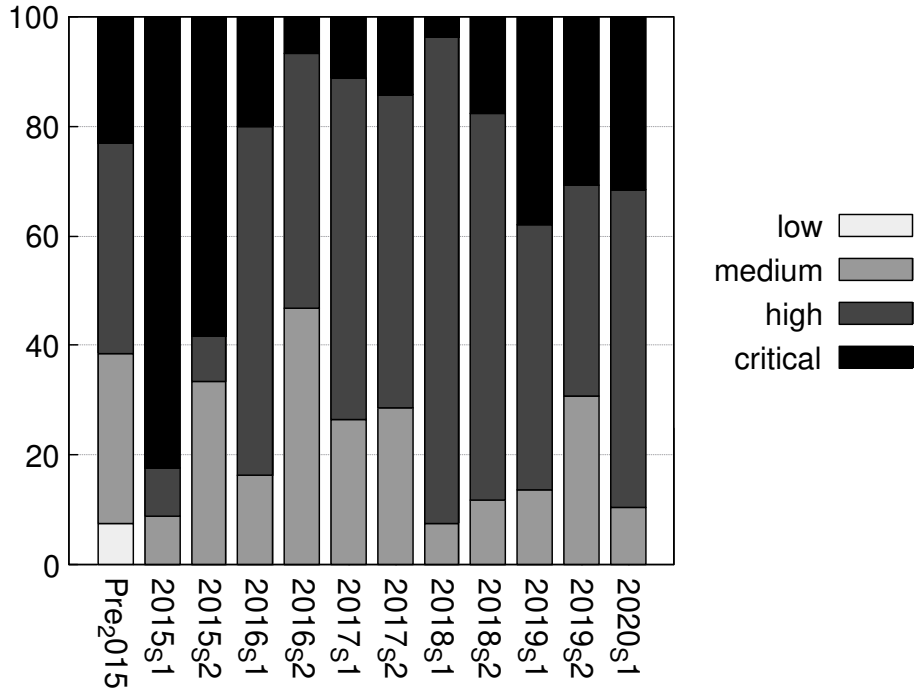


Figure 3.10: Evolution of Memory Corruption Vulnerabilities CVSS over time

Conclusion Memory Corruption is a concerning vulnerability type affecting AOSP. If figures are decreasing, related CVEs have not vanished and usually have high impact on the system.

4 Vulnerability Detection at Commit Level

In this chapter, we address the issue of software vulnerability detection.We aim to replicate the machine-learning based VCCFinder [30], that several tool considered since as state-of-the-art, refer to without directly comparing given the absence of code availability. We thus provide a replicable baseline for futur comparison. In a second time, we explore parameters and algorithm selection, in attempt to improve resulting performance. Finally, we also attempt to change the set of features and to address the issue induced by the over-population of unlabeled set in the data set.

Contents

4.1 Motivation

Software development is a complex engineering activity. At any stage of the software lifecycle, developers will introduce bugs, some of which will lead to failures that violate security policies. Such bugs are commonly known as *software vulnerabilities* [219] and are one of the main concerns that our ever-increasingly digitalised world is facing. Detecting software vulnerabilities as early as possible has thus become a key endeavour for software engineering and security research communities [26, 27, 28, 29]. Typically, software vulnerabilities are tracked during code reviews, often with the help of analysis tools that narrow down the focus scope by flagging potentially dangerous code. On the one hand, when such tools build on static analysis (either deciding based on code metrics or matching detection rules), the number of false positives can be a deterrent to their adoption. On the other hand, when the tools build on dynamic analysis (e.g., for pinpointing invalid memory address), they are operated on the entire software which may not scale to the frequent evolutions of software.

To address the aforementioned challenges that static and dynamic tools face in finding vulnerabilities, [30] have proposed the VCCFinder approach with two key innovations: (1) the focus is made on code commits, which are "the natural unit upon which to check whether new code is dangerous", allowing to implement early detection of vulnerabilities just when they are being introduced; (2) the wealth of metadata on the context of who wrote the code and how it is committed is leveraged together with the code analysis to refine the detection of vulnerabilities.

VCCFinder is a machine learning approach that trains a classification model, which can discriminate between safe commits and commits that lead to the code being vulnerable. The experimental assessment presented by the authors has shown great promise for wide adoption. Indeed, by training a classifier on vulnerable commits made in 2011 on open source projects, VCCFinder was demonstrated to be capable of precisely flagging a majority of vulnerable commits that were made between 2011 until 2014. VCCFinder further produced 99% less false positives than the tool the authors decided to compare their implementation to, namely FlawFinder [82]. Finally, the authors reported that VCCFinder flagged some 36 commits to which no CVE was attached, one of which has been indeed confirmed as a vulnerability introducing commit.

VCCFinder constitutes a literature milestone in the research direction of vulnerability detection at commit-time. Their overall detection performance, presented in the form of Recall-to-Precision curve, however indicates that the problem of vulnerability finding remains largely unsolved. Indeed, when precision is high (e.g., around 80%), recall is dramatically low (e.g., around 5%). This high precision is a promise that security experts' time will be spent on likely Vulnerability-Contributing Commits. This is how to make the best of their skills. Similarly, when aiming for high recall (e.g., at 80%), precision is virtually null.

Unfortunately, since the publication of VCCFinder, and despite the tremendous need and appeal of automatically detecting commits that introduce vulnerability, this field has not attracted as much interest, and therefore as much progress, as one could have imagined.

Thus, to date, it remains unclear (1) whether the ability of VCCFinder to detect Vulnerability-Contributing Commits can be replicated[1], (2) whether, given some varia-

[1]Throughout this chapter, we use the words *reproduction* (different team, same experimental setup) and *replication* (different team, different experimental setup) as defined in the ACM

tions in the datasets or in the algorithm implementation, the produced classification model is stable, and (3) whether some adaptations of the learning (e.g., to account for data imbalance) can improve the achievable detection performance.

In this section. We perform a study on the state of the art of vulnerability finding at commit-time in order to inform future research in this direction. To that end, we first report on a replication attempt of VCCFinder. Replication attempt for which we tried to stick as much as possible to the original work. Then, we present an exploratory study on alternative features from the literature as well as the implementation of a semi-supervised learning scenario. We contribute to the research domain in several axes:

- We perform a replication study of VCCFinder, highlighting the different steps of the methodology and assessing to what extent our results conform with the authors published findings.
- We rebuild and share a clean, fully reproducible pipeline, including artefacts, for facilitating performance assessment and comparisons against the VCCFinder state-of-the-art approach. This new baseline might help unlock the field.
- We explore the feasibility of assembling a new state of the art in vulnerability-contributing commit identification, by assessing a new feature set.
- We identify one issue to be the lack of labelled data, and we explore the possibility to leverage a specialised technique, namely co-training, to mitigate that issue.

The main findings of this work are as follows:

- The VCCFinder publication lacks sufficient information and artefacts to enable replication.
- Despite our best experimental efforts, we were unable to replicate the results reported in the publication, suggesting some generalisation issues due to high sensitivity of the approach to dataset selection and learning process.
- A semi-supervised learning approach based on our new feature set (inspired by a recent work [1] that is targeting the detection of vulnerability fix commits, rather than the detection of Vulnerability-Contributing Commits, or VCCs) does not achieve the same detection performance as reported in the state of the art. Nevertheless, our approach constitutes a **reproducible baseline for this research direction.**

> *While our work contains a replication study, it also acknowledges the limits of the replicated approach (i.e., VCCFinder) and, more importantly, it tries to unlock this important research field by providing a reproducible setup. Data, code and instructions are available. It also demonstrates that the artefacts we provide allow for new experiments to advance the state of the field.*

The rest of this section is organised as follows:

- We first focus on describing the VCCFinder approach: what resources are available, what we had to guess, and how we reimplemented it (Section 4.2). We compare the achieved results with the originally presented ones.
- We then propose and evaluate in Section 4.3 a new approach, built with another feature set, and co-training.

Artifact Review and Badging Document. We further note that this terminology was updated in August 2020; We use the updated version. `https://www.acm.org/publications/policies/artifact-review-and-badging-current`

- We finally summarise our contributions in Section 4.4.

4.2 Replication Study of VCCFinder

The first objective of our work is to investigate to what extent the VCCFinder [30] state-of-the-art approach can be replicated (different team, different experimental setup) and/or reproduced (different team, same experimental setup). VCCFinder[2] is a machine learning-based approach aiming at detecting commits which contribute to the introduction of vulnerabilities into a C/C++ code base.

As most machine learning-based approaches, VCCFinder relies on several building blocks:

1. A labelled dataset of commits which is used to train a supervised learning model;
2. A feature extraction engine that is used to extract relevant characteristics from commits;
3. A machine learning algorithm that leverages the extracted features to yield a binary classifier that discriminates vulnerability-contributing commits from other commits.

In the following, we present, for each of the aforementioned three building blocks, the descriptions of operations in the original paper. We then discuss to what extent we were able to replicate these operations. Subsequently, we present the results of our replication study.

4.2.1 Datasets

4.2.1.1 Datasets - VCCFinder Paper

A key contribution in the VCCFinder publication is the construction of two labelled datasets of C/C++ commits.

- A dataset of commits that contribute vulnerabilities (VCCs) into a code base;
- A dataset of commits that fix vulnerabilities that exist within a code base.

With the assumption that a commit that fixes a vulnerability does not introduce a new one, the authors consider the second dataset as a negative dataset (i.e., the corresponding dataset of non-vulnerability-contributing commits). To build both datasets, the paper reports that 66 open-source git repositories of C and C++ projects were considered. Overall, these repositories included some 170 860 commits. For the creation of the *vulnerability-fixing commits* data set, the authors gather all the CVEs[3] related to these repositories. They selected CVEs that are linked to a fixing commit. With this method, 718 vulnerability fixing commits were collected.

Collecting commits contributing to a vulnerability is less straightforward. Indeed, usually, commits introducing vulnerability are not tagged as such, and there are no direct information in the commit message that indicates the vulnerable nature of the commit.

To overcome this difficulty, the authors follow an approach defined by [111] and called SZZ. The principle is to start from vulnerable lines of code. Such vulnerable lines of code are identified thanks to the vulnerability fixing commits: indeed, it is reasonable to assume that the lines that have been fixed were previously vulnerable. Then the `git blame` command is used on these identified lines of code. The `git blame` command allows finding the last commit that modified a given line. The assumption

[2]VCCFinder means Vulnerability-Contributing Commit Finder

[3]CVEs: Common Vulnerabilities and Exposures are publicly available cybersecurity vulnerabilities.

here is that the last modification made on a vulnerable line of code is the modification that introduced the vulnerability.

Thanks to this method, 640 vulnerability-contributing commits (VCC) have been collected. Note that the numbers of vulnerability-contributing commits and vulnerability fixing commits are different simply because one commit can potentially contribute to more than one vulnerability.

In the VCCFinder paper, both datasets have been divided into a training set and a testing set (following a two-third, one-third ratio). All commits created before January, 1st 2011 are put in the training set, and the remaining in the test set. The numbers of commits of each dataset are presented in the left part of Table 4.1. Note that among the whole dataset of 170 860 commits, only 1258 (640 + 718) commits have been classified. The 468 (219 + 249) labelled commits in the test set is used as ground truth, notably to compute Precision and Recall performance metrics.

All other commits that are not categorised into the two first datasets (169 502) are put in a third dataset named *unlabelled* dataset. This dataset of unlabelled commits is also split into two datasets. All commits created after January, 1st 2011 are in a test set. In the original paper, this unlabelled test set is used to try to uncover yet-undisclosed vulnerabilities. The authors claim VCCFinder was able to flag 36 commits as VCCs. They detail one VCC for which they received confirmation from the development team that it was indeed a VCC. At the time they wrote the presentation of their work, they had not received confirmation for the others.

4.2.1.2 Datasets - Availability

The dataset of the original VCCFinder article is not directly accessible.

Online investigation may direct to a specific Github repository[4] that holds the name of the tool and the name of one of the authors. However, the original paper does not mention this repository. The code present in this repository is not fully documented, as was already mentioned by a prior work whose authors noted some major challenges to exploit its contents [39]. After carefully analysing this repository, we came to the conclusion that the artefacts in this repository would not allow us to re-construct the exact same dataset as the one used in the original VCCFinder. Moreover, it would not even allow to construct a *different* dataset, as parts of the features extraction process is missing (to the best of our knowledge).

4.2.1.3 Datasets - Our Replication Study

At the time we reached a conclusion about the available Github repository, we had already contacted the authors of VCCFinder who offered to provide directly the output of their feature extraction pipeline. We accepted their offer, as it seemed that it was the only viable solution.

This dataset provided to us by VCCFinder's authors is a database export that contains three tables:

- A table listing 179 public repositories of C/C++ projects;
- A table listing 351 400 commits, each commit being linked to a repository thanks to the use of a repository id;
- A table listing the CVEs used to identify the vulnerability fixing commits.

Note that over those 179 repositories, all commits are related to an existing repository. However, only 50 repositories have at least one declared commit (i.e., 129 repositories

[4] `https://github.com/hperl/vccfinder`

Table 4.1: Datasets comparisons

	VCCFinder Paper 66 repositories			**Replication** 38 repositories		
	Training	Test	Total	Training	Test	Total
Positive (vuln. contr. commit)*	421	219	640	470	253	723
Negative (vuln. fixing commit)	469	249	718	389	879	1268
Unlabelled	*90 282*	*79 220*	*169 502*	*229 381*	*119 489*	*348 870*
Total			170 860			350 861

* Vulnerability-Contributing Commit

have no related commit).

Furthermore, out of these 50 repositories, only 38 repositories contain at least one vulnerability fixing or vulnerability-contributing commit. Among these 38 repositories, only 27 are linked to both a vulnerability contributing commit and its relevant vulnerability fixing commit.

While no such process is mentioned by original authors, we opted to discard commits that do not modify any code file, as they are very unlikely to be involved in any vulnerability fixing or introducing. We used a simple heuristic that discards commits with no modification to a file whose extension is either `.h`, `.c`, `.cpp`, or `.cc`.

Table 4.1 presents a comparison between a) the number of commits that have been involved in our replication attempt, and b) the dataset described in VCCFinder original paper.

We note that the dataset provided to us is significantly different than the one described in the VCCFinder paper. We also note that we are unable to evaluate whether there is any overlap between the dataset we had access to and the original one.

> *As shown in Table 4.1, the datasets used in the VCCFinder paper and the ones used in our replication study are not identical. Even if the number of positive and negative samples in the training and test sets are close (same order of magnitude), we can notice significant differences regarding: (1) the number of repositories presenting a fixing commit (66 vs 38), (2) the number of negative samples (i.e. fix commits) in the Test sets (249 in the VCCFinder paper, 879 in our replication study).*
> *This fact alone guarantees that we will not be able to obtain exactly identical results. Given* how much *the datasets are different, we even* expect *our results to be potentially significantly different.*

4.2.1.4 Use of the data sets

The aforementioned *ground truth* notion is important as VCCFinder's authors opted to both report performance metrics computed against this ground truth, and metrics computed on data they had no ground truth for (we do not know how they did this). Original authors were contacted but did not come back to us on the matter. As a result, we faced huge difficulty to clearly understand the notion of ground truth as used in the

Table 4.2: Dataset repartition scenarios

		Training	Test
Unlabelled Train Replication	positive	470	253
	negative	229 770 (389 + 229 381)	120 368 (879 + 119 489)
Unlabelled Replication	positive	470	253
	negative	389	120 368 (879 + 119 489)
Ground Truth Replication	positive	470	253
	negative	389	879

original VCCFinder paper.

Since our understanding of their notion of ground truth is based on deduction and guesswork, and not on a clear authoritative description from original authors, we now carefully detail on what we trained our classifiers on, and on what they were tested on. More specifically, we performed three different experiments:

1. What we think the original experiment was;
2. A less coherent setup;
3. A more traditional setup.

We note that we cannot definitely affirm which of the first or the second setup VCCFinder original paper used, as both are coherent with the figures reported. The repartition is presented in Table 4.2, and detailed in the following paragraphs:

Unlabelled Train Replication: A classifier is trained on the whole training set, including the unlabelled commits created before 2011. This first one is the one we think to match the most with the description of the original experiment. The negative label (i.e., not VCC) is associated with those unlabelled commits before training. The resulting classifier is tested on the whole test set, including the unlabelled commits from 2011 and newer. Similarly, those unlabelled commits are associated with the negative label. The goal being to find VCCs, if the resulting classifier predicts one originally unlabelled commit to be a VCC, this will display as a *False Positive*.

Unlabelled Replication: This setup is very similar to the previous one, with the exception that the unlabelled commits created before 2011 are not used in the training phase. Those related to after 2011 are used in the test set (and associated with the negative label). This scenario would enable to analyse the model's behaviour once facing security neutral commits. That is to say, commits that are neither VCCs nor fixing commits, the latter having to be written with a security mindset. Still, the model would train on the closest we have to a ground truth. This setup is less coherent in the sense that unlabelled commits are not treated similarly in the training than in the testing.

Ground Truth Replication: In this more traditional setup, a classifier is trained on the train set for which we have a ground truth, i.e., excluding the unlabelled commits. Similarly, the resulting classifier is tested on the test set for which we have a ground truth, i.e., excluding the unlabelled commits.

4.2.2 Features

4.2.2.1 Features - VCCFinder Paper

The second main step of the VCCFinder approach consists in extracting the relevant features that will feed the machine learning algorithm. Among the selected features, VCCFinder considers *code metrics* and *meta-data* related to both a particular commit and the whole repository.

Regarding the commit[5] itself, the patch code and the commit message are both considered. Note that a specific section of the original paper is dedicated to asserting the relevance of the features by comparing their frequency in vulnerability-contributing commits and other commits.

Regarding code metrics, for a given commit m from a repository R, VCCFinder extracts:

- The number of structural keywords of C/C++ programs (such as `if`, `int`, `struct`, `return`, `void`, `unsigned`, `goto`, or `sizeof`, etc) present in m. Overall, 62 keywords are referenced;
- The number of hunks[6] in m;
- The number of additions in m;
- The number of files changed in R.

Regarding metadata, for a given commit m from a repository R, VCCFinder considers:

- The total number of commits in R;
- The percentage of commits in R performed by the author of m;
- The number of changes performed on the files modified by m after m was applied;
- The number of changes performed on the files modified by m before m was applied;
- The number of authors altering the files impacted by m;
- The number of stargazers, forks, subscribers, open issues and others, including the commit message itself.

4.2.2.2 Features - Availability

The earlier mentioned git repository ends up registering commits in a database, though as already stated (Section 4.2.1.2), we are unsure whether the resulting database would have all the information needed, in particular, we have been unable to locate code that would compute all the features required. Furthermore, the original paper does not contain enough details to fully re-implement the full feature extraction ourselves.

Therefore, regarding the extraction of features, we have to rely on the fields present in the database given by the original authors.

4.2.2.3 Features - Our Replication Study

As already explained, the original paper does not precisely list all the features extracted leading to a situation where we were unable to re-implement a feature extraction engine, and thus unable to re-use their approach on another dataset.

However, the database that was shared with us already contains the features computed by VCCFinder authors themselves. We hence directly used those features.

[5] We remind that a commit is composed of a patch (i.e., the "diff" representing the code changes), and a commit message (explaining the modification performed by the patch)

[6] a hunk is a block of continuous added lines

Since the VCCFinder authors sent us datasets with the features already extracted, our replication study leveraged exactly the same features as the VCCFinder approach. However, since we did not obtain or re-implement the feature extraction engine, we are not able to extract features from other datasets of commits.

4.2.3 Machine Learning Algorithm

4.2.3.1 Machine Learning Algorithm - VCCFinder Paper

The VCCFinder approach leverages an SVM algorithm (through its LibLinear [220] implementation) to learn discriminating vulnerability introducing commits from other commits.

This algorithm builds a hyper-plan that would separate, in our case, vulnerability introducing commits from others. To classify a given commit, a distance is computed between the feature vector of this commit (i.e., a point in the hyper-space) and this hyper-plan.

The sign of this distance determines whether this commit contributes to a vulnerability or not.

Given a commit and the extracted features, we describe now the generation of the feature vector of this commit that is used as input of the machine learning algorithm.

This process follows a generalised bag-of-words approach that normalises the features' values into boolean vectors. Regarding the normalisation, for each feature, commits are categorised into bins based on the occurrences of the feature. Then a string is built by concatenating the name of the feature and the bin identifier.

Finally, joining all these newly created strings together with the texts formed by the patch code and/or commit message, a considerable string is built and fed to a tool named SALLY [221]. SALLY is a binary tokenisation tool which generates a high-dimensional sparse vector of booleans from a string, computing a hash for each split-on-space sub-string. At the end of this process, each commit is represented now by, first, a boolean, indicating its class (vulnerability-contributing commit or not) and a succession of pairs `(feature_hash/binary value)` that represent a sparse vector of the features.

The VCCFinder authors mention they used a handicap value C of 1 and weight for this one-class problem of 100 as "*the best values*" (last sentence of their section 4.2).

Eventually, the authors present their results on the test set with a Recall-to-Precision curve for which the actual parameter is the threshold in Figure 4.1. After computing the distance from the hyperplane for each commit in the test set and by incrementally lowering the threshold, the commits the closest to the hyperplane will be classified as VCCs. Lowering the threshold results in increasing the number of True Positives, but might also quickly bring more False Positives.

The higher the Recall-to-Precision curve, the more precise, and the more horizontal, the more the model is not sacrificing precision for recall.

4.2.3.2 Machine Learning Algorithm - VCCFinder Availability

As already explained, VCCFinder authors did not release code that perform all the required steps of their approach. Even in the repository found on the Internet (but not mentioned in the VCCFinder paper), the code that orchestrates the training of the classifier and its usage is absent.

However, as noted above, authors provide some of the parameters in the paper. We

note that the embedding step (i.e., tokenisation and discretisation) is almost adequately described in the original paper, with the exception of the number of bins (cf. below).

4.2.3.3 Machine Learning Algorithm - Our Replication Study

The VCCFinder authors mentioned they used the LibLinear [220] library to run the SVM algorithm. However, several front-ends of LibLinear exist. We decided to use the `LinearSVC`[7] implementation included in the popular framework scikit-learn.

Regarding the construction of the feature vectors, and more specifically regarding the normalisation step, the authors do not specify the number of bins they use, nor on which features this step was performed. We decided to consider 10 bins per feature containing each, as much as possible, the same number of commits. This was done with scikit-learn's `preprocessing.QuantileTransformer` facility, assigning the value of 10 to `n_quantiles` parameter, and `'uniform'` to the `output_distribution` parameter.

We then apply LinearSVC classifier with C parameter equals to one, the weight of the class one to 100 over 200 000 iterations.

> *With the exception of the exact usage of the unlabelled commits, we are rather confident that our own implementation of the machine learning algorithm building blocks mimics the VCCFinder one. However, we cannot evaluate if the differences have a significant impact on the results obtained.*

4.2.4 Results

In this section, we detail the results yielded by VCCFinder in the original paper, as well as the results that we obtain when we replicate VCCFinder.

4.2.4.1 VCCFinder Paper

To assess the performance of their machine learning-based approach, the authors keep about two-thirds of their datasets for training, and use one-third of the datasets for testing. Table 4.1 presents the exact numbers. Note that, as explained in Sub-Section 4.2.1, we are not sure about what the training and testing sets are composed of.

The original results are presented in Figure 4.1, which is directly extracted from the paper [30]. The plot is obtained by measuring/computing precision and recall values when varying the threshold.

In the original paper, the authors compare VCCFinder against a then-state-of-the-art tool named flawfinder(in red in Figure 4.1). Flawfinder is a static analyser tool that looks for dangerous calls to sensitive C/C++ APIs in the code as `strcpy` and flags them.

Figure 4.1 shows that VCCFinder greatly outperforms Flawfinder. The authors also set their tool to the same level of recall that Flawfinder is capable of for this dataset, 24%, and show that their approach presents then a precision of 60%. In comparison, Flawfinder can only achieve 1% in such conditions. For a recall of 84%, VCCFinder has a precision of 1%.

With precision and recall values extracted from Figure 4.1, an F1-score can be computed thanks to the following formula:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

[7] `https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html`

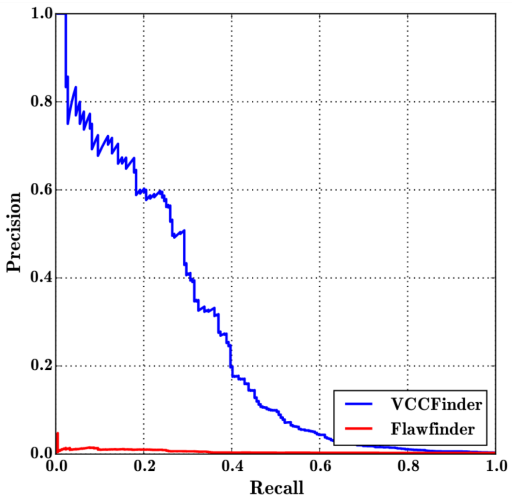


Figure 4.1: Extracted from the VCCFinder paper: precision/recall performance profile of VCCFinders

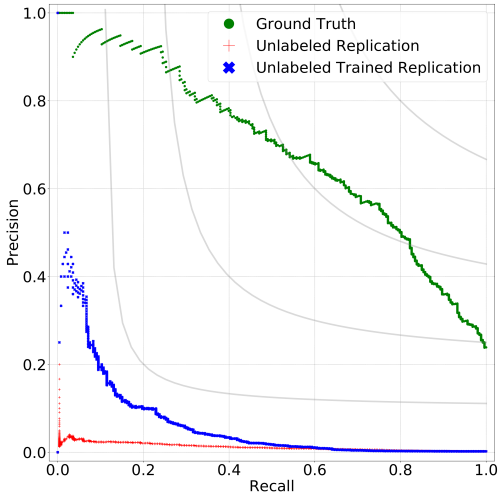


Figure 4.2: Precision/recall performance profile of VCCFinder's Replication

We can notice that the maximal F1-score of VCCFinder seems to be lower than 0.4, with a maximum of either (Recall;Precision) =(0.25;0.6) or (Recall;Precision)=(0.3;0.5). Those lead to an F1-score of either 0.35 or 0.375.

Table 4.3 describes several metrics (extracted from the original paper) such as True Positive, False Positive, etc computed on the test set. VCCFinder flagged 53 commits that are, according to the ground truth, actually introducing a known vulnerability. Applying VCCFinder to the larger set of unclassified commits, 36 commits were flagged as suspicious. Among those 36 potential VCCs, one was described by authors as confirmed by the project maintainers, who had already patched this vulnerability. Authors opted not to comment on the other 35 commits, invoking "responsible disclosure".

These 36 commits are presented as belonging all to the post-January 2011 unclassified set. Thus, on what they define themselves as the ground truth, no false positive is met.

4.2.4.2 Our Replication Study

The results presented in Figure 4.2 show the precision per recall we obtain on the 3 different test sets while diminishing the threshold. One can understand the threshold as the minimum distance from the hyperplane for a commit to be considered as VCC. The grey curves represent the lines for a constant F1-score at 0.2, 0.4, 0.6 and 0.8. We now details the results for each of the 3 test sets presented in 4.2.1.4:

Ground Truth Replication:
The replication achieves a maximum F1-score of 0.63 for a recall of 0.76 and a precision of 0.54 (see line 2 of Table 4.3 and green dots in Figure 4.2). We also set ourselves, for the purpose of comparison, to the reference recall used in VCCFinder's original paper of 0.24 to find a precision of then 0.92. In these conditions, the F1-score is of 0.38. It presents a progressive decline and correctly tags 61 commits as VCCs.

Unlabelled Replication:
This attempt trains on the ground truth but is tested on both ground truth and beyond 2011 unclassified is drawn in red in Figure 4.2. We can see it perform very poorly, presenting more than three thousand false positives, once set to the same recall of 0.24. The precision is then barely of 2% and the F1 score of 0.037.

Table 4.3: Results of replication on updated test set

	True Positive(VCC*)	False Positives	False Negatives	True Negatives†	Precision	Recall
VCCFinder	53	36	166	79 184	0.60	0.24
Ground_Truth Replication	61	5	192	885	0.92	0.24
Unlabelled Replication	61	3145	192	157 224	0.02	0.24
Unlabelled Trained Replication	61	695	192	159 674	0.08	0.24

* VCC: Vulnerability-Contributing Commit

† Vulnerability-Fixing Commit and post-2011 Unlabelled

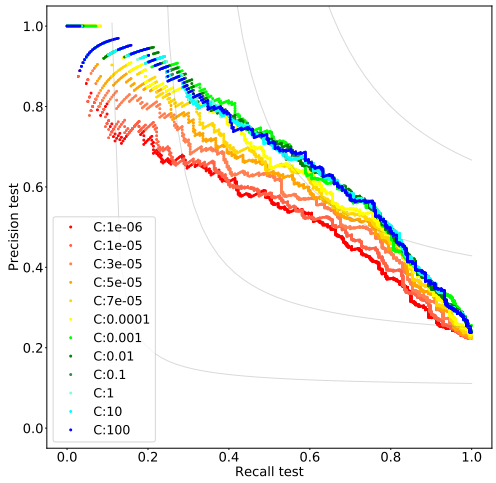


Figure 4.3: Precision/recall performance profile of VCCFinder's replication for varying values of C parameter

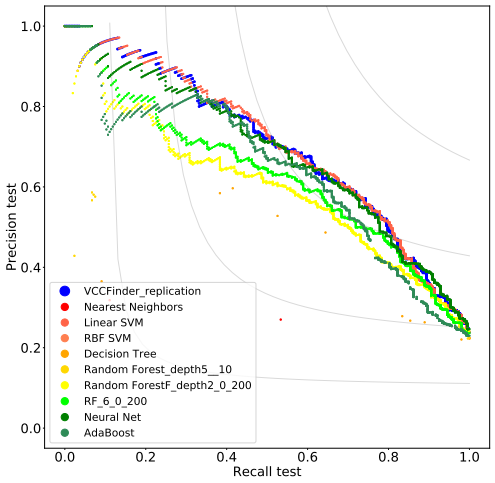


Figure 4.4: Precision/recall performance profile for comparing classifying algorithms

Unlabelled Train Replication:
It is after assessing how poorly the last experiments performed that we decided to include unclassified in the training, forcing them as non-VCCs. The results are illustrated thanks to the blue curve in Figure 4.2 and the last row of Table 4.3. It improves sensibly the performances without reaching the level of the original. The precision for fixed recall is of 8%, leading to an F1-score of 0.12.

4.2.4.3 Parameters Exploration

Besides the results on the 3 different test sets, we took the opportunity of this replication attempt of VCCFinder to investigate the impact of various parameters.

Exploration over parameter C:
In the original paper it is just stated that the optimal conditions are for a cost parameter C of 1. We experiment for different values of C on the basis of the Ground Truth Replication. We experiment for values from $C = 10^{-6}$ to 100, and obtain the values presented in Figure 4.3.

It appears that the behaviour seems to tend toward an optimal behaviour starting at $C = 10^{-2}$ and higher. Thus, as advocated by the VCCFinder authors, using a value of C at 1 makes sense.

Exploration over class weight parameter:
Altering the weight of the positive class (VCCs) from 0.1 to 100, we saw no difference

in the output using the same other settings. There is, thus, no reason to deviate from the original paper declared values.

Exploration with other algorithms:
We also experimented with a variety of different machine learning algorithms. Results are presented in Figure 4.4. We note that SVM—that is used by the original VCCFinder paper—is among the algorithms that produce the best results.

4.2.5 Analysis

We discuss the experimental results of our replication attempt of the VCCFinder approach.

RQ: *Is our reproduction of VCCFinder successful?*

According to the terminology used by ACM's *Artifact Review and Badging* guidelines, a *Reproduction* requires the same experimental setup [222]. We recognise that some elements of our setup were different from the setup in VCCFinder publication. We have therefore documented the differences.

We note that the combination of a) an implementation of the approach, and b) the exact dataset used originally would have allowed us—and any other researcher—to positively validate the results reported by VCCFinder's authors.

> *We have been unable to* Reproduce *VCCFinder.*

RQ: *Does the present work constitute a successful* Replication *of VCCFinder?*

The ACM's terminology states that researchers conducted a successful *Replication* when they "obtain the same result using artifacts which they develop completely independently"[8].

We were unable to obtain the same results, mostly because we were unable to re-implement ourselves the code based on the paper. This is caused by the lack of details and/or of clarity of the original paper. As an example, even if we had had access to the software that collects the code repositories and built a database[9], we would still miss the complete list of repositories that were involved in the original experiment.

> *We have been unable to* Replicate *the results in the VCCFinder publication.*

Given that the differences in experimental results between our replication study and the original VCCFinder publication may be due to the variations in the dataset or in the learning process, we propose to investigate an alternative approach, that we would make available to the research community, and that could yield similar performance to the promising one reported in the VCCFinder paper.

4.3 Research for Improvement

VCCFinder is an important milestone in the literature of vulnerability detection. Indeed, departing from approaches that regularly scanned source code to statically find vulnerabilities, VCCFinder initiated an innovative research direction that focuses on code changes to flag vulnerabilities while they are being introduced, i.e., at commit

[8]https://www.acm.org/publications/policies/artifact-review-and-badging-current

[9]Note that the link provided in footnote 1 of page 3 in the original post-print publication raises a 404 error.

time. Unfortunately, its replicability challenges advances in this direction. By investing in an attempt to fully replicate VCCFinder and making all artefacts publicly available, we unlock the research direction of vulnerability detection at commit-time and provide the community with support to advance the state of the art.

Considering our released artefacts of **a new replicable baseline**, we propose to investigate some seemingly-appealing variations of the VCCFinder approach to offer insights to the community. Thus, in this section, we go beyond a traditional replication by:

(1) Studying the impact of leveraging a different feature set that was claimed to be relevant to vulnerabilities [1], thus proposing a new approach to compare against VCCFinder (in Section 4.3.1);

(2) Trying to overcome the problem of unbalanced datasets, i.e., the fact that there are much more unlabelled samples than labelled ones (in Section 4.3.2).

4.3.1 Using an alternate feature set

As described above, the feature set used in VCCFinder is not sufficiently documented to be re-implemented, and the VCCFinder authors did not release a tool that is able to extract features from a collection of commits.

In this section, we investigate the use of an alternate feature set, described in a recent publication [1] that is targeting **the detection of vulnerability fix commits, rather than the detection of VCC**. To reduce ambiguity when needed, we refer to this alternate feature set as *New Features*, while the VCCFinder feature set is denoted *VCC Features*.

In this experiment, the settings of the machine learning stay the same as in the replication (LinearSVC with C=1 and the class weight set to 100).

RQ: ***How a less extensive but more security-focused feature set alters the VCCFinder approach?***

4.3.1.1 New Feature Set

The *New Feature set* is made of three types of features: Text-based features, Security-Sensitive features and Code-Fix features. They are all shown in Table 4.4

- Code metrics: A difference between the two feature sets concerning the code is that the new feature set focuses on 17 characteristics of the code, while VCCFinder collects 62 keywords. Though, for each, it also computes whether they are added, removed, the difference of those two factors and their addition.
  Taken individually, most of them are common to the two feature sets. Except for the count of elements under parenthesis, function calls, keywords: `INTMAX`, `define` and `offset`, VCCFinder's feature set includes them all and beyond.
- Commit message: In *New Features*, only the ten most significant words present in the commit message corpus, as obtained through a term-frequency inverse-document-frequency (TFIDF) analysis, are captured.

Note that we tried to normalise the features (as recommended in [223]). The results of detection along the test set were the same or slightly worse with this normalisation step. Thus we decided not to normalise the features.

4.3.1.2 Results

Figure 4.5 and Table 4.5 present the performances with the New Feature Set.

Table 4.4: Alternate set of features (adapted from [1])

ID	code-fix	ID	security-sensitive
F1	#commit files changed	S1	#sizeof added
F2	#loops added	S2	#sizeof removed
F3	#loops removed	S3	S1−S2
F4	F2−F3	S4	S1+S2
F5	F2+F3	S5-S6	Like S1−S2 for continue
F6-F9	Like F2-F5 for *if*	S7-S8	Like S1-S2 for break
F10-F13	Like F2-F5 for Lines	S9-S10	Like S1-S2 for INTMAX
F14-F17	Like F2-F5 for Parenthesized expression	S11-S12	Like S1-S2 for goto
F18-F21	Like F2-F5 for Boolean operators	S13-S14	Like S1-S2 for define
F22-F25	Like F2-F5 for Assignements	S15-S18	Like S1-S4 for struct
F26-F29	Like F2-F5 for Functions call	S19-S20	Like S1-S2 for offset
F30-F33	Like F2-F5 for Expressions	S21-S24	ike S1-S4 for void
ID	text		
W1-W10	Most recurrent top 10 word		

Table 4.5: Confusion Table for New Features

	True Positive(VCC)	False Positives	False Negatives	True Negatives	Precision	Recall
VCCFinder	53	36	166	79 184	0.60	0.24
Ground_Truth New Features	61	9	192	854	0.871	0.241
Unlabelled New Features	61	5 672	192	120 346	0.010	0.241

By considering the Ground Truth only (second line of Table 4.5 and green curve in Figure 4.5), the New Features are less performant than VCC Features. For, still, a recall of 0.24, the precision is only 67% while it used to top at 92% in such a case.

Here again, because of the doubt on what is the actual test set in the original paper (cf. Section 4.2.1.4), we also tested on both the ground truth and the unclassified commits post January, 1st 2011 (red curve in Figure 4.5 and last row in Table 4.5).

> *Our feature set does not allow to outperform our VCCFinder replication.*

4.3.2 Adding Co-Training

A major issue with any VCC detection endeavour is the lack of labelled data, with less than one per cent of the data being labelled. While researchers can collect many hundreds of thousands commits, acquiring even a modest dataset of known VCCs requires a massive effort.

One field of machine learning focuses on the usability of the unlabelled data. The study by [224] states that it is possible, in some case, to leverage unlabelled samples to improve a machine learning model. [225] investigated the potential for gaining information from unlabelled data. This last study concludes that so called active-methods have already proven theoretical efficiency.

In our case, depending on the interpretation of the use of the dataset as explained earlier, unlabelled commits for training (before 2011) are either discarded (*Ground Truth* experiment) or incorporated in the non-VCCs set (*Unlabelled Replication* and *Unlabelled Train Replication*).

> **RQ:** ***Can semi-supervised sorting of unlabelled data improve the VCCFinder approach?***

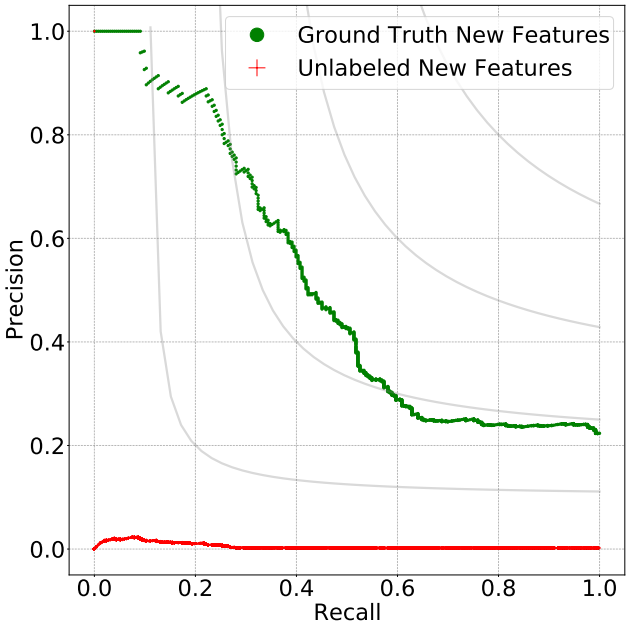


Figure 4.5: Precision-recall performances using New Features

One semi-supervised learning approach, called co-training and introduced by [226], could help answer this question. On a Web page classification problem, [226] used two classifiers in parallel to complete training sets with unlabelled data. They ended up with an error rate of just 5% based on both the page content and hyperlinks over a test set of 265 pages: only 12 pages labelled (3 as positives course-pages, 9 negatives) and around 800 unlabelled. They demonstrated that Co-Training achieved performances on this problem that was unmatched by standard, fully-supervised machine learning methods. It is a technique that has industrially proven a reduction of false positive by a factor 2 to 11 on specific element detection on a video [227], and for which conditions of maximum efficiency it induces were analysed [228].

4.3.2.1 Co-Training Principle

When trying to detect VCCs, an important point is that unlabelled commits are unlabelled not because they are not VCCs, but because it is unknown whether they are VCCs. Arguably, in any large-enough collection of commits, it is reasonable to assume at least some of them are actually VCCs.

The insight behind trying Co-Training with VCC detection is the following: By building two preliminary and independent VCC classifiers, the unlabelled commits predicted to be VCCs by both classifiers could be used to augment the training set. By repeating this step, it might be possible to leverage the vast space of unlabelled commits.

4.3.2.2 Description of the algorithm

[226] showed that the co-training algorithm works well if the feature set division of dataset satisfies two assumptions: (1) each set of features is sufficient for classification, and (2) the two feature sets of each instance are conditionally independent given the class.

Both the VCC Features set and the alternate feature set can be split into two subsets

of features: One based on code metrics, and one based on the commit message.

Previous work on security patches detection showed that, for the New Feature set, the two resulting feature subsets are independent, and thus satisfy the two main assumptions for Co-training [1].

Once these two assumptions are satisfied, the Co-training algorithm considers these two feature sets as two different, but complementary *views*. Each of them is used as an input of one of two classifiers used in Co-training: One focused on code metrics, and the other on commit messages. The algorithm is given three sets: a positive set, a negative set, and a set of unlabelled.

As described in Algorithm 1, and shown in Figure 4.6, the training process is an iterative process in which each classifier (**h1** and **h2** on Figure 4.6) is initialised being just given the labelled inputs **LP**, that is used as the ground truth. From the whole set of unlabelled, a subset **U'** is randomly selected.

At every round, each classifier is trained on a labelled set (**LP** for the first round). Then a number of unlabelled commits from **U'** are classified with those two classifiers. When both classifiers agree on a commit, this commit is added to the ground truth, i.e., it will be used to augment the training set in the next round. The process keeps going until we reach a predetermined size of the labelled set.

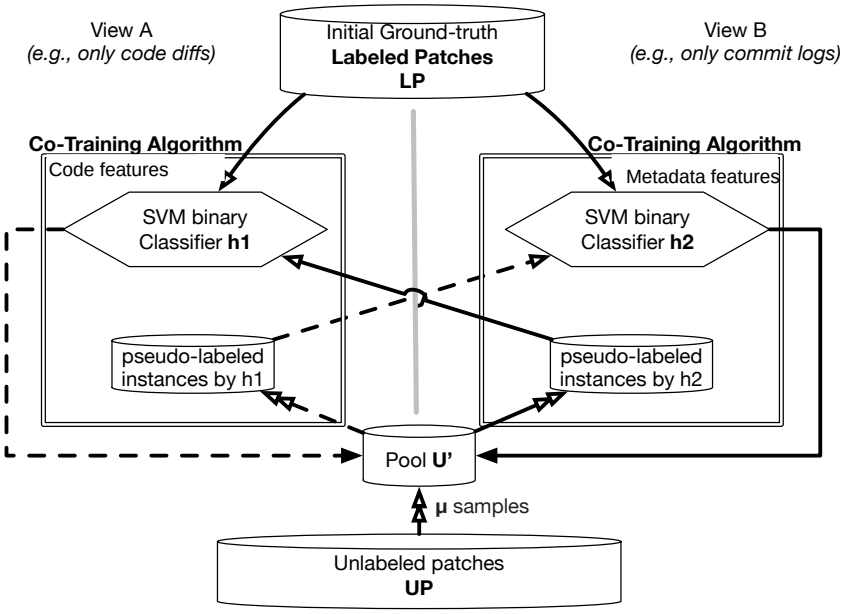


Figure 4.6: Co-Training (Figure extracted from [1])

4.3.2.3 Implementation

For the implementation of the Co-training, we select two Support Vector Machines (SVM) [229] as classification algorithms. We also perform experiments using three different size limits of the training set: by 1000, 5000 and 10 000 unlabelled commits added.

This variation enables us to compare the effect of this variable in prediction performance. To respect temporality, the unlabelled commits were all taken before January, 1st 2011, as was for the original unaltered training set. For both sets of features, the co-training occurs after the extraction of features. One classifier trains on the code metrics and the other on the metadata. We finally use, as for the replication, a LibLinear

Algorithm 1: Steps for each Co-Training iteration. (extracted from [1])

input : training set (LP), unlabelled data (UP)
input : pool U'

output : U': updated pool
output : LP: updated training set

Function *getView(x, classifier)*
  if $classifier = C_1$ **then**
    return $Text\_features(x)$
  return $Code\_features(x)$

Function *buildClassifier(first)*
  $vectors = \emptyset$;
  if $first = True$ **then**
    foreach $x \in LP$ **do**
      $vectors = vectors \cup getView(x, C_1)$;
  else
    foreach $x \in LP$ **do**
      $vectors = vectors \cup getView(x, C_2)$;
  $classifier \leftarrow train\_model(\text{SVM}, vectors)$;
  return $classifier$;

$h_1 \leftarrow buildClassifier(True)$; $h_2 \leftarrow buildClassifier(False)$;
$(P_1, N_1) \leftarrow classify(h_1, U')$; $(P_2, N_2) \leftarrow classify(h_2, U')$;
$LP \leftarrow LP \cup random\_subset(\#p, P_1) \cup random\_subset(\#p, P_2)$;
$LP \leftarrow LP \cup random\_subset(\#n, N_1) \cup random\_subset(\#n, N_2)$;
$U' \leftarrow U' \cup random\_subset(\#2 * (p + n), UP)$;

model to classify the commits of the test set. For the latter values of C is 1 and, still, the weight of the class to 100.

4.3.2.4 Co-Training Results

4.3.2.4.1 Co-Training with VCC Features

Performance is improved slightly (cf. Figure 4.7 vs Figure 4.2) when Co-Training is used in conjunction with VCC Features. This improvement, however, does not appear to change with the size increase of the training set (whether 1000 or 10 000).

When testing with the Unlabelled Test, performance drops for all attempts. Therefore, no improvement can be concluded in this aspect.

4.3.2.4.2 Co-Training with New Features

Figure 4.8 presents the results for a Co-Training process based on New Features. It includes variations for the training set (with 1000 and 10 000 unclassified commits) and, tests with and without the unclassified commits. On testing without the unlabelled Test set, one can conclude that the increase of 1000 unlabelled already helps perform better than the baseline green curve of Figure 4.5. An increase of the dataset by 10 000 is further contributing to detect more VCCs.

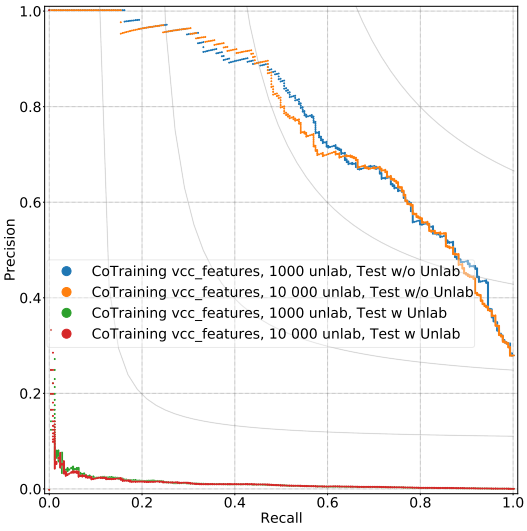


Figure 4.7: Co-Training Performance using VCC Features' set

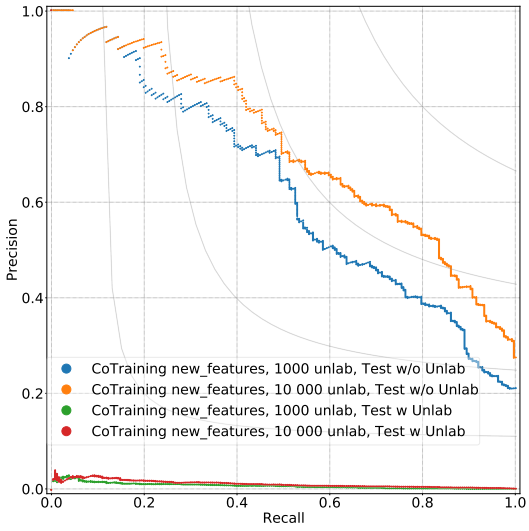


Figure 4.8: Co-Training Performance using New Features set

4.3.2.5 Co-Training Analysis

> *The Co-Training we implemented does not seem to be of particular help for the identification of VCCs.*

This finding is clear when we consider the unclassified commits, in which cases the performance metrics dramatically drop. There seems to be an effect, though, for the New Features when only considering the Ground Truth.

4.4 Conclusion

Vulnerability detection is a key challenge in software development projects. Ideally, vulnerabilities should be discovered when they are being introduced, i.e., by flagging the suspicious vulnerability-contributing commits. VCCFinder, presented in 2015 at the CCS flagship security conference held the promise of detecting vulnerability-contributing commits at scale using machine learning. Since the research direction that this approach initiated has not boomed since then, we have proposed to revisit it. First, we attempted (and failed) to replicate the approach and to replicate the results. Then, we propose to build an alternative approach for the detection of vulnerability-contributing commits using a new feature sets (whose extraction is clearly replicable) and a semi-supervised learning technique based on co-training to account for the existence of a large set of unlabelled commits. Our experimental results indicate that the proposed approach does not yield as good performance as the ones reported in the VCCFinder publication. Nevertheless, it constitutes a strong and reproducible baseline for the research community. Our artefacts are publicly available at `https://github.com/Trustworthy-Software/RevisitingVCCFinder`

5 Prevention Efficiency: Defense Mechanism, the case of AOSP

Besides vulnerability detection, the implementation of defense mechanisms is an alternative to limit the exploitation of vulnerabilities. In this chapter, we review the defense mechanisms proposed by the AOSP developers over a period of ten years. First, thanks to available information, we trace the introduction and evolution of these defense mechanisms. We further verify at the binary level with dedicated tools, which mechanisms related to memory corruption were implemented and where. Then, we confront the evolution of Memory Corruption related vulnerabilities disclosure with the introduction of Defense Mechanisms.

Contents

5.1 Motivation

Code exploitation is a long recognised issue. In 1972, the U.S. Air Force released a report [144] describing how resources shared on a system, such as memory, could be accessed and modified by unintentional user[1]. This practice gained further publicity in 1996, with an article [230] providing explicit methodology for hijacking the flow of execution by overwriting the stack.

As we did with VCCFinder, one can attempt to detect predisposing conditions to infringe to the security policy (i.e. vulnerabilities, see Section 2.1.1). It is, nonetheless, not the only way to fight this battle. Another approach is to implement rules, tools and methods that will prevent the exploitation of the vulnerability or cut the flow of this exploitation before it achieves its targets.

Additionally, vulnerability detection will enable to correct one or a few vulnerabilities while a defence mechanism hardens the system against a range of vulnerabilities. This way around, efforts can be maximised implementing well-motivated defence mechanisms.

In this chapter, we focus on the defence mechanisms that AOSP implemented over more than ten years. After a description of our methodology, we list them chronologically. We also classify them according to the type the vulnerability they attempt to protect from. We both use information collected on the internet and analyse AOSP binaries. Finally, we investigate the influence of these defence mechanisms on the system safety through the evolution of disclosed vulnerabilities.

5.2 Methodology

5.2.1 Listing of Defense Mechanisms implemented

The first part of the work, resides in gathering the list of defense mechanism that have been implemented in AOSP up to API-level 30. To that end we manually extract information that are provided in the following blogs, websites and articles:

- Android versions security releases[2].
- Google Security Blog [231]
- Android developers blog [232]
- Android Security Experts publications [164, 233]
- Clang website [234]

5.2.2 Binary extraction

As the information provided in these blog may be incomplete and of varying quality, we decided to get data one level deeper. We do so by analysing Android binaries for API-levels 10 to 28. We downloaded one x86 system-image for each Android API-level available in Android Studio. Then we analyse the binaries with the tools **hardening-check**[3] and **checksec**[4].

- `hardening-check` is a utility developed for Debian that "*examine*[s] *a given set of ELF binaries and check for several security hardening features*". It was developed by Kees Cook for debian and is since 2012 under GNU General Public Licence version 2. `hardening-check` enables to verify if the binary file:

[1] `https://csrc.nist.gov/csrc/media/publications/conference-paper/1998/10/08/proceedings-of-the-21st-nissc-1998/documents/early-cs-papers/ande72a.pdf`

[2] As for Android 8.0: `https://source.android.com/security/enhancements/enhancements80`

[3] `https://manpages.debian.org/testing/devscripts/hardening-check.1.en.html`

[4] `https://www.trapkit.de/tools/checksec/`

Table 5.1: Number of binaries per Android version in system.img

API-level	10	15	16	17	18	19	21	22	23	24	25	26	27	28
Nb binaries	212	274	306	323	313	337	365	370	384	461	464	499	529	660

  - is compatible with ASLR (through what is called **PIE**),
  - implements stack canaries (checking that this memory is not being tampered with)
  - is written with a modified set of instruction provided by **Fortify_Source** so to prevent a significant number of buffer overflows
  - provides instructions to mark certain areas as *read-only* for execution time (called **RelRO**)
  - further improves the **RelRO** by ordering the program to resolve all addresses before the program is executed
  - prevents the *Heap* and the *Stack* to overwrite each other (**Stack Clash**)
  - checks for a `-fcf-protection` flag that indicates the binary is instrumented to protect the execution of the flow of instructions with Intel's implementation of **CFI**, called Control-flow Enforcement.
- `Checksec.sh` is a tool initially developed by Tobias Klein and that knew its first release in January 2009. It is now under BSD Licence.
  `Checksec.sh` can also provide with PIE, Fortify_Source, Stack Canaries, RelRO but it can also provide with whether the Stack is executable or not(**NX**), with RPath and RunPath.

The analysis starts by mounting each system-image and select all the binaries. Those binaries are the **E**xecutable and **L**inkable **F**ormat identifiable by their header, starting by 0x7F 0x45 0x4c 0x46, or 0x7F 'E' 'L' 'F'. The number of resulting binaries for each API-level is provided in Table 5.1. On each of those binaries, we run the security check tools and collect the results in file summarising their level of protection. We further provide our results in Figure 5.2.

We exclude **.odex** files as they are adjacent-to-applications files that optimise applications' boot up. They are only generated when an application is installed on the device. An isolated analysis of earlier-mentioned **.odex** files confirm that these files would rather confuse the reading has `hardening-check` only finds them to implement **PIE**.

5.3 Timeline of the implemented Defence Mechanism over Android versions

In this section we describe Defence Mechanisms presented in the Figure 5.1 timeline using the information collected as described in Section 5.2.1. We further detail these mechanisms, the moment of their introduction and improvement(s) following five categories that differentiated as we collected the defense mechanims. We start with (i) Cryptographic Improvements that gather the differente methods to encrypt data, either to send it or store it. We continue with (ii) Access Control that we separate in three as in Google's developers article [164]. We mention therefore the evolution for Android Permissions, Unix Access Control and SELinux Access Control. We also explain how (iii) Authentification improved with new features. We list the Memory Corruption related defense mechanisms in (iv). We finish with a category regrouping (v) Other mitigations techniques.

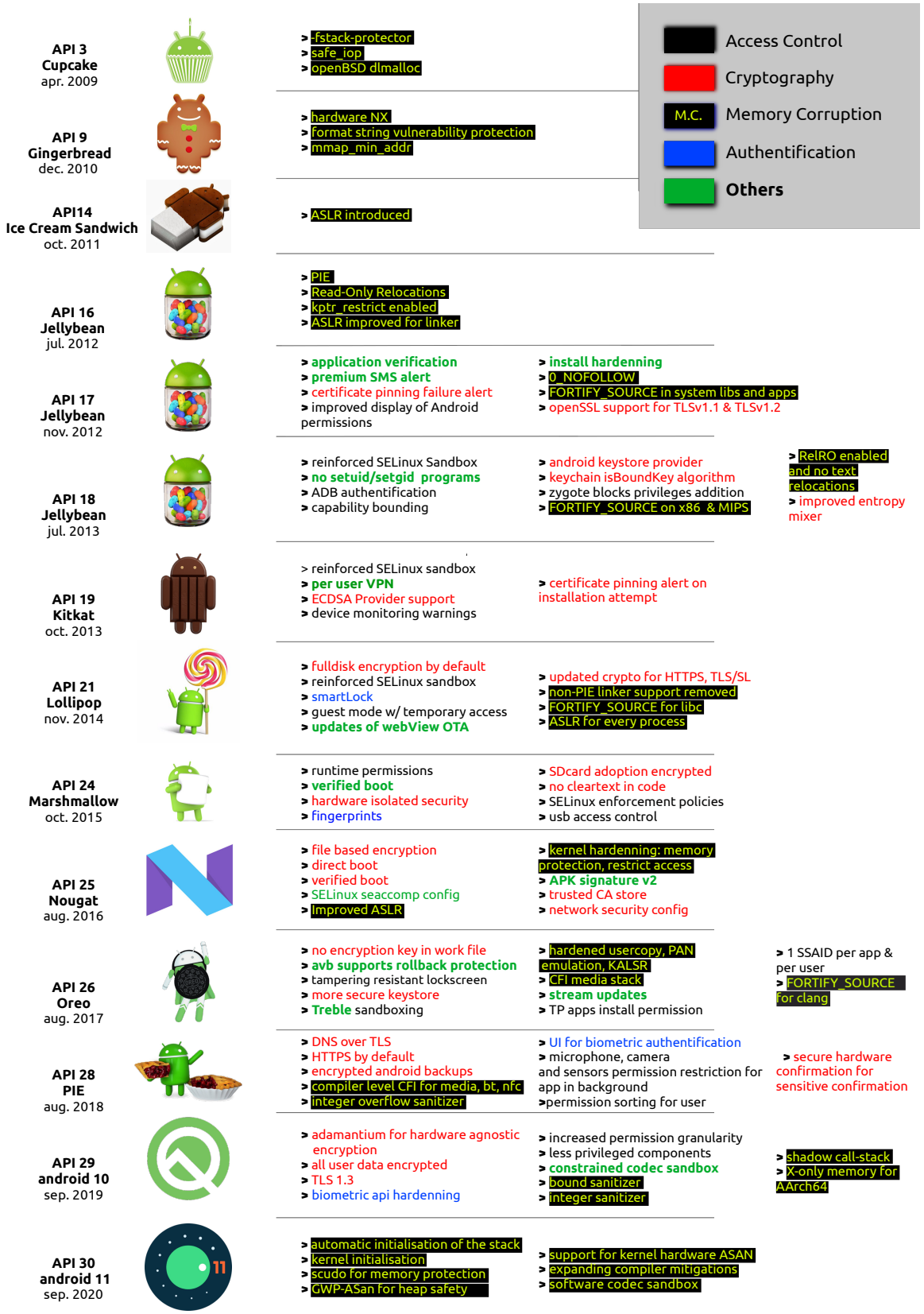


Figure 5.1: Defence Mechanisms Timeline in AOSP

5.3.1 Cryptographic Improvements

Data becomes encrypted by default, and as a block, in the local storage by API-level 21, in 2014. In 2015, the SDcard can also be encrypted as an extension of the local storage. In API-level 25, the practice is updated with a finer granularity as they are now encrypted at file level. It is first the backed-up applications' data of users that is encrypted starting API-level 28; then any user-related data by API-level 29.

Another narrative relates to regular updates to the latest and safer cryptographic algorithms and libraries. In API-level 17 (2012), OpenSSL support for TLSv1.1 and TLSv1.2 was introduced (respectively defined in 2006 and 2008). This cryptographic protocol is extended to the DNS request in API level 28 (2018) and updated to TLSv1.3 in API-level 29. API 19 (2013) introduces support to Elliptic Curves.

A last angle provided relates to certificates. The user notified that a certificate verification failed before rendering the content the device attempts to access in November 2012. A year later, the system starts to monitor the installation of certificates attempting to impersonate Google's authority. In 2016, Android is provided with an independent, trusted Certificate Authority store for system activities[5]. To be reached, this Trust CA Store needs either a rooted device or an emulated Android phone[6].

5.3.2 Access Control

Android Permission System

Android higher layers follow a Discretionary Access Policy control based on permissions. Over the years, much effort was put into increasing users' control, and understanding of permissions. It also appeared with the aim of better controlling the way processes inherit their privileges from parent processes.

Almost looking like an answer to recommendations provided by recent research [166], the hierarchy of android permission was changed and the display improved in API-level 17[7].

Then, by API-level 24 Android permissions were no more granted once and for all at installation time but could also be granted at runtime, upon request. Permissions became individually revocable and not only as a bulk. Beforehand, rejection of one permission meant rejection of the application's installation.

Eventually, by API-level 29 it became possible to grant specific permissions to the app only in usage. In other words, applications with such permission type granted cannot accede the same functionalities when they are in the background.

Permissions are divided into several risk categories; elevating its permissions without properly requesting them is a whole field of attacks that AOSP is vulnerable to [235].

Reducing the privileges of specific components (API-level 29, Sept.2019), plus the gain in granularity prevented different Confused Deputy attacks. These attacks enabled a third-party application to abuse other applications' permissions they want to access unknowingly of the user and the device [236].

Unix Access Control

In API-level-18 (2013), processes created from zygote (the process from which all processes are forked) cannot inherit the permissions of the file's owner. One desired

[5] `https://android-developers.googleblog.com/2016/07/changes-to-trusted-certificate.html`

[6] `https://medium.com/hackers-secrets/adding-a-certificate-to-android-system-trust-store-ae8ca35`

[7] Section Security Changes: `https://developer.android.com/about/versions/jelly-bean#android-4.2`

consequence is that these generated processes cannot change file and group ownership.

In API-level 19 (2014), a design flaw is corrected as users can now set up a VPN for themselves. API-level 24 (2015) saw a change in the policy of the home directory, passing to root only [164]. In API-level 26 (2017), an android advertisement identifier (SSAID) is provided per application and user. It can, however, be shared between applications from the same developer and enables to share states between these applications (i.e., logging in one app, logs in the other).

SELinux Access Control

SELinux Mandatory Access Contol was introduced in 2013 (API-level 18). It coincides with criticism of the lack of control over the origin of Kernel calls [237]. At first, it concerned only four processes: `installd`, `netd`, `vold` and `zygote`. These are respectively: installation daemon, network connectivity daemon, volume events daemon and process creation. API-level 21 (2014) has expanded its application has been expanded to all userspace processes. Several rules were implemented in API-level 25 (2016) to tighten the application sandbox. In API-level 26 (2017), the introduction of Treble permitted SOC vendors to update their own SELinux rules more easily and independently from the rest of the OS.

5.3.3 Authentication

The use of fingerprints was enabled in API level 24 for devices with supporting hardware. It was further improved by API-level 28 to make it available to all applications as a secure way to access applications resources. In API-level 29, face recognition was provided with the same level of security fingerprints were given in API-level 28. Through Gatekeeper, the hardware provides a specific Trusted-Execution Environment (an isolated trusted architecture) in which the checks are executed[8]. The smartLock mechanism described in Section 2.3.1.3 was introduced in 2014.

5.3.4 Memory Corruption Prevention

Since the first Android versions, many defences mechanisms have been deployed which have made the exploitation of memory corruption vulnerabilities more difficult.

The Stack Canaries were implemented early, by API-level 3 in 2009. Canaries are memory locations on the stack initialised with specific values which are not supposed to be altered during the execution of the program. If a canary is modified, the system assumes that an undesired behaviour attempts to overwrite the stack, and the execution stops. At the same time, a first sanitising of integer operations was introduced in the code verification with `safe_iop`[9].

The stack is also made non-executable in API-level 9 (in 2010), which means data on the stack cannot be interpreted as a list of instructions. Later, in API-level 29, and for AArch64 architecture, this mechanism is hardened by restricting the spans in memory that can be interpreted as code, rather than specifying which spans cannot.

API-level 16, July 2012, saw the introduction of Read-Only Relocation (or RelRO). Executables, given different devices and different executions, may be loaded in memory at different addresses. One element loaded in memory is the Global Offset Table, which retains the different offsets enabling to access appropriate elements independently from the changing context. RelRO hardens the GOT to protect it from being overwritten.

[8] `https://source.android.com/security/authentication/gatekeeper`

[9] `https://android.googlesource.com/platform/external/safe-iop/+/b805514f31a231a0e78a18f296c0454fcadead1a/src/safe_iop.c`

This defence mechanism can be improved with Immediate Binding. The latter forces addresses to be resolved at load time. This closes the window between load-time and run-time for the alteration of the GOT.

Another mechanism, ASLR, was developed iteratively over AOSP's layers and sometime needed corrections to be functional. When introduced in API-level 14 (2011), it was flawed. It only had an impact on the position of the stack but not on the heap, on the linker, nor on libraries[10]. Furthermore, the memory layout was shared between all Android applications because the randomisation only happened during boot [238]. Thus, leaking the memory layout of one application made it possible to guess any application's layout. Since 2014 (API 21), all binaries are required to be **P**osition **I**ndependent **E**xecutable. Thus both the code and the Process Linkage Table (i.e., the table that holds the addresses where other code functions are written) are randomised. ASLR is further strengthened in API-level 25 (2016) with an improvement of randomisation functions for libraries.

Other defence mechanisms were developed following an iterative process, starting in most vulnerable modules. So happens for Control-Flow Integrity. CFI was first deployed in the media stack in API-level 26 and then in Bluetooth and NFC components by API-level 28. It is a possibility provided by the change of compiler from GCC to LLVM's Clang, achieved in API-level 26. API-level 29 saw a complementary improvement of CFI called Shadow Call-Stack. The Call Stack is duplicated in a secure memory location and used to check unusual changes in the regular stack.

`safe_iop` was only dropped in 2018, when API-levels 28, 29 and 30 hardened the system with better sanitising. Integer Sanitisers were, first, expanded to more libraries than just the media ones (i.e., `libui, libnl, mediaplayerservice, libexif, ...`) in API-level 28[11] and, then, to software codecs in API-level 30 (i.e., `FLAC, hevdec, mpeg2, ...`). The development team also introduced Bound Sanitisers in API-level 29, which will enforce the check of indexes and sizes of arrays. Last but not least of these sanitisers to be mentioned are Address Sanitisers, implemented in randomly selected system process and the kernel at boot. They help collect use-after-free and heap-buffer overflows. It is a technique that has already managed to unravel several vulnerabilities in Google's Chrome Web Browser[12].

5.3.5 Other Improvements

Finally, the Others category regroups spare categories but nonetheless relevant to how widespread and diverse is the range of action aiming to increase Android security level. Treble, a system to deliver OS patches independently from the device vendors, was implemented in API-level 26 [233]. It is the generalisation of an exploit prevention technique that already had happened with API-level 21 for a specific and sensitive component: webView. This component is used to render Internet resources through an application. It was a favoured gate to access a device's resources as providing the best and the worst of both worlds between applications and dynamically loaded web-content [239, 240] It was further bypassing checks for installation of non-explicitly authorised content[13]. Thus in 2014, the Webview component was made updatable aside

[10] https://threatpost.com/analyzing-aslr-android-ice-cream-sandwich-40-022112/76239/
[11] https://android-developers.googleblog.com/2018/06/compiler-based-security-mitigations-in.html
[12] `https://bugs.chromium.org/p/chromium/issues/list?q=Hotlist%3DGWP-ASan&can=1`
[13] https://labs.f-secure.com/archive/webview-addjavascriptinterface-remote-code-execution/

from the rest of the system through the PlayStore (or Over The Air).[14]

Google also realised that 90% of the vulnerabilities in the upstream kernel had already been patched in the upstream kernel [193]. Better than developing their own patch, or cherry-picking the fixing patch from the upstream updated kernel into an Android Common Kernel, it appeared more advantageous to keep the kernel updated with the latest version of the LTS upstream kernel. Hence the kernel was now divided into two clear parts: the AOSP Generic Kernel Image and vendor modules. Each can now be updated independently, reducing further the exposure window.

Version rollback attacks are targeted starting API-level 24, in case an undesired third party gains physical access to the device.

We can also observe several improvements made to make the user aware of doubtful action,such as when sending premium SMS (API-level 17, Nov.2012), or installation of applications with unverifiable origin(e.g., API-levels 17 and 26).

5.4 Binary Analysis

The collection of information sources to provide the above timeline is heterogeneous, providing data of different quality. This data could be incomplete, or for instance, specific defence mechanisms may only partially cover some layers or modules of AOSP. Therefore, specifically for Memory Corruption, we undertake an analysis of the binaries of each version of AOSP from API-level 10 to 28, for which we found an file-system image in Android Studio.

As explained in Section 5.2.2, we gathered the binaries present in the x86 `system.img` file-system for each Android version available for download in **Android Studio**. To these binaries we apply the tools `hardening-check` and `checksec.sh`.

The evolution of defence mechanisms is illustrated in Figure 5.2. The black line represents the total number of binaries analysed in for each file-system. We observe that there is a constant increase of the number of binaries and an overall increase of the proportion of binaries featuring defence mechanisms. In the following, we describe our observations for the different defence mechanisms.

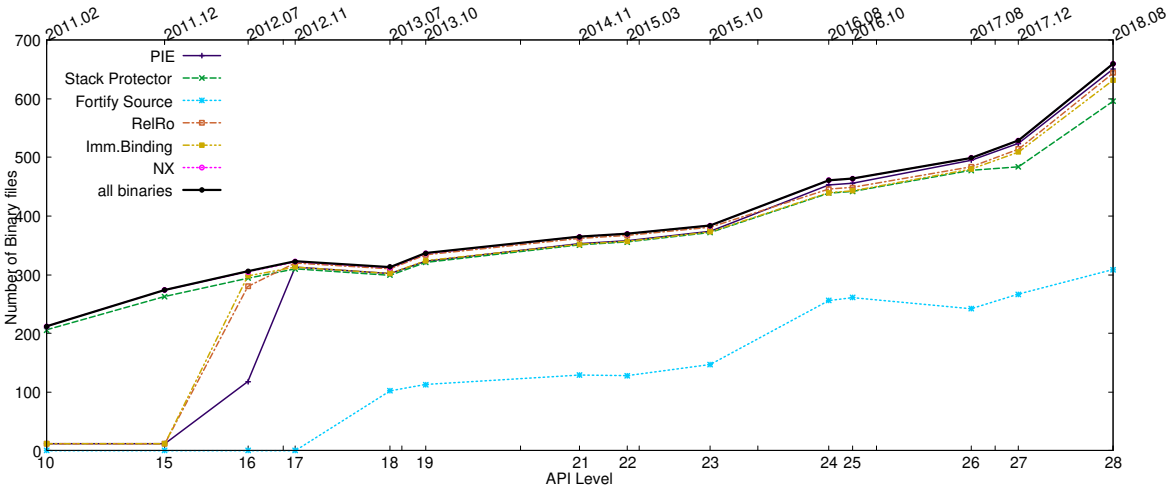


Figure 5.2: Evolution of mechanisms implemented in binaries over APIs 10 to 28

Non-e**X**ecutable: All binaries of all studied Android versions from API-level 10

[14] `https://support.google.com/work/android/thread/103741642/force-managed-play-update-of-google-chrome-and-system-webview?hl=en`

to 28 have their stack non-executable. This matches with Figure 5.1 as the stack is enforced to be non executable since API-level 9

PIE: Our analysis indicates that **P**osition **I**ndependant **E**xecutables have been introduced in API-level 16. However, in Android API-level 16, PIE is activated on only 36% of the binaries. 95% of the binaries without PIE are from the `lib` folder, and almost all will be found with PIE activated in the following version, released four months later. In API-level 17, PIE is indeed applied to almost all binaries, with only 10 binaries not PIE-compliant. For all the following versions, the number of binaries for which we found no evidence of being compliant to PIE oscillates around 10. The binaries that do not provide signs to be Executable Independently from the Position tend to be the same over the versions. We note these could be artefacts of using versions from Android Studio for emulation. In android API-level 28, they are six if we do not count the 64 to 32 versions of these: `adbd` (android debug bridge debugging), `gdbserver`, `mdnsd` (local broadcasting), `bpf_kern.o` (monitoring and debugging the kernel [15]), `micro_bench_static`, and `simpleperf` (code performance analysis).

Stack Protector: From Android API 10 to API 26, the proportion of binaries with *Stack Protector* activated remains stable around 96%.

However, we notice that starting with API 27, this proportion slightly decreases, lowering to 90% in API 28.

This early implementation confirms information provided in Figure 5.1 with the early implementation of this Defence Mechanism in API-level 3.

Fortify Source: Our analysis of binaries finds that the first binaries using Fortify Source in the API-level 18. This confirms information provided in Figure 5.1 for x86 architecture. However, only 33% of the binaries provided do implement Fortify Source in this version of Android. This number only slowly increases until API-level 24, in which 55% of the binaries are protected. The absolute number of binaries protected then stays overall constant until API-level 26. So to remember, it is in API 26 that the Android development team provides full support of Fortify for LLVM's Clang after a switch from GCC. From then, the absolute number of binaries protected rises in absolute while representing still around 50% of all binaries. Overall, the implementation of Fortify Source is different from other defence mechanisms. It seems to take its own pace and to be more independent from the global number of binaries. For example, the number of binaries protected in API-level 25 is higher than in API-level 26, while the global number of binaries rises.

Read-Only Relocation: This defence mechanism started to be implemented in API-level 16 with 91% and keeps representing this percentage of all binaries, if not higher when reaching 95% with API-level 28. One explanation for why some binaries do not hold such protection could be that we are working from file-system images provided by Android Studio that could hold specific debugging binaries.

Immediate Binding: Starting with API-level 17 and until API-level 26, the number of binaries implementing Immediate Binding is almost equal to the number of binaries with only 10 to 20 binaries present that do not provide it in each version. Strengthening of RelRo, covers almost all the binaries holding RelRO, but not all.

CFI: The hardening-check tool seeks Intel's implementation of CFI flag, called **C**ontrol-flow **E**nforcement. Therefore it does not enable us to acknowledge the introduction of LLVM's implementation of CFI in Figure 5.2. However, AOSP's source code make it possible to see the spread of CFI along the different components. First, by December

[15] `https://source.android.com/devices/architecture/kernel/bpf`

2016[16], CFI was implemented on media libraries `libmedia` and `libstagefright`, only to be released in API-level 27 (December 2017). It is since spreading among components as provides a dedicated makefile[17] from April 2018. The last extension regards the layer responsible for abstracting the hardware (called HAL), set under protection of CFI in January 2021[18]. It makes now twenty-one components covered.

Duration of project from 0% to 100%: From our analysis, we can conclude that, generally, once the Android development team decides to introduce a given Defence Mechanism in AOSP, this mechanism ends up being applied to the vast majority of binaries in the span of only 2 or 3 versions. Fortify_Source stands as an outlier to that regard. Indeed, after a long-running effort spanning more than 10 versions, Fortify_Source is applied to only around half the binaries. Of all the defences we measured here, it is also the only one that saw a period of regression, where both the absolute number of covered binaries, and the proportion of covered binaries decreased between API-level 25 and API-level 26. One likely hypothesis for this slower development is that the team working on Fortify_Source was focusing on the change to LLVM and not on increasing the system coverage. Nonetheless, the project of using Fortify_Source in more and more binaries seems to still be dedicated resources as the number of covered binaries keeps increasing in the latest versions we analysed.

5.5 Investigating impact of Defence Mechanisms on CVEs

The category of defence mechanisms that received the most attention is Memory Corruption. They represent 35 different implementations and/or improvements in Figure 5.1 versus 25 for cryptography. It is an understandable behaviour given the figure provided by Google (see Section 2.3.1.1) that they considered by 2016 that 86% of the vulnerabilities affecting android to be related to Memory Corruption.

The defence mechanisms most often target the risk for vulnerable code to be exploited better than necessarily preventing the apparition of vulnerable code. We can, however, expect these defence mechanisms to make the exploitation more complex: requiring more time and effort. As we discussed for bypassing ASLR (needing a leak of the memory mapping to exploit a buffer overflow), it becomes required to combine vulnerabilities to succeed in the exploitation of vulnerable code. The reduction of available techniques to attackers, and the increasing skills required to manage them, could be correlated with a reduction of the number of vulnerabilities that need to be fixed.

We thus try to observe if their is a correlation between the number of exploitable code and the introduction of Memory Corruption defence mechanisms.

5.5.1 Memory Corruption

5.5.1.1 Description

Figure 5.3 presents the introduction timeline of memory corruption related defence mechanisms (presented in Figure 5.1) and the evolution of declared memory corruption related CVEs (as seen in Figure 3.9). We focus on the different steps followed in the implementation of ASLR, Fortify_Source and CFI.

[16] `https://android.googlesource.com/platform/frameworks/av/+/a4a6d63ec590a3be60a60527c619fb0bf7870b59^!`

[17] `https://android.googlesource.com/platform/build/+/e003a0a6cec52c2a8bd561673509f3a34bc5c052^!`

[18] `https://android.googlesource.com/platform/build/+/08764606dee8784d6251a92459cdae673001f5cb`

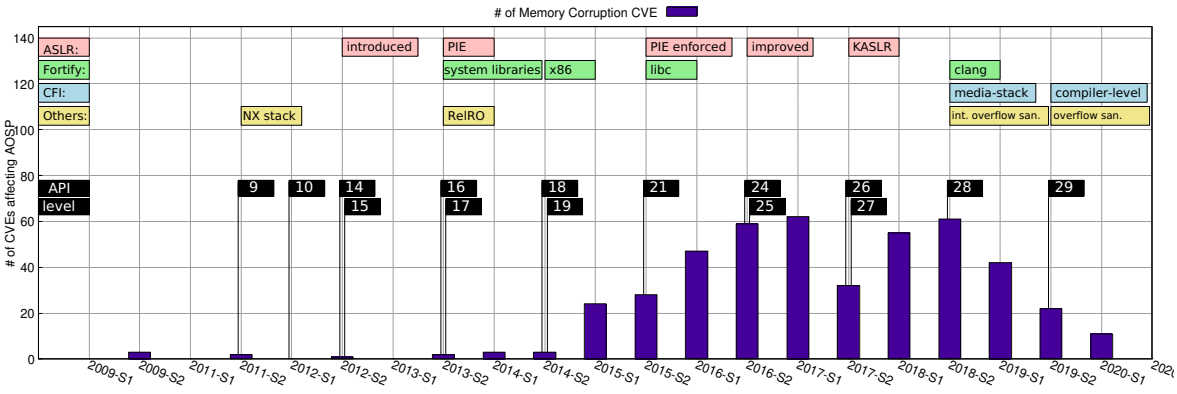


Figure 5.3: Evolution of the number of Memory Corruption related Vulnerabilities related to the introduction of Memory Corruption related Defence Mechanisms

Many of the defence mechanisms were implemented, or knew a first implementation, before 2016. It affects the capacity to conclude as the most meaningful pool of data came in 2015 with Android bulletins.

We saw in Section 3.3.6 that the number of Memory Corruption vulnerabilities has been rising up to 2017, and began a noticeable decrease in 2019.

There is a time of high variations, between those two periods during which GCC's FORTIFY_SOURCE was already implemented, PIE was already enforced and that saw ASLR's entropy to be improved (API-level 25).

Also, the apparent decrease happens after the release of API-level 27 that, as we saw, provided a fully-capable FORTIFY_SOURCE for clang, and the introduction of CFI in the media stack.

5.5.1.2 Analysis

We cannot attribute directly any effect on the curbs of CVEs to any introduction of a specific defence mechanism. First, because the data quality is not homogeneous over time (before vs after Android bulletins). Second, because of the introduction and the improvements of different mechanisms overlay. Third, it is not possible to exclude external causes to the focus we provide. As a reminder, the global count of vulnerabilities drops from 2018 on.

There is however a significant drop. More progressive than the one of the second semester of 2017, and not recovering as far as the data we analyse go. This trend covers a year and a half, thus 18 months and as many Android bulletins, and can be considered encouraging. Nonetheless, we cannot consider it definitive because of the before-mentioned limitations.

Another piece of information highlights that the Android team keeps chasing memory corruptions regardless of the trend up to API-level 28. In the last versions, API-levels 29 and 30 (over the scope of CVEs presented in Figure 5.3) studied in Figure 5.1. The development team keeps hardening the system with each version against Memory Corruption. It can also be seen as a further encouraging signal that now, Google wants to anticipate potential existing attack vectors.

5.6 Discussion

This discussion is divided into three different points. We first mention threats to this study, reminding the hypotheses on which the analyses are based. We then list

key points in the story of the implementation of defence mechanisms we believe to be worth remembering and exporting to other programs. And finally, we prospect to what can also be done for AOSP from where it stands.

Threats to the study: We begin reconsidering the nature of the relationship between the evolution of vulnerabilities and the implementation of defence mechanisms.

[Actual Impact of Def.Mech.] Defence mechanisms target exclusively the introduction of a vulnerability. They can very well make the target program resilient to the impact or the exploitability more complex for triggered vulnerabilities. Hence a vulnerability, as a Buffer Overflow, may still be exploitable after the implementation of certain defence mechanisms, as making the stack Non-Executable. In other cases, as with the Clang's UBSanitizer for integer sanitising [241] (API-level 29 and 30 in Figure 5.1), the source code may present the possibility of an overflow. Nonetheless, the compiler will instrument the code to monitor the evolution of integers and be redirected in error-catching branches in case a program leads to an overflow of one of these values; therefore preventing the exploitation. We have to acknowledge that the link of causality is not complete.

[To patch or not to patch] In contrast, when security experts consider the exploitation to be rendered extremely unlikely or when the resulting advantage does not compensate for the effort to complete the exploit: then the decision of the developing team can be not to patch the vulnerability. In such a case, vulnerabilities hardly impacting after the implementation of a defence mechanism will not appear in our statistics (as not patched as a security issue). Consequently, the evolution of the number of patched CVEs may be more correlated to the actual level of exposure of a system than implied in the paragraph just above.

[Context] This last point underlines another threat to our analysis: a patched CVE does not carry the same impact in different contexts. Not only due to its severity (i.e. end-impact on the system), but a vulnerability that needs to be patched in early versions does not mean the same as a vulnerability exploitable when several defence mechanisms have already been implemented. In the second case, the vulnerability is still exploitable while the cost of implementing several defence mechanisms has already been engaged. In the first case, late defence mechanisms may have prevented the exploitation. Thus, representations as in Figure 3.1 consider with the same weight vulnerabilities of different nature.

[Data Quality] Additionally, we take most of our data in Android Security Bulletins. Our analyses depend on the level of transparency they adopt on vulnerability patches. We also, *de facto*, integrate without clarifying it, the definition of what is a vulnerability that needs to be patched and what is a vulnerability that does not (or that is considered a bug and patched as such).

[Detection Efforts] Finally, the relevance of using patched vulnerabilities is based on the crucial hypothesis that analysts deploy enough effort to detect vulnerabilities that are detectable with the current State-of-the-Art techniques, and that they disclose these vulnerabilities responsibly. For instance, in case the attention of analysts shifts to other products, or if there is no incentive to detect vulnerabilities in AOSP, the reduction of the number of vulnerabilities cannot be considered as a consequence of an actual hardening of the platform [76]. The level of adoption of Android and the diversity of security-needy features provided by Android protects us from this threat, but it has not been precisely measured. This gap also underlines the necessity for new detection methods to be tested and implemented. With time, the easiest-to-spot

vulnerabilities are detected, and it becomes incrementally complex, and costly, to detect newest ones. Additionally, these vulnerabilities that evade defence mechanisms are particularly worrying, and underline that defence mechanisms and vulnerability detection are complementary.

Good habits: We cannot measure precisely either what would be the vulnerability trends, had Android developers not implemented these defence mechanisms. We can, however, notice several measures that ought to produce beneficial outcomes, regardless of our capacity to measure them.

[Monitoring & Transparency] First, the list of defence mechanisms integrated into the Android Open Source Project, however spreading over more than ten years, demonstrates a genuine will to anticipate future threats; either these unknown threats are existing, yet uncovered, vulnerabilities or ones introduced in the future. It covers all the types of vulnerabilities we have observed AOSP to be subject to. In the case of Memory Corruption, it also demonstrates a continuous focus on developing newest, safest, even if costly, defence mechanisms. Regardless of the cost and the time (over two years until release), a decision was made to re-implement Fortify_Source for Clang as Clang further enables the use of Overflow Sanitizers (integer, bound, address) and of Control-Flow Integrity.

The motivation to undertake such consequent measure can be built on the knowledge provided by the publication of monthly security Bulletins.

- This monitoring provides the developing team and the public with up-to-date figures. It helps trigger alarms if the system, or one of its modules, appears particularly exposed to a particular vulnerability type. This exposure can be understood as the cost of not increasing the safety of the system.
- Engaging in transparency through the publication of fixed vulnerabilities is also beneficial for the development and the adoption of Android as an OS. First, engaging in such a procedure brings trust to the product. Developers recognise the product used to be vulnerable, but exposure is monitored, and action to fix the issue is undertaken. Second, such practice engages the developers to continue this procedure, resulting in a constant improvement of the product.
- Further, the acknowledgement from whom the vulnerability disclosure originates underlines a discussion with analysts and public recognition of their work. Thus, the detection of vulnerability is valued and is encouraged. It answers the above-mentioned requirement to keep the focus on the detection of vulnerabilities on Android.

[Modularisation and Updates] Second, we have acknowledged in the Section 2.1.3 the period of time between the disclosure of a vulnerability and the eventual update of users' devices to have the most impact on both end-users and product developers. For instance, in the introduction of this thesis (Section 1.1), we mentioned Eternal Blue. The attack occured on June 27th 2017, while Microsoft had emitted a fixing patch by March 14th 2017. Thus the late (or lack of) adoption of updates (3 months after release) that partially explains the extent to which this malware affected a country and the global economy. In Figure 5.1, we can observe how much effort has been developed into modularising AOSP and fluidising the independent update of these modules. It was first undertaken for WebView. As an open gate to web content, and thus to web threats, the modularisation and update through the Play Store made the system defence more responsive to newest threats. This capacity to react has been followed up with project Trebble and with the Generic Kernel Image project for the kernel. They differentiate

more clearly AOSP components from third parties elements (e.g. device, vendors). The system is since autonomous regarding the update and can, for instance, month after month provide the latest security update available for download to all devices. AOSP thus answered the main lead for improvement from *Mazuera et al.* [78].

Perspectives: In this paragraph, we mention what could be further done to either strengthen the system or assess the efficiency of defence mechanisms. Both would help software developers to prioritize the implementation of defence mechanisms.

[Integration Testing] As mentioned by *Mazuera et al.* [78], further implementing integration (authors say "*just-in-time*") quality control tools and techniques can be implemented (as a commit-level detection algorithm). Another profound change, more consequent than the development of Fortify_Source for Clang, would consist in changing coding languages for more resilient ones. For instance, strengthening toward memory corruptions could benefit from a change from C to Rust in lower layers of the AOSP Stack.

[Behaviour Monitoring] The focus could be ported to the applications by making a remote copy of users' device and, on this copy, scanning for Potentially Harmful Applications (PHAs) and producing a dynamic taint analysis [242]. The paper focuses on practical aspects as minimising the useful trace and the size of the data to be sent. The suggestion yet has significant drawbacks as the battery life was reduced by 30%.

[Restructuration] Applications, still, could be the subject of an architectural re-foundation [243]. They could implement their own SELinux rules, as the original paper describes and provides templates for. It however requires to grant the developer of the application with a higher level of trust, which brings other trust-related issues.

[Precise Measurement of the Impact of Def.Mech.] In relationship with the threats evoked above: we have not properly assessed the efficiency of later-introduced defence mechanisms toward the vulnerabilities that required a fix. One way to acknowledge the actual efficiency of defence mechanisms would consist in providing, per defence mechanism, the number of CVEs they would have prevented, or made the exploitation more complex, had they been implemented earlier. To do so, we need a pool of exploits for each CVEs. We would then roll back specifically the patch and observe if the exploitation is altered on the newly compiled AOSP version. The resulting score could be seen both as a criticism of not anticipating enough specific weaknesses, or demonstrating the capacity to have taken the right decision to implement a specific defence mechanism. Proper meaning to these figures would nonetheless also require a considerable number of CVEs to test for, and also to compare with other systems' results to qualify the reactivity.

6 Conclusion and Perspectives

In this final chapter, we conclude regarding the challenges that software vulnerabilities brings to the research community and to becoming resilient. We collect the main elements to remember and put them in further perspective. We underline how they nurture further detection of vulnerabilities and nourish detection of exploitation of these known vulnerabilities.

Contents

6.1 Conclusion

In this thesis, we, first, contribute to the understanding of what is a vulnerability and to analyse them. Our work allows for the creation of usable and **updatable consistent vulnerability data sets**. Thus, we produced a manual categorisation of the Android Open Source Project vulnerabilities that clearly states to follow a *vulnerability analysis*. We focus on the predisposing condition creating a vulnerability in the code before the patch happens. From this set of vulnerabilities and their classification, we analyse the evolution of vulnerability disclosures over more than ten years of the Android Open Source Project. The first fact is that there are still vulnerabilities needing a patch, as of June 2020, and that their severity does not diminish. Also, efforts did not manage either to make any root CWE affecting AOSP to properly vanish; not even memory corruption. Nonetheless, and while our analyses are circumvented by the data time span, there is a trend to the reduction of the number of CVEs that needs to be patched in AOSP, at least since 2017. This encouraging sign affects all CWEs and particularly memory corruption and the native libraries layer. Their lifetime also has significantly shortens.

We explain to have chosen a *vulnerability oriented* analysis with the ambition to tackle the challenge of **vulnerability detection**. As such, a machine-learning-based detection system, such as VCCFinder [30], that we attempted to replicate, could be trained with information that we could statically consider as among the most significant. Confronted with the lack of control over the data set graciously provided by the original authors, we could not replicate nor reproduce the approach. We nonetheless distribute our code as a reproducible baseline and attempted to out-perform original results with exploration over parameters, algorithms and to attempt to face the issue of imbalanced data set, regarding the set of unlabeled commits, using co-training. With our code available it is now possible to evaluate and compare with more recent tools that built on the success of VCCFinder to determine most successful and most promising approaches in vulnerability detection.

We eventually analyse how the AOSP development team hardened AOSP over time and further anticipated both the discovery of existing vulnerabilities and the introduction of newest ones. Instead of only patching disclosed vulnerabilities, the implementation of defence mechanisms **prevents the exploitation** of specific weaknesses in the components they are applied to. We further verify quantitatively that this information does match with the binaries of AOSP from API-level 10 to 28. Developing these protections may take time, can spread over different versions, and may necessitate rebuilding the project from early stages, but, as far as we can observe, these defence mechanisms are eventually beneficial. Specifically for Memory Corruption related vulnerabilities, the introduction of specific defence mechanisms is correlated with a reduction of the number of vulnerabilities needing to be patched. An observation that could not be noted in other studies, regarding other systems, regardless of the efforts to observe such tendencies [59, 53, 76], hence contributing to AOSP's widespread adoption. Nonetheless, Android stands as an Operating System. As such, not all the practices adopted and not all the defence mechanisms are applicable to any software. However, a consistent approach would scale down from AOSP's list in case of motivated uselessness better than building up by cherry-picking.

6.2 Perspectives

In this final part, we will describe several challenges that this thesis paves the way to. Some of these objectives where originally ambitioned to be part of the thesis. Other where only revealed through the learning process.

6.2.1 Detecting vulnerability clones in the Android OS

6.2.1.1 Detection of new vulnerabilities

On the one hand, we have a machine learning-based vulnerability detector at the commit level that we could not confirm as a proper replication of the original work, partially because of the lack of command over the database provided by the original authors. On the other hand, we have a database of around a thousand commits fixing a vulnerability. The combination of these two elements could serve several purposes:

1. Finish to assess the precision and recall of our revisited VCCFinder implementation with real cases of native (C/C++) vulnerabilities in AOSP.

2. After the determination of the conditions for a good enough ratio of True Positives over False Positives, we could analyse the rest of the native layers of AOSP. The resulting commits, sorted with most confidence by the algorithm, as VCCs could be manually verified to assess whether they are True or False Positives. In other words, if the vulnerable version pointed at is exploitable.

   As vulnerabilities are classified regarding their CWEs, and we have the connection to parents CWEs, the revisited VCCFinder algorithm could be trained separately for different CWEs.

Challenges:

- If we have the commit that fixes the vulnerability, there is one step further to produce to pass from a vulnerable version to the actual(s) VCC(s). This implies that an SZZ-like algorithm would have to be implemented and that, for each vulnerability, provided VCCs would have to be manually verified.

6.2.1.2 Detection of Vulnerability Clones

This list of vulnerable commits for Android could be helpful through another type of vulnerability detection, namely code clone detection.

1. A first exploration of the code would consist in referencing Abstract Syntax Trees of these vulnerabilities and implementing a code comparison analysis with the code of AOSP using the same language as the vulnerability, as Deckard does [92].

   This technique could be further expanded to include Program Dependence Graphs and Control Flow Graphs to produce analogue methods like *Yamaguchi et al.* [91]. Further promising improvement uses the Code Property Graphs from raw code with Joern [244]. VGRaph [245] is another perspective for which, to be considered vulnerable, the target code has to be close enough to the code shared between the unpatched and patched version, but looking, however, closer (i.e. above a specific threshold), to the vulnerable version, and not too close to the patched version of the code.

2. Another tool, earlier mentioned (Section 2.2.1), sounds promising regarding the expectation to detect vulnerability clones at the function level, namely MVP [87].

MVP is based on both the syntax and the semantic of vulnerable code. It first needs to cut programs into functions and then hashes the syntax of normalised lines of the code. It also keeps score of the relationships between lines of the same code(semantic). Eventually, MVP confronts resulting abstractions with a database of vulnerability signatures. As for VGraph, specific rules are established so the code is close enough from the vulnerable version of the code, and not so close from the patched version, both syntactically and semantically.

Challenges:

- The code of MVP [87] is not publicly released. Even though the article provides a respectable amount of details about the implementation, others might be lacking (like the hashing function used). Further, any replication, or attempt, could only focus on the C++ code as the tool depends heavily on the Joern [244]. Evaluation of the eventual implementation would suffer the same challenges as our VCCFinder-revisited for both reproduction and replication.

6.2.2 Reference a library of exploit specific to these vulnerabilities

Another aspect that this thesis provides the bases for is either or both referencing and developing exploits for these vulnerabilities.

The development of such exploits can help train security experts to measure the severity of a vulnerability accurately. Several exploit libraries already exist and are available online [246, 247]. They can serve as a base to start tackling faster other issues such as characterising threats and suspicious programs

6.2.2.1 Exploit Generation

Provided with the patch, we can deduce the sub-goals or input to feed the program with so to exploit the vulnerable version of the code.

The development of such exploit can also be generated faster with fuzzing. The location of the issue in the code and the analysis of the patch can enable to constraint the inputs to reach the vulnerable lines. Then we could adapt existing tools implementing afl into android [105] to explore the desired code locations until an exploit is generated.

6.2.2.2 Assess the maximum potentiality of vulnerabilities

As we have stated, the appropriate importance to attribute to the resolution of a vulnerability in the code can be given if the report contains accurate information about the vulnerability. Therefore, stating the actual severity of a vulnerability is crucial to the resolution of the issue. From one exploit, Koobe [15] is capable of fuzzing the Linux kernel to find other exploits that match the maximum capability of exploitation of Out-of-Bound Overwrite on the Heap of the Linux kernel. With the code available, the principle might be applicable to AOSP native code and extended to other memory corruption weaknesses.

Challenges:

- Koobe relies first on the availability of a first exploit. As such, either we already achieved the steps in Section 6.2.2.1, or enough exploits for the specific CWE we tackle have been gathered.
- If the tool is available[1], the eventual exploitability relies heavily on what the original authors call *feng-shui*. It consists in using various techniques to set objects on the heap in ideal disposition for the overflow to occur in an exploitable manner.

[1] `https://github.com/seclab-ucr/KOOBE`

Their *feng-shui* is not the focus of the article and thus is not detailed. This absence could imply complications in reproducing the chain before and when further attempting to apply it to other types of vulnerabilities in other algorithms. However MAZE [248] could be used as a resource so to use various techniques on the heap.

6.2.3 Abstract exploits and detection of Android applications exploiting known vulnerabilities

The aforementioned steps lead to an eventual application in the wild of the database of vulnerabilities, their exploits and the knowledge gathered relative to their characteristics. With millions of android applications, certainly several of these applications have attempted, or still do attempt, to exploit these referenced vulnerabilities. Detecting these applications having a, properly speaking, malicious behaviour could first enable blacklisting the developers behind these applications. Such detection could also provide a more direct answer as one device used in the world in two runs on a version of AOSP older than 2 years. In some cases, this implies that these vulnerabilities might not receive the last security patch anymore and thus be at risk of the latest vulnerabilities. Increasing user's exposure.

A parade would reside in detecting the exploitation of these vulnerabilities in the code of applications. Either statically, or dynamically this requires two steps: 1. Abstraction of the exploitation of the vulnerability 2. Detection of the exploitation

- Statically, the code of the exploits gathered or created above could be the source of the abstraction. We could use several of the methods overviewed in Section 6.2.1.2. Such as applying an MVP-like [87] approach from vulnerability clones to exploit clones, with the limitation that the exploit fully holds in one function, and there might not be an equivalent for the *patched* code, that serves as a rejection. Limitations of these techniques are, in the first case, the high rate of false positives if the abstraction becomes too general. Another limitation is the granularity. An MVP-like approach can only detect the exploit if it entirely fits in one function.
- Dynamic detection could use Enhanced Attack Trees mentioned in Section 2.2.3. Better than abstracting the code of the exploit, a *medium* could be to abstract the actions the exploit has to take and sub-goals the exploit has to capture so that the end-of-line goal can be achieved. Once such representations are archived for several vulnerabilities (e.g. Enhanced Abstract Trees [108]) the OS can be recompiled with a watchdog program verifying, at runtime, actions performed and permissions granted to applications. The trade-off is as follows: the defence would not depend on the semantic nor on the syntax of the exploit but only on the application's behaviour on the system. The detection is independent of the granularity from when the actions can be traced back to their original commander. For such instrumentation **MADAM** [214] stands as an example of on-device monitoring (see 2.4.4).

  Challenges:

  However, the abstraction of so many vulnerabilities requires the implementation of an automated abstraction.

List of papers

List of papers included in this dissertation

Accepted Papers

T. Riom, A. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, and J. Klein, “Revisiting the vccfinder approach for the identification of vulnerability- contributing commits”, *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–30, 2021.

Forthcoming Submission

T. Riom, K. Allix, A. Bartel, T. F. Bissyand and J. Klein, "10-years of Software Vulnerabilities Evolution in the Android Open-Source Project", *We are considering submitting this paper to Empirical Software Engineering, JSS, or MSR*

T. Riom, K. Allix, A. Bartel, T. F. Bissyand and J. Klein, "Defense Mechanisms in AOSP: Do they impact the number of Vulnerabilities", *We are considering submitting this paper to Empirical Software Engineering, JSS, or MSR*

List of Papers not included in this dissertation

L. Li, T. Riom, T. F. Bissyandé, H. Wang, J. Klein, Y. Le Traon, "Revisiting the impact of common libraries for android-related investigations", *Journal of Systems and Software*, vol.154, August 2019, pp. 157-175.

Bibliography

[1] A. D. Sawadogo, T. F. Bissyandé, N. Moha, K. Allix, J. Klein, L. Li, and Y. Le Traon, "Learning to catch security patches," 2020.

[2] HackerNoon, "7 reasons why telegram is insecure by design but millions still flock to it ignoring privacy concerns." `https://hackernoon.com/7-reason-why-telegram-is-insecure-by-design-but-millions-still-flock-to-it-ig` Last Accessed on:.

[3] Wired, "Whatsapp fixes its biggest encryption loophole." Last Accessed on:.

[4] FRSecure, "Zoom risk vulnerabilities and security best practices." Last Accessed on:.

[5] E. F. Foundation, "Telegram harm reduction for users in russia and ukraine." Last Accessed: May 4th, 2022.

[6] IPCC, "Climate change 2021 – the physical science basis – summary for policymakers." Last Accessed on: May 4th, 2022.

[7] IPCC, "Climate change 2021 – the physical science basis – summary for policymakers." Last Accessed on: May 4th, 2022.

[8] NASA, "Mars 2020 mission perseverance rover." `https://mars.nasa.gov/mars2020/`. Last Accessed: May 4th, 2022.

[9] D. Magazine, "Apollo 11's "1202 alarm" explained." https://www.discovermagazine.com/the-sciences/apollo-11s-1202-alarm-explained. Last Accessed on: May 4th, 2022.

[10] P. A. NASA, "Apollo 11 program alarms." `https://www.hq.nasa.gov/office/pao/History/alsj/a11/a11.1201-pa.html`. Last Accessed: May 4th 2022.

[11] P. Kidwell, "Stalking the elusive computer bug," *IEEE Annals of the History of Computing*, vol. 20, no. 4, pp. 5–9, 1998.

[12] R. Shirey, "Internet security glossary." `https://www.packetizer.com/rfc/rfc2828/`. Last Accessed on: May 4th 2022.

[13] N. Cam-Winget, R. Housley, D. Wagner, and J. Walker, "Security flaws in 802.11 data link protocols," *Commun. ACM*, vol. 46, p. 35–39, may 2003.

[14] A. Stubblefield, J. Ioannidis, and A. D. Rubin, "Using the fluhrer, mantin, and shamir attack to break wep," in *NDSS*, 2002.

[15] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1093–1110, USENIX Association, Aug. 2020.

[16] F. Culture, "Pegasus : 12 chefs d'État et de gouvernement ciblés." `https://www.franceculture.fr/geopolitique/pegasus-12-chefs-detat-et-de-gouvernement-cibles`. Last Accessed on: May 4th 2022.

[17] T. Guardian, "Phone of top catalan politician targeted." `https://www.theguardian.com/world/2020/jul/13/phone-of-top-catalan-politician-targeted-by-government-grade-spyware`. Last Accessed on: May 4th 2022.

[18] T. Verge, "Spyware scandal rocks polish government." https://www.theverge.com/2021/12/27/22855390/poland-pegasus-spyware-opposition-brejza-nso. Last Accessed on: May 4th 2022.

[19] Haaretz, "Revealed: Israel's cyber-spy industry helps world dictators hunt dissidents and gays." `https://www.haaretz.com/israel-news/.premium.MAGAZINE-israel-s-cyber-spy-industry-aids-dictators-hunt-dissidents-and-gays-1.6573027`. Last Accessed on May 4th 2022.

[20] sleepya, "Exploit for cve-2017-0144, i.e. eternalblue." https://www.exploit-db.com/exploits/42030. Last Accessed on May 4th, 2022.

[21] Forbes, "Notpetya ransomware attack cost shipping giant maersk over $200 million." https://www.forbes.com/sites/leemathews/2017/08/16/notpetya-ransomware-attack-cost-shipping-giant-maersk-over-200-million/. Last Accessed on: May 4th 2022.

[22] GData, "Merck wins not petya claim – but the future of cybersecurity insurance is complicated." https://www.gdatasoftware.com/blog/2022/01/37232-merck-wins-not-petya-claim-but-the-future-of-cybersecurity-insurance-is-complicated. Last Accessed on: May 4th 2022.

[23] R. Anderson and T. Moore, "The economics of information security," *Science*, vol. 314, no. 5799, pp. 610–613, 2006.

[24] N. Daoudi, K. Allix, T. F. Bissyandé, and J. Klein, "A deep dive inside drebin: An explorative analysis beyond android malware detection scores," vol. 25, may 2022.

[25] J. Samhi, A. Bartel, T. F. Bissyandé, and J. Klein, "Raicc: Revealing atypical inter-component communication in android apps," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1398–1409, 2021.

[26] X. Zhu, X. Feng, T. Jiao, S. Wen, Y. Xiang, S. Camtepe, and J. Xue, "A feature-oriented corpus for understanding, evaluating and improving fuzz testing," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pp. 658–663, 2019.

[27] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (USA), p. 209–224, USENIX Association, 2008.

[28] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, (USA), p. 18, USENIX Association, 2005.

[29] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, (USA), USENIX Association, 2001.

[30] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), p. 426–437, Association for Computing Machinery, 2015.

[31] NVD, "Common vulnerability scoring system calculator." `https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator`.

[32] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pp. 1–2, IEEE, 2021.

[33] M. Jimenez, R. Rwemalika, M. Papadakis, F. Sarro, Y. Le Traon, and M. Harman, "The importance of accounting for real-world labelling when predicting software vulnerabilities," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), p. 695–705, Association for Computing Machinery, 2019.

[34] R. Croft, Y. Xie, and M. A. Babar, "Data preparation for software vulnerability prediction: A systematic literature review," *arXiv preprint arXiv:2109.05740*, 2021.

[35] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang, "Understanding the reproducibility of crowd-reported security vulnerabilities," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pp. 919–936, 2018.

[36] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, and D. Phung, "Deep domain adaptation for vulnerable code function identification," in *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8, 2019.

[37] M. Gegick, L. Williams, J. Osborne, and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM Workshop on Quality of Protection*, QoP '08, (New York, NY, USA), p. 31–38, Association for Computing Machinery, 2008.

[38] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[39] K. Hogan, N. Warford, R. Morrison, D. Miller, S. Malone, and J. Purtilo, "The challenges of labeling vulnerability-contributing commits," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 270–275, IEEE, 2019.

[40] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 403–414, 2015.

[41] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 218–227, 2008.

[42] R. Paul, A. K. Turzo, and A. Bosu, "Why security defects go unnoticed during code reviews? a case-control study of the chromium os project," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1373–1385, 2021.

[43] K. Allix, Q. Jerome, T. F. Bissyandé, J. Klein, R. State, and Y. L. Traon, "A forensic analysis of android malware – how is malware written and how it could be detected?," in *2014 IEEE 38th Annual Computer Software and Applications Conference*, pp. 384–393, 2014.

[44] W. Hu, D. Octeau, P. D. McDaniel, and P. Liu, "Duet: Library integrity verification for android applications," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless & Mobile Networks*, WiSec '14, (New York, NY, USA), p. 141–152, Association for Computing Machinery, 2014.

[45] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pp. 221–238, USENIX Association, Aug. 2020.

[46] Google, "Syzkaller." `https://github.com/google/syzkaller`. last accesssed: May 4th 2022.

[47] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *SIGPLAN Not.*, vol. 46, p. 265–278, mar 2011.

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

[49] Y. Chen and X. Xing, "Slake: Facilitating slab manipulation for exploiting vulnerabilities in the linux kernel," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, (New York, NY, USA), p. 1707–1722, Association for Computing Machinery, 2019.

[50] C. of Europe, "Convention on cybercrime (ets no. 185)." https://www.coe.int/en/web/conventions/full-list?module=treaty-detail&treatynum=185. Last Accessed on May 4th 2022.

[51] E. F. Foundation, "Vulnerabilities equities process (vep)." https://www.eff.org/document/vulnerabilities-equities-process-january-2016. Last Accessed on: May 4th 2022.

[52] CISA, "Alert (aa22-117a) – 2021 top routinely exploited vulnerabilities." https://www.cisa.gov/uscert/ncas/alerts/aa22-117a. Last Accessed on: May 4th 2022.

[53] E. Rescorla, "Is finding security holes a good idea?," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 14–19, 2005.

[54] O. H. Alhazmi and Y. K. Malaiya, "Modeling the vulnerability discovery process," in *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pp. 10–pp, IEEE, 2005.

[55] O. Alhazmi and Y. Malaiya, "Quantitative vulnerability assessment of systems software," in *Annual Reliability and Maintainability Symposium, 2005. Proceedings.*, pp. 615–620, 2005.

[56] R. C. B. Robert M. Brady, Ross J. Anderson, "Murphy's law, the fitness of evolving species, and the limits of software reliability," tech. rep., University of Cambridge, 1999.

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

[58] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *2015 IEEE symposium on security and privacy*, pp. 692–708, IEEE, 2015.

[59] A. Ozment and S. E. Schechter, "Milk or wine: does software security improve with age?," in *USENIX Security Symposium*, vol. 6, pp. 10–5555, 2006.

[60] L. Bilge and T. Dumitraş, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 833–844, 2012.

[61] A. Arora, A. Nandkumar, and R. Telang, "Does information security attack frequency increase with vulnerability disclosure? an empirical analysis," *Information Systems Frontiers*, vol. 8, no. 5, pp. 350–362, 2006.

[62] A. Arora, R. Telang, and H. Xu, "Optimal policy for software vulnerability disclosure," *Management Science*, vol. 54, no. 4, pp. 642–656, 2008.

[63] H. Cavusoglu, H. Cavusoglu, and J. Zhang, "Security patch management: Share the burden or share the damage?," *Management Science*, vol. 54, no. 4, pp. 657–670, 2008.

[64] M. Monperrus, "The Living Review on Automated Program Repair," Technical Report hal-01956501, HAL Archives Ouvertes, 2018.

[65] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie, "Autopag: Towards automated software patch generation with source code root cause identification and repair," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, ASIACCS '07, (New York, NY, USA), p. 329–340, Association for Computing Machinery, 2007.

[66] Z. Huang, D. Lie, G. Tan, and T. Jaeger, "Using safety properties to generate vulnerability patches," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 539–554, 2019.

[67] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 502–511, 2013.

[68] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *Computer Security – ESORICS 2017* (S. N. Foley, D. Gollmann, and E. Snekkenes, eds.), (Cham), pp. 229–246, Springer International Publishing, 2017.

[69] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim, and p. chin, "Learning to repair software vulnerabilities with generative adversarial networks," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, Curran Associates, Inc., 2018.

[70] Z. Chen, S. J. Kommrusch, and M. Monperrus, "Neural transfer learning for repairing security vulnerabilities in c code," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[71] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, *A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries*, p. 508–512. New York, NY, USA: Association for Computing Machinery, 2020.

[72] G. Bhandari, A. Naseer, and L. Moonen, "Cvefixes: Automated collection of vulnerabilities and their fixes from open-source software," in *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, (New York, NY, USA), p. 30–39, Association for Computing Machinery, 2021.

[73] N. Dissanayake, A. Jayatilaka, M. Zahedi, and M. A. Babar, "Software security patch management - a systematic literature review of challenges, approaches, tools and practices," *Information and Software Technology*, vol. 144, p. 106771, 2022.

[74] T. Scholte, D. Balzarotti, and E. Kirda, "Quo vadis? a study of the evolution of input validation vulnerabilities in web applications," in *Financial Cryptography and Data Security* (G. Danezis, ed.), (Berlin, Heidelberg), pp. 284–298, Springer Berlin Heidelberg, 2012.

[75] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2201–2215, 2017.

[76] N. Alexopoulos, S. M. Habib, S. Schulz, and M. Mühlhäuser, "The tip of the iceberg: On the merits of finding security bugs," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 1, pp. 1–33, 2020.

[77] M. Jimenez, M. Papadakis, T. F. Bissyandé, and J. Klein, "Profiling android vulnerabilities," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 222–229, 2016.

[78] A. Mazuera-Rozo, J. Bautista-Mora, M. Linares-Vásquez, S. Rueda, and G. Bavota, "The android os stack and its vulnerabilities: an empirical study," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2056–2101, 2019.

[79] D. Wu, D. Gao, E. K. T. Cheng, Y. Cao, J. Jiang, and R. H. Deng, "Towards understanding android system vulnerabilities: Techniques and insights," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, (New York, NY, USA), p. 295–306, Association for Computing Machinery, 2019.

[80] S. Farhang, M. B. Kirdan, A. Laszka, and J. Grossklags, "An empirical study of android security bulletins in different vendors," in *Proceedings of The Web Conference 2020*, pp. 3063–3069, 2020.

[81] A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 161–168, IEEE, 2017.

[82] D. A. Wheeler, "Flawfinder," 2001. accessed April 2020.

[83] J. Signoles, P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, and B. Yakobowski, "Frama-c: a software analysis perspective," vol. 27, 10 2012.

[84] G. J. Holzmann, "Uno: Static source code checking for userdefined properties," in *In 6th World Conf. on Integrated Design and Process Technology, IDPT '02*, 2002.

[85] L. Torvalds, J. Triplett, C. Li, and L. V. Oostenryck, "Sparse - a semantic parser for c," 2003. accessed january 2020.

[86] M. Martin, B. Livshits, and M. S. Lam, "Finding application errors and security flaws using pql: A program query language," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), p. 365–383, Association for Computing Machinery, 2005.

[87] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 1165–1182, USENIX Association, Aug. 2020.

[88] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, pp. 595–614, 2017.

[89] J. Jang, A. Agrawal, and D. Brumley, "Redebug: Finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*, pp. 48–62, 2012.

[90] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), p. 259–269, Association for Computing Machinery, 2014.

[91] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, May 2014.

[92] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, 2007.

[93] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, (New York, NY, USA), p. 708–719, Association for Computing Machinery, 2016.

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

[95] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, p. 82–90, Feb. 2013.

[96] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), p. 213–223, Association for Computing Machinery, 2005.

[97] C. Y. Cho, D. Babiundefined, P. Poosankam, K. Z. Chen, E. X. Wu, and D. Song, "Mace: Model-inference-assisted concolic exploration for protocol and vulnerability discovery," in *Proceedings of the 20th USENIX Conference on Security*, SEC'11, (USA), p. 10, USENIX Association, 2011.

[98] H. Li, J. Oh, H. Oh, and H. Lee, "Automated source code instrumentation for verifying potential vulnerabilities," in *ICT Systems Security and Privacy Protectio* (J.-H. Hoepman and S. Katzenbeisser, eds.), (Cham), pp. 211–226, Springer International Publishing, 2016.

[99] N. Kosmatov and J. Signoles, "A lesson on runtime assertion checking with Frama-C," in *Runtime Verification. RV 2013. Lecture Notes in Computer Science*, vol. 8174 LNCS, (Rennes, France), pp. 386–399, Sept. 2013. Conference of 4th International Conference on Runtime Verification, RV 2013 ; Conference Date: 24 September 2013 Through 27 September 2013; Conference Code:100802.

[100] M. Zalewski, "American fuzzy lop." `http://lcamtuf.coredump.cx/afl/`, 2017.

[101] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2123–2138, 2018.

[102] S. Groß, "Fuzzil: Coverage guided fuzzing for javascript engines," Master's thesis, Karlsruhe Institute of Technology, 2018.

[103] R. Padhye, C. Lemieux, and K. Sen, "Jqf: coverage-guided property-based testing in java," pp. 398–401, 07 2019.

[104] R. Kersten, K. S. Luckow, and C. S. Pasareanu, "Poster: Afl-based fuzzing for java with kelinci," in *ACM Conference on Computer and Communications Security*, 2017.

[105] ele7enxxh, "Android-afl." `https://github.com/ele7enxxh/android-afl`.

[106] Google, "Oss-fuzz." `https://google.github.io/oss-fuzz/`.

[107] B. Liu, C. Zhang, G. Gong, Y. Zeng, H. Ruan, and J. Zhuge, "FANS: Fuzzing android native system services via automated interface analysis," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 307–323, USENIX Association, Aug. 2020.

[108] B. Schneier, "Attack trees." `https://www.schneier.com/academic/archives/1999/12/attack_trees.html`.

[109] S. A. Camtepe and B. Yener, "Modeling and detection of complex attacks," in *2007 Third International Conference on Security and Privacy in Communications Networks and the Workshops - SecureComm 2007*, pp. 234–243, 2007.

[110] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–36, 2017.

[111] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, (New York, NY, USA), p. 1–5, Association for Computing Machinery, 2005.

[112] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, (New York, NY, USA), p. 529–540, Association for Computing Machinery, 2007.

[113] R. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Transactions on Software Engineering*, vol. 34, pp. 579–596, Sep. 2008.

[114] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308–320, Dec 1976.

[115] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 421–428, April 2010.

[116] Y. Shin and L. Williams, "An initial study on the use of execution complexity metrics as indicators of software vulnerabilities," in *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pp. 1–7, 2011.

[117] S. Moshtari, A. Sami, and M. Azimi, "Using complexity metrics to improve software security," *Computer Fraud & Security*, vol. 2013, no. 5, pp. 8–17, 2013.

[118] D. Wijayasekara, M. Manic, J. L. Wright, and M. McQueen, "Mining bug databases for unidentified software vulnerabilities," in *2012 5th International Conference on Human System Interactions*, pp. 89–96, June 2012.

[119] D. Wijayasekara, M. Manic, and M. McQueen, "Vulnerability identification and classification via text mining bug databases," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, pp. 3612–3618, Oct 2014.

[120] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, pp. 993–1006, Oct 2014.

[121] I. Kononenko, "On biases in estimating multi-valued attributes," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, (San Francisco, CA, USA), p. 1034–1040, Morgan Kaufmann Publishers Inc., 1995.

[122] I. Medeiros, N. Neves, and M. Correia, "Dekant: A static analysis tool that learns to detect web application vulnerabilities," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), p. 1–11, Association for Computing Machinery, 2016.

[123] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, (New York, NY, USA), p. 480–491, Association for Computing Machinery, 2016.

[124] H. Jégou, M. Douze, C. Schmid, and P. Pérez, "Aggregating local descriptors into a compact image representation," in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 3304–3311, 2010.

[125] M. Slaney and M. Casey, "Locality-sensitive hashing for finding nearest neighbors [lecture notes]," *IEEE Signal Processing Magazine*, vol. 25, no. 2, pp. 128–131, 2008.

[126] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 3289–3297, July 2018.

[127] X. Ban, S. Liu, C. Chen, and C. Chua, "A performance evaluation of deep-learnt features for software vulnerability detection," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 19, p. e5103, 2019. e5103 cpe.5103.

[128] K. Goseva-Popstojanova and J. Tyo, "Identification of security related bug reports via text mining using supervised and unsupervised classification," in *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pp. 344–355, July 2018.

[129] S. Kim, E. J. Whitehead, Jr., and Y. Zhang, "Classifying software changes: Clean or buggy?," *IEEE Transactions on Software Engineering*, vol. 34, pp. 181–196, March 2008.

[130] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 65–74, Oct 2013.

[131] K. Yamamoto, "Vulnerability detection in source code based on git history," 2018. (Unpublished).

[132] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, (New York, NY, USA), p. 201–213, Association for Computing Machinery, 2016.

[133] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, (San Francisco, CA), USENIX Association, Dec. 2004.

[134] Y. Zhou and A. Sharma, "Automated identification of security issues from commit messages and bug reports," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, (New York, NY, USA), p. 914–919, Association for Computing Machinery, 2017.

[135] M. Alohaly and H. Takabi, "When do changes induce software vulnerabilities?," in *2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC)*, pp. 59–66, Oct 2017.

[136] K. Nigam, A. K. McCallum, S. Thrun, and T. Mitchell, "Text classification from labeled and unlabeled documents using em," *Mach. Learn.*, vol. 39, p. 103–134, may 2000.

[137] L. Yang, X. Li, and Y. Yu, "Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pp. 1–7, Dec 2017.

[138] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 579–582, 2018.

[139] L. Wan, *Automated vulnerability detection system based on commit messages.* PhD thesis, 2019.

[140] D. H. Aristizabal, D. M. Rodriguez, and R. Y. Guevara, "Measuring aslr implementations on modern operating systems," in *2013 47th International Carnahan Conference on Security Technology (ICCST)*, pp. 1–6, IEEE, 2013.

[141] S. Forrest, A. Somayaji, and D. H. Ackley, "Building diverse computer systems," in *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pp. 67–72, IEEE, 1997.

[142] B. Spengler, "Pax: The guaranteed end of arbitrary code execution," *G-Con2: Mexico City, Mexico*, 2003.

[143] stein, "Guide for conducting risk assessments." `https://isopenbsdsecu.re/mitigations/aslr/`, 2019. Accessed: 12-21-2021.

[144] H. Meer *et al.*, "Memory corruption attacks: The (almost) complete history," *Blackhat USA.(Jul. 2010)*, 2010.

[145] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–13, IEEE, 2016.

[146] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*, pp. 227–242, 2014.

[147] T. Müller, "Aslr smack & laugh reference," in *Seminar on Advanced Exploitation Techniques*, 2008.

[148] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "A theory of secure control flow," in *International Conference on Formal Engineering Methods*, pp. 111–124, Springer, 2005.

[149] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 941–955, USENIX Association, Aug. 2014.

[150] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, "Pt-cfi: Transparent backward-edge control flow violation detection using intel processor trace," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, (New York, NY, USA), p. 173–184, Association for Computing Machinery, 2017.

[151] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[152] T.-c. Chiueh and F.-H. Hsu, "Rad: A compile-time solution to buffer overflow attacks," in *Proceedings 21st International Conference on Distributed Computing Systems*, pp. 409–417, IEEE, 2001.

[153] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[154] S. Sharma, "Enhance application security with fortify_source." Red Hat Blog, `https://www.redhat.com/en/blog/enhance-application-security-fortifysource`, 2014. Accessed: 03-02-2022.

[155] C. Herley and P. Van Oorschot, "A research agenda acknowledging the persistence of passwords," *IEEE Security Privacy*, vol. 10, no. 1, pp. 28–36, 2012.

[156] M. L. Mazurek, S. Komanduri, T. Vidas, L. Bauer, N. Christin, L. F. Cranor, P. G. Kelley, R. Shay, and B. Ur, "Measuring password guessability for an entire university," CCS '13, (New York, NY, USA), p. 173–186, Association for Computing Machinery, 2013.

[157] B. Ur, S. M. Segreti, L. Bauer, N. Christin, L. F. Cranor, S. Komanduri, D. Kurilova, M. L. Mazurek, W. Melicher, and R. Shay, "Measuring Real-World accuracies and biases in modeling password guessability," in *24th USENIX Security Symposium (USENIX Security 15)*, (Washington, D.C.), pp. 463–481, USENIX Association, Aug. 2015.

[158] J. Yu, L. Lu, Y. Chen, Y. Zhu, and L. Kong, "An indirect eavesdropping attack of keystrokes on touch screen through acoustic sensing," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 337–351, 2021.

[159] G. Ye, Z. Tang, D. Fang, X. Chen, K. I. Kim, B. Taylor, and Z. Wang, "Cracking android pattern lock in five attempts," in *Proceedings of the 2017 Network and Distributed System Security Symposium 2017 (NDSS 17)*, Internet Society, 2017.

[160] C. Yuan, X. Sun, and R. Lv, "Fingerprint liveness detection based on multi-scale lpq and pca," *China Communications*, vol. 13, no. 7, pp. 60–65, 2016.

[161] C. Yuan, Z. Xia, L. Jiang, Y. Cao, Q. M. Jonathan Wu, and X. Sun, "Fingerprint liveness detection using an improved cnn with image scale equalization," *IEEE Access*, vol. 7, pp. 26953–26966, 2019.

[162] C. Herley, "So long, and no thanks for the externalities: the rational rejection of security advice by users," in *Proceedings of the 2009 workshop on New security paradigms workshop*, pp. 133–144, 2009.

[163] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android.," in *Ndss*, vol. 310, pp. 20–38, 2013.

[164] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Transactions on Privacy and Security (TOPS)*, vol. 24, no. 3, pp. 1–35, 2021.

[165] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, (New York, NY, USA), p. 627–638, Association for Computing Machinery, 2011.

[166] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[167] A. O. Freier, P. Karlton, and P. C. Kocher, "The ssl protocol version 3.0." `https://www.paulkocher.com/doc/SSLv3.pdf`, 1996. Last accessed on: February 18th 2022.

[168] Y. Chen, Y. Zhang, Z. Wang, and T. Wei, "Downgrade attack on trustzone," *arXiv preprint arXiv:1707.05082*, 2017.

[169] D. R. Thomas, A. R. Beresford, and A. Rice, "Security metrics for the android ecosystem," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pp. 87–98, 2015.

[170] B. University, "Bu-102: Early innovators." `https://batteryuniversity.com/article/bu-102-early-innovators`.

[171] M. S. Ziegler and J. E. Trancik, "Re-examining rates of lithium-ion battery technology improvement and cost decline," *Energy Environ. Sci.*, vol. 14, pp. 1635–1651, 2021.

[172] E. storage news, "Europe and us will shave c.10% off china's li-ion production capacity market share by 2030." `https://www.energy-storage.news/europe-and-us-will-shave-c-10-off-chinas-li-ion-production-capacity-market-share`

[173] W. Westerman, *Hand tracking, finger identification, and chordic manipulation on a multi-touch surface*. Citeseer, 1999.

[174] Zytronic, "Aboutus." `https://www.zytronicplc.com/about-us/`.

[175] G. Walker, "Fundamentals of projected-capacitive touch technology." `https://walkermobile.com/SID_2014_Short_Course_S1.pdf`.

[176] Zytronic, "A brief history of touchscreen technology." `https://www.zytronic.co.uk/insights/article/history-of-touchscreen-technology/`.

[177] O. H. Alliance, "Faq." https://www.openhandsetalliance.com/oha_faq.html.

[178] Android, "Announcing the android 1.0 sdk, release 1." https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html.

[179] R. A. for Arstechnica, "Google's iron grip on android: Controlling open source by any means necessary." https://arstechnica.com/gadgets/2018/07/googles-iron-grip-on-android-controlling-open-source-by-any-means-necessary/.

[180] "Android vs ios market." https://leftronic.com/blog/android-vs-ios-market-share/.

[181] "Google pay." https://pay.google.com/intl/fr_fr/about/.

[182] Payconiq, "La nouvelle façon de payer." https://payconiq.lu/fr/.

[183] Android, "Developer policy center." https://play.google.com/about/developer-content-policy/.

[184] Android, "Application signing." https://source.android.com/security/apksigning.

[185] Android, "User guide - sign your app." https://developer.android.com/studio/publish/app-signing.

[186] Samsung, "One ui - samsung app store." https://www.samsung.com/fr/apps/.

[187] F-Droid, "F-droid - free and open source android app repository." https://f-droid.org/en/packages/. Last Accessed on: May 25th 2022.

[188] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 468–471, ACM, 2016.

[189] G. Gong, "Fuzzing android system services by binder call to escalate privilege," *BlackHat USA*, vol. 2015, 2015.

[190] D. Cotroneo, A. K. Iannillo, and R. Natella, "Evolutionary fuzzing of android os vendor system services," *Empirical Software Engineering*, vol. 24, no. 6, pp. 3630–3658, 2019.

[191] Linux, "Kernel." https://www.kernel.org/.

[192] Android, "Kernel overview." https://source.android.com/devices/architecture/kernel/.

[193] Android, "The generic kernel image (gki) project." https://source.android.com/devices/architecture/kernel/generic-kernel-image.

[194] A. developer, "Android 1.5." https://android-doc.github.io/about/versions/android-1.5-highlights.html.

[195] A. Developer, "Android manifest overview." https://developer.android.com/guide/topics/manifest/manifest-intro.

[196] A. Developer, "Android services." https://developer.android.com/reference/android/app/Service.

[197] A. Developer, "Broadcast." https://developer.android.com/guide/components/broadcasts.

[198] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), p. 73–84, Association for Computing Machinery, 2010.

[199] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering*, 2012.

[200] G. Developer, "Goole play protect." https://developers.google.com/android/play-protect/potentially-harmful-applications.

[201] G. Developer, "Play protect cloud-based protection." https://developers.google.com/android/play-protect/cloud-based-protections.

[202] D. O. Mo Yu and C. R. from Android Security & Privacy Team, "Combating potentially harmful applications with machine learning at google: Datasets and models." https://android-developers.googleblog.com/2018/11/combating-potentially-harmful.html.

[203] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 280–291, 2015.

[204] B. Liu, B. Liu, H. Jin, and R. Govindan, "Efficient privilege de-escalation for ad libraries in mobile apps," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '15, (New York, NY, USA), p. 89–103, Association for Computing Machinery, 2015.

[205] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. Le Traon, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*, 2017.

[206] J. Samhi, T. F. Bissyande, and J. Klein, "Triggerzoo: A dataset of android applications automatically infected with logic bombs," in *19th International Conference on Mining Software Repositories, Data Showcase, (MSR 2022)*, Association for Computing Machinery, may 2022.

[207] L. Li, A. Bartel, J. Klein, and Y. Le Traon, "Automatically exploiting potential component leaks in android applications," in *Proceedings of the 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2014)*, IEEE, 2014.

[208] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Le Traon, *Improving privacy on android smartphones through in-vivo bytecode instrumentation.* PhD thesis, University of Luxembourg, 2012.

[209] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, "Empirical assessment of machine learning-based malware detectors for android," *Empirical Software Engineering*, vol. 21, pp. 183–211, Feb 2016.

[210] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "Drebin: Effective and explainable detection of android malware in your pocket," in *NDSS*, 2014.

[211] M. Hurier, K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "On the lack of consensus in anti-virus decisions: Metrics and insights on building ground truths of android malware," in *Detection of Intrusions and Malware, and Vulnerability Assessment* (J. Caballero, U. Zurutuza, and R. J. Rodríguez, eds.), (Cham), pp. 142–162, Springer International Publishing, 2016.

[212] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "piggybacked" mobile applications," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, (New York, NY, USA), p. 185–196, Association for Computing Machinery, 2013.

[213] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, *Borrowing Your Enemy's Arrows: The Case of Code Reuse in Android via Direct Inter-App Code Invocation*, p. 939–951. New York, NY, USA: Association for Computing Machinery, 2020.

[214] A. Saracino, D. Sgandurra, G. Dini, and F. Martinelli, "Madam: Effective and efficient behavior-based android malware detection and prevention," *IEEE Transactions on Dependable and Secure Computing*, vol. 15, no. 1, pp. 83–97, 2018.

[215] R. Shirey, "Rfc2828: Internet security glossary," 2000.

[216] M. Linares-Vásquez, G. Bavota, and C. Escobar-Velásquez, "An empirical study on android-related vulnerabilities," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pp. 2–13, IEEE, 2017.

[217] NIST's Computer Security Division, Information Technology Laboratory, "Guide for conducting risk assessments." `https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-30r1.pdf`, 2012. Accessed: 2020-09-16.

[218] M. Jimenez, M. Papadakis, and Y. Le Traon, "Enabling the continous analysis of security vulnerabilities with vuldata7," in *Proceedings of the 18th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2018, Madrid, Spain, September 23-24, 2018*, 2018.

[219] I. V. Krsul, *Software vulnerability analysis.* Purdue University West Lafayette, IN, 1998.

[220] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *J. Mach. Learn. Res.*, vol. 9, p. 1871–1874, June 2008.

[221] K. Rieck, C. Wressnegger, and A. Bikadorov, "Sally: A tool for embedding strings in vector spaces," *J. Mach. Learn. Res.*, vol. 13, p. 3247–3251, Nov. 2012.

[222] Association for Computer Machinery, "Artifact review and badging," 2020. Accessed November 27, 2020.

[223] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A practical guide to support vector classification," tech. rep., Department of Computer Science, National Taiwan University, 2003.

[224] V. Castelli and T. M. Cover, "On the exponential value of labeled samples," *Pattern Recognition Letters*, vol. 16, no. 1, pp. 105–111, 1995.

[225] T. Zhang and F. Oles, "The value of unlabeled data for classification problems," in *Proceedings of the Seventeenth International Conference on Machine Learning,(Langley, P., ed.)*, vol. 20, p. 0, Citeseer, 2000.

[226] A. Blum and T. Mitchell, "Combining labeled and unlabeled data with co-training," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, COLT' 98, (New York, NY, USA), p. 92–100, Association for Computing Machinery, 1998.

[227] A. Levin, P. Viola, and Y. Freund, "Unsupervised improvement of visual detectors using co-training," in *null*, p. 626, IEEE, 2003.

[228] M.-F. Balcan and A. Blum, "A pac-style model for learning from labeled and unlabeled data," in *International Conference on Computational Learning Theory*, pp. 111–126, Springer, 2005.

[229] V. Vapnik, *The nature of statistical learning theory*. Springer science & business media, 2013.

[230] Aleph One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[231] Google, "Google security blog." `https://security.googleblog.com/`.

[232] A. Developers, "Android developers blog." `https://android-developers.googleblog.com/`.

[233] K. S. Yim, I. Malchev, A. Hsieh, and D. Burke, "Treble: Fast software updates by creating an equilibrium in an active software ecosystem of globally distributed stakeholders," *ACM Trans. Embed. Comput. Syst.*, vol. 18, oct 2019.

[234] Clang Contributors, "Clang static analyzer." `https://clang-analyzer.llvm.org/`.

[235] M. Elsabagh, R. Johnson, A. Stavrou, C. Zuo, Q. Zhao, and Z. Lin, "FIRMSCOPE: Automatic uncovering of Privilege-Escalation vulnerabilities in Pre-Installed apps in android firmware," in *29th USENIX Security Symposium (USENIX Security 20)*, pp. 2379–2396, USENIX Association, Aug. 2020.

[236] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Information Security* (M. Burmester, G. Tsudik, S. Magliveras, and I. Ilić, eds.), (Berlin, Heidelberg), pp. 346–360, Springer Berlin Heidelberg, 2011.

[237] A. Armando, A. Merlo, and L. Verderame, "An empirical evaluation of the android security framework," in *Security and Privacy Protection in Information Processing Systems* (L. J. Janczewski, H. B. Wolfe, and S. Shenoi, eds.), (Berlin, Heidelberg), pp. 176–189, Springer Berlin Heidelberg, 2013.

[238] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *2014 IEEE Symposium on Security and Privacy*, pp. 424–439, 2014.

[239] T. Luo, H. Hao, W. Du, Y. Wang, and H. Yin, "Attacks on webview in the android system," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, (New York, NY, USA), p. 343–352, Association for Computing Machinery, 2011.

[240] J. Hu, L. Wei, Y. Liu, S.-C. Cheung, and H. Huang, "A tale of two cities: How webview induces bugs to android applications," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 702–713, 2018.

[241] Clang, "Undefined behavious sanitizer." `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`.

[242] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th annual computer security applications conference*, pp. 347–356, 2010.

[243] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi, "Seapp: Bringing mandatory access control to android apps," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[244] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*, pp. 590–604, 2014.

[245] B. Bowman and H. H. Huang, "Vgraph: A robust vulnerable code clone detection system using code property triplets," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 53–69, 2020.

[246] ele7enxxh, "List of android poc." `https://github.com/ele7enxxh/poc-exp`.

[247] Rapid7, "Metasploit." `https://www.metasploit.com/`.

[248] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards automated heap feng shui," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1647–1664, USENIX Association, Aug. 2021.

[249] T. Riom, A. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, and J. Klein, "Revisiting the vccfinder approach for the identification of vulnerability-contributing commits," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–30, 2021.

Annexe 1: List of Memory Corruption CWEs

CWE-288	Authentication Bypass Using an Alternate Path or Channel
CWE-305	Authentication Bypass by Primary Weakness
CWE-1084	Invokable Control Element with Excessive File or Data Access Operations
CWE-1127	Compilation with Insufficient Warnings or Errors
CWE-115	Misinterpretation of Input
CWE-179	Incorrect Behavior Order: Early Validation
CWE-408	Incorrect Behavior Order: Early Amplification
CWE-440	Expected Behavior Violation
CWE-444	Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')
CWE-480	Use of Incorrect Operator
CWE-483	Incorrect Block Delimitation
CWE-698	Execution After Redirect (EAR)
CWE-783	Operator Precedence Logic Error
CWE-835	Loop with Unreachable Exit Condition ('Infinite Loop')
CWE-837	Improper Enforcement of a Single, Unique Action
CWE-841	Improper Enforcement of Behavioral Workflow
CWE-385	Covert Timing Channel
CWE-257	Storing Passwords in a Recoverable Format
CWE-349	Acceptance of Extraneous Untrusted Data With Trusted Data
CWE-130	Improper Handling of Length Parameter Inconsistency
CWE-166	Improper Handling of Missing Special Element
CWE-167	Improper Handling of Additional Special Element
CWE-168	Improper Handling of Inconsistent Special Elements
CWE-182	Collapse of Data into Unsafe Value
CWE-229	Improper Handling of Values
CWE-233	Improper Handling of Parameters
CWE-237	Improper Handling of Structural Elements
CWE-241	Improper Handling of Unexpected Data Type
CWE-409	Improper Handling of Highly Compressed Data (Data Amplification)
CWE-471	Modification of Assumed-Immutable Data (MAID)
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-88	Improper Neutralization of Argument Delimiters in a Command ('Argument Injection')
CWE-140	Improper Neutralization of Delimiters

CWE-188	Reliance on Data/Memory Layout
CWE-463	Deletion of Data Structure Sentinel
CWE-641	Improper Restriction of Names for Files and Other Resources
CWE-791	Incomplete Filtering of Special Elements
CWE-1083	Data Access from Outside Expected Data Manager Component
CWE-544	Missing Standardized Error Handling Mechanism
CWE-617	Reachable Assertion
CWE-783	Operator Precedence Logic Error
CWE-129	Improper Validation of Array Index
CWE-179	Incorrect Behavior Order: Early Validation
CWE-183	Permissive List of Allowed Inputs
CWE-184	Incomplete List of Disallowed Inputs
CWE-606	Unchecked Input for Loop Condition
CWE-641	Improper Restriction of Names for Files and Other Resources
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-123	Write-what-where Condition
CWE-124	Buffer Underwrite ('Buffer Underflow')
CWE-125	Out-of-bounds Read
CWE-126	Buffer Over-read
CWE-127	Buffer Under-read
CWE-131	Incorrect Calculation of Buffer Size
CWE-786	Access of Memory Location Before Start of Buffer
CWE-787	Out-of-bounds Write
CWE-121	Stack-based Buffer Overflow
CWE-122	Heap-based Buffer Overflow
CWE-805	Buffer Access with Incorrect Length Value
CWE-806	Buffer Access Using Size of Source Buffer
CWE-128	Wrap-around Error
CWE-190	Integer Overflow or Wraparound
CWE-191	Integer Underflow (Wrap or Wraparound)
CWE-192	Integer Coercion Error
CWE-193	Off-by-one Error
CWE-197	Numeric Truncation Error
CWE-198	Use of Incorrect Byte Ordering
CWE-369	Divide By Zero
CWE-681	Incorrect Conversion between Numeric Types
CWE-839	Numeric Range Comparison Without Minimum Check
CWE-1077	Floating Point Comparison with Incorrect Operator
CWE-466	Return of Pointer Value Outside of Expected Range
CWE-587	Assignment of a Fixed Address to a Pointer
CWE-823	Use of Out-of-range Pointer Offset
CWE-501	Trust Boundary Violation
CWE-341	Predictable from Observable State
CWE-412	Unrestricted Externally Accessible Lock
CWE-73	External Control of File Name or Path
CWE-502	Deserialization of Untrusted Data

CWE-763	Release of Invalid Pointer or Reference
CWE-770	Allocation of Resources Without Limits or Throttling
CWE-774	Allocation of File Descriptors or Handles Without Limits or Throttling
CWE-789	Uncontrolled Memory Allocation
CWE-908	Use of Uninitialized Resource
CWE-909	Missing Initialization of Resource
CWE-1188	Insecure Default Initialization of Resource
CWE-828	Signal Handler with Functionality that is not Asynchronous-Safe
CWE-15	External Control of System or Configuration Setting
CWE-372	Incomplete Internal State Distinction
CWE-134	Use of Externally-Controlled Format String
CWE-135	Incorrect Calculation of Multi-Byte String Length
CWE-681	Incorrect Conversion between Numeric Types
CWE-843	Access of Resource Using Incompatible Type ('Type Confusion')
CWE-450	Multiple Interpretations of UI Input