

12-2021

Static and Dynamic Analysis in Cryptographic-API Misuse Detection of Mobile Application

Kunyang Li
William & Mary

Follow this and additional works at: <https://scholarworks.wm.edu/honorsthesis>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Li, Kunyang, "Static and Dynamic Analysis in Cryptographic-API Misuse Detection of Mobile Application" (2021). *Undergraduate Honors Theses*. William & Mary. Paper 1739.
<https://scholarworks.wm.edu/honorsthesis/1739>

This Honors Thesis -- Open Access is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Static and Dynamic Analysis in Cryptographic-API Misuse Detection of Mobile Application

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Computer Science from
William & Mary


by

Kunyang (Ella) Li

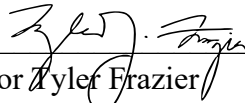
Accepted for _____ Honors _____
(Honors, High Honors, Highest Honors)



Professor Adwait Nadkarni, Committee Chair
Department of Computer Science



Professor Bin Ren
Department of Computer Science



Professor Tyler Frazier
Department of Data Science

Williamsburg, VA
December 6, 2021

Static and Dynamic Analysis in Mobile Cryptographic-API Misuse Detection

Kunyang (Ella) Li

Department of Computer Science

College of William & Mary
December, 2021

ABSTRACT

With Android devices becoming more advanced and gaining more popularity, the number of cryptographic-API misuses in mobile applications is escalating. Numerous snippets of code in Android are from Stack Overflow and over 90% of them contain several crypto-issues. Various crypto-misuse detectors come out aiming to report vulnerabilities of apps and better secure users' privacy. These detectors can be broadly classified into two categories based on the analysis strategies employed to catch misuses – static analysis (i.e., by scanning the code base) and dynamic analysis (i.e., by executing the code). However, there are not enough research on comparing their underlying differences, making it difficult to explain the pervasiveness of static crypto-detectors in both academia and industry. The lack of studies potentially limits the improvement of crypto-detection efficiency. In this study, a holistic evaluation and comparison on static and dynamic analysis' underlying mechanisms, robustness, and efficiency are carried out. A systematic empirical experiment is implemented on testing 1003 popular Android applications across 21 categories from Google Play. We find that 93.3% of the apps make at least one mistake using cryptographic APIs and closely analyze top four cryptographic rules reported to be violated most frequently by static crypto detector. Instead of merely comparing statistics such as false positives (i.e., false alarms), we focus on examining the crypto rules whose number of violations reported by static and dynamic crypto detectors diverge greatly. In addition, we firstly posit a new taxonomy schema that classifies cryptographic rules based on how they are inspected rather than their attack type or severity level. This schema will be useful to both researchers and practitioners to decide how to efficiently combine static and dynamic techniques to improve the reliability and accuracy of crypto-detection.

TABLE OF CONTENTS

Acknowledgments	iv
1 Introduction	1
1.1 General development of crypto-detectors	3
1.1.1 Types of crypto-detectors	3
1.1.2 Current dilemma of crypto-detectors	3
1.2 Major contribution	4
1.3 Overview of the following chapters	7
2 Background	8
2.1 Android OS and Applications	8
2.2 Cryptographic Algorithms	10
2.3 Mechanism of Crypto-detectors	12
2.4 High-level evaluation of crypto-detectors	13
2.4.1 Static analysis tools	14
2.4.2 Dynamic analysis tools	14
3 Related Work	16
3.1 Cryptography misuses and rules	16
3.2 Current Crypto-detectors	17
3.2.1 Static analysis	17
3.2.2 Dynamic analysis	18
3.3 Evaluation research	19

4	RC1: Comparison in mechanisms	21
4.1	Static tools	21
4.1.1	Our example: Cryptoguard	22
4.1.2	Support for the detection of native code	22
4.1.3	Transform DEX bytecode to readable Java source code	23
4.2	Dynamic tools	24
4.2.1	Our example: Crylogger	24
4.2.2	Generate execution traces for subsequent analysis	25
4.2.3	Automatic test input generation	26
5	RC2: Evaluation on accessibility and robustness	28
5.1	General perspectives	29
5.2	Dynamic tool: Crylogger	31
5.2.1	Experiment process 1: Remote headless server	31
5.2.2	Experiment process 2: Local VirtualBox Ubuntu	34
5.3	Static tool: Cryptoguard	35
6	RC3: Implementation, data, and plots	37
6.1	Implementation	37
6.2	Data and plots	39
6.3	Cryptoguard vs. Crylogger	41
7	Evaluation	43
7.1	Top four violations detected by Cryptoguard	43
7.1.1	Randomness: Cryptographically insecure PRNG	44
7.1.2	Integrity: Insecure cryptographic hash	46
7.1.3	Confidentiality: Predictable/Constant cryptographic key and password for PBE	49

7.1.4	SSL/TLS MitM Attack: Occasional use of HTTP	51
7.2	Overview on research questions (RQs)	54
8	Results and findings	55
8.1	RQ1: Crypto-misuses results among 21 different types of applications .	55
8.2	RQ2 - RQ4: New taxonomy for cryptographic rules	56
8.2.1	First stage: General taxonomy	56
8.2.2	Second stage: Crypto rule analysis	58
8.2.3	Third stage: Refined taxonomy	60
9	Limitation and Future works	62
9.1	Limitation	62
9.2	Directions of future works	63
9.2.1	Integration of static and dynamic analysis strategies	63
9.2.2	Automatically repair of the misuses	63
9.2.3	Machine learning and deep learning in vulnerability detection	64
10	Conclusion	65
	Bibliography	67

ACKNOWLEDGMENTS

First of all, I would like to express my sincere gratitude to my advisor Professor Adwait Nadkarni for his dedicated mentorship, who inspiringly guided me throughout the research and helped refine my thesis. Without his continuous patience, encouragement and immense knowledge, it would not have been possible.

Next, I would like to thank the rest of my Honors defense committee, Professor Ren Bin and Professor Tyler Frazier, for their valuable time, insightful comments and inspiring questions, and Charles Center for supporting me over the summer with research grant.

Last but not least, I would like to thank my parents who are always there to support and hearten me.

Chapter 1

Introduction

Attention on Android security and privacy, with a long history, has further boomed over the past few decades [1]. One of the essential reasons is the proliferation of Android in the advent of the Age of Big Data. As the world leading operating system in smartphone industry, Android takes more than 70% of all mobile devices, both the least and most expensive ones, running on Google-powered OS and has more than 2 billion monthly active users [4]. According to the latest data, Android is supported by a series of markets that boast about 2.79 million applications providing various functionality ranging from social networking to health and finance [54].

As its popularity strikes, security and privacy issues become more threatening to the public and attracts tremendous attention from academia. The security research community has invested significant effort in improving security of Android applications over the last couple of years [47]. This endeavor has effectively addressed a wide range of problems and led to creation of tools for application analysis.

Most of the related security problems belong to one of the following three threat models (i.e., criterion to partition security threats by identifying their objectives and vulnerabilities):

- **Ignorance of benign developer:** This could happen when developers do not have sufficient expertise or because of their carelessness. Benign developers may acciden-

tally misuse API in developing stages or introduce more vulnerabilities while trying to fix the original ones [3].

- **Evasion of attacker/developer:** This could happen when there are hired attackers by competing companies or by malicious developers within the company. The attackers may use strategies to break the existing system or steal private information for later usage; while evasive developers might purposefully leave a vulnerability in a software and escape their colleagues' scrutiny [3].
- **Threats from high tech – 5G:** 5G (i.e., the fifth generation of cellular networks) leads to the proliferation of end users, expansion of attack surface, and escalation of availability of personally identifiable data. Although it brings foreseeable benefits to the world, 5G is making cybersecurity more susceptible and its threat should not be overlooked.

Aiming at mitigate the damage of these problems, there are many tools emerging in academia and industry. One of the most popular ones is Bouncer released by Google. It quietly and automatically scans apps and developer accounts in Google Play with its reputation engine and cloud infrastructure to keep malicious apps off the official Android app store. According to Google, Bouncer was responsible for 40% drop of malign apps in Google Play [36].

We are going to narrow down various Android malware to cryptographic API misuses specifically in this thesis. A large portion of Android included snippets of code taken from Stack Overflow and 98% of these snippets contain several crypto issue [23]. Further, multiple studies have shown that there are a vast majority of Android applications misusing cryptographic APIs and libraries in the real world and are causing devastating security and privacy implications [2].

1.1 General development of crypto-detectors

Researchers have been spending too much time and effort in manually searching for those bugs. Gradually, cryptographic-API misuse detectors (crypto-detectors), which is designed to automate the process of finding vulnerabilities, gain ground.

1.1.1 Types of crypto-detectors

Generally speaking, there are two major types of crypto-detectors widely used in both academia and industry – static crypto-detectors and dynamic crypto-detectors. While having the same purpose – accurately and efficiently catching as many vulnerabilities as possible in Android applications, their fundamental differences are the underlying mechanisms to achieve this goal.

Static tools use various slicing and searching techniques in order to capture bugs by thoroughly scanning through the code of applications without actually executing them. In comparison, dynamic tools execute apps to check if there are misuses along the way. More details on mechanisms will be discussed in Chapter 2.

1.1.2 Current dilemma of crypto-detectors

Admittedly, although both of these tools have certain strength and weakness, neither of them are perfect in practice. They could generate multiple false positives, i.e., alarms raised on licit API calls, and false negatives, i.e., evasion of misuses, under difference scenarios. For example, some purposefully injected dead malicious code could easily escape detection of dynamic tools. Native library is a popularly used source of code to evade static tools' detection.

Besides their occasional inefficiency, risks of inadvertently injecting malign code while using crypto-detectors should not be overlooked. The code downloaded automatically from external platforms during execution could be both hard to discern and harmful to

the machine running it. Although its possibility is relatively low, it would be detrimental to the base system once it happens.

1.2 Major contribution

To the best of our knowledge, this thesis is the first work focusing on comparing dynamic and static crypto-detectors from such multifaceted perspectives. One of our major focuses is finding their discrepant results in checking certain cryptographic rules (i.e., detecting certain types of vulnerabilities) and analyzing the underlying reasons. This could be further used to not only improve the efficiency of individual tools but more effectively combine static and dynamic techniques in building crypto-detectors. In stark contrast, most of the existing works either focus on discussing the mechanism of their own artifacts or generally make comparison on dynamic and static tools by looking at their false positives and false negatives.

During the process of this thesis, We have done systematic analysis and evaluation both theoretically and pragmatically. Extensive literature review has been conducted covering most cutting-edge crypto-detectors and benchmark tools and existing evaluation studies. Additionally, we have carried out empirical experiments on over 1000 popular Android apps with both remote server and local virtual environment to attest our hypothesis and get final conclusions.

This thesis tends to address three key Research Challenges (RCs) bellow as well as four Research Questions (RQs), which will be discussed in depth in Chapter 7 and Chapter 8.

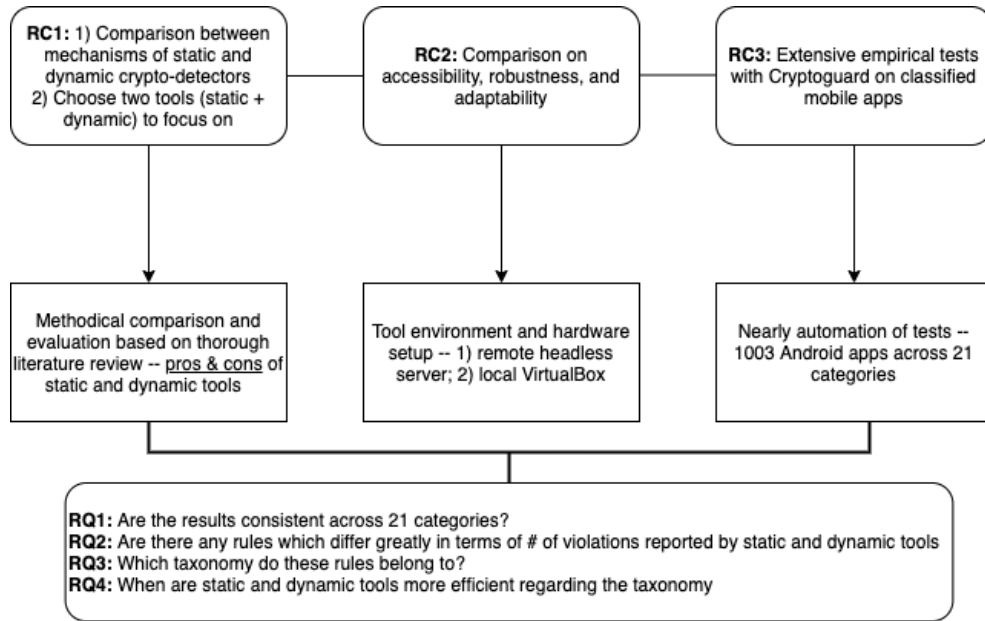


Figure 1.1: Research challenges & approaches

RC1: *Conduct in-depth comparison between mechanisms of static and dynamic tools and select two representative tools to focus on.* In addition to understanding the mechanisms' differences of the two types of crypto-detectors on the surface, deeper analysis on implication of these disparities on their efficiency is more critical. This requires detailed review on the structure and functionality of the tools. Besides, we need to read ample literature to select two most exemplary and advanced tools (i.e., one dynamic tool and one static tool) to focus on.

RC2: *Compare their accessibility, robustness, and adaptability (i.e., how easy to be used).* In order to make the comparison more comprehensive, analysis on robustness of crypto-detectors should be as essential as that on efficiency. This is also paramount in practice – how acceptable crypto-detectors are to software developers. Analysis on level of expertise required and hardware compatibility of dynamic and static tools are key components involved.

RC3: *Perform extensive empirical tests with one static tool (i.e., Cryptoguard) on classified apps.* Empirical experiment is indispensable to get a persuasive data-driven eval-

uation. This entails large-scale collection and implementation of tests on popular Android applications.

To address these research challenges, this thesis makes the following contributions:

- **Methodical comparison on their mechanisms (pros & cons):** We have narrowed down a number of differences of techniques static and dynamic crypto-detectors used to 2 key aspects for static and dynamic tools, separately – static tools: i) transform DEX bytecode to readable Java bytecode; and ii) detect native code; dynamic tools: i) execution trace for analysis; and ii) input generation automation. We finalized the tools – Cryptoguard (static) and Crylogger (dynamic) – that we are going to conduct more detailed research on. (addresses **RC1**).
- **Tool environment and hardware setup:** We contextualize the tools in both remote server and local virtual environment given the resource available for this thesis. Dynamic tools have more restrictions on its running conditions – i) ABI compatibility; ii) parameter tuning; and iii) emulator unstableness, while static tools, in general, are more adaptive and robust in practices without rigor requirement on the architecture and released versions (addresses **RC2**).
- **Nearly automation of substantial empirical tests:** We carried out tests on 1003 popular Android applications from 21 categories from Google Play Store in order to make our analysis more cogent and convincing. The automation of downloading process failed because given certain restrictions of headless server, we could only use third-party website, which is inherently unstable, to fulfill the downloading process. However, we successfully automated the process to perform the tests and to systematically collect the data with Shell and python packages. According to the data, there are 4 top rules which have been violated the most by Cryptoguard, which will be discussed later in Chapter 6. (addresses **RC3**)

1.3 Overview of the following chapters

In next chapter, we discuss the comparison and evaluation process in more details. In Chapter 2, we introduce more related information about Android architecture, common security vulnerabilities, as well as mechanism and high-level evaluation of static and dynamic crypto-detectors. In Chapter 3, we discuss the related work. In Chapter 4, 5, and 6, we address the three research challenges in depth, respectively. In Chapter 7, we analyze the results from empirical experiment and raise four research questions. In Chapter 8, we further analyze the results and address the four RQs. In Chapter 9, we discuss limitation of our work and future works before concluding in Chapter 10.

Chapter 2

Background

2.1 Android OS and Applications

As a pioneer of open source ecosystem, Android operating system has gained a wide familiarity globally in recent years. It takes up to 72% of all mobile devices, including smartphones and tablets, and owns 2.8 billion active users by 2021 [4] [14]. Android's popularity also comes from its innovative features like customizability, low-cost, and increasingly sensing and powerful computing ability.

Android is a mobile operating system with a Linux-based software stack created for a wide array of devices and form factors. It has six key components [17]:

- **The Linux Kernel** is responsible for underlying functionalities such as multi-threading and low-level memory management.
- **Hardware Accelerating Layer (HAL)**, with multiple library modules, shows device hardware capabilities (e.g., camera, accelerometer) to the higher-level Java API-framework by offering standard interfaces.
- **Android Runtime (ART)** optimizes garbage collection and utilizes different types of compilation in order to run multiple virtual machine on low-memory devices.

- **Native C/C++ Libraries** are base library modules that support various core system components and services (e.g., HAL, ART).
- **Java API Framework** makes the entire Android feature-set accessible to developers/users, who can reuse key library modules and certain components and services, including Content Provider, Activity Manager, View System, Resource Manager, and Notification Manager.
- **System Application** consists of all applications running in the system, ranging from most basic ones (e.g., calendars, contacts) to advanced third-party apps (e.g., social media, travelling).

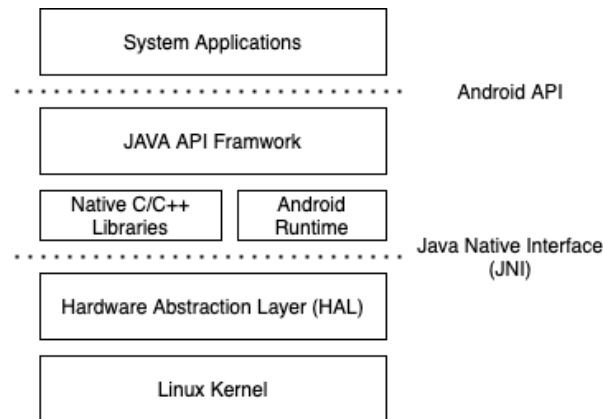


Figure 2.1: Android software stack

Android applications, which run on top of the stack framework, uses the provided APIs to access facilities without dealing with low level details of the operating system. They are mainly written in Java with some native code written in C/C++ for high performance. The four major different types of components in its Manifest.xml file are Activities, Services, Broadcast Receivers and Intents, and Content Providers. During the building process, Java source code is first compiled into Java bytecode, then translated into Dalvik bytecode, and finally stored in .DEX format. Apps are packaged/compressed and distributed in the form of apk files, which are compressed folders containing DEX files, optional native code, and other application resources [60].

The increasing prevalence of Android has also led the users of devices with Android platform to be especially susceptible to an enormous portion of malicious attackers. This situation is partially due to the simplicity to install any third-party applications to Android without sufficient scrutiny [41]. Admittedly, the newest version of Android (i.e., Android 11) has been making improvement on this – they use Google Play Protect to scan for bad apps. Nevertheless, since malware detection is considered as a crucial requirement to protect the users from personal privacy leakage, it is still an indispensable topic for Android. This is also one of the major reasons that we focus our research on Android platform.

2.2 Cryptographic Algorithms

Malware is defined as software that is specifically designed to damage the smartphone or gain authorized access to steal personal information [41]. With multiple types of malware present, we are going to focus on detecting cryptographic API misuses, which is one of the most fundamental and prevalent genres. We found that 93.3% of applications downloading from Google Play, the most commonly used app store, make at least one mistake using cryptographic APIs. This result is comparable to that – 88% overall – in another study [20].

Cryptographic algorithms can be generally classified into three groups – encryption algorithms, hashing algorithms, and signature algorithms. Altogether, they are aiming to secure data confidentiality, integrity, and authenticity. [57] The simplified process is shown in Figure 2.2.

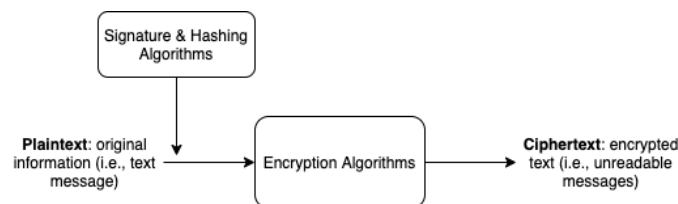


Figure 2.2: Encryption process

- **Encryption algorithms**, including block ciphers (i.e., transfers and encrypts data as an entire block) and stream ciphers (i.e., executes data as individual units), are used to encrypt and proffer confidentiality. Common stream ciphers are RC4 and ISAAC, while common block ciphers include DES, IDEA, and RC5/RC6. Most Advanced Encryption Standard (i.e., AES, which is used by the US government and others as standard algorithms for protecting highly sensitive data) candidates are block ciphers, which are commonly used in Android applications.
- **Hashing algorithms** (i.e., hash functions) are used to create a unique digital fixed-sized "fingerprint", which varies once the original message has been modified. There are several different types of hashing – division-remainder, digit rearranging, and folding. Standard hashing algorithms consist of MD2, MD4, MD5, and Secure Hash Algorithm (SHA). Among these, MD2 and SHA1 are considered insecure and strongly discouraged (more details in Chapter 7).
- **Signature algorithms** are used to sign the data to authenticate the message sender is the person he/she claims to be. Digital signatures, which is popularly used recently, include both signature algorithms and hashing algorithms.

Additionally, as a key property of many encryption algorithms, ciphertext indistinguishability is paramount. Based on IND-CPA security (i.e., indistinguishability under chosen-plaintext attack), an encryption scheme must be either probabilistic or stateful to be indistinguishable under chosen plaintext attack. If a cryptosystem lacks of ciphertext indistinguishability, ciphertext, which will render certain amount of information of plaintext, will easily leak those pieces of information to the attackers. Attackers will then have a higher probability to narrow down to obtain the actual plaintext than they could by random guessing.

2.3 Mechanism of Crypto-detectors

As cryptographic algorithms has been prevalently used to secure information authenticity, confidentiality, and integrity, it is indispensable to ensure they have been properly and safely used. A vast amount of cryptographic API misuses detectors have been created, and they aim to make this vulnerability-catching task more efficient and effective by automating the entire process. Crypto-detectors are usually categorized into two groups – static crypto-detectors and dynamic crypto-detectors – based on separate approaches used.

Static tools need to scan through the entire applications’ source code without actual execution to check certain cryptographic rules (shown in Figure 2.3). During the process, they use various slicing strategies, including backward/forward slicing and interprocedural/introprocedural slicing. These techniques differ mostly on how they slice the program either with flow graph (i.e., control-flow graph) or dependence graph (i.e., program/system dependence graph) to search for specific class/method in order to examine rules. These slicing strategies facilitate the process of searching for key words during misuse detection.

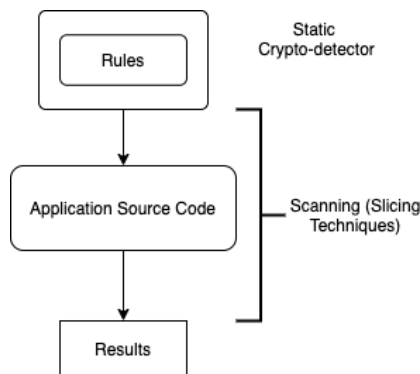


Figure 2.3: Mechanism of Static Crypto-detectors

Dynamic tools, in contrast, need to execute the applications with emulator to catch vulnerabilities (shown in Figure 2.4). In this case, it is more demanding on the environment and hardware (e.g., architecture, memory). Logger and checker are used to facilitate this process by systematically collecting and inspecting runtime log messages [42]. Logger, which modifies low level Java libraries, monitors the APIs of the crypto algorithms and

stores the value of relevant parameters to a log file during runtime; while checker analyzes the the log file and reports the violated crypto rules offline. The online-offline combination effectively optimizes its efficiency and memory space needed.

Besides, automated input generator is another core component, it is used to generate random screen inputs during the execution time in order to trigger as many API calls as possible for later examination. These are effective at exposing security vulnerabilities. The most frequently used tool to test Android is Monkey [59], which is part of Android developers toolkit and thus does not require any additional installation effort. It is a program that runs on emulators or devices and generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events. It has an upper limit of number of events users want to generate, once which have been met, Monkey stops. [60]

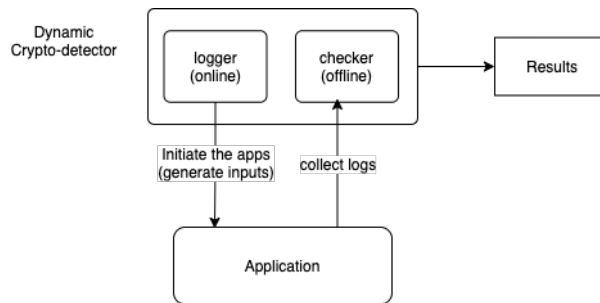


Figure 2.4: Mechanism of Dynamic Crypto-detectors

Moreover, there are crypto-detectors which combine static and dynamic strategies. But the number of them are limited due to the difficulty in combining these two fundamentally different approaches.

2.4 High-level evaluation of crypto-detectors

In this section, we evaluate static and dynamic crypto-detectors' strength and weaknesses from a high level (more comparisons on low level mechanisms will be in Chapter 4).

2.4.1 Static analysis tools

Static strategies have been well studied outside the field of computer science, with multiple fields having researchers interested in such topics. This is partially because its appealing ability to scan *all* of the source code efficiently. Without executing applications, static crypto-detectors can be used to detect errors in an initial stage of agile software development cycle (i.e., with incomplete deliverables), and this can thus allow immediate fix during implementation. Another advantage of detection without execution is that developers/researchers do not have to install applications before examining them. This not only simplify the time-consuming process but could also avoid possible malware attacks. Static tools, in general, are highly scalable, they can be easily adapted to accommodate different environment (i.e., architecture, version) and be used with a large code base.

The drawbacks of static tools due to the absence of execution are conspicuous, nonetheless. Since they only look at source code, no user experience is involved, which is detrimental to some industry such as entertainment and consulting firms. They could not collect any first-hand feedback with static analysis tools. Moreover, consistent usage of static tools are not very frequent in the industry because of numerous elusive feedback [7] [27]. Even though inspecting the code manually expects more efforts, some developers prefer not to use those tools to avoid confusion while interpreting the results. Static tools also tend to generate a high proportion of false positives, which could only be eliminated manually. Additionally, from the perspective of developers of the applications, static analysis tools could lead to sensitive information leakage to some degree, since the entire source code has to be exposed.

2.4.2 Dynamic analysis tools

Contrary to static tools, dynamic analysis tools inspect applications by actual execution. Besides using automated input generation techniques, developers can choose to interact with the UI directly. This can tremendously help identify vulnerabilities that don't ad-

here business context or standard from real user experience. During the execution either automated or manual, developers are capable of monitoring system memory, functional behavior, response time, and overall performance of the system. Since the dynamic crypto-detectors will run the application directly, there is no need to look for source code, which would be helpful in some situations (e.g., working with remote server). In general, dynamic tools generate less false positives compared to static ones.

Despite the benefit of using dynamic approaches, certain defects could not be overlooked. To successfully use dynamic tools requires more efforts. Some expertise is required to setup environment and build the model. Some limitations come from automated generation of UI events because they can't deal with login screen (i.e., they are randomized input and cannot deal with login information). Once these programs got stuck in one problem, the whole process has to be reinitiated. Further, higher accuracy rates could be achieved with more UI events and API calls triggered. This will make the process more time-consuming.

Thanks to the distinct characteristics of static and dynamic analysis tools, both researchers in academia and industry practitioners are constantly handling their trade-offs, and their performance has always been improving.

Chapter 3

Related Work

This chapter discusses related work from three areas: cryptographic misuses and common rules, static and dynamic current crypto-detectors, and evaluative studies on current detectors.

3.1 Cryptography misuses and rules

There are an increasing number of studies focusing on examining android developers' usage of cryptographic components. A critical milestone is Egele's *et al.* study [20] on whether developers use the cryptographic APIs in a fashion that provides typical cryptographic notions of security (e.g., IND-CPA security). They found that 88% applications which uses cryptographic APIs make at least one mistake. This alarming result is not novel. Krüger *et al.* [30] found 95% of apps have at least one misuses in their cryptographic libraries. Chatzikonstantinou's *et al.* study [10] shows that 87.8% of the applications present some kind of misuse, while no cryptographic usage was detected for the rest of them. The study of Lazar *et al.* [31] claims that 83% of mistakes they found are because of cryptographic libraries misusing by individual applications. Our own results confirm these high values of cryptographic misuse by examining 1003 Android applications from Google Play.

Most of these studies focus on inspecting several key cryptographic rules. A very

common one is that if a symmetric cipher (e.g., Advanced Encryption Standard, AES) is configured with Electronic Code Book (ECB) mode, presence of patterns in plaintexts will be leaked in ciphertexts [9], and this will violate IND-CPA security. Besides, confidentiality will be compromised if hard coded or reused IVs are used or if constant password for PBE or other cryptographic keys are used; integrity could not be guaranteed if insecure hash functions such as SHA1 and MD2 are used; and randomness will be violated if constant seeds are given to pseudorandom number generators (PRNGs) [10] [45] [20]. There are many other common cryptographic rules as well, which will be discussed in more details in Chapter 7. Chatzikonstantinou *et al.* [10] has classified these misuses into four categories – use of weak cryptography, weak implementation, use of weak keys, use of weak cryptographic parameters.

3.2 Current Crypto-detectors

Many crypto-detectors exist to detect cryptographic misuses. Among them, tools using static analysis have outnumber those with dynamic analysis. As mentioned above, this is largely because of the static analysis tools could scan the entire body of code and is regarded to be relatively more comprehensive, which, however, does not always lead to a more accurate detection result.

3.2.1 Static analysis

During the last couple of years, there are several static crypto-detectors emerging. While trying to improve the detection accuracy, they all possess unique features. MalloDroid [22] [38] and FlowDroid [6] focus on detecting misuses of SSL/TLS protocol. CryptoLint [20], as one of the earliest ones, uses static program slicing to identify flows between cryptographic keys, initialization vectors, and similar cryptographic material and the cryptographic operations. Without getting access to application source code, it checks for typical cryptographic misuses quickly and accurately by disassembling raw Android binaries. CMA [50]

performs static analysis on Android apps, selects the branches that invoke cryptographic APIs, and then checks the usage of those API calls by following the target branches. CrySL [30] is a specialization language which bridges the gap between cryptographic experts and developers by enabling crypto experts to specify the secure usage of certain crypto libraries and translating those CrySL specialization into static analysis. CogniCrypt [29] is a more well-rounded tool supporting developers with the use of cryptographic APIs from two ways. It generates code that implements the respective tasks in a secure manner and, meanwhile, continues running static analysis in the background to ensure a secure integration of the generated code in developers' workspace. Among all, CryptoGuard [45], to the best of our knowledge, covers the highest number of cryptographic rules and guarantees the highest detection efficiency. It is build on specialized forward and backward program slicing techniques, which are implemented by using flow-, context-, field-sensitive data-flow analysis. One of its outstanding features is a set of refinement algorithms that systematically discard false positives. This is a huge improvement compared to other static tools.

3.2.2 Dynamic analysis

Other tools employ dynamic analysis in crypto misuses detection process. SMV-Hunter [52] detects protocol misuses in SSL/TLS certificate validation process which could otherwise make applications vulnerable to SSL/TLS Man-in-the-Middle attacks. Similarly, AndroSSL [24] also focuses on security issues of SSL/TLS connections established by Android apps. Meanwhile, iCryptoTracer [33] detects cryptographic misuses in iOS apps with combined static and dynamic analyses which entails complex implementation with API hooking techniques. K-Hunt [32] is a more recent tool which identifies insecure keys such as deterministically generated keys, insecurely negotiated keys, and recoverable keys through analyzing binary executables. TaintDroid [21] operates an efficient and system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. It focuses on reducing leakage of sensitive data by inspecting how third-party applications collect and share users' private data. Crylogger [42], among

all, is the first open-source tool to detect crypto misuses dynamically in a comprehensive way. It examines 26 crypto rules and is implemented both during the execution and offline to optimize the efficiency.

3.3 Evaluation research

Various evaluation research is conducted with thorough literature review and designed benchmark tools. Reaves *et al.* [47] performed evaluation on Android security research which analyzes applications and characterize the work published in top venues. Rahaman *et al.* [46] created a comprehensive benchmark – CryptoAPI-Bench – with both basic and advanced unit test cases in order to compare leading analysis tools. Further, Ami *et al.* [3] designed MASC framework which enables a systematic evaluation of crypto-detectors with data-driven taxonomy and generation of usage-based mutation operators and threat-based mutation scope.

Braga *et al.* [9] claims that both tool builders and software developers underestimate security issues related to public-key cryptographic in general. They used statistics such as precision, recall, and F-Measure as calculated measurements to compare distinct static tools directly, uncover their limitations, gaps and overlaps, and to determine the impact of static analysis tools in positively influencing the correct use of cryptographic in software development. Johnson *et al.* [27] also analyzed the reasons for cases of underused static crypto-detectors. They found false positives of static analysis could potential outweigh the true positives in volume. Besides, the way in which the warnings are presented are barriers to use. This would require developers to spend a lot of time trying to figure out what needs to be done to fix problems. According to Poeplau *et al.* [44], another latent drawback of static tools is that the design of android system allows applications to load additional code from external sources at runtime. Malware can use this capability to add malicious functionality after being inspected by official application store or anti-virus engine at installation time, while developers of benign applications can inadvertently

introduce vulnerabilities.

Compared to evaluative studies on static analysis tool, those on dynamic crypto-detectors eclipsed in volume. Most of them focuses on assessing the efficiency of automated input generations. Choudhary *et al.* [60] perform a thorough comparison on the effectiveness and corresponding techniques (i.e., code coverage, ability to detect faults, ability to work on multiple platforms, and ease of use) of major existing test input generation tools for Android. Zheng *et al.* [60] analyze one specific automatic test input generation for Android – Monkey, which is under spotlight of research and ways to improve on its limitations. According to Chatzikonstantinou *et al.* [10], one major benefit of dynamic analysis is that developer are able to monitor system memory, functional behavior, response time, and overall performance of the system through a dynamic test.

Chapter 4

RC1: Comparison in mechanisms

In general, the number of studies on static analysis tools is significantly larger than that of dynamic tools. What is the underlying rationale behind this? Does this lead to the conclusion that dynamic analysis is inferior to static analysis? In this chapter, we aim to tackle RC1 – *comparison between mechanisms of static and dynamic crypto-detectors and finalize two representative tools (one static and one dynamic) to focus on*. After examining their working procedures from both empirical experiments and literature, we summarized some features (pros and cons) of the process below.

4.1 Static tools

The distinct feature that set static analysis tool apart is its capability to thoroughly scan the source code without execution. This is beneficial since it could cover code more comprehensively and does not require any modification on low level code (e.g., native Java libraries, etc.), which is important since the working environment would not be seriously changed. However, the potential drawbacks are apparent as well. For example, they have to figure out how to accurately detect untriggered but malicious native code. Besides, in order to obtain readable Java source code from DEX files, static crypto-detector has to properly and efficiently transform DEX bytecode. We will be focusing on these two aspects in the following sections.

4.1.1 Our example: Cryptoguard

Among all static crypto-detectors, we decided to choose Cryptoguard [45] on which to conduct more research. To the best of our knowledge, Cryptoguard is one of the most advanced static tool and has significantly decreased false positives, which is a major setback of static tools. In order to achieve high accuracy rate, it has refined strategies for forward and backward slicing and for discarding false alerts. However, as one of state-of-the-art static crypto-detectors, Cryptoguard still has certain limitations as disclosed below.

We used Cryptoguard as our example to further inspect static crypto-detectors' detailed features and performance. Besides assess its mechanism specifically, we utilized it to systematically conduct tests on 1003 Android applications from Google Play and looked into its source code to analyze why certain rules that have been violated the most.

4.1.2 Support for the detection of native code

One major limitation of static analysis tools is its support for the detection of native code. Native code is widely used in practice, however, it severely complicates the process of static analysis. Researchers found that 14% of application with fewer than 50,000 downloads contained at least one native library, whereas 70% of applications with more than 50 million downloads contained at least one native library [58]. Nevertheless, native code could be inherently challenging for static tools to detect. Some code from native libraries are simply hidden with concealed execution path and could escape the scanning of static crypto-detectors; others are obfuscated by some obfuscators binaries such as OLLVM [47]. Obfuscation is not novel to static analysis tools, but there are only few tools focusing on native code obfuscation or recovery of the original control flow graph [28]. These scenarios will largely increase the difficulty of security analysis process.

It clearly indicates that there is a gap in static crypto-detectors and would encourage attackers to inject more malicious code into native code in order to evade security detection. However, many of the extant static tools such as Cryptoguard overlooked native code

obfuscation during their detection process. Although their accuracy is not significantly damaged so far, the potential harm from it could be irreparable once it succeeds and expands.

4.1.3 Transform DEX bytecode to readable Java source code

Another challenge unique to static tools is to transform DEX bytecode to readable Java source code, and Java bytecode is a crucial medium. Although Java is the widely-used programming language for Android applications, Java bytecode is not contained in the application package. This is because that aiming to run the applications in Android-specific Dalvik Virtual Machine, the Android SDK has to include an extra step to transform Java bytecode into DEX bytecode [47]. Inaccurate transformation of the executable source may severely degrade the program analysis performance and obscure the results [5]. Since a vast majority of static analysis tools are designed to operate on either Java bytecode or Java source code which is further derived from Java bytecode, the need of an appropriate reverse engineering technique is ample and urgent.

The most common strategy applied by static analysis tools to solve this is to include a *decompiler* [41], which is used to decompile both the Manifest.xml and DEX file to generate an easily-readable version of Java code. However, rather than directly decompiling DEX bytecode to Java source code, the transforming process is markedly different, since the former process could result in hard-to-read source code with infinite while loops and break statements [47].

There are several existing tools, dealing with this complex process, and they could be classified into two groups – one-step process and two-step process [53]. Strategies of the latter group have been commonly utilized and the representative tool is *DEX2jar* [15]. It first converts DEX file to jar file and then to Java code, facilitated by *jd-gui* [18], which is popularly adopted to display Java code. *DEX2jar* could generate greatly readable code but it is hard to modify or recompile the code with it. In comparison, *jadx* [16] directly produces Java source code from DEX file. It is considered to be relatively more robust and

preferred by some developers since, without generating jar file, it is capable to circumvent certain problems of broken code.

Although there are constant improvement, the assumption that this challenge has been solved is still far from true. Several tools are created to optimize the Java bytecode before further decompilation to Java source code, but they remarkably increase the performance time [47]. For example, Cryptoguard uses Soot optimization, which sacrifices performance for more readable code, to decompile apk file to Java source bytecode. Thus, this reverse engineering process remains to be one of the limitations of static analysis tools.

4.2 Dynamic tools

Dynamic tools are designed to detect security vulnerabilities in a running system, and one inherent drawback of this is its potential low code coverage. It would be unlikely for dynamic crypto-detectors to cover the entire code base. However, as seen in the experiment, with *logger*, the *checker* submodule is much more straightforward to be understood, implemented, and modified. Its main purpose is to check offline if messages in *logger* are legit given the predefined cryptographic rules. However, we found that the demand to modify native Java libraries could cause some issue. We noticed that, during the experiment, nearly half of the fellow developers have overlooked that they are required to modify native libraries such as *libcore* before execution. There are other limitations due to the involvement of runtime *logger* and automated input generation strategies, which would be discussed in details in the following subsections.

4.2.1 Our example: Crylogger

We chose Crylogger [42] as our example to further analyze dynamic crypto-detector, meanwhile, we will also take Monkey, which is fully automated, lightweight, and common among developers, as an example for automatic input generation. As a noticeably recent tool which is published in 2020, Crylogger is the first open source tool to detect cryptographic

misuses dynamically and, further, it supports large number of rules (i.e., 26 crypto rules). All these features set Crylogger distinct from its counterparts and make it a perfect target for our research.

We analyzed techniques used by Crylogger in depth. In addition, we set up Crylogger both on remote server and on local virtual machine to test its applicability. However, due to limited resource, we only focused on its accessibility and robustness during our experiment and decided to collect results from its authors to further compare its performance results to those of Cryptoguard. Compared to static tools, except for its relatively inferior robustness, Crylogger significantly decreases false positives and since it combines online *logger* and offline checker, its performance is not largely compromised. However, admittedly, it is inevitable to confront challenges from execution traces management and automatic input generation.

4.2.2 Generate execution traces for subsequent analysis

Dynamic tools examines applications by inspecting information captured at runtime, therefore, generating and collecting execution traces for succeeding analysis is paramount. These traces can happen at any level in the software stack during runtime, including native processor instructions, virtual machine instructions, system calls, and Android API calls [47]. A full view of application behavior could be obtained from the information provided by these traces (i.e., certain parameters, locations of the traces, etc.).

There are usually two general approaches to deal with traces – on-line trace analysis (i.e., traces are analyzed immediately during execution) and off-line trace analysis (i.e., traces are stored for later analysis) [19]. Crylogger uses the latter technique by implementing an online *logger* and offline *checker*. It separates the process of obtaining traces and that of analyzing traces in order to improve performance but it also unavoidably sacrifices some memory space for storage.

In all, the major tasks for dynamic tools are i) how to narrow down and generate specific traces to acquire only desired information; and ii) how to convert information in

original traces, which is presented in low-level machine instructions, to readable high-level semantics for later analysis. Instead of collecting the entire piece of arcane information, *logger* in Crylogger tackles these two challenges through monitoring the API calls and collecting only the relevant parameters that are passed to the APIs of the crypto algorithms. Those parameters are presented in a much more straightforward manner, which can ease the checking process.

4.2.3 Automatic test input generation

There are many existing tools which aim to test inter-application communication by randomly generating input values. These tools have slightly different purposes. Monkey [59], as a frequently used tool, uses black-box to generate UI events and is easy to use without additional installation efforts. Dynodroid [35] is smarter than Monkey in several perspectives – it takes the context and frequency of events into account so that events that are relevant in more context will be selected often. In comparison, Intent Fuzzer [48] mainly tests how applications interact with other applications on mobile devices. We chose Monkey as our example not only because Crylogger chose it as well but also due to its great performance in both simplicity and efficiency.

There are several apparent setback of Monkey. First of all, it can easily get stuck in some activities during execution and could not continue to explore the rest of the application until restart the process. Manual work is required here. Secondly, it cannot deal with registration or login page which appears before the application would be used. We have taken this into consideration during our empirical experiment and have eliminated some types of applications. In addition, Monkey usually obtains low code coverage – the input generation process is randomized and finite (e.g., 10k, 30k random events), it is unlikely to cover the entire code base and some malicious code could successfully evade detection. One remedy for this is to set a large number of UI events for Monkey to generate, however, this could substantially slow down the performance.

Nevertheless, according to Crylogger [42], although Monkey achieves on average about

25% of line coverage, the number of reports on crypto rules violations is comparable to that of other static tools. This result is a little counterintuitive but successfully reveals that the seeming drawbacks of dynamic analysis tools might not always hold water. Admittedly, Monkey has inherently limited code coverage but this does not necessarily compromise its efficiency in triggering most cryptographic APIs for catching vulnerabilities. We assumed that this could be caused by the fact that cryptographic API calls are crucial and commonly used in mobile applications so that they can be easily triggered. Furthermore, the results of Crylogger which runs with 10k, 30k, and 50k UI events are actually similar, and this helps confirm our assumption.

In all, given the above analysis, developers should not simply discard the option of designing or applying dynamic analysis tools because of the limitation of automatic test input generation. This also partially substantiates one of our primary hypotheses that it might be promising to combine static and dynamic analysis techniques.

Chapter 5

RC2: Evaluation on accessibility and robustness

In this chapter, we aim to address RC2 – *compare crypto-detectors’ accessibility, robustness, and adaptability*. We are going to discuss over the accessibility (i.e., how easy the tools can be learned/used) and robustness (i.e., how strong the systems of the tools are while handling diverse and unexpected cases) of current crypto-detectors, and specifically we will compare Cryptoguard and Crylogger from these perspectives. Both accessibility and robustness are crucial in the life cycle of crypto-detectors since they determine whether one tool will be widely adopted by developers in industry and academia at the starting point and whether it will survive after intense competition. In order to make our evaluation more tenable, we put it into practical context by conveying empirical experiments with Crylogger and Cryptoguard and recording technical issues along the way.

Despite all of the technical issues occurred, we were constantly inspired to carry out the analysis to a deeper level to find out the underlying problems. In summary, Cryptoguard turns out to be more robust and adaptive in practice with only rare failures in the testing process.

5.1 General perspectives

Generally speaking, we assessed tools’ accessibility and robustness from four aspects in three phases of the tool’s life cycle as shown in Figure 5.1. We then applied this evaluation schema to Cryptoguard and Crylogger.

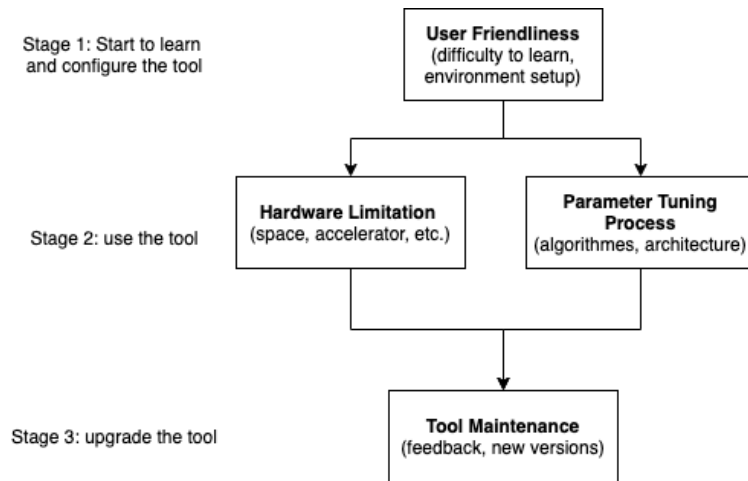


Figure 5.1: General aspects and stages of tools’ life cycle

User friendliness (i.e., Stage One) includes indicators of how easy one tool could be learned in terms of environment understanding and setup. In practice, tool’s level of difficulty to be learned and used by novices with limited specialized knowledge is a determining factor of its future success. Usually the user groups consist of undergraduate/graduate students and employees in industry with only shallow expertise [47]. Furthermore, even with open source code, there is often a steep learning curve before the tool can be DEXterously applied. Although Crylogger claims to be largely improved and simplified upon its process of environment setup [42], a visible gap still exists during our experiment process. Many incompatible issues get in the way of our research experiment such as ABI incompatibility with emulator and AOSP setup problems on virtual machine. Supports from hardware (e.g., Linux machine) and software (e.g., emulator setup) are particularly entailed and tend to be easily mishandled by users. In comparison, getting started with Cryptoguard is more trouble-free. Only JDK and Android SDK are listed in prerequisites. One solution

to make tools more user friendly is to release a virtual machine image together with the artifact. However, many technical issues still occurred when Crylogger did release a virtual image.

Hardware limitation is a crucial component of Stage Two. This applies especially when emulators are involved. Accelerators, even though are not required theoretically, are strongly recommended in practice since limited speed is an inherent challenge of emulators. Besides, lack of sufficient memory space also limits the usage of certain tools. For example, Crylogger entails AOSP (i.e., Android Open Source Project) which offers source code to create customized Android variant. The requirements of AOSP includes 500 GB of free disk space (i.e., 250 GB to check out the code and 150 GB to build it) and recommended 64 GB of available RAM [12]. Other common hardware requirements includes virtualization techniques, KVM acceleration, GUI supports such as XQuartz, and so on. These conditions may be easily satisfied in common lab settings but it caused substantial problems when we tried to configure it virtually with remote lab servers during the pandemic.

In addition to restraints on hardware, software parameter tuning process has comparable importance in Stage Two. Researches has to understand the underlying mechanism of the tool in order to appropriately tune the parameters. For example, knowledge of both base machine, emulator, and mobile applications' architecture and major techniques of crypto-detector are in demand so as to input proper parameters for either emulators (i.e., *adb* commands) or cryptographic algorithms.

Tool maintenance belongs to the last phase. Most tools are open-source tools with source code available on Github, and the frequency of their updates on code fixes are, therefore, apparent. Among crypto-detectors, there is a wide discrepancy exists – some have been actively updated while others are outmoded, fragile, or only exists in theory (i.e., some research release no actual artifact beyond an academic paper). Cryptoguard has more than 3 versions while Crylogger has only one version available. However, release time is also needed to be taken into consideration since Crylogger is more recently released than Cryptoguard. Many technical issues including corner cases, which are hidden or

overlooked by developers for some reason, will be easily revealed by developers or researchers in public once released. To solve those issues not only is favored by practitioners who need the tools to guarantee security issues in real world but can also make the tool more compatible, applicable, and robust in order to become more popular. In this case, original tool developers should take the responsibility to constantly update their tool.

5.2 Dynamic tool: Crylogger

The empirical experiment we have carried out with Crylogger involves two phases. In the first phase, we used remote lab server to install and conduct tests with Crylogger, while in the second phase we switched to use local virtual machine (i.e., VirtualBox) after confronting several technical obstacles. Crylogger, as a representative of dynamic crypto-detectors, is more restricted on hardware requirements than static counterparts are. Although we did not end up conducting large-scale testing on Android applications with Crylogger due to limited resource available during the pandemic (e.g., limited virtualization, KVM accelerator, and GUI support with remote server), we collected information about each problematic case and analyzed them respectively to find out its fundamental causes.

5.2.1 Experiment process 1: Remote headless server

We started with using remote headless server. The process involves downloading Android SDK and JDK, downloading AOSP and installing Crylogger by making required modifications on AOSP, installing OpenGApps, building and starting emulator, and, finally, installing and testing applications. However, the above process went through only with the sample application provided by the authors but not with any other published Android applications. In short, none of the applications can be installed on the emulator besides the sample application. After discussing with other related researchers, who conducted research on Crylogger before, through online forum and in-depth consultation with the

original author of Crylogger, we hypothesized that this may be caused by connecting to the server remotely and without GUI support.

During our experiment, we managed to overcome most of the problems by adjusting parameters with emulator, but the only bridle that we could not surpass is *Broken Pipe (32)* and *Transaction failure* errors while trying to install Android applications onto the emulator through command line. Slightly different errors were met by other researchers as well but not by the authors. After consulting the author, we learned that the environment he worked in while developing Crylogger is "a physical machine (no virtual machine) with Ubuntu 18.04.1 with kvm installed and with the possibility of opening the graphical user interface of the emulator". He then suggested me to either get physical access to Linux machines or export graphical interface (i.e., GUI) through SSH. However none of these worked out for me – i) because of the pandemic, I did not have the chance to work physically on Linux machines in lab; and ii) since I was physically distant from the server (overseas), any graphical interface tools were so slow that this option turned out to be impractical. However, this does not discourage me from continuing the research, on the contrary, it accidentally provides me a unique perspective to assess dynamic crypto-detectors. By simply switching workspace might bypass this specific problem but the fundamental issue still exists, and might appear elsewhere to others in the future as well. Thus, I have conducted further analysis on how and why these obstacles come across and whether there are ultimate solutions for them.

We have come up with four hypotheses why errors such as *Broken pipe* and *Transaction failure* could occur which are shown in Figure 5.2.

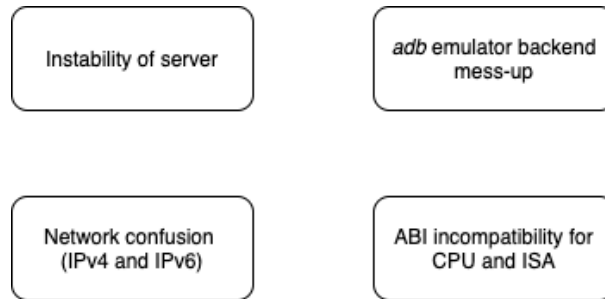


Figure 5.2: Four hypotheses for errors related to Crylogger

- **Instability of server** – The first and most intuitive hypothesis is that since emulator is running on remote server which is connected through SSH across long distance, it is innately unreliable. The most common solution is to reboot the server to start over the process. However, there are still various error messages after rebooting – *broken pipe (32)*, which is the most usual one, *failed transaction*, and *can't find service (package)*.
- **adb emulator backend data messed up** – This is similar to the first hypothesis but focuses more on emulator complications. Given that emulator is running with virtualization support, it tends to slow down or crash when backend data behaves oddly for some unknown reason. One of the possible reasons is that there are previously installed apps with the same name as the current one or the current apps have already been downloaded before. To avert this, we have used commands such as `adb -s emulator-5554 uninstall xxx.apk` and `./gradlew clean` to constantly check and uninstall unnecessary apps. Besides, we also used commands such as `adb kill-server` $\&\&$ `adb start-server` to restart the emulator. This worked for few developers as reported online over StackOverflow, but is not helpful in our scenario.
- **Network confusion (IPv4 and IPv6)** – Same type of issues has also been brought up by Android developer official website [13]. They presume that this *broken pipe* error is caused by failure of Gradle sync process and, more specifically, is because of network confusion (i.e., Gradle is trying to use IPv4 rather than IPv6). Since IPv4

and IPv6 use different bit-size IP addresses, they enable unique communication approaches among devices and applications. Thus, we set Java option to IPv6 in profile configuration for shell. However, by doing this does not solve the problem. There might either be more complicated network confusion issue that could not be simply addressed by choosing IPv6 or this has nothing to do with network configuration.

- **ABI incompatibility** – This explanation ended up being the most possible and fundamental cause of our problem after our analysis. Android Application Binary Interface (ABI, for short) typically includes the CPU instruction set, the endianness of memory stores and loads at runtime, conventions for passing data between applications and system, and format of executable binaries, etc [11]. Since different Android devices use different CPUs, which in turn support different Instruction Set Architectures (ISAs, for short), each combination of CPU and ISA has its own ABI. While working with native code, hardware matters. The Native Development Kit (NDK, for short) ensures that developers are compiling for the right architectures and CPUs by providing a them a variety of ABIs to choose from. Incompatibility of ABI between applications and emulator might inherently limit the apps that could be installed to the emulator and be further tested by Crylogger. This could not be simply solved by paying more attention to match the version and architecture of Android applications while downloading to that of the emulator.

5.2.2 Experiment process 2: Local VirtualBox Ubuntu

Because of the above mentioned obstacles, we were not able to continue our experiment with remote server and chose to use Ubuntu 20.04.2.0 with local virtualization (i.e., VirtualBox) instead in order to work on Linux operating system. However, this also introduced several issues. The two main problems are caused by i) nested virtualization and ii) Virtual Private Network (VPN, for short) availability.

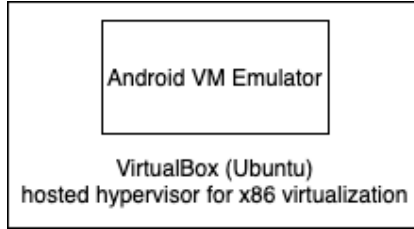


Figure 5.3: Nested virtualization scenario

Nested virtualization appeared when we were trying to run a virtual machine instance (i.e., android emulator) inside of another virtualization environment (i.e., Ubuntu with VirtualBox). This scenarios could potentially cause several problems. While some hyper-visors can nest within KVM, this does not work well in our case with VirtualBox. We were not able to use hardware accelerator (i.e., KVM) again this time. Besides, because of physical limitation (i.e., conducting this experiment abroad), we could not download Android AOSP directly from official website but need mirror images. However, to our best knowledge, all of the mirror images in public has been either broken/unavailable or outdated so that they are not well-suited with the image provided by Crylogger’s authors. Thus, we were again incapable of moving forward with Crylogger in our experiment.

Despite that we were unable to conduct extensive tests with Crylogger after many attempts, the whole process inspired us to not only try out different approaches to circumvent certain issues but also lead us to carry out deep analysis on the underlying reasons why those problems occurred and whether those are intrinsic weakness of dynamic crypto-detectors.

5.3 Static tool: Cryptoguard

When it comes to configure and test with Cryptoguard, the process becomes relatively light-weighted and smooth. This is partially because the essence of static crypto-detectors – it only needs to statically scan the source code of applications without execution. Thus,

there is no need to setup virtualization environment for emulators.

The environment setup process includes adjusting JDK and Python versions. More specifically, we downgraded Java version from Java 8 to Java 7 and upgraded gradle version from 4.4.1 to 6.5.1 (but not to the newest version because maven is depreciated in version 7). In additional, we do not have to check low-level architecture of either base machine or applications. As long as the source code is available, Cryptoguard could almost succeed to conduct the analysis on the target app.

Chapter 6

RC3: Implementation, data, and plots

This chapter, aiming to address RC3 – *extensive empirical tests with Cryptoguard on classified apps*, covers the procedure of automating crypto-misuse detection on 1003 Android applications and data collection, presents the results efficiently, and makes comparison between the results of Cryptoguard and those of Crylogger. We will start by discussing about how to automate the process of conducting tests with Cryptoguard on applications at a large scale and effectively collect their results. Then the reports of the results will be showed with graphs focusing on different perspectives, namely across both different categories and different cryptographic rules. Finally, we will compare the number of violations detected by Cryptoguard and Crylogger.

6.1 Implementation

In order to ensure the exhaustiveness and persuasiveness of our analysis, we conducted extensive tests on 1003 popular Android applications across 21 different categories (shown in Figure 6.1) downloaded from Google Play Store. The categories covers nearly every aspect of people’s daily life, ranging from finance and education to game, social media,

and traveling. For each category, we collected 40 applications on average with certain degree of variation and exception.

	counts
app_type	
bank	46
book	28
browser	59
calculator	50
cartoon	75
dictionary	57
education	58
fit	37
food	37
game	60
invitation	26
map	49
medical	118
music	35
music-player	28
photo	35
real estate	37
shopping	80
social-media	36
travel	37
weather	14
total	1003

Figure 6.1: Categories of Android applications downloaded

We packaged all of the applications to be detected together to automate the testing process. However, there are some occasional cases when Cryptoguard failed at testing certain apps and manual effort is required then. In comparison, the procedure to collect and pre-process the data is relatively more complicated. We used Python and Shell scripts to separate results of individual app from the package, filter the results to only keep the information of violated rules and ignore the unnecessary one, and export the results to excel/csv file for further analysis. While analyzing the collected data, we utilized several python packages such as Pandas, Numpy, and Matplotlib.

6.2 Data and plots

93.3% of 1003 applications have violated at least one cryptographic rule as shown from our results. Rule 9, 16, 1&2, and 7 are the 5 rules (highlighted in Figure 6.3) which are most likely to be violated – with possibility higher than 50%. More detailed analysis on these rules will be discussed in Chapter 7.

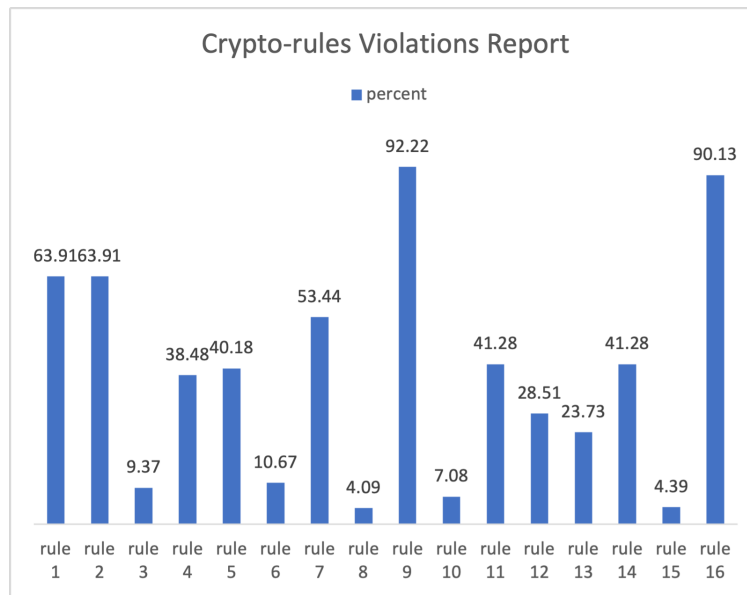


Figure 6.2: Cryptographic rules violated by Android applications tested using Cryptoguard

1	Predictable/constant cryptographic keys
2	Predictable/constant password for PBE
3	Predictable/constant password for KeyStore
4	Custom Hostname verifiers to accept all hosts
5	Custom TrustManager to trust all certificates
6	Custom SslSocketFactory w/o manual Hostname verification
7	Occasional use of HTTP
8	Predictable/constant PRNG seeds
9	Cryptographically insecure PRNGs (e.g., java.util.Random)
10	Static salts in PBE
11	ECB mode in symmetric ciphers
12	Static Ivs in CBC mode symmetric ciphers
13	Fewer than 1,000 iterations for PBE
14	64-bit block ciphers (e.g., DES, IDEA, Blowfish, RC4, RC2)
15	Insecure asymmetric ciphers (e.g., RSA, ECC)
16	Insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2)

Figure 6.3: Cryptographic rules examined [45]

We also inspect whether there are huge discrepancies or strong correlation among both

various categories of applications and different cryptographic rules. As shown in Figure 6.4, there is surprisingly high correlations among different categories of apps. Nearly all types of applications have similar potential to violate certain rules. This is somewhat counterintuitive while considering simple offline calculators and complex online video games are similar in probability of containing cryptographic misuses. It can be partially explained by that developers of video games are especially advertent about security issues including cryptography usage and they have sound project management system to supervise this. However, in contrast, calculator developing team may not pay enough attention on security issues. More explanations are needed to be studied further.

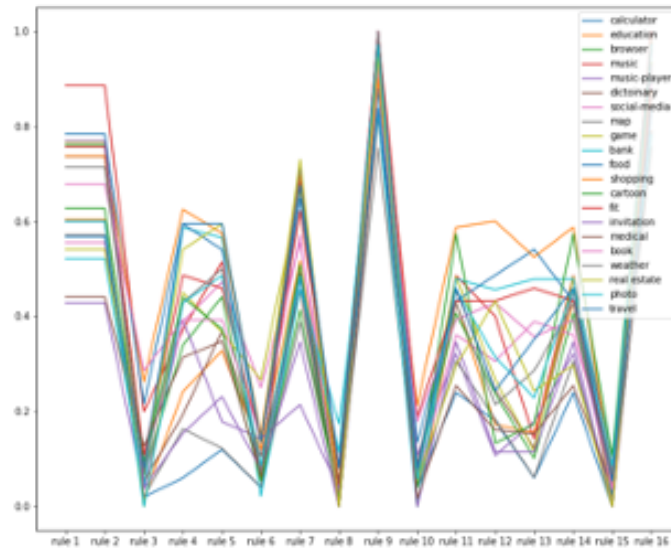


Figure 6.4: Crypto-misuses across different rules (extract categories)

The plots are more spread out in Figure 6.5 (the x-axis is a little blurry because of scaling but it does not matter in our analysis), which reveals that there is a huge variation of number of cryptographic rules violated across different categories. In other words, rules do not have a homogeneously regular trend over all types of applications. Generally speaking, the top two lines (i.e., rule 9 and rule 16) have particularly similar tendency, the same is for rule 12 and 13 (green and orange lines in the middle).

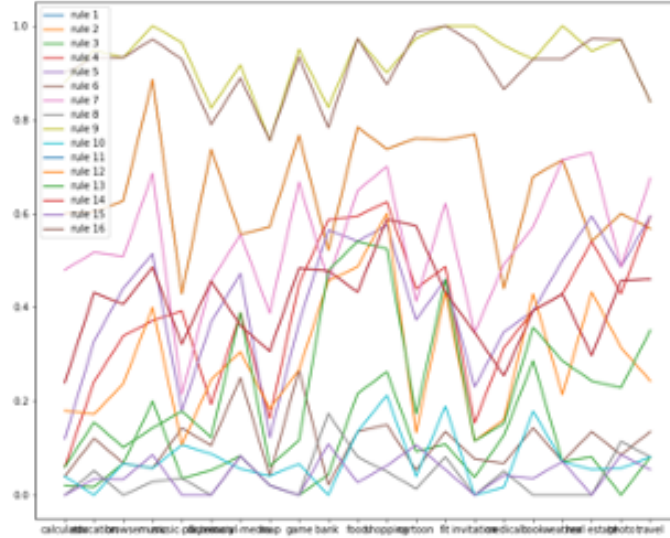


Figure 6.5: Crypto-misuses across different categories (extract rules)

6.3 Cryptoguard vs. Crylogger

The result of Cryptoguard has been collected and processed as described above and Crylogger’s result is obtained from the authors due to our technical issues discussed in Chapter 5. Although the applications have not been completely overlapped, both of them tested popular apps from a wide variety of categories from Google Play, so that we assume the results can be properly compared. Both of the resulting datasets have been put into the same graph in order to make direct comparison.

However, given the two datasets having different scale (i.e., distinct number of applications), we applied the formula $\frac{\text{Number of apps violating Rule } x}{\text{Total number of apps examined}}$ here to make them compatible while comparing. Normalization or standardization is not used here because we want the discrepancy to be more distinct and sharper rather than smoother and subtle.

Generally speaking, the results of Cryptoguard and Crylogger are highly similar (as shown in Figure 6.6). They have alike peaks and valleys with only relatively large divergence for a few rules, namely rule 4, 5, and 14.

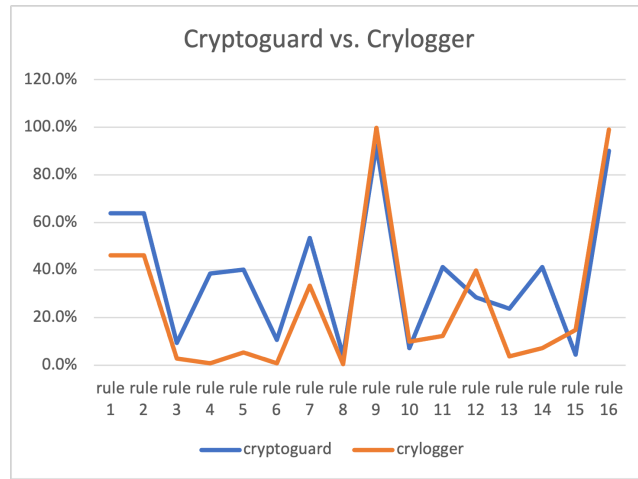


Figure 6.6: Comparison on the number of violations reported – Cryptoguard vs. Crylogger

Figure 6.7 reveals that Cryptoguard reports more violations than Crylogger. More specifically, there are significantly more violations caught by Cryptoguard than did Crylogger for rule 4, 14, 5, and 11; while more violations of rule 12 and 15 are caught by Crylogger. We think this result and its corresponding analysis (Chapter 8) are particularly crucial since they can lead to the exploration of the fundamental difference between static and dynamic crypto-detectors and their strength and weakness.

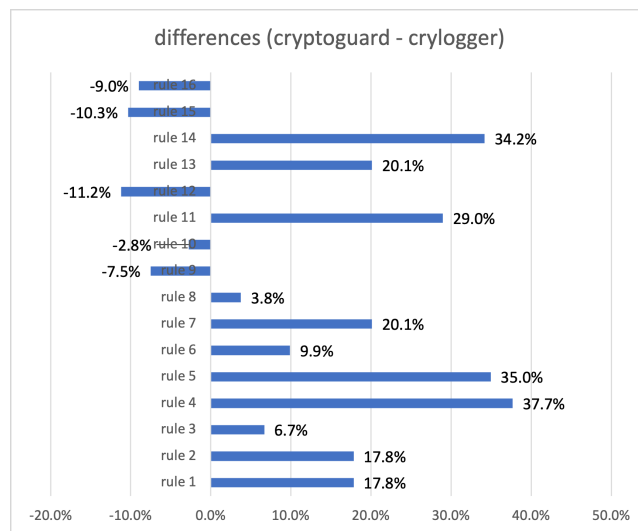


Figure 6.7: Difference Bar Chart (Cryptoguard - Crylogger)

Chapter 7

Evaluation

In this chapter, we further evaluate the results obtained in the previous section. We start off by having an in-depth analysis on the top four violations reported by Cryptoguard. Given that these four violations nearly overlap with the top four reported by Crylogger (based on Figure 6.6), they deserve to be rigorously studied in order to understand why they are especially susceptible to be breached within mobile apps and what procedure should be carried out to effectively prevent these violations and make the apps securer. In addition, we also introduce the outline of our four research questions (RQs) derived from the results by demonstrating their originality and corresponding methodologies.

7.1 Top four violations detected by Cryptoguard

The top four violations reported by Cryptoguard while detecting Android applications during our empirical experiment are i) use of predictable cryptographically insecure PRNGs, ii) use of insecure cryptographic hash, iii) use of predictable/constant cryptographic keys or password for PBE, and iv) occasional use of HTTP. These violations fall into four distinct perspectives of security, namely randomness, integrity, confidentiality, and SSL/TLS Man-in-the-middle Attack. Besides, they be closely inspected from their fundamental definition, historical evolution, security considerations and problems, detection methodology used by crypto-detectors, and proposed solutions.

7.1.1 Randomness: Cryptographically insecure PRNG

Pseudorandom number generator (PRNG for short) has been extensively used in mobile applications, which require numerous random numbers for common operations such as key or salts generation. The more entropy and the higher-level uniqueness these random number can achieve, the securer and more stable the applications are in terms of cryptographic communication. However, several significant vulnerabilities relating to weak random number generation have been found in widely used software. In 1996, the Netscape browser's SSL implementation was found to use fewer than a million possible seeds for its PRNG, which makes it highly vulnerable [26]. *Java.util.Random* is a commonly-used class for generating random numbers, while it is proven to be cryptographically insecure.

Rule 9 – use of cryptographically insecure PRNG – checks specifically the use of *Java.util.Random* in mobile apps. The technique used to catch the use of *Random* in Cryptoguard is locating the class, method, and then string orderly and checking if the string contains insecure PRNG such as *Java.util.Random: void <init>* and *Java.lang.Math: double random*. A code snippet is as follows. This procedure has certain limitations – it tends to result in false positives (i.e., redundant alerts which are not harmful). It will output violation reports as long as predefined untrusted PRNGs are found in the apk file, however, in some cases those algorithms are reasonable to be applied with respect to performance and on small-scale tokens and will not cause any insecure issues. Having 92.22% of applications violating it, this rule is most likely to be breached as shown in the result. This reveals that there is still substantial usage of *Random* class to generate random numbers and its underlying mechanism and insecurity need to be further exposed.

```
for (String prng : UNTRUSTED_PRNG){
    if (unit.toString().contains(prng)){
        analysis.add(unit);
    }
}
```

Random() creates a new random number generator when predefined seed can be passed in as parameter to generate different data types such as int, float, etc. Those generators output a stream of pseudorandom numbers. However, it implies that with the same seed and same sequence of method calls, it will output identical sequence of numbers. The class has a 48-bit internal state and thus will generate a 48-bit seed. It will repeat about 2^{48} calls and will not be able to produce all possible *longs* or *doubles*. Besides, it uses system clock to generate the seed and this feature could be taken advantage of by attackers, who knows the time at which the seed was generated, to predict the seed easily. In stark contrast, the class *Java.security.SecureRandom* has been proven to be cryptographically secure. It uses random data from people's os such as keystrokes and other data saved in */dev/random* or */dev/urandom* directory as seeds. Attackers can rarely collect these information unless they can fully get access to the target os. Besides, all output sequences are non-deterministic and, therefore, cryptographically strong. However, the operation of this class is noticeably slower than that of *Random* and this is one of the key reasons why *Random* has not lost its popularity in practice.

With partial knowledge, one possible solution has come out and been commonly adopted for a while. It suggests developers combine *Java.security.SecureRandom* and *Random* class by feeding a cryptographically secure seed generated by *SecureRandom* class to *Random* in order to guarantee the security of the pseudorandom number generation process. However, this does not completely solve the problem. In addition to predictable seeds, *Random* also employs linear congruential generator (LCG, for short), which has high predictability, to produce random numbers. In optimal cases, it is easy and fast for a modern computer (i.e., in a second) to predict future values within a full cycle. Therefore, the cryptography security of applications will still be compromised no matter how safely the seed is generated – either using other cryptographically secure procedures such as *Java.security.SecureRandom* or rolling a die randomly.

As a result, *Java.security.SecureRandom* should be publicized and used more in getting a cryptographically secure PRNG for security-sensitive applications. While *Java.util.Random*

does have merits, since it is cryptographically vulnerable, it should only be used to generate small tokens in few cases with experts' confirmation and close surveillance.

7.1.2 Integrity: Insecure cryptographic hash

In addition to randomness, data integrity of mobile apps is a crucial part of a secure system. However, it would be severely compromised while insecure cryptographic hash functions such as SHA1, MD5, MD4, and MD2 are used. A hash function is a function that takes an arbitrary amount of input and produces an output, which is known as message digest, of fixed size. The standard hash function serves as a basis for cryptographic hash functions. In recent years, MD5 and variants of SHA are commonly used cryptographic hash algorithms. With the help of these functions, users are able to generate message digests to detect and further prevent the unauthorized changes in files. This is especially important for critical system and sensitive databases [43].

Rule 16 – use of insecure cryptographic hash – checks if those insecure hash functions are used in the applications. It uses slicing strategies to scan the application in order to look for the usage of 6 predefined unsafe hash algorithms (i.e., BROKEN HASH) mainly in 3 cryptographic methods of *MessageDigest* class as shown below. However, since MD5 remains suitable for other non-cryptographic purposes such as determining the partition for a particular key in a partitioned database[43]. Alike cases could imply that the potential false positives could be generated by the straightforward method Cryptoguard used.

```
BROKEN_HASH = ("MD2", "MD5", "MD4", "SHA-1", "SHA1", "SHA")
criterial.setClassName("Java.security.MessageDigest");
criterial.setMethodName("Java.security.MessageDigest getInstance(Java.lang.String)");
criterial.setMethodName("Java.security.MessageDigest getInstance(Java.lang.String,
Java.lang.String)");
criterial.setMethodName("Java.security.MessageDigest getInstance(Java.lang.String,
Java.security.Provider)");
```

Message Digest 5 (MD5), together with its two predecessors (i.e., MD2 and MD4), are commonly described as cryptographically broken but are still widely used to produce 128-bit values. It is found to suffer from extensive vulnerabilities such as collision attacks. There are various types of collisions for MD5 [56]. One of the critical reasons for its insecurity is its speed. MD5 can be operated fast and thus, it only takes a small amount of time to be broken by attackers – simple MD5 collisions can now be found in seconds on a standard desktop. Compared to MD5, Secure Hash Algorithm 1 (SHA-1) emerges later and is a more complex algorithm and it can produce a longer hash value (i.e., 160-bit). It is a widely used NIST cryptographic hash function standard that was officially deprecated by NIST in 2011 due to fundamental security weakness demonstrated in various analyses and theoretical attacks. Despite its deprecation, SHA-1 remains widely used in 2017 for document and TLS certificate signatures and in many software such as Git version control system for integrity and backup purposes [55]. MD5 and SHA1 are susceptible to hash collision and pre-image attacks. Commercially available rainbow tables (shown in Figure 7.1) allow attackers to easily obtain pre-images of MD5 and SHA1 and get authenticated without knowing the actual plaintext. Those collisions enable attackers to forge digital signatures or break the integrity of messages.

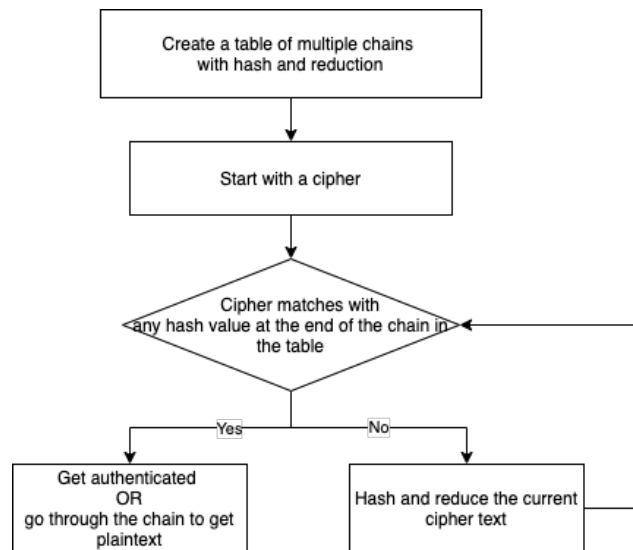


Figure 7.1: Rainbow table attack process

In spite of the vulnerabilities, we further analyzed why there are substantial reluctance to replace MD5 and SHA-1. Many industry practitioners are disinclined to give up using these two hash algorithms for other safer alternatives. The reasons could be grouped into three categories – outdated information, restricted usage, and operational constraints.

- **Outdated information** – Because of the history and popularity of SHA1 and MD5, many developers still consider them secure. They falsely assume there are no known techniques to find collisions and break those algorithms except via brute force [43]. Finding an actual collision seems to be extremely inefficient and impractical for the past years due to high complexity and computational cost of the task. However, they does not gain the cutting-edge information that SHA-1 collisions have finally become common with known instance of collisions [55]. Besides, some of them made an assumption that using mainstream protocols such as TLS, IKE, and SSH, which relies only on second pre-image resistance, are unaffected by collisions. However, they are ignorant of a new class of transcript collision attacks [8] which can obtain second pre-image of SHA1 and MD5.
- **Restricted usage** – Some developers are aware of the weakness of these algorithms and they have only used those functions in restricted cases assuming there will not be presence of active adversaries. For example, MD5 is used for the per-block checksums for Hadoop files systems’ consistency and setup. However, those kinds of premature excuses should not be made to put cryptographic security in jeopardy.
- **Operational constraints of hardware** – Take backward compatibility for clients as an example. Apache Tomcat server has to use MD5 in its digest authentication code because major browsers do not support secure hash functions. In addition, there is a lack of support for SHA2 on systems running Windows XP SP2 or older [43].

Therefore, safer alternatives such as SHA2 and SHA3 deserve more attention both from industry and academia. While SHA2 includes a significant number of changes from

its predecessor, SHA3 is internally different from the MD5-like structures of SHA1 and SHA2 [43]. It can be directly substitute for SHA2 in current applications to significantly improve the robustness of NIST's overall hash algorithm toolkit.

7.1.3 Confidentiality: Predictable/Constant cryptographic key and password for PBE

Above all, software with predictable or constant cryptographic keys and passwords are inherently insecure. Cryptoguard combines the testing of rule 1 and 2 because both of them examine the constancy perspective of the applications. There are other rules in Cryptoguard which also focus on confidentiality and predictability – static IVs in CBC mode symmetric cyphers (rule 12, 25.31%) and static salt in PBE (rule 10, 5.02%). These rules have relatively low violation rate than rule 1 and 2, which has 63.91%.

More specifically, these two rules focuses on two classes – *SecretKeySpec* and *PBEKeySpec*. It inspects if there are static strings or instances of other data types in five methods of these two classes (shown in the snippet). This strategy might overlook certain insecure cases since it is challenging to define and look for predictable keys. Compared to the two rules discussed above, rule 1 and rule 2 does not specify a particular insecure algorithm that should not be used, but, instead, it aims to prevent constant and predictable input provided by developers.

```
class: Javax.crypto.spec.SecretKeySpec
criteria1.setMethodName("void <init>(byte[], Java.lang.String)");
criteria1.setMethodName("void <init>(byte[], int, int, Java.lang.String)");

class: Javax.crypto.spec.PBEKeySpec
criteria1.setMethodName("void <init>(char[])");
criteria1.setMethodName("void <init>(char[], byte[], int, int)");
criteria1.setMethodName("void <init>(char[], byte[], int)");
```

The cryptographic functionality in Java is mainly provided by two libraries – Java Cryptographic Architecture (JCA) and Java Cryptography Extension (JCE). JCA is tightly integrated with the core Java API and delivers some primary cryptographic features; while JCE provides various advanced cryptographic operations [37]. *SecretKeySpec* and *PBEKeySpec*, which located in `Javax.crypto` package, are both provided by JCE. These two classes are commonly used cryptographic classes. However, there are wide uses of constant symmetric encryption key in these two classes while random key generation algorithm should be used instead. *SecretKeySpec* specifies a secret key in a provider-independent fashion. It converts raw secret keys in byte arrays to a `SecretKey` directly without having to go through a provider based `SecretKeyFactory` [39]. *PBEKeySpec* generates a cryptographic key from a user-chosen password with password-based encryption (PBE, for short) [40]. It stored passwords as char arrays rather than as string object.

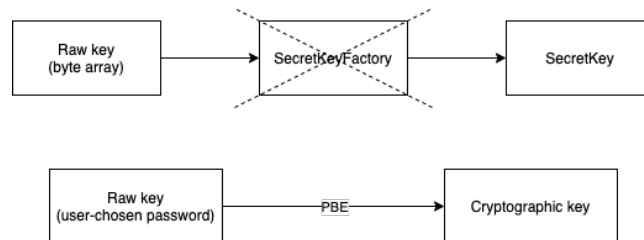


Figure 7.2: *SecretKeySpec* (upper) and *PBEKeySpec* (lower) mechanisms

One of the possible reasons for their high rate is that insecurity occurs more frequently in commonly known robust classes, which developers usually pay less attention to. Since these two classes are claimed to be secure by official website such as Oracle and by multiple software developers on widely used online forum such as StackOverFlow, practitioners tend to use less secure strategies (e.g., predictable/constant keys) within them.

In addition to cryptographic insecure scenarios, there are some other features of *PBEKeySpec* class which might cause some potential vulnerabilities – read incompatibility (i.e., unrepeatable read) and manual garbage collection. As described above, an instance of the class *PBEKeySpec* stores the raw key material from which an encryption mechanism can

derive a cryptographic key. Since the password is stored as a mutable char array, it can be easily overwritten when it is no longer needed. Problem occurs when two threads share an instance of *PBEKeySpec* and one of the threads tries to read the password while the other is clearing or modifying it. The thread that is trying to read the password may acquire an unexpected one. To avoid these types of complication, the password returned by *PBEKeySpec.getPassword()* method should match the one given to the constructor, otherwise, an exception (i.e., *IllegalStateException*) should be thrown as a warning. Besides, *PBEKeySpec* does not automatically clear the password after its usage [29]. It expects the developer to call *clearPassword()* after the object has fulfilled its purpose. It provides flexibility in certain cases but may lead to problems such as read incompatibility or memory leak when people omit to manually clear the garbage.

In short, random key generation algorithm, rather than static or predictable raw key material, should be used together with *SecretKeySpec* and *PBEKeySpec* in order to ensure the cryptographic confidentiality of mobile applications.

7.1.4 SSL/TLS MitM Attack: Occasional use of HTTP

The fourth most likely to be violated rule is rule 7 – occasional use of HTTP – which belongs to SSL/TLS Man-in-the-middle attack. HTTP and HTTPS are both protocols used while transferring information between server and browser. The information, either requests or responses, includes website content and API calls. The major distinction between them is that HTTPS secures communication over a computer network while HTTP is inherently insecure since it can lead to severe data leakage. After the initiation of HTTP in 1989 by Tim Berners-Lee, Google recommended sites to be switched to HTTPS in 2014, but only sites with e-commerce pages back then used HTTPS. In recent years, HTTPS is widely used. Cryptoguard searches for string 'http' in five methods from four classes to check whether HTTP has been used.

```

class: Java.net.URL
criteria1.setMethodName("void <init>(Java.lang.String)");
criteria2.setMethodName("void <init>(Java.lang.String, Java.lang.String,
Java.lang.String)");
criteria3.setMethodName("void <init>(Java.lang.String, Java.lang.String,
init, Java.lang.String)");

class: okhttp3.Request$Builder
criteria4.setMethodName("okhttp3.Request$Builder url(Java.lang.String)");

class: retrofit2.Retrofit$Builder
criteria5.setMethodName("retrofit2.Retrofit$Builder baseUrl(Java.lang.String)")

```

Hypertext Transfer Protocol (HTTP, for short) is an application layer protocol, which prescribes order and syntax for presenting information, in the Internet protocol suite model for distributed, collaborative, hypermedia information system. In short, it is used for transferring data over network. However, it is not encrypted and thus is vulnerable to MitM and eavesdropping attacks. Attackers can gain access to website accounts and sensitive information and even modify webpages to inject malware or advertisement easily with HTTP. To put it simply, it has priorities on trust among human other than security, which is strong discouraged from scientific perspective. As problems unremittingly emerging, HTTPS (i.e., Hypertext Transfer Protocol Secure) has been introduced. As a combination of HTTP and encryption, it extends HTTP by using TLS (SSL) to encrypt normal HTTP requests and responses. Thus, attackers will only get access to cypher texts rather than simply plain texts. Authentication of accessed website and protection of the privacy and integrity of the exchanged data while in transits are both guaranteed. This characteristics of HTTPS effectively protects against MitMA and the bidirectional encryption of communication between a client and server protects the communication against eavesdropping

and tampering.

We have summarized three potential reasons of continuing usage of HTTP – lack of knowledge, inadequate security checks, and performance consideration.

- **Risks of re-enabling and redirection** – HTTPS is set by default on Android versions greater than 8 and could be changed by setting *cleartextTrafficPermitted*. However, a high percentage of applications used the flag to re-enable HTTP [38] and some developers without enough expertise tend to follow the norm and end up using HTTP inadvertently. In addition, in most situations where HTTPS would have been possible, the hosts will redirect from HTTP to HTTPS serving the same content. However, the risk to leak information is not significantly reduced or eliminated by simply redirecting. Despite that, many developers seem to suffer from misconception and underestimation of the threat of MitMA in the presence of this type of redirections.
- **Inadequate security checks by official platforms** – Many authoritative platforms do not constrain developer’s use of HTTP. Android OS with version lower than 9 could allow developers using HTTP without custom workarounds. Not until 2016 did Google announced specific security safeguards and policies to prohibit insecure certification validation code. Although customization provides developers certain degree of flexibility, applications’ security might be harmfully impacted. Thus, the need for more restricted security surveillance should be enforced by official platforms.
- **Performance considerations** – Another reason why HTTPS is not widely adopted is about concerns relating to performance. Encryption increases the response time of two popular Web servers Netscape Enterprise Server 3.5.1 and Microsoft IIS 4.0 by at most 22%. Nevertheless, this additional delay imposed by encryption is considered to be moderate [25]. With typical PCs, encrypted Web communications using SSL and RC4 can transfer data at speed similar to non-encrypted HTTP. Websites are still encouraged to use HTTPS to secure communication.

7.2 Overview on research questions (RQs)

We aim to i) inspect if different categories of applications have similar cryptographic misuses and ii) formulate a taxonomy of cryptographic misuses with respect to which is tendentious to be detected by static and dynamic tools, respectively. Thus, we propose four research questions below:

- **RQ1:** *Are the results specific for different types of apps or consistent across different types?*
- **RQ2:** *Are there any rules which differ greatly in terms of number of violations between static and dynamic?*
- **RQ3:** *Which taxonomies do these rules belong to?*
- **RQ4:** *When are static and dynamic tools more efficient regarding the taxonomies?*

No existing studies, to the best of my knowledge, focus on examining whether there are discrepancies among different types of mobile apps in terms of their tendency to violate cryptographic API rules. However, this is critically important to consider because if the divergence does appear, it is necessary to inform developers of different industries to be alert of certain type of cryptographic misuses and to enforce more surveillance accordingly. Although various studies evaluate the performance and accuracy of static and dynamic crypto-detectors, none of them compares the efficiency of static and dynamic tools via looking at their capability of detecting the violations of different rules – instead, most of them focus on calculating various statistics such as the rate of false positives. We intend to design a schema to better combine static and dynamic analysis strategies to improve the efficiency of crypto-detectors through classifying cryptographic rules and recognizing the strength of static and dynamic analysis strategies.

To answer RQ1→RQ4, we analyzed the data obtained from our empirical experiments described in Chapter 6. We redesigned a new taxonomy based on the one utilized by Crylogger [42].

Chapter 8

Results and findings

8.1 RQ1: Crypto-misuses results among 21 different types of applications

As presented in Fig. 6.4, the number of crypto-misuses is surprisingly *consistent* across 21 different types of applications ranging from browsers to cartoon to book and real estate. They all have similar trend and global and local maximums and minimums. There are several different factors that could be attributable to this result –

1. Susceptibility to break cryptographic rules are evenly distributed across all categories of applications. Since cryptographic usages are extremely fundamental and common in developing mobile apps, they have alike tendency to violate certain rules no matter how complicated their functionalities are.
2. The propensity to violate different cryptographic rules does varies tremendously with respect to the levels of complexity of different types of apps, but the manpower employed during the development process is also significantly disproportional. In other words, the development and supervision team of more complicated applications are more structured, systematic, and rigorous compared to less complicated apps.

The analysis obtained is correlational rather than causal since it is unlikely to have

enough control variable in our experiment – the development team of the applications are not the same but may have some fundamental differences. There might be many other factors leading to this result, but we did not go further into this direction since as long as they are consistent as shown by the results, no dramatic change is necessary for security reasons to be imposed to status quo.

8.2 RQ2 - RQ4: New taxonomy for cryptographic rules

While there are substantial amount of studies on detection of cryptographic API misuses and diverse designs of crypto-detectors, the categories of cryptographic rule they used are pervasively consistent. Some common classifications are attack type (e.g., predictable secrets, SSL/TLS MitM, CPA, and brute force), crypto property (e.g., confidentiality, integrity, and randomness, etc.), severity (e.g., high, medium, low), and analysis method (e.g., forward slicing and backward slicing).

Unlike these categorization methods focusing on the content of the rules (i.e., security features the rules are based on), our new taxonomy is established on the approaches used to inspect the rules. In other words, we take the procedures used in detection process and the efficiency of static and dynamic analysis strategies into account while formulating this new taxonomy. We surmise this novel proposition of taxonomy may be used to contrive a schema to effectively combine static and dynamic tools to improve the accuracy and robustness of crypto-detection.

8.2.1 First stage: General taxonomy

According to how cryptographic rules are being inspected, we initially classified them into two general categories – insecure values and static values.

- **Insecure values** – Usage of known/proved broken algorithms or ciphers such as SHA1, MD5, and RSA or unsafe protocols such as HTTP. There are repository of

values that are claimed to be insecure, and the rules checking if any of them are used in applications are classified as insecure values.

- Methodology: Store all broken algorithms and values → get the relevant values used in the application → check if that value belongs to predefined insecure value repository – if so, report corresponding rule violation; otherwise, continue.

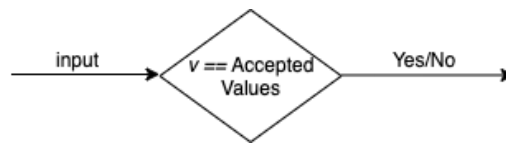


Figure 8.1: insecure value examination process

- **Repetitive values** – Usage of predictable or constant values in an application potentially compromise security, especially when they are used as keys, passwords, or IVs. These values are not predefined but are simultaneously stored when encountered during the detection.

- Methodology: store relevant value the first time it is encountered → when the same variable/API is met again, compare its value to the ones' that are stored orderly – if it is repetitive, report corresponding rule violation; otherwise, store that value to the repository

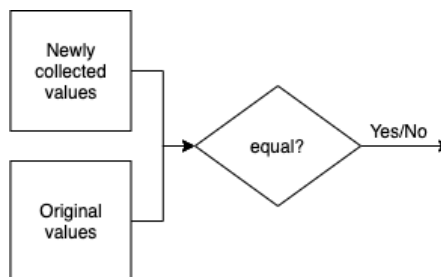


Figure 8.2: repetitive value examination process

8.2.2 Second stage: Crypto rule analysis

The logic of the methodology is the same for both static and dynamic crypto-detectors, but how they realize it is different. The distinct approaches they used lead to different results (i.e., number of reports). In this section, we focus on the rules whose number of violations diverge the most between Cryptoguard and Crylogger and the ones that are most susceptible to be violated. Generally speaking, the vulnerabilities reported by Cryptoguard outnumber those reported by Crylogger. This can be partially attributable to some specific techniques utilized by dynamic and static crypto-detectors discussed in Chapter 4. Admittedly, there might be false positives and false negatives in the results, but we did not prioritize these edge cases in our analysis.

We scrutinize how the following selected rules are examined by Cryptoguard and Crylogger, respectively. The analysis of rules falling into *insecure values* category is presented below.

- **Rule 9: Untrusted PRNG** – Use *Java.securityRandom* rather than *Java.util.Random*. Cryptoguard checks if there are strings belonging to predefined *UNTRUSTED_PRNG* while scanning the file. It aims to catch the insecure usage of *Random*. Crylogger checks if the value from logger satisfies *RandomGenerator.alg*=*'Secure'*. It aims to examine if the secure one has been used.
- **Rule 16: Broken Hash** – Do not use SHA1, MD5, MD4, and MD2. Cryptoguard checks if method *MessageDigest* uses predefined *Broken_Hash*. Crylogger checks if the value from the logger satisfies *MessageDigest.alg*!={*'SHA1'*, *'MD5'*, ...}. They both checks if insecure hash algorithms are used in the application.
- **Rule 4: Dummy Hostname Verifier** – Do not return true directly within *verify()*. This rule inspects if Hostname Verifier actually implement the verification process instead of accepting all host indiscriminately. Cryptoguard checks if *verify* method has used parameter *Java.net.ssl.SSLSession* to verify hosts. Crylogger passes some

erroneous values (i.e., NULL, empty strings) to check if the function is not implemented naively. This examination is relatively indirect by using test cases rather than get the actual parameter used.

- **Rule 14: Broken Symmetric Crypto Algorithms** – Do not use DES, IDEA, Blowfish, RC2, and RC4. Symmetric encryption algorithms has inherent hardship to properly transmit key used to encrypt and decrypt data between parties. Cryptoguard checks if predefined *BROKEN_CRYPTO* is used. Crylogger checks if the value from logger satisfies *SymmEncryption!={‘DES’, ‘IDEA’, ...}*. They both checks if symmetric encryption algorithms are used in the application.
- **Rule 5: Dummy Cert. Validation** – Verify host names and certificates properly (i.e., do not trust all of them). The purpose of this rule is similar to that of Rule 4. Cryptoguard checks if exception, expiration, and valid list of certificates are scrutinized appropriately. It simplifies the tasks by examining some concrete cases. Crylogger checks if the value from logger satisfies *SSL/TLS/Cert.allcert=False*.
- **Rule 11: ECB Mode for Symmetric Crypto Encryption** – Do not use operation mode ECB with AES. ECB mode encryption should be avoided since it can leak information about plaintext – same ciphertext will be acquired if same plaintext is put into ECB mode encryption. Cryptoguard checks if method uses predefined *BROKEN_CRYPTO*. Crylogger checks if the value from logger satisfies *SymmEncryption.mode!=‘ECB’* or *SymmEncryption.#block=1*. They both checks if ECB mode is used in the application.

The analysis of rules falling into *repetitive values* category is presented below.

- **Rule 12: Static IVs** – Do not use constant Initial Vectors in symmetric encryption. Cryptoguard uses hash maps to keep track of all constant sources in *Java.crypto.spec.IvParameterSpec*. Crylogger compares the value collected in first log with that in the second log – $\{SymmEncryption.IV\}^1 \cap \{SymmEncryption.IV\}^2$.

They both store the previous values and check if the current value has already been stored before.

While Rule 4, 5, 9, 11, 14, 16 all belong to *insecure values* category, only Rule 12 belongs to *repetitive values* category. This indicates that discrepancy between results of static and dynamic analysis strategies occurs within *insecure values* category more often. In addition, dynamic crypto-detectors, overall, tend to detect more rules examining constant values while static tools are better at examining broken values.

8.2.3 Third stage: Refined taxonomy

As shown in the above section, static and dynamic analysis shows distinct efficiency within one category – while all 6 rules belong to *insecure value* category, Cryptoguard detects more violations of rule 4, 5, 11, 14 and Crylogger detects more violations of rule 9 and 16. More detailed classification is needed. We further refined the existing *insecure value* category by dividing it into fore-encryption group and encryption and communication group.

- **Fore-encryption** – Rule 9 and 16. This category focuses on examining value and algorithm used before the actual encryption. Crylogger detected approximate 9% more violations than Cryptoguard did.
- **Encryption and communication** – Rule 4, 5, 11, 14. This category concentrates on inspecting functions and algorithms used during the encryption and message transmission process. Cryptoguard detected around 34% more violations than Crylogger did.

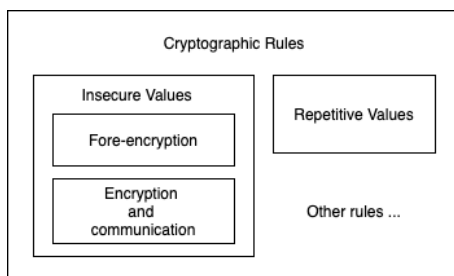


Figure 8.3: New taxonomy of crypto rules

Our new taxonomy is presented above in Figure 8.3. Overall, dynamic analysis is more efficient in detecting rules that could be checked explicitly. It can directly log the relevant information such as *RandomGenerator.alg* and *MessageDigest.alg* to check whether broken values or strings are used in the application. The rules belonging to *fore-encryption* category can be examined more straightforwardly. In contrast, it is relatively challenging to directly log the values required to be checked during the process of encryption and message transmission. Take rule 4 as an example. While static tools can easily scan the code and get the parameters of certain functions they are inspecting, the relevant information for dynamic crypto-detectors is largely limited – its only option is to use test cases (i.e., input certain value) to test if the output is as expected. However, this circuitous approach may miss many insecure cases and constrain its efficiency.

Chapter 9

Limitation and Future works

While we have reduced the gap in studies of cryptographic misuses in the field of mobile application security, there are certain limitations of our research and large space for future work to make further improvement. In this chapter, we focus on discussing our current limitations and pointing out potential related research topics.

9.1 Limitation

Hardware constraints of dynamic crypto detectors in experiment. Due to restricted hardware accessibility during our empirical experiment, we were not capable of testing Android applications with Crylogger virtually, instead, we chose to use the data from the author [42] as a workaround and compared that with the results of Cryptoguard from our experiment. Although this inspires in-depth analysis on the underlying reasons for the implementation issues, it would potentially cause the comparison to be deviating from the ground truth – the applications tested by Crylogger and Cryptoguard can not be guaranteed to be exactly the same. In order to lower this side effect, we attempted to maximize the overlap between Android applications used in our experiment and those used by Crylogger [42] by downloading the most popular applications and across a wide variety from Google Play.

Less emphasis on false positives. One of the most critical contributions of our research is the design of a new taxonomy of cryptographic rules. Its uniqueness comes from our

emphasis on analyzing the divergence of number of violations of each rule reported by static and dynamic crypto detectors rather than the mere comparison on their statistics such as false positives. However, it introduces bias meanwhile – without checking false positives manually, the number of violations reported might not be as accurate as expected.

9.2 Directions of future works

9.2.1 Integration of static and dynamic analysis strategies

This is the most natural succeeding direction of future work leads by our research – combination of static and dynamic strategies in cryptographic misuses detection. The new taxonomy indicates which categories of crypto rules could be more efficiently examined by dynamic analysis strategies and static analysis strategies. Thus, instead of the design schema used by some previous works (i.e., combining them by using dynamic techniques to confirm the results of static techniques sequentially), the new strategies should effectively partition the rules so that the crypto detector checks some rules by static techniques and others by dynamic techniques. In this way, the efficiency and resources could be substantially boosted and saved, respectively, without compromising the accuracy, which, instead, might be improved as well.

9.2.2 Automatically repair of the misuses

In addition to the detection of cryptographic API misuses, the automatic reparation may be equally attractive to software developers, practitioners and, especially, common users without much expertise. One of the potential techniques to automatically repair the vulnerabilities found focuses on modifying lower-level bytecode. The logic is that, in templates of misuses, they first locate where the actual variables or registers should be, allocate placeholders accordingly, and finally fix code by replacing placeholders with the mapped variables or registers [34]. While this is feasible under some circumstances, it cannot overhaul the entire system. There are certain misuses that cannot be corrected by simple

replacement. For example, via merely replacing HTTP with HTTPS, the link would become unreachable or invalid. Overall, this complex topic is potentially meaningful and deserves more attention from both academia and industry.

9.2.3 Machine learning and deep learning in vulnerability detection

While machine learning (ML) are gaining popularity increasingly over the recent decades and they have been pervasively applied in the detection of mobile application malware, the application of ML models in the specific field – cryptographic misuse detection – is largely limited. They are critically important because of its ability to detect new or unknown malware that are infected by various Trojans, worms, and spyware [51]. There are about 38 ML models ranging from supervised learning to reinforcement learning such as support-vector machine, random forest, and neural network [49] and 7 general stages in its life cycle including problem definition, data acquisition, feature selection, model selection, training, and evaluation, and so on. ML models can improve automatically through experience and usage of data. Its unique features greatly improve the prospects for misuse detection of Android application and should be specialized into catching cryptographic misuses in the near future.

Chapter 10

Conclusion

In this thesis, we have studied static and dynamic analysis techniques used in detecting cryptographic-API misuse in Android mobile applications through extensive literature review, systematic empirical study, and in-depth analysis and comparison. We found that 93.3% of applications contains at least one cryptographic misuses, and the results are generally consistent across different types of mobile apps. We summarized strength and weakness of static and dynamic crypto detectors used in public. In particular, we looked into some challenges confronted by these two strategies – static analysis: i) how native code are detected and ii) how DEX bytecode is transformed into Java source code; dynamic analysis: i) how execution traces are generated during execution and ii) limitations of automatic input generation. These topics are critical causes of inaccuracies/defects of each analysis strategy, respectively.

During the course of our experiment with Crylogger, we faced substantial hardware constraints and, thus, identified and analyzed underlying limitations of dynamic crypto detectors such as nested virtualization and ABI incompatibility, etc., that caused the issues. We concluded that static crypto detectors outperforms dynamic ones regarding to their robustness in implementation. Also, we closely examined four top crypto rules that are most frequently violated according to the report of static crypto detector Cryptoguard. Surprisingly, they all belong to separate crypto properties (i.e., randomness, integrity,

confidentiality, etc.). This further confirms the need to create a new taxonomy according to their performance in reality.

In order to bridge the research gap, we found regularity in the results and posited a new taxonomy schema of cryptographic rules based on how they are examined by analysis tools in practice instead of commonly used schemas such as attack type and severity level. This is an especially meaningful blueprint on how to efficiently combine static and dynamic analysis in crypto detection. Instead of using dynamic strategies to confirm the results of static ones sequentially, the new combination techniques will substantially boost the efficiency, soundness, and accuracy of the detection process. In addition, this study will be continued. Aspects of future work such as integration, automatic reparation, and application of machine learning algorithms are increasingly expanding and promising.

Bibliography

- [1] YASEMIN ACAR, MICHAEL BACKES, SVEN BUGIEL, SASCHA FAHL, PATRICK MCDANIEL, AND MATTHEW SMITH. Sok: Lessons learned from android security research for appified software platforms. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 433–451, 2016.
- [2] SHARMIN AFROSE, SAZZADUR RAHAMAN, AND DANFENG YAO. Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61, 2019.
- [3] AMIT AMI, NATHAN COOPER, KAUSHAL KAFLE, KEVIN MORAN, DENYS POSHYVANYK, AND ADWAIT NADKARNI. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques, 07 2021.
- [4] ANONYMOUS. How android os became the world’s most popular?
- [5] YAUHEN LEANIDAVICH ARNATOVICH, LIPO WANG, NGOC MINH NGO, AND CHARLIE SOH. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. *IEEE Access*, 6:12382–12394, 2018.
- [6] STEVEN ARZT, SIEGFRIED RASTHOFER, CHRISTIAN FRITZ, ERIC BODDEN, ALEXANDRE BARTEL, JACQUES KLEIN, YVES LE TRAON, DAMIEN OCTEAU, AND PATRICK 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*, pages 259–269, New York, NY, USA, 2014. Association for Computing Machinery.
- [7] NATHANIEL AYEWAH, WILLIAM PUGH, DAVID HOVEMEYER, J. DAVID MORGENTHALER, AND JOHN PENIX. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
- [8] KARTHIKEYAN BHARGAVAN AND GAETAN LEURENT. Transcript collision attacks: Breaking authentication in tls, ike, and ssh, 01 2016.
- [9] ALEXANDRE BRAGA, RICARDO DAHAB, NUNO ANTUNES, NUNO LARANJEIRO, AND MARCO VIEIRA. Understanding how to use static analysis tools for detecting

- cryptography misuse in software. *IEEE Transactions on Reliability*, 68(4):1384–1403, 2019.
- [10] ALEXIA CHATZIKONSTANTINOY, CHRISTOFOROS NTANTOGIAN, GEORGIOS KAROPOULOS, AND CHRISTOS XENAKIS. Evaluation of cryptography usage in android applications. In *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*, BICT'15, pages 83–90, Brussels, BEL, 2016. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).
 - [11] ANDROID COMMUNITY. Android abis.
 - [12] ANDROID COMMUNITY. Aosp requirements.
 - [13] ANDROID COMMUNITY. Known issues with android studio and android gradle plugin.
 - [14] DAVID CURRY. Android statistics (2021), June 2021.
 - [15] GITHUB DEVELOPERS. dex2jar.
 - [16] GITHUB DEVELOPERS. jadx - dex to java decompiler.
 - [17] GOOGLE DEVELOPERS. Android platform architecture.
 - [18] ONLINE DEVELOPERS. Jd project.
 - [19] ONLINE DEVELOPERS. Tracerjd: A framework for generic trace-based dynamic dependence analysis with fine-grained logging.
 - [20] MANUEL EGELE, DAVID BRUMLEY, YANICK FRATANTONIO, AND CHRISTOPHER KRUEGEL. An empirical study of cryptographic misuse in android applications, 11 2013.
 - [21] WILLIAM ENCK, PETER GILBERT, SEUNGYEOP HAN, VASANT TENDULKAR, BYUNG-GON CHUN, LANDON P. COX, JAEYEON JUNG, PATRICK MCDANIEL, AND ANMOL N. SHETH. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 32(2), June 2014.
 - [22] SASCHA FAHL, MARIAN HARBACH, THOMAS MUDERS, LARS BAUMGÄRTNER, BERND FREISLEBEN, AND MATTHEW SMITH. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12*, pages 50–61, New York, NY, USA, 2012. Association for Computing Machinery.
 - [23] FELIX FISCHER, KONSTANTIN BÖTTINGER, HUANG XIAO, CHRISTIAN STRANSKY, YASEMIN ACAR, MICHAEL BACKES, AND SASCHA FAHL. Stack overflow considered harmful? the impact of copy amp;paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, 2017.

- [24] FRANÇOIS GAGNON, MARC-ANTOINE FERLAND, MARC-ANTOINE FORTIER, SIMON DESLOGES, JONATHAN OUELLET, AND CATHERINE BOILEAU. Androssl: A platform to test android applications connection security, 10 2015.
- [25] ARTHUR GOLDBERG, ROBERT BUFF, AND ANDREW SCHMITT. A comparison of http and https performance.
- [26] NADIA HENINGER, ZAKIR DURUMERIC, ERIC WUSTROW, AND J. ALEX HALDERMAN. Mining your ps and qs: Detection of widespread weak keys in network devices. In *Proceedings of the 21st USENIX Conference on Security Symposium, Security'12*, page 35, USA, 2012. USENIX Association.
- [27] BRITTANY JOHNSON, YOONKI SONG, EMERSON MURPHY-HILL, AND ROBERT BOWDIDGE. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013.
- [28] ZELIANG KAN, HAOYU WANG, LEI WU, YAO GUO, AND GUOAI XU. Deobfuscating android native binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 322–323, 2019.
- [29] STEFAN KRÜGER, SARAH NADI, MICHAEL REIF, KARIM ALI, MIRA MEZINI, ERIC BODDEN, FLORIAN GÖPFERT, FELIX GÜNTHER, CHRISTIAN WEINERT, DANIEL DEMMLER, AND RAM KAMATH. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936, 2017.
- [30] STEFAN KRÜGER, JOHANNES SPÄTH, KARIM ALI, ERIC BODDEN, AND MIRA MEZINI. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.
- [31] DAVID LAZAR, HAOGANG CHEN, XI WANG, AND NICKOLAI ZELDOVICH. Why does cryptographic software fail?: a case study and open problems. In *APSys*, 2014.
- [32] JUANRU LI, ZHIQIANG LIN, JUAN CABALLERO, YUANYUAN ZHANG, AND DAWU GU. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 412–425, New York, NY, USA, 2018. Association for Computing Machinery.
- [33] YONG LI, YUANYUAN ZHANG, JUANRU LI, AND DAWU GU. icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In *Network and System Security*, Man Ho Au, Barbara Carminati, and C.-C. Jay Kuo, editors, pages 349–362, Cham, 2014. Springer International Publishing.
- [34] SIQI MA, DAVID LO, TENG LI, AND ROBERT H. DENG. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM*

- on Asia Conference on Computer and Communications Security, ASIA CCS '16*, pages 711–722, New York, NY, USA, 2016. Association for Computing Machinery.
- [35] ARAVIND MACHIRY, ROHAN TAHILIANI, AND MAYUR NAIK. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 224–234, New York, NY, USA, 2013. Association for Computing Machinery.
 - [36] TREND MICRO. A look at google bouncer.
 - [37] ZMIEŃ NA JĘZYK POLSKI. Cryptography in java, March 2020.
 - [38] MARTEN OLTROGGE, NICOLAS HUAMAN, SABRINA AMFT, YASEMIN ACAR, MICHAEL BACKES, AND SASCHA FAHL. Why eve and mallory still love android: Revisiting TLS (in)security in android applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4347–4364. USENIX Association, August 2021.
 - [39] ORACLE. Class secretkeyspec.
 - [40] ORACLE. <https://docs.oracle.com/javase/8/docs/api/javax/crypto/spec/pbekeyspec.html>.
 - [41] ZINAL D. PATEL. Malware detection in android operating system. In *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pages 366–370, 2018.
 - [42] LUCA PICCOLBONI, GIUSEPPE DI GUGLIELMO, LUCA P. CARLONI, AND SIMHA SETHUMADHAVAN. Crylogger: Detecting crypto misuses dynamically, July 2020.
 - [43] PRASHANT P. PITTALIA. A comparative study of hash algorithms in cryptography, 2019.
 - [44] SEBASTIAN POEPLAU, YANICK FRATANTONIO, ANTONIO BIANCHI, CHRISTOPHER KRUEGEL, AND GIOVANNI VIGNA. Execute this! analyzing unsafe and malicious dynamic code loading in android applications, 01 2014.
 - [45] SAZZADUR RAHAMAN, YA XIAO, SHARMIN AFROSE, FAHAD SHAON, KE TIAN, MILES FRANTZ, MURAT KANTARCIUGLU, AND DANFENG (DAPHNE) YAO. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery.
 - [46] SAZZADUR RAHAMAN, YA XIAO, SHARMIN AFROSE, KE TIAN, MILES FRANTZ, NA MENG, BARTON P. MILLER, FAHAD SHAON, MURAT KANTARCIUGLU, AND DANFENG (DAPHNE) YAO. Poster: Deployment-quality and accessible solutions for cryptography code development. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 2545–2547, New York, NY, USA, 2019. Association for Computing Machinery.

- [47] BRADLEY REAVES, JASMINE BOWERS, SIGMUND ALBERT GORSKI III, OLABODE ANISE, RAHUL BOBHATE, RAYMOND CHO, HIRANAVA DAS, SHARIQUE HUSSAIN, HAMZA KARACHIWALA, NOLEN SCAIFE, BYRON WRIGHT, KEVIN BUTLER, WILLIAM ENCK, AND PATRICK TRAYNOR. *droid: Assessment and evaluation of android application analysis tools. *ACM Comput. Surv.*, 49(3), October 2016.
- [48] RAIMONDAS SASNAUSKAS AND JOHN REGEHR. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. Association for Computing Machinery.
- [49] JANAKA SENANAYAKE, HARSHA KALUTARAGE, AND MHD OMAR AL-KADRI. Android mobile malware detection using machine learning: A systematic review. *Electronics*, 10(13), 2021.
- [50] SHAO SHUAI, DONG GUOWEI, GUO TAO, YANG TIANCHANG, AND SHI CHENJIE. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.
- [51] HRITIK SONI, PRANJAL ARORA, AND D RAJESWARI. Malicious application detection in android using machine learning. In *2020 International Conference on Communication and Signal Processing (ICCSP)*, pages 0846–0848, 2020.
- [52] DAVID SOUNTHIRARAJ, JUSTIN SAHS, GARRETT GREENWOOD, ZHIQIANG LIN, AND LATIFUR KHAN. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps, 01 2014.
- [53] STACKOVERFLOW DEVELOPERS. decompiling dex into java sourcecode.
- [54] STATISTA. Number of available applications in the google play store from december 2009 to february 2016, September 2021.
- [55] MARC STEVENS, ELIE BURSZTEIN, PIERRE KARPMAN, ANGE ALBERTINI, AND YARIK MARKOV. The first collision for full sha-1. In *Advances in Cryptology – CRYPTO 2017*, Jonathan Katz and Hovav Shacham, editors, pages 570–596, Cham, 2017. Springer International Publishing.
- [56] MARC STEVENS, ARJEN LENSTRA, AND BENNE DE WEGER. Chosen-prefix collisions for md5 and colliding x.509 certificates for different identities. In *Advances in Cryptology - EUROCRYPT 2007*, Moni Naor, editor, pages 1–22, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [57] JOHN R. VACCA. Scene of the cybercrime (second edition), 2008.
- [58] NICOLAS VIENNOT, EDWARD GARCIA, AND JASON NIEH. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, June 2014.

- [59] ANDROID OFFICIAL WEBSITE. Ui/application exerciser monkey.
- [60] XIA ZENG, DENG FENG LI, WU JIE ZHENG, FAN XIA, YU ETANG DENG, WING LAM, WEI YANG, AND TAO XIE. Automated test input generation for android: Are we really there yet in an industrial case? In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 987–992, New York, NY, USA, 2016. Association for Computing Machinery.