

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Beyond Android Crypto API Misuse report: A study on Origin and Reachability

Lucas Sousa Borges

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2025



Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Beyond Android Crypto API Misuse report: A study on Origin and Reachability

Lucas Sousa Borges

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Daniel de Paula Porto
CIC/UnB

Pedro Henrique Teixeira Costa
CIC-UnB

Prof. Dr. Guilherme N. Ramos
Coordenador do Curso de Engenharia da Computação

Brasília, 24 de Junho de 2025

Dedicatória

Dedico esse trabalho às pessoas que em algum momento da minha graduação chegaram a desejar o meu sucesso mais do que eu mesmo. Em especial, dedico esse trabalho aos meus pais Paulo e Marinely, que me ensinaram, principalmente com o exemplo, a fazer tudo com ética e dignidade, como homem, como cristão, como ser humano.

Agradecimentos

Agradeço primeiramente ao meu professor orientador Rodrigo Bonifácio, que me conduziu por esse projeto sempre com seriedade, paciência e interesse, e ao colega de pesquisa Luis Amaral, que me ensinou as bases da pesquisa.

Agradeço aos demais parentes que sempre se fizeram presentes durante minha graduação.

Ao longo dessa jornada fiz diversas amizades que tenho o orgulho de dizer que mantenho até hoje, e que foram essenciais para que eu tivesse forças de chegar até o fim. Sem querer me arriscar a deixar algum de fora, não me comprometerei a citar nomes, mas espero que saibam que são muito importantes pra mim.

Por fim, agradeço a você, caro leitor, que se dispôs a prestigiar este trabalho.

Resumo

Este trabalho apresenta uma pesquisa conduzida no intuito de investigar o uso incorreto de funções criptográficas em aplicativos Android, um fator crucial para a proteção de dados sensíveis. Propõe-se investigar a prevalência dessas falhas, determinar a proporção de vulnerabilidades que se originam de bibliotecas de terceiros (TPLs) e verificar quantas das falhas relatadas estão em código inalcançável, sugerindo que podem ser falsos positivos. O mérito desta proposta reside em fornecer informações úteis para os desenvolvedores de ferramentas de análise, destacando limitações relacionadas à escalabilidade, atribuição de origem e redução de falsos positivos, e para desenvolvedores de software, oferecendo orientação prática na interpretação de resultados de análise e priorização de esforços de correção. A proposta foi avaliada através de três estudos: o primeiro investigou a prevalência de usos indevidos de API criptográfica em aplicativos populares do Google Play usando CogniCrypt e CryptoGuard ; o segundo explorou heurísticas para diferenciar vulnerabilidades de código de aplicativo e TPL ; e o terceiro investigou se os usos indevidos de API criptográfica são alcançáveis a partir do código do aplicativo. Os resultados indicam que uma parcela significativa das vulnerabilidades relatadas se origina de TPLs, e que uma grande proporção dessas vulnerabilidades em TPLs é inalcançável, sugerindo que podem ser falsos positivos. A pesquisa foi aprovada pelo professor orientador e o artigo se encontra pronto para submissão em conferência adequada. Melhorias futuras podem incluir a utilização de ferramentas mais recentes para detecção de TPLs e alcançabilidade, além da incorporação dessas ferramentas às ferramentas de análise estática.

Palavras-chave: Android, Vulnerabilidades criptográficas, Bibliotecas externas, Alcançabilidade

Abstract

This paper presents a research conducted to investigate the incorrect use of cryptographic functions in Android applications, a crucial factor for protecting sensitive data. The study proposes to investigate the prevalence of these flaws, determine the proportion of vulnerabilities originating from third-party libraries (TPLs), and verify how many of the reported flaws reside in unreachable code, suggesting they might be false positives. The merit of this proposal lies in providing useful information for analysis tool developers, highlighting limitations related to scalability, origin attribution, and false positive reduction, and for software developers, offering practical guidance in interpreting analysis results and prioritizing remediation efforts. The proposal was evaluated through three studies: the first investigated the prevalence of cryptographic API misuses in popular Google Play applications using CogniCrypt and CryptoGuard; the second explored heuristics to differentiate vulnerabilities in application code from those in TPLs; and the third investigated whether cryptographic API misuses are reachable from the application code. The results indicate that a significant portion of the reported vulnerabilities originate from TPLs, and that a large proportion of these vulnerabilities within TPLs are unreachable, suggesting they may be false positives. The research was approved by the advising professor, and the paper is ready for submission to an appropriate conference. Future improvements could include the use of more recent tools for TPL detection and reachability analysis, as well as the incorporation of these tools into static analysis tools.

Keywords: Android, crypto API misuse, Third Party Libraries, Reachability.

Sumário

1 Introdução **1**

2 Artigo: "Beyond Android Crypto API Misuse report: A study on Origin and Reachability" **3**

2.1 Abstract . . . 3
2.2 Introduction . . . 4
2.3 Background . . . 6
2.4 Study Design . . . 8
2.5 Preliminary Study: Android App Developers' Perception . . . 10
2.5.1 Settings . . . 10
2.5.2 Results . . . 10
2.6 Study 1: Prevalence (RQ1) . . . 13
2.6.1 Settings . . . 13
2.6.2 Results . . . 13
2.7 Study 2: Third-party Libraries (RQ2) . . . 15
2.7.1 Settings . . . 15
2.7.2 Results . . . 15
2.8 Study 3: Unreachable Code (RQ3) . . . 18
2.8.1 Settings . . . 18
2.8.2 Results . . . 19
2.9 Discussion . . . 21
2.9.1 Implications of the Results . . . 21
2.9.2 Threats to Validity . . . 22
2.10 Related Work . . . 23
2.11 Final Remarks . . . 26

3 Conclusões **27**

Referências **28**

Lista de Figuras

2.1 Overall research workflow. 9
2.2 Example of a secret gist sent to an Android project. 11

Lista de Tabelas

2.1 Example of Cognicrypt's Output for app Hola VPN. 13
2.2 Example of part of CryptoGuard's Output for app Hola VPN. 13
2.3 Segment of LibScout's Output for app Hola VPN 15
2.4 Most frequently reported cryptographic API misuse methods from third-party libraries, as identified by CogniCrypt. The *Total* column indicates the number of distinct apps in which each class/method pair was flagged for a cryptographic API misuse. 17

Capítulo 1

Introdução

Este documento apresenta o artigo resultante das atividades desenvolvidas no âmbito do Trabalho de Conclusão de Curso (TCC) em Engenharia de Computação. O autor foi responsável por uma parte significativa do estudo que fundamentou os métodos e resultados apresentados no artigo, com especial atenção à análise de Bibliotecas de Terceiros (TPLs), à compilação dos dados e à redação geral do manuscrito. Este trabalho é resultado da colaboração entre os membros de um grupo de pesquisa do Departamento de Ciência da Computação da Universidade de Brasília.

Diversas ferramentas de análise estática e dinâmica, como CogniCrypt e CryptoGuard [1, 2], foram desenvolvidas para identificar o uso incorreto de bibliotecas criptográficas. Essas ferramentas identificam o uso incorreto de bibliotecas criptográficas, e utilizam técnicas avançadas de análise de programa para detectar padrões de mau uso de APIs criptográficas ou desvios de regras de uso correto. Embora estudos prévios [2–4] tenham evidenciado a eficácia dessas ferramentas na conscientização sobre segurança e na confiança na qualidade do software, revelando a presença de pelo menos um mau uso de API criptográfica em uma alta porcentagem de aplicativos Android analisados, eles também apontaram que uma parte significativa dessas vulnerabilidades reportadas ocorre em métodos pertencentes a TPLs.

Apesar do reconhecimento de que muitos maus usos de APIs criptográficas se originam de TPLs, pesquisas anteriores não adotaram uma abordagem sistemática para diferenciar entre o código da aplicação e o código das bibliotecas. Mais importante, não examinaram se as vulnerabilidades reportadas em bibliotecas são realmente alcançáveis a partir do código da aplicação. De fato, um estudo preliminar conduzido com desenvolvedores de aplicativos Android sugeriu o oposto, com alguns participantes expressando ceticismo quanto à relevância das questões reportadas, especialmente quando se referiam a bibliotecas internas ou métodos não diretamente invocados pelo aplicativo. Essa percepção dos desenvolvedores, de que as advertências podem se originar de TPLs e residir em partes

inalcançáveis do código, motivou a presente investigação.

Este trabalho apresenta os resultados de três estudos distintos. O primeiro investiga a prevalência de usos indevidos de API criptográfica em aplicativos Android populares, contrastando com achados anteriores na literatura ao observar falhas nas ferramentas de análise para completar a avaliação de uma porcentagem considerável de aplicativos. A segunda parte explora duas heurísticas para determinar se um mau uso de API criptográfica se origina do código da aplicação ou de TPLs: a primeira emprega um detector de TPLs Android (LibScout), e a segunda examina a frequência com que o mesmo mau uso aparece em diferentes aplicativos. Por fim, o terceiro estudo investiga se os usos indevidos de API criptográfica originados de TPLs são realmente alcançáveis a partir do código da aplicação.

Os resultados demonstram que, mesmo com as limitações de escalabilidade das ferramentas de análise, há uma presença generalizada de usos indevidos de API criptográfica nos aplicativos analisados. Ao combinar as heurísticas propostas, uma proporção substancial das vulnerabilidades reportadas é atribuída a bibliotecas de terceiros. Adicionalmente, uma grande parte dos maus usos de API criptográfica provenientes de TPLs foi considerada inalcançável a partir do código da aplicação, sugerindo que muitos podem ser falsos positivos. As implicações desses achados são significativas: para os desenvolvedores de ferramentas, são destacadas as limitações relacionadas à escalabilidade, atribuição de origem e redução de falsos positivos, apontando direções claras para aprimoramento. Para os engenheiros de software, os resultados oferecem orientação prática para interpretar os resultados das análises e priorizar os esforços de correção de forma mais eficiente.

O artigo da pesquisa foi redigido em inglês, seguindo o formato padrão de publicações do IEEE, com o objetivo de submissão a uma conferência que adota tal padrão. O manuscrito encontra-se finalizado e preparado para o processo de submissão. O capítulo subsequente apresenta a transcrição completa deste artigo.

Capítulo 2

Artigo: "Beyond Android Crypto API Misuse report: A study on Origin and Reachability"

2.1 Abstract

The correct use of cryptographic functions is a key component of safeguarding sensitive data. To assist developers in identifying cryptographic misuse, various static and dynamic analysis tools—such as Cognicrypt and Cryptoguard—have been developed. While such tools have improved awareness of security practices, recent studies reveal that many reported vulnerabilities originate from Third-Party Libraries, and a significant number of flagged methods may not even be invoked by the application. This study examines cryptographic misuse within a set of Android applications, aiming to determine the extent to which reported vulnerabilities stem from third-party libraries and how many of the flagged methods are actually unreachable within the application context.

2.2 Introduction

Ensuring the protection and confidentiality of data in software is essential and should be a priority throughout the development process. This is especially important when considering data privacy in Android applications (or apps, for short)—the most widely used platform for mobile devices such as tablets and smartphones. To achieve this, whether using built-in functions or third-party libraries (TPLs), cryptographic functions must be correctly implemented and utilized to prevent code vulnerabilities. In recent years, researchers have developed static and dynamic analysis tools to identify the incorrect usage of cryptographic libraries (or APIs) [1, 2, 4, 5]. These tools rely on advanced program analysis techniques that either identify code patterns revealing crypto API misuses [2] or code that deviates from rules that state the correct usage of Crypto APIs [1], allowing these tools to detect vulnerabilities and specify the affected method or class.

Studies such as [2–4] highlight the impact of static analysis tools on security awareness and confidence in software quality. These studies analyzed different sets of applications and reported at least one cryptographic API misuse in 96%, 91%, and 99.7% of the analyzed Android apps, respectively. However, as noted by the same studies, a significant portion of the reported misuses occur in methods that belong to TPLs used within the applications. Although these studies acknowledge that many detected cryptographic API misuses originate from TPLs, they do not adopt a systematic approach to differentiate between application code and library code. More importantly, previous research does not examine whether the reported vulnerabilities in libraries are actually reachable from the application code. In fact, the results of a preliminary study we conducted with Android app developers (Section 2.5) suggest the opposite. Some participants expressed skepticism about the relevance of the reported issues, with one remarking: *"the warning seems hard to locate. I can't find any direct use of com.android.volley.InternalUtils in the app. It might be used internally by other methods from com.android.volley, maybe it should be pointed to android developpers then, since these methods don't raise any warning?"*

In this paper, we report the results of three studies. The first investigates the prevalence of cryptographic API misuses in popular Android apps available on Google Play. In contrast to previous findings in the literature, both CogniCrypt and CryptoGuard fail to complete the analysis for a non-negligible percentage of the apps: execution either results in an error or does not finish within a 30-minute timeout (Section 2.6). The second study explores two heuristics that we integrate into the outputs of CogniCrypt and CryptoGuard to determine whether a cryptographic API misuse originates from the application code or from TPL code. The first heuristic utilizes an Android TPL detector (LibScout [6]), while the second examines how frequently the same misuse appears across different apps. Results show that, when combining both heuristics, 61% of the misuses

reported by CogniCrypt and 55% of those reported by CryptoGuard originate from TPLs (Section 2.7). The third study investigates whether the cryptographic API misuses originating from TPLs are actually reachable from the application code. We found that a large portion of these misuses occur in code that is unreachable from the app, suggesting they are likely false positives (Section 2.8).

Our findings offer actionable insights for both researchers and practitioners. For tool developers, we highlight limitations related to scalability, source attribution, and false positive reduction—suggesting clear directions for improvement. For software engineers, our results provide practical guidance for interpreting analysis results and prioritizing remediation efforts, especially in resource-constrained settings. We discuss the implications of our findings and the limitations of our research in Section 2.9.

2.3 Background

The widespread misuse of cryptographic functions in Java and Android applications leads to devastating data security breaches. Developers frequently struggle with low-level cryptographic APIs, resulting in insecure code, even when the algorithms themselves are conceptually secure and correctly implemented [7]. This highlights that simply having strong cryptographic algorithms is not enough; they must also be used securely within the application. To address this challenge, researchers have developed both static [1–3] and dynamic solutions [4, 5] for checking cryptographic rules and detecting these critical misuses. Examples of these tools are CogniCrypt and CryptoGuard.

CogniCrypt checks application compliance with cryptographic standards by analyzing call graphs, typestates, and variable values, and then verifies rule constraints and predicates written in a domain-specific language (CrySL) [1, 3]. CryptoGuard also detects cryptographic API misuses; however, it does so using specialized forward and backward program slicing techniques based on flow-, context-, and field-sensitive data-flow analysis [2]. Both CogniCrypt and CryptoGuard are static analysis tools with the same goal, but they cover different sets of cryptographic rules and employ distinct program analysis techniques. This complementarity justifies the use of both tools in our study. By employing them together, we were able to analyze a larger number of applications and compare their results.

Previous research has also highlighted that reusing TPLs can propagate vulnerabilities [8, 9], including those arising from cryptographic API misuses [3, 10, 11]. Indeed, third-party libraries play a crucial role in software development by reducing the developers' workload in re-implementing common functionalities. Despite that, their presence may be a source of vulnerabilities and risks to end-user privacy [8]. Several studies have introduced software solutions for detecting TPLs in Android applications [6, 12–15]. Leveraging diverse techniques, these tools employ various strategies aiming to accurately identify the exact versions of libraries embedded within applications, each one seeking to surpass the limitations of previous tools and keeping pace with evolving obfuscation techniques utilized by obfuscator programs and compilers such as ProGuard [16], DashO [17], Allatori [18], or the R8 compiler in Android Studio [19].

LibScout is a representative example of a TPL detector for Android [6]. It implements a lightweight detection approach that uses Merkle Trees to extract structured representations of libraries. According to its authors, LibScout is resilient to common obfuscation techniques such as identifier renaming and API hiding. In a large-scale study involving 2,065 versions of 164 distinct libraries and a dataset of 3,590 Android apps, the authors report an average detection of 13.1 distinct libraries per app. They also found that 70.40%

of the included library versions were outdated relative to the app's release date, among other findings.

Other examples of TPL detectors include Orlis [13], LibBloom [14], and LibScan [15], which are considered state-of-the-art tools for TPL detection, particularly due to their resilience against modern obfuscation techniques. In a preliminary study, we experimented with LibScout, LibScan, and LibBloom. However, LibScan failed to produce results for several Android apps in our dataset, and we encountered difficulties aligning the output formats of both LibScan and LibBloom with those of CogniCrypt and CryptoGuard. For these reasons, we chose to use LibScout in our research, as it successfully completed analysis for all apps in our dataset and produced results in a format that could be easily integrated with the outputs of CogniCrypt and CryptoGuard.

2.4 Study Design

The primary goal of this research is to characterize the prevalence of cryptographic API misuses in Android apps, with a particular focus on how these misuses are distributed across third-party libraries and the extent to which they occur in unreachable code. This investigation is motivated by findings from a preliminary study aimed at understanding Android developers' perceptions of warnings reported by the static analysis tools CogniCrypt and CryptoGuard. Unexpectedly, many developers were unfamiliar with the code flagged by these tools, indicating that the issues may originate from TPLs and could potentially reside in unreachable parts of the application. Further details about the preliminary study are provided in Section 2.5. Altogether, our research answers the following general research questions:

(RQ1) What is the prevalence of cryptographic API misuses in Android apps? Previous studies suggest that more than 95% of Android apps contain at least one misuse. Answering this research question helps confirm or refute findings from prior work, providing a more accurate understanding of how widespread these issues are and laying the groundwork for answering research questions RQ2 and RQ3.

(RQ2) To what extent do cryptographic API warnings originate from TPL code rather than the app's own code? Answering this question helps quantify the impact of TPLs on the total number of reported misuses. It also offers valuable insight for tool developers, who may use this information to improve the clarity of static analysis reports—for example, by distinguishing whether a warning originates from application code or from a dependency.

(RQ3) To what extent are the cryptographic API warnings from TPLs located in unreachable code? This question is key to understanding how many reported misuses could potentially be exploitable at runtime. It may also support improvements in static analysis tools by helping to identify likely false positives, thus improving the quality and actionability of the reported results.

To answer the research questions, we gather from the AndroZoo dataset [20] the 1000 most popular Android apps from the Google Play Store by number of downloads and positive reviews. We then execute CogniCrypt and CryptoGuard to collect crypto-apis misuse. Additionally, we calculate the proportion of these warnings that originate from TPLs and those that are located in unreachable code. These absolute numbers enable us to address RQ1, RQ2, and RQ3. Figure 2.1 illustrates the Overall research workflow, which comprises the stages of cryptographic vulnerability analysis, TPL identification, and reachability analysis to determine the origin and relevance

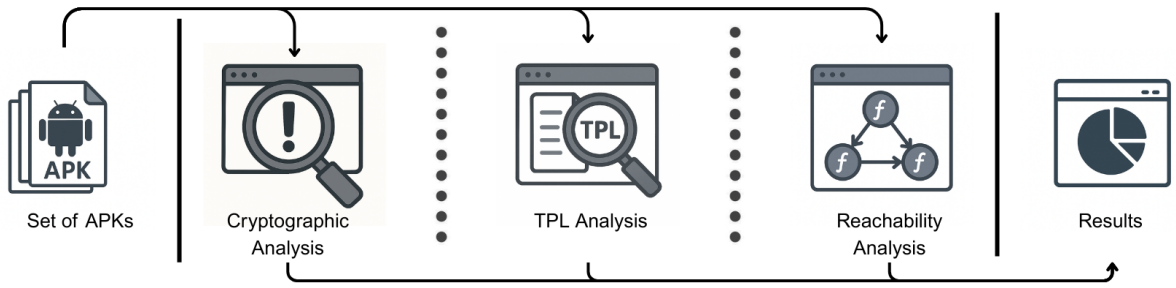


Figura 2.1: Overall research workflow.

of the reported misuses. Surprisingly, during our study, we observed that both CogniCrypt and CryptoGuard did not complete their analysis for 66.1% and 77.5% of the apps. Therefore, we also report descriptive statistics on the success rate of CogniCrypt and CryptoGuard executions. We make all artifacts available at this GitHub repository: `https://github.com/luisamaralh/sast-paper-android`

2.5 Preliminary Study: Android App Developers' Perception

2.5.1 Settings

In the preliminary study, we aimed to understand how open-source Android developers perceive the warnings reported by CogniCrypt and CryptoGuard. To do this, we used a dataset comprising all open-source apps in the *Money*, *Security*, *Connectivity*, *Phone & SMS*, and *System* categories from *F-droid*[1]—a catalog of FOSS (Free and Open Source Software) applications for the Android platform. Our rationale for selecting apps from these categories is that they are likely to involve privacy and security concerns.

Regarding data collection, we executed CogniCrypt and CryptoGuard to identify crypto API misuses in 363 Android apps from our F-Droid dataset. CogniCrypt and CryptoGuard reported an average of 47.49 warnings per project (median of 18). We then submitted issues to the GitHub repository of the projects, reporting that we (a) had possibly found vulnerabilities, and (b) documented the warnings in private gists for non-disclosure purposes.

If app developers expressed interest in evaluating the gists, we asked them for a suitable way to share the information privately. For those who agreed to receive general information, we randomly selected five warnings per project. After reviewing the initial gists, we asked whether they were interested in receiving a complete report with the remaining warnings. Figure 2.2 shows an example of a documented gist that we shared with the developers of a project.

2.5.2 Results

The analysis revealed that many of the issues identified by CogniCrypt were not inherent in the Android applications themselves but rather stemmed from third-party libraries utilized by these apps. Out of 20 open-source projects surveyed, 12 had issues originating from library code. Developer reactions to these library-related security problems varied: three developers committed to—or had already—updated the pertinent dependencies, while two others trusted renowned library providers so implicitly that they dismissed the warnings as inconsequential.

In another eight cases, developers reviewed the warnings and ultimately decided that the issues posed no threat to their applications. They either considered the alerts to be false positives from CogniCrypt or judged the vulnerabilities as not critical within the

[1]https://www.f-droid.org/en/packages/

CogniCrypt (report 1)

- Class: **hidden class name**
- Method: `md5`
- Line: 351
- Issue details: Constraint Error
  - Violating CrySL rule for `MessageDigest`
  - The parameter should be one of {SHA-256, SHA-384, SHA-512}

Code

```
public static String md5(String s) {
    try {
        MessageDigest md = MessageDigest.getInstance("MD5");
        md.update(s.getBytes());
        byte[] messageDigest = md.digest();

        StringBuilder hexString = new StringBuilder();
        for(byte b : messageDigest) {
            hexString.append(Integer.toHexString(0xFF & b));
        }
        return hexString.toString();
    } catch (NoSuchAlgorithmException e) {
        Timber.e(e);
    }
    return "";
}
```

Questions:

1. How likely might this warning reveal a security threat to this app?
   (a) Very unlikely
   (b) Unlikely
   (c) I cannot evaluate this
   (d) Likely
   (e) Very likely
2. Are you likely to accept a patch that fixes this particular issue?

Figura 2.2: Example of a secret gist sent to an Android project.

context of their application. For instance, one developer set aside a warning about MD5

hash usage because it was being applied in a low-risk, non-security-sensitive component.

It is also noteworthy that developers directly recognized the validity of a flagged problem in only four instances. Among these, just one developer took the initiative to implement a fix, while the others indicated no plans to address the issue.

Beyond these specific reactions, a key finding was that many developers struggled to fully understand the problems reported by CogniCrypt. The tool's explanations were often too brief, leaving them uncertain about the underlying rationale for the warnings. Consequently, developers advocated for more detailed, actionable guidance and clear suggestions for how to resolve the issues. Some even recommended that CogniCrypt's reports should explicitly differentiate between problems originating in the application's own code and those arising from TPLs.

2.6 Study 1: Prevalence (RQ1)

2.6.1 Settings

As mentioned earlier, this study aims to verify the prevalence of crypto-API misuse detected by CogniCrypt and CryptoGuard in Android apps. We downloaded from Androzoo the 1,000 high-popularity Android apps from the Google Play Store, considering the following selected criteria: We first selected 10,000 apps based on the number of downloads (10M to 10B downloads). We further refined the selection to the 1,000 apps with the highest number of positive reviews (1.2M to 202M). Thus, the selection process considers both the popularity and reputation of the applications.

To collect crypto-api misuse from the apps, we ran four concurrent instances, both for CogniCrypt (v4.2.2) and CryptoGuard (commit 92551ee), with a 30-minute timeout on a Debian virtual machine with 32 cores and 128 GB of RAM.

We executed the analysis according to the command-line execution and Java version recommendations in both CogniCrypt [21] and CryptoGuard's [22] documentation, configured with a maximum Java heap size of 32 GB and a thread stack size of 1 GB. For CogniCrypt, we specifically selected the Java Cryptography Architecture (JCA) rules.

Tabela 2.1: Example of Cognicrypt's Output for app Hola VPN.

Index	ErrorType	Class	Method	Message
1	ConstraintError	com.facebook.f0.c	java.lang.String h(java.lang.String)	(...) First parameter (with value "MD5") should be any of {SHA-256, SHA-384, SHA-512}
2	ConstraintError	com.appsflyer.internal.aa	java.lang.String h(java.lang.String)	
3	ConstraintError	org.hola.q9.e.a	byte[] q(byte[])	

Tabela 2.2: Example of part of CryptoGuard's Output for app Hola VPN.

Index	ErrorType	Class	Method	Message
1	Found broken hash functions	e.c.a.k0.n0	i	Found: "SHA-1"
2	Found broken hash functions	org.apache.http.impl.auth.DigestScheme	c	Found: "MD5"
3	Found broken hash functions	com.facebook.f0.c	h	Found: "MD5"
4	Found broken hash functions	org.hola.q9.e.a	q	Found: "MD5"

2.6.2 Results

In the experiment involving the execution of CogniCrypt on our dataset of 1000 apps, the analysis failed with an error for 241 apps (24.10%), timed out for 420 apps (42.00%), and successfully completed for 339 apps (33.90%). Among the 339 apps for which the execution completed successfully, 299 contained at least one cryptographic API misuse, corresponding to 88.20% of the successfully analyzed apps. Even more concerning, CogniCrypt reported a total of 13 470 crypto API misuses. This huge number of warnings might be partially explained because CogniCrypt generates different warnings for a given

crypto API misuse—since a crypto API misuse might violate different CrySL rules. After further analysis, we found that CogniCrypt reports 6780 unique crypto API misuses. The number of unique crypto API warnings reported by CogniCrypt ranges from a minimum of 1 to a maximum of 112, with an average of 22.67. Recall that, when analyzing an Android app, static analysis tools reveal crypto API misuses both in the app's code and in the code of the libraries it uses. We further explore this issue in the next section. Most JCA violations originate from the CrySL rules for the improper use of `MessageDigest` (33.65% of all warnings), `SSLContext` (13.78% of all warnings), and `Cipher` (10.82%) classes of the API.

When executing CryptoGuard on our dataset, the tool led to execution errors in 716 cases (71.60%) and timeouts in 159 instances (15.90%). The analysis was completed successfully for only 125 apps (12.50%). CryptoGuard reported at least one cryptographic API misuse in 122 of the 125 apps (97,6%) for which the analysis completed successfully. Unlike CogniCrypt, CryptoGuard reports only one warning for each crypto API misuse, totaling 4277 unique crypto API misuses. Most warnings correspond to violations of the *Used untrusted PRNG* rule, accounting for 2275 reported misuses (53.17% of the total number of warnings). However, it is also important to note that CryptoGuard applies an overly strict constraint for pseudo-random number generators, reporting a warning for every use of the `java.util.Random` class, regardless of whether the random value is actually used in a cryptographic context—some of these warnings may even be considered false positives. The *Found broken hash functions* rule has the second highest number of violations (1400), followed by *Used constant keys in code* (164), *Used HTTP Protocol* and *Found broken crypto schemes* (98). For CryptoGuard, the number of unique crypto API warnings per app varies between 1 and 102, with an average of 35.05.

Finding 1: Our results indicate that the static analysis tools CogniCrypt and CryptoGuard are unable to complete the assessment for a large proportion of apps. However, among the apps for which the analysis succeeds, both tools report a widespread presence of cryptographic API misuses: CogniCrypt reports that over 88% of the analyzed Android apps contain at least one violation, while CryptoGuard reports that more than 95% of the successfully analyzed apps exhibit at least one misuse.

2.7 Study 2: Third-party Libraries (RQ2)

2.7.1 Settings

In this study, we aim to determine the proportion of classes containing methods with reported warnings that originate from Third-party Libraries, building upon findings from the Prevalence study. To this end, we utilize the same set of applications and results from the previous study, in addition to the LibScout tool for TPL detection.

LibScout, like most TPL detectors, relies on a reference set of libraries for comparison. For this study, using the scripts available in LibScout's official GitHub repository, we obtained 13,225 versions of 432 distinct libraries, some of which come from popular organizations such as Facebook and OkHttp.

Given an application and a predefined set of libraries, LibScout identifies which of these libraries are present in the application. The limitations associated with this functionality will be discussed in the discussion section.

Based on the outputs of CogniCrypt, CryptoGuard, and LibScout, classes containing methods with reported warnings were classified as originating from a TPL if their fully qualified name started with the root package of a TPL identified by LibScout within the set of analyzed applications.

Fig. 2.3 provides a segment of a LibScout output file, indicating the detection of five libraries and their root packages in the Hola VPN application, whose root package is named `com.hola`. Based on this figure, along with Tables 2.1 and 2.2, and applying the classification method described, we confirm, for example, that classes with a qualified name starting with `com.appsflyer.internal.aa` originate from a TPL.

Tabela 2.3: Segment of LibScout's Output for app Hola VPN

Line	Content
499	- Full library matches:
500	**name:** AppsFlyer
501	**category:** Analytics
502	**version:** 6.2.3 [OLD VERSION]
503	**release-date:** 04.03.2021
504	**comment:** Mobile Attribution and Marketing Analytics
505	**lib root package:** "com.appsflyer"

2.7.2 Results

After integrating the results from LibScout with those from CogniCrypt and CryptoGuard, we found that 35.73% of the cryptographic API misuses reported by CogniCrypt originate from third-party library code. This proportion is even higher for CryptoGuard: of the 4277 cryptographic API misuses detected by CryptoGuard, 49.47% stem from

TPLs. We believe these percentages are conservative, as LibScout may fail to detect libraries that it is not configured to recognize. Nevertheless, we envision that tools like CogniCrypt and CryptoGuard should include additional information in their reports, indicating whether a vulnerability is likely to originate from TPLs. Libraries such as AppsFlyer, Facebook Sharing, Facebook Login, and OkHttp are among the top-ranked libraries most frequently associated with cryptographic API misuses. Using this conservative approach, we found that out of the 299 apps in which CogniCrypt reported a vulnerability, 40 contained vulnerabilities originating exclusively from TPLs. In contrast, among the 122 apps flagged by CryptoGuard, only two had warnings that stemmed exclusively from TPLs.

> **Finding 2:** LibScout's analysis reveals that a non-negligible share of cryptographic API misuses stem from third-party libraries: 35.73% of the warnings reported by CogniCrypt and 49.47% of those flagged by CryptoGuard are linked to TPLs.

Since using the LibScout's default configuration is limited to 432 distinct libraries, we decided to explore a different approach to identify warnings that are likely to come from third-party libraries. We started from the observation that some popular libraries are responsible for cryptographic API misuses detected in methods that appear across many apps in our dataset. For instance, as shown in Table 2.4, our analysis using CogniCrypt revealed that the method `hashWithAlgorithm` from the class `com.facebook.internal.Utility` was flagged in 137 distinct apps. The table highlights the methods from third-party libraries that most frequently appear in the outputs of CogniCrypt across different apps. On average, methods identified as vulnerable and originating from TPLs—according to both CogniCrypt and LibScout—appear in 5.41 distinct apps. For CryptoGuard and LibScout, this average is 2.86 distinct apps. In contrast, methods not classified as part of TPLs appear in fewer apps: 1.71 on average for CogniCrypt and 1.20 for CryptoGuard.

To identify methods likely originating from third-party libraries, we defined a second heuristic based on the distribution of warning occurrences across apps—complementing the first heuristic, which relies on LibScout. Specifically, we computed the upper bound for outliers using the interquartile range (IQR) method. Methods whose number of warning occurrences in distinct apps exceeds this upper bound are classified as originating from TPLs. This approach reflects the intuition that such methods appear with abnormally high frequency and are likely reused across multiple apps via external libraries. We apply this heuristic only to methods not already classified as third-party code by LibScout. For the methods associated with warnings reported by CogniCrypt and CryptoGuard, the computed upper bound is 3.50. Therefore, we also label as third-party library code

Tabela 2.4: Most frequently reported cryptographic API misuse methods from third-party libraries, as identified by CogniCrypt. The *Total* column indicates the number of distinct apps in which each class/method pair was flagged for a cryptographic API misuse.

Class	Method Name	Total
com.facebook.internal.Utility	hashWithAlgorithm	137
com.facebook.internal.security.CertificateUtil	getCertificateHash	132
com.facebook.appevents.internal.HashUtils	computeFileMd5	130
com.facebook.appevents.internal.HashUtils	computeChecksumWithPackageManager	117
com.facebook.appevents.AppEvent$Companion	md5Checksum	114
com.applovin.impl.sdk.utils.StringUtils	a	93
com.google.android.gms.internal.consent_sdk.zzcl	zzb	70
com.facebook.unity.FB	getKeyHash	52
okhttp3.internal.platform.ConscryptPlatform	newSSLContext	51
okhttp3.internal.platform.OpenJSSEPlatform	platformTrustManager	51

any method that (i) appears in more than 3.50 distinct apps and (ii) was not previously identified as third-party by LibScout.

Applying the second heuristic to the CogniCrypt outputs reveals a total of 1718 additional crypto API misuses that are likely to come from TPLs, and 245 additional third-party crypto API misuses for CryptoGuard. Altogether, combining both heuristics, we found that 61% of the CogniCrypt detected crypto API misuses are likely to come from TPLs, while 55% of the crypto API misuses reported by CryptoGuard are likely to originate from TPLs.

Finding 3: The percentage of vulnerabilities originating from TPLs ranges from 35% for CogniCrypt and 49.47% for CryptoGuard when considering only our first heuristic. When both heuristics are applied, these figures increase to 61% for CogniCrypt and 55% for CryptoGuard. These percentages are lower than those reported in a previous study [2], which employs a less systematic approach for classifying cryptographic API misuses as originating from TPLs.

2.8 Study 3: Unreachable Code (RQ3)

2.8.1 Settings

In this study, we aim to determine the proportion of cryptographic API misuse warnings, previously identified by CogniCrypt and CryptoGuard, that are located in unreachable code. This question is crucial to understanding how many reported misuses could potentially be exploitable at runtime, and it directly supports improvements in static analysis tools by helping to identify likely false positives, thereby enhancing the quality and actionability of the reported results. To this end, we build upon the results from the previous studies, using the same set of applications and reported warnings to address RQ3. For the detection of reachable code, we employed the static analysis tool *Androguard*. Androguard is a comprehensive Python-based framework designed for reverse engineering and automated static analysis of Android applications, capable of parsing APK and Dalvik Executable (DEX) files. The process involved, for each application, first identifying its main entry points. These entry points are crucial as they represent the initial starting points for potential execution paths within the application. We identified several categories of entry points by analyzing the APK using AnalyzeAPK() to obtain the APK object (a) and the Analysis object (dx):

- **Manifest-Declared Components and Lifecycle Callbacks:** We extracted all declared Activities, Services, Broadcast Receivers, and Content Providers from the AndroidManifest.xml file using the APK object's methods (e.g., a.get_activities(), a.get_services(), a.get_receivers(), a.get_providers()). For these components, we specifically identified their lifecycle callback methods (e.g., onCreate, onStart, onResume, onReceive), which are automatically invoked by the Android system during component state transitions.
- **Exported Components:** We identified components explicitly marked as android:exported="true the AndroidManifest.xml. Such components can be invoked by other applications, making their public methods potential external entry points.
- **Native Methods (JNI):** Methods declared as native (Java Native Interface) were considered entry points, as they bridge Java/Kotlin code with underlying native (C/C++) libraries. These were identified by inspecting method access flags.
- **Dynamic Code Loading (DCL):** We identified methods responsible for dynamically loading code (e.g., using DexClassLoader or PathClassLoader), as these can introduce new execution paths at runtime. This was achieved by searching for calls to relevant loader class constructors using dx.find_methods().

All identified methods from these categories were aggregated to form the comprehensive set of true entry points for each application. Based on these entry points, Androguard constructs the application's static call graph. The Analysis object (dx) is instrumental in generating this graph using the dx.get_call_graph() method. This method produces a networkx.MultiDiGraph object, where nodes represent individual methods within the application's bytecode, and directed edges signify method calls. NetworkX is a powerful Python package for the creation, manipulation, and study of complex networks, enabling the representation and analysis of these intricate code structures. A method is then considered reachable if a path exists in this call graph from at least one of the identified true entry points to the method in question. This static reachability analysis involves examining the application's codebase without executing it, evaluating which parts of the code might call or invoke vulnerable functions based solely on the code's structure and dependencies. With the list of reachable methods for each application, we performed the classification of the warnings reported by CogniCrypt and CryptoGuard. A warning was considered to be located in unreachable code if the method containing it was not present in the list of reachable methods generated by *Androguard*. This intersection allows us to quantify the proportion of alerts that would likely not be triggered at runtime, directly answering RQ3 and providing insights into the actionability of static analysis warnings by reducing false positives.

2.8.2 Results

By integrating the outputs of CogniCrypt and CryptoGuard—which report cryptographic API misuses—with CryptoGuard's reachability analysis, we were able to determine which misuses originating from third-party libraries are actually reachable from the application code. We hypothesize that non-reachable misuses are likely false positives. Therefore, cryptographic API detectors should indicate the reachability status of reported issues, as this could help mitigate a specific class of false positives in Android-focused cryptographic misuse detection tools. The results presented in the previous section show that CogniCrypt detected 2,423 cryptographic API misuses originating from TPLs (35.73% of the total). Among these, 1,675 were found to be unreachable according to AndroGuard's reachability analysis. In other words, 69.12% of the cryptographic API misuses from TPLs detected by CogniCrypt are considered unreachable by AndroGuard. Considering CryptoGuard, out of the 2,116 cryptographic API misuses reported as originating from TPLs, 2,003 (94.65%) were considered unreachable by AndroGuard.

Finding 4: A large proportion of cryptographic API misuses reported by CogniCrypt and CryptoGuard from third-party libraries—69.12% and 94.65%, respectively—were found unreachable by AndroGuard, suggesting many may be false positives.

Out of the 299 apps for which CogniCrypt detected at least one cryptographic API misuse, 31 (10.36%) had misuses occurring exclusively in unreachable methods. Similarly, for the 122 apps flagged by CryptoGuard, 37 (30.33%) contained only unreachable misuses.

2.9 Discussion

2.9.1 Implications of the Results

Our findings offer several significant implications for both researchers and practitioners in the field of Android application security. For tool developers, our study highlights a critical need for improving the scalability and robustness of static analysis tools, as both CogniCrypt and CryptoGuard exhibited high rates of analysis errors and timeouts (66.1% and 77.5%, respectively). Future development should prioritize handling diverse and complex application structures to ensure more complete analyses. Furthermore, the substantial proportion of cryptographic API misuses originating from Third-Party Libraries —61% for CogniCrypt and 55% for CryptoGuard when both heuristics are applied—underscores the importance of clear source attribution in tool reports. Tools should explicitly differentiate between issues in application code and those in TPLs to help developers focus their remediation efforts, directly addressing feedback from our preliminary study where participants desired such distinction. The discovery that a large percentage of TPL-related misuses are unreachable (69.12% for CogniCrypt and 94.65% for CryptoGuard) strongly suggests that incorporating reachability analysis into static analysis tools is crucial for reducing false positives. By filtering out unreachable code, tools can provide more actionable and trustworthy results, thereby improving developer confidence and reducing the perceived “noise"of warnings. Our successful use of two heuristics—LibScout for TPL detection and an outlier-based approach for identifying frequently occurring TPL methods across apps—demonstrates effective strategies for more accurately attributing misuses, which tool developers could integrate to enhance their detection capabilities.

For software engineers and security practitioners, our results provide practical guidance for prioritizing security remediation efforts. Given that a significant portion of reported vulnerabilities stem from TPLs and a substantial amount of these are unreachable, engineers can more effectively allocate resources by focusing on misuses in their own application code or reachable TPL code. The study reinforces the notion that while TPLs reduce development workload, they are also a significant source of vulnerabilities and risks. Developers should exercise caution when incorporating TPLs, regularly update dependencies, and be aware that security warnings may often point to library-specific issues rather than direct application code flaws. Finally, our findings offer crucial context for interpreting static analysis reports. Developers should be aware that a high number of warnings may include many false positives, especially those related to unreachable code or TPLs that are not critical to the application’s functionality. This understanding can prevent unnecessary effort in addressing non-critical alerts.

2.9.2 Threats to Validity

While our study provides valuable insights, it is important to acknowledge certain limitations that may affect the broader applicability and interpretation of our findings. A primary threat to external validity is the selection bias of applications: our study utilized the 1000 most popular Android apps from the Google Play Store, which, while aiming for representativeness of widely used applications, may not fully capture the entire app landscape, as it excludes the diversity of cryptographic misuse patterns found in less popular or enterprise-specific applications.

Regarding construct validity, limitations exist within the static analysis tools themselves. The incompleteness of analysis is a concern, as CogniCrypt and CryptoGuard failed to complete analysis for a substantial percentage of apps, meaning our prevalence findings are based on a subset of the dataset. This could potentially skew overall prevalence rates. Furthermore, CogniCrypt and CryptoGuard employ distinct program analysis techniques and cover different sets of cryptographic rules. While complementary, this means misuses detected by one tool might not be detected by the other, and reported numbers are specific to their capabilities. Like all static analysis tools, they are susceptible to false negatives and false positives. Although our reachability analysis addresses a specific class of false positives, others may still exist, potentially influencing the reported prevalence of misuses.

Limitations also apply to TPL detection. Our primary heuristic relied on LibScout, which depends on a predefined reference set of 432 distinct libraries. If an application uses a TPL not in this set, it would not be identified, leading to a potential underestimation of TPL-related misuses and making our reported percentages conservative. While LibScout is reported to be resilient to common obfuscation techniques, sophisticated obfuscation could still hinder accurate TPL identification. Additionally, our preliminary experimentation with other TPL detectors encountered issues, and relying solely on LibScout might introduce a specific bias.

Finally, the use of a static analysis tool for our reachability analysis presents inherent uncertainties. While it effectively identifies methods that can be reached, it does not guarantee that these methods will always be invoked at runtime under real-world usage scenarios. Complex execution flows and user interactions can influence actual code paths, meaning some “reachable"code might still rarely or never be executed. Although we identified a comprehensive set of entry points, it is possible that some less common or dynamically determined entry points were missed, potentially affecting the completeness of the call graph and thus the reachability analysis.

2.10 Related Work

Muslukhov et al. [23] were among the first to examine the influence of third-party libraries on the propagation of cryptographic API misuses in Android applications. Their study reports a substantially higher proportion of misuses originating from TPLs compared to our findings. Several methodological differences between their study and ours may account for this divergence. First, they developed BinSight [24], a tool designed to detect six categories of cryptographic API misuses within the Java Cryptography Architecture. In contrast, our study employs state-of-the-art static analysis tools—CogniCrypt and CryptoGuard—that detect a broader spectrum of cryptographic API misuses. Second, the two studies differ in how they attribute misuses to applications versus TPLs. BinSight primarily classifies call sites as TPL-related based on package names, whereas our methodology combines LibScout [6] with additional heuristics to perform this classification. Our approach is intentionally more conservative, which may reduce false positives but can also increase the likelihood of false negatives.

A substantial body of research has investigated the prevalence of cryptographic API misuses and the effectiveness of tools designed to detect them. Early studies introducing CogniCrypt and CryptoGuard revealed widespread occurrences of such misuses in Java libraries and Android applications [2,3,10]. For instance, Krüger et al. analyzed a dataset of 10,000 Android applications from AndroZoo and found that 8,422 contained at least one cryptographic misuse [10]. Many of these originated from widely used libraries; to mitigate this bias, the authors excluded misuses within classes declared in `com.android`, `com.facebook.ads`, `com.google`, and `com.unity3d`. After applying this filter, the number of affected applications dropped to 4,349, corresponding to 43% of the dataset.

Similarly, Rahaman et al. found that 39 of 46 Apache projects contained at least one instance of misuse [2]. In another study involving 6,181 popular Android applications, the same authors identified 130,845 cryptographic vulnerabilities, 95% of which originated from TPLs. Complementing these findings, Hazhirpasand et al. analyzed 489 open-source Java projects using CogniCrypt [25] and reported that 487 exhibited at least one misuse. While several issues were disclosed to project maintainers, only a small fraction were addressed, with most rejections citing the issues as non-security-sensitive or redirecting responsibility to TPL maintainers [25].

To facilitate systematic comparisons of misuse detection tools, researchers have developed dedicated benchmarks [26–28]. For instance, Afrose et al. introduced *CryptoBench* [28], a benchmark suite combining two widely adopted components: the *CryptoAPI Bench* and the *Apache Crypto-API Bench*. The former contains 181 synthetic unit tests spanning 18 misuse types, while the latter comprises 10 Apache projects with 121 curated vulnerabilities. Using these benchmarks, Afrose et al. evaluated four tools, inclu-

ding CogniCrypt and CryptoGuard, and reported that CryptoGuard achieved superior performance overall. They also observed that, among the tools studied, only SpotBugs and CryptoGuard scaled effectively to large projects.

Expanding on this evaluation, Zhang et al. assessed a broader range of static analysis tools across multiple benchmarks, including the *Crypto-API Bench*, *Apache Crypto-API Bench*, *MUBench*, and the *OWASP Benchmark* [29]. Their findings indicate that no single tool consistently outperforms others across all benchmarks: while CryptoGuard performs best on the *Crypto-API Bench* and *Apache Crypto-API Bench*, Xanitizer achieves the highest F1-scores on the remaining benchmarks. The authors also attempted to contribute fixes for the detected issues via pull requests, though most were ultimately not accepted by the target projects.

A different research group introduced *MASC Bench*, a benchmark consisting of 105 synthetic cryptographic API misuses, organized into nine categories and generated through mutation testing [27]. Using this benchmark, the authors evaluated nine cryptographic API misuse detectors, including CogniCrypt and CryptoGuard, and identified several limitations of these tools in detecting specific classes of misuses from the *MASC Bench*. Similarly, Schlichtig et al. developed the *CamBench* dataset, which comprises two benchmarks: *CamBench_Real*, featuring real-world applications with labeled cryptographic API usages, and *CamBench_Cap*, a synthetic collection of programs that exercise Java cryptographic APIs. In a preliminary assessment, the authors compared the detection capabilities of several static analysis tools, including CogniCrypt, CryptoGuard, FindSecBugs, and SonarQube, finding that CogniCrypt demonstrated superior effectiveness in identifying a broader range of cryptographic misuses [30].

In a more recent work, Xiao et al. evaluated an extension of Parfait [31], demonstrating its effectiveness on the *Crypto-API Bench* with a precision of 86.62% and a recall of 98.40%. Further evaluations of nine Oracle projects and two open-source systems (Spring Security and Orchid) showed that the Parfait extension scales to systems with hundreds of thousands of lines of code and outperforms CogniCrypt in terms of accuracy. Additionally, moving beyond static analysis approaches, Torres et al. investigated runtime verification as an alternative strategy for detecting cryptographic API misuse [5]. Their study, which evaluated five benchmarks including the *Crypto-API Bench*, *Apache Crypto-API Bench*, and the *MASC bench*, demonstrated that runtime verification achieves higher precision than static-analysis-based detectors. Additionally, the authors identified and corrected inconsistencies in the ground truth of CryptoBench [26] and re-evaluated prior findings, concluding that the scalability issues previously reported for CogniCrypt [26] were not observed in their experiments.

Recent studies have begun to investigate the use of foundation models for detecting

cryptographic API misuses [32, 33]. Xia et al. [32], for example, found that applying models such as GPT-4-turbo and CodeLlama without providing explicit misuse definitions resulted in a high number of false positives. However, when the models were supplied with additional context—such as descriptions of the specific vulnerability types to be detected—their precision improved significantly. In this enhanced setting, GPT-4-turbo achieved a precision of 0.87 and a recall of 0.90, demonstrating the potential of prompt engineering to improve model performance for this task.

2.11 Final Remarks

In this study we investigated cryptographic API misuses in Android applications, focusing on the impact of Third-Party Libraries (TPLs) and code reachability. Our analysis confirmed the widespread presence of cryptographic misuses, confirming the suspicions brought by the preliminary study: A large part of the crypto API misuses originate from TPLs, and most of them are actually unreachable.

The implications of our findings are clear: for tool developers, there's a need for improved scalability, robustness, and better origin attribution, along with integrating reachability analysis to reduce false positives. For app developers, understanding that many warnings may stem from unreachable TPL code can help in prioritizing remediation efforts and interpreting security reports more effectively. Ultimately, our research advocates for more precise and actionable security insights to enhance the Android ecosystem's security posture.

For future work, several research paths are suggested by this study. One promising direction involves developing new techniques or improving existing ones to enhance the accuracy of TPL detection and reachability analysis, especially when facing modern obfuscation techniques. Another avenue would be to investigate methods for integrating reachability analysis directly into cryptographic API misuse detection tools, which could significantly reduce false positives at the point of origin.

Capítulo 3

Conclusões

A pesquisa revelou um panorama complexo sobre a segurança de dados sensíveis na plataforma móvel mais difundida. Uma das descobertas mais impactantes foi a constatação de que uma proporção considerável dessas vulnerabilidades reportadas tem sua origem em bibliotecas de terceiros (TPLs). Este achado ressalta a importância de considerar o ecossistema de dependências no desenvolvimento de software, visto que as TPLs, embora acelerem o processo de desenvolvimento, podem introduzir riscos de segurança.

Mais crucial ainda, a pesquisa evidenciou que uma grande parte das vulnerabilidades de API criptográfica provenientes de TPLs reside em código inalcançável a partir da lógica principal do aplicativo. Isso sugere que muitas das sinalizações emitidas por ferramentas de análise estática podem, na verdade, ser falsos positivos.

Ao longo do estudo, fizemos as escolhas que acreditamos que nos conduziriam por um caminho que nos levaria às respostas que buscávamos e que valorizariam as nossas motivações e esforços. No entanto, as limitações de tempo e recursos nos levaram a adotar ferramentas e uma quantidade de aplicativos que nos limitaram a um resultado conservador.

Em termos de trabalhos futuros, abrem-se alguns caminhos de pesquisa a partir deste estudo. Uma direção promissora envolve o desenvolvimento de novas técnicas ou aprimoramento das existentes para aprimorar a precisão na detecção de TPLs e na análise de alcançabilidade, especialmente diante das técnicas de ofuscação modernas. Outra direção seria investigar métodos para integrar a análise de alcançabilidade diretamente nas ferramentas de detecção de mau uso de API criptográfica, o que poderia reduzir significativamente os falsos positivos no ponto de origem.

Referências

[1] Krüger, Stefan, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler e Ram Kamath: *Cognicrypt: Supporting developers in using cryptography*. Em *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, páginas 931–936, 2017. 1, 4, 6

[2] Rahaman, Sazzadur, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Danfeng, Yao e Murat Kantarcioglu: *Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects*, 2019. `https://arxiv.org/abs/1806.06881`. 1, 4, 6, 17, 23

[3] Krüger, Stefan, Johannes Späth, Karim Ali, Eric Bodden e Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis*. IEEE Transactions on Software Engineering, 47(11):2382–2400, 2021. 1, 4, 6, 23

[4] Piccolboni, Luca, Giuseppe Di Guglielmo, Luca P Carloni e Simha Sethumadhavan: *Crylogger: Detecting crypto misuses dynamically*. Em *2021 IEEE Symposium on Security and Privacy (SP)*, páginas 1972–1989. IEEE, 2021. 1, 4, 6

[5] Torres, Adriano, Pedro Costa, Luis Amaral, Jonata Pastro, Rodrigo Bonifácio, Marcelo d'Amorim, Owolabi Legunsen, Eric Bodden e Edna Dias Canedo: *Runtime verification of crypto apis: An empirical study*. IEEE Transactions on Software Engineering, 49(10):4510–4525, 2023. 4, 6, 24

[6] Backes, Michael, Sven Bugiel e Erik Derr: *Reliable third-party library detection in android and its security applications*. Em *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, páginas 356–367, New York, NY, USA, 2016. ACM, ISBN 978-1-4503-4139-4. `http://doi.acm.org/10.1145/2976749.2978333`. 4, 6, 23

[7] Nadi, Sarah, Stefan Krüger, Mira Mezini e Eric Bodden: *"jumping through hoops": Why do java developers struggle with cryptography apis?* Em *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, páginas 935–946, 2016. 6

[8] Wang, Ying, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu e Yang Liu: *An empirical study of usages, updates and risks of third-party libraries in java projects*. Em *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, páginas 35–45, 2020. 6

[9] Gkortzis, Antonios, Daniel Feitosa e Diomidis Spinellis: *A double-edged sword? software reuse and potential security vulnerabilities*. Em Peng, Xin, Apostolos Ampatzoglou e Tanmay Bhowmik (editores): *Reuse in the Big Data Era*, páginas 187–203, Cham, 2019. Springer International Publishing, ISBN 978-3-030-22888-0. 6

[10] Krüger, Stefan, Johannes Späth, Karim Ali, Eric Bodden e Mira Mezini: *Crysl: An extensible approach to validating the correct usage of cryptographic apis*. Em Millstein, Todd D. (editor): *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 de *LIPIcs*, páginas 10:1–10:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. https://doi.org/10.4230/LIPIcs.ECOOP.2018.10. 6, 23

[11] Gao, Jun, Li Li, Pingfan Kong, Tegawendé F. Bissyandé e Jacques Klein: *Understanding the evolution of android app vulnerabilities*. IEEE Trans. Reliab., 70(1):212–230, 2021. https://doi.org/10.1109/TR.2019.2956690. 6

[12] Zhan, Xian, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo e Yang Liu: *Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications*. Em *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, páginas 1695–1707, 2021. 6

[13] Wang, Yan, Haowei Wu, Hailong Zhang e Atanas Rountev: *Orlis: Obfuscation-resilient library detection for android*. Em *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, páginas 13–23, 2018. 6, 7

[14] Huang, Jianjun, Bo Xue, Jiasheng Jiang, Wei You, Bin Liang, Jingzheng Wu e Yanjun Wu: *Scalably detecting third-party android libraries with two-stage bloom filtering*. IEEE Transactions on Software Engineering, 49(4):2272–2284, 2023. 6, 7

[15] Wu, Yafei, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma e Peicheng Wang: *Libscan: Towards more precise third-party library identification for android applications*. Em *Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23)*, páginas 3385–3402. USENIX Association, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei. 6, 7

[16] PreEmptive: *"proguard the original java optimizer for android apps."*, 2025. https://www.preemptive.com/products/dasho/, acesso em 05/05/2025. 6

[17] Guardsquare: *"java security and android obfuscation with dasho."*, 2016-2025. https://www.guardsquare.com/proguard, acesso em 04/05/2025. 6

[18] Inc., Smardec: *"allatori java obfuscator."*, 2005-2025. https://allatori.com/, acesso em 05/05/2025. 6

[19] Google: *"shrink, obfuscate, and optimize your app."*, 2005-2025. https://developer.android.com/studio/build/shrink-code, acesso em 05/05/2025. 6

[20] Allix, Kevin, Tegawendé F Bissyandé, Jacques Klein e Yves Le Traon: *Androzoo: Collecting millions of android apps for the research community*. Em *Proceedings of the 13th international conference on mining software repositories*, páginas 468–471, 2016. 8

[21] CROSSINGTUD: *"cognicryptsast"*, 2017-2025. `https://github.com/CROSSINGTUD/CryptoAnalysis`, acesso em 17/07/2025. 13

[22] CryptoGuardOSS: *"cryptoguard"*, 2018-2025. `https://github.com/CryptoGuardOSS/cryptoguard`, acesso em 17/07/2025. 13

[23] Muslukhov, Ildar, Yazan Boshmaf e Konstantin Beznosov: *Source attribution of cryptographic api misuse in android applications*. Em *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, página 133–146, New York, NY, USA, 2018. Association for Computing Machinery, ISBN 9781450355766. `https://doi.org/10.1145/3196494.3196538`. 23

[24] "Muslukhov, I. e others": *Cambench repository*, 2025. `https://github.com/iim/binsight`, acesso em 2025-07-18. 23

[25] Hazhirpasand, Mohammadreza, Mohammad Ghafari e Oscar Nierstrasz: *Java cryptography uses in the wild*. Em *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '20, New York, NY, USA, 2020. Association for Computing Machinery, ISBN 9781450375801. `https://doi.org/10.1145/3382494.3422166`. 23

[26] Afrose, Sharmin, Ya Xiao, Sazzadur Rahaman, Barton P. Miller e Danfeng Yao: *Evaluation of static vulnerability detection tools with java cryptographic api benchmarks*. IEEE Transactions on Software Engineering, 49(2):485–497, 2023. 23, 24

[27] Ami, Amit Seal, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk e Adwait Nadkarni: *Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques*. Em *2022 IEEE Symposium on Security and Privacy (SP)*, páginas 614–631, 2022. 23, 24

[28] Schlichtig, Michael, Anna Katharina Wickert, Stefan Krüger, Eric Bodden e Mira Mezini: *Cambench – cryptographic api misuse detection tool benchmark suite*, 2022. `https://arxiv.org/abs/2204.06447`. 23

[29] Zhang, Ying, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao e Na Meng: *Automatic detection of java cryptographic api misuses: Are we there yet?* IEEE Transactions on Software Engineering, 49(1):288–303, 2023. 24

[30] "Schlichtig, M. e others": *Cambench repository*, 2025. `https://github.com/CROSSINGTUD/CamBench/blob/main/README.md`, acesso em 2025-07-18. 24

[31] Xiao, Ya, Yang Zhao, Nicholas Allen, Nathan Keynes, Danfeng (Daphne) Yao e Cristina Cifuentes: *Industrial experience of finding cryptographic vulnerabilities in large-scale codebases*. Digital Threats, 4(1), março 2023. `https://doi.org/10.1145/3507682`. 24

[32] Xia, Yifan, Zichen Xie, Peiyu Liu, Kangjie Lu, Yan Liu, Wenhai Wang e Shouling Ji: *Exploring automatic cryptographic api misuse detection in the era of llms*, 2024. `https://arxiv.org/abs/2407.16576`. 25

[33] Baek, Heewon, Minwook Lee e Hyoungshick Kim: *Cryptollm: Harnessing the power of llms to detect cryptographic api misuse*. Em Garcia-Alfaro, Joaquin, Rafał Kozik, Michał Choraś e Sokratis Katsikas (editores): *Computer Security – ESORICS 2024*, páginas 353–373, Cham, 2024. Springer Nature Switzerland. 25