To Fix or Not to Fix: A Critical Study of Crypto-misuses in the Wild

Anna-Katharina Wickert[*], Lars Baumgärtner[*], Michael Schlichtig[†], Krishna Narasimhan[*], Mira Mezini[*]
[*]Technische Universität Darmstadt
Darmstadt, Germany
Email: <wickert,baumgaertner,kri.nara,mezini>@cs.tu-darmstadt.de,
[†]Heinz Nixdorf Institute, Paderborn University
Paderborn, Germany
Email: michael.schlichtig@uni-paderborn.de

arXiv:2209.11103v2 [cs.CR] 23 Sep 2022

***Abstract*—Recent studies have revealed that 87 % to 96 % of the Android apps using cryptographic APIs have a misuse which may cause security vulnerabilities. As previous studies did not conduct a qualitative examination of the validity and severity of the findings, our objective was to understand the findings in more depth. We analyzed a set of 936 open-source Java applications for cryptographic misuses. Our study reveals that 88.10 % of the analyzed applications fail to use cryptographic APIs securely. Through our manual analysis of a random sample, we gained new insights into *effective false positives*. For example, every fourth misuse of the frequently misused JCA class `MessageDigest` is an *effective false positive* due to its occurrence in a non-security context. As we wanted to gain deeper insights into the security implications of these misuses, we created an extensive vulnerability model for cryptographic API misuses. Our model includes previously undiscussed attacks in the context of cryptographic APIs such as DoS attacks. This model reveals that nearly half of the misuses are of high severity, e.g., hard-coded credentials and potential Man-in-the-Middle attacks.**

***Index Terms*—API-misuses, cryptography, false positives**

I. Introduction

Many applications need to protect sensitive and high-value data, such as passwords or financial transactions, using cryptography (hereafter referred to as *crypto*). For this purpose, crypto APIs provide access to crypto tasks, protocols, and primitives in all major programming languages. However, several studies have revealed that developers struggle to use crypto APIs [14], [17] correctly and introduce misuses, meaning that an API usage may be used syntactically correct but problematic from a security perspective. A common crypto misuse is the use of the *Electronic Code Book (ECB)* mode for encryption. Although it has been known for a long time that *ECB* is insecure [5], it was, e.g., found in the widely used Zoom video conferencing system until May 2020[1].

To address this problem, static analyzers such as *SpotBugs*[2], *CryptoGuard* [19], and $CogniCrypt_{SAST}$ [13], have been proposed to support developers and security researchers in checking for such misuses. These tools have been used in various empirical studies that have produced worrying insights into the state of crypto usages in the wild [5], [13], [19], [9]. However, none of the studies performed a qualitative examination of the validity and severity of the findings. Such an examination is, however, essential for both getting a more realistic picture of the state of crypto usages in the wild and to improve future studies and analyzes. It is well-known that static analyzes may produce false positives, which are considered their *Achilles heel* [11], [24]. Moreover, in a security context, the validity of any finding should be judged from the perspective of a threat model to get a feeling for its severity.

This paper contributes to closing this gap by designing and conducting a study focusing on qualitative examination of reported crypto misuses. With regard to false positives, we are particularly interested in reports that are specific to their concrete usage. An API may be used in different – non-crypto - contexts and not the same constraints apply in all contexts. For example, the most misused JCA class `MessageDigest` from previous studies [9], [13], [19] may be used to compute hashes independent of a security context, e.g., to compute the hash of a file. However, a static analysis cannot know this and has to be conservative. Thus, it may produce false positives and draw an inaccurate picture of the security of our software. With regard to qualitatively judging the severity of findings, we formulate and use a novel, comprehensive threat model. Specifically, we designed and conducted a study to answer the following research questions:

RQ1: What are common (effective) false positives arising from misuses, e.g., due to a non-security context?

RQ2: How severe are the vulnerabilities introduced by crypto API misuses in applications?

We studied crypto misuses of two Java crypto providers – the Java Cryptography Architecture (JCA) and Bouncy Castle (BC) – in open-source Java projects from GitHub using $CogniCrypt_{SAST}$ [13]. We only included projects that are mainly developed and maintained by professionals to address the generalizability problem of open-source studies [21]. Altogether, we collected 936 Java projects, of which 210 use a crypto API and 88.10 % have at least one misuse.

[1]https://support.zoom.us/hc/en-us/articles/360043770412-Updating-your-Zoom-Rooms-to-version-5-0-5

[2]https://spotbugs.github.io/

©2022 IEEE

https://t.me/learningnets

```java
private byte[] signByte(byte[] dataToSign){
  byte[] signedBytes;
  Signature s = Signature.getInstance("SHA1WithRSA");
  s.initSign(getPrivateKey());
  s.update(dataToSign);
  // Call to signedBytes = s.sign() missing.
  return signedBytes;
}

// Get a PrivateKey object with an insecure key length.
private PrivateKey getPrivateKey() {
   return shortKey();
}
```

Listing 1: Code snippet that signs a byte array using a key.

To qualitatively analyze the misuses, we randomly picked 157 misuses for review. We observed that one fourth of the misuses of the class `MessageDigest` – the most misused class according to previous studies [13], [19], [9] and the second most in our study – occur in a non-security context. Thus, we can consider these misuses as *effective false positives* [20] that may not or cannot be fixed by developers.

To judge the severity of all findings, we defined a threat model that connects API misuses reported by $CogniCrypt_{SAST}$ to security vulnerabilities. This model subsumes the existing vulnerability model for crypto API misuses by Rahaman et al. [19] and includes new threats, e.g., *Denial of Service* (DoS) attack and *Chosen-Ciphertext Attacks* (*CCA*). Overall, our model marks 42.78 % of the misuses as high severity.

In summary, this paper makes the following contributions:

- We studied crypto misuses found in a large, representative data set of applications [21].
- We provide empirical evidence that the reported misuses contain a significant amount of *effective false positives*. Our disclosure confirms this and reveals that many may not be fixed even though some are ranked as high severity.
- We created a novel, comprehensive threat model mapping crypto API misuses to vulnerabilities, introducing previously undiscussed threats like DoS attacks and CCAs.

II. Background

In this section, we provide background information regarding crypto API misuses in Java and introduce $CogniCrypt_{SAST}$ [13], the crypto API misuse analyzer we used for our study. In addition, we introduce the term *effective false positives*.

A. Misuses of Java Crypto APIs

The JCA provides a set of extensible cryptographic components ranging from encryption over authentication to access control, enabling developers to secure their applications. It is implementation-independent by using a "provider" architecture; developers can plug and play their implementation of crypto primitives for use with this architecture. A commonly used provider besides the default that is shipped with the Java Development Kit (JDK) is the Bouncy Castle library.

Listing 1 illustrates a usage of the JCA to sign a byte-array `dataToSign`. For this, the `Signature` object is initialized with a signature algorithm (Line 3) and a private key, passed via the function `getPrivateKey`, is added to the `Signature` object `s` (Line 4). Next, the byte-array `dataToSign` is passed to `s` to actually compute the signature of the data (Line 5) before the function returns the signed bytes. Unfortunately, the call to `sign` that would return the signature is missing (Line 6).

A *crypto misuse*, hereafter just misuse, is a usage of a crypto API that is considered insecure by experts. A misuse may be syntactically correct, a working API usage, and may not even raise an exception. We will briefly discuss the error types defined by Krüger et al. [13] to illustrate some misuses.

1) **Constraint Errors** (Listing 1, Line 3): Crypto APIs use parameters to let developers select crypto algorithms when initializing crypto objects. These parameters are often passed as strings in a specific format.
2) **Incomplete Operation Errors** (Listing 1, Line 6): The security of crypto objects may rely on a specific protocol. For instance, `Signature` crypto objects, once initialized and filled with data via a call to `update(byte[])`, require a call to the `sign` method to complete the signing operation. Thus, the required calls to complete the use of an initialized crypto object are missing.
3) **Required Predicate Errors** (Listing 1, Line 4): Crypto objects often depend on each other. For example, `Signature` objects require a correctly generated `Key` object. In order for a composed crypto solution to be secure, it is required that its components on which it depends are secure. Thus, composing a crypto object with required but insecure objects results in a misuse.
4) **Never Type of Error:** Sensitive information, e.g., a secret key, should never be of type `java.lang.String`, as strings are considered insecure compared to mutable byte arrays. Strings are immutable and stay in memory until collected by Java's garbage collector. Thus, they are longer visible in memory for attackers than necessary and outside of the direct control of the developer [3].
5) **Forbidden Method Errors:** Certain methods of crypto objects should never be called for security reasons. For example, the `PBEKeySpec` object generates keys from passwords and requires a crypto salt while initializing the object. Therefore, constructors that are not parameterized with a salt cause a misuse.
6) **Type State Error:** Such an error occurs when an object moves into an insecure state as the result of an improper method call sequence. For example, a `Signature` object requires a call to the `initSign` method prior to any number of calls to the `update` method.
   The key difference to an *Incomplete Operation Error* is that the missing and expected call is within the call sequence, while for an *Incomplete Operation Error* the expected call sequence is not finished.

[3]https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/spec/PBEKeySpec.html, accessed 19.09.2022

https://t.me/learningnets

Vulnerabilities	Attack Type	Severity	Novel	C	IO	RP	NT	FM	TS
Predictable/constant crypto keys	Predictability Through Initialization	H	○	●	○	●	○	○	●
Predictable/constant passwords for PBE		H	○	○	○	○	○	●	○
Predictable/constant passwords for KeyStore		H	○	●	○	○	○	○	○
Cryptographically insecure PRNGs		M	○	○	○	●	○	○	○
Missed to finish crypto function	Predictability Through Usage	H	●	○	●	○	○	○	●
Missed to pass data		M	●	○	●	○	○	○	●
Insecure TrustManager	MitM Attacks on SSL/TLS	H	○	○	○	●	○	○	○
Insecure SSL/TLS standard		H	●	●	●	○	○	●	●
Static Salts in PBE	Chosen-Plaintext Attack (CPA)	M	○	○	○	●	○	○	○
ECB mode in symmetric cipher		M	○	●	○	○	○	○	○
Static IVs in CBC mode symmetric ciphers		M	○	○	○	●	○	○	○
Padding Oracle	Chosen-Aiphertext Attack (CCA)	M	●	●	○	○	○	○	○
Fewer than 10,000 iterations for PBE	Bruteforce Attacks	L	○	●	○	○	○	○	○
64-bit block ciphers		L	○	●	○	○	○	○	○
64-bit authentication tag GCM		L	●	●	○	○	○	○	○
Insecure cryptographic ciphers		L	○	●	○	●	○	○	○
Insecure cryptographic signature		L	●	●	○	○	○	○	○
Insecure cryptographic MAC		L	●	●	○	○	○	○	○
Insecure cryptographic hash		H	○	●	○	●	○	○	○
Usage of String	Credential Dumping	L	●	○	○	○	●	○	○
Missed to clear password		L	●	○	●	○	○	○	○
Trigger Exception	DoS Attacks	M	●	●	○	○	○	○	●

TABLE I: A model of vulnerabilities which can be detected with *CogniCrypt*$_{SAST}$. For novel ○ marks if this vulnerability is discussed by Rahaman et al. [19] and ● if not. For C: Constraint Error, IO: Incomplete Operation Error, RP: Required Predicate Error, NT: Never Type of Error, FM: Forbidden Method Error and TS: Type State Error ● marks if this vulnerability can be caused by the respective error type and ○ if not.

B. *CogniCrypt*$_{SAST}$

For our study, we used the crypto misuse detector *CogniCrypt*$_{SAST}$ which follows an allowlisting approach. In contrast to denylisting approaches such as *CryptoGuard* [19] that describe vulnerabilities, allowlisting approaches describe all secure API usages. Violations of the defined rules are reported as misuses [13]. Previous studies reported a precision of 85 % to 94 % [13], [10]. Further, *CogniCrypt*$_{SAST}$ supports the BC library next to the JCA.

C. *Effective False Positives*

Past research has shown that developers consider false positives as the "Achilles heel" of static analyzes [11], [2]. However, in practice, the definition of false positives varies: From a static analysis perspective, a false positive is a finding which is incorrectly identified by the analysis. For developers on the other hand, some findings can not be fixed in the application, e.g., due to a broken standard. Sadowski et al. [20] introduce the term *effective false positives* to cover reported misuses on which a user will not take further action.

For an example of an *effective false positive*, consider a usage of *MD5* that is correctly flagged by the static analysis. However, the usage of *MD5* at hand happens in a non-security context as the concrete call cannot be influenced by external factors and is not essential for the security of the software.

III. Threat-model of vulnerabilities introduced by API misuses

To reason about the potential impact of the reported misuses, we contribute a threat model extending existing models [19] with more vulnerabilities caused by API misuses. Specifically, we derive our model from a study of CrySL rules written by crypto experts, the respective APIs, and the misuses observed in our study. In combination with standard attacks and crypto misuses from previous work, our model covers a wide variety of crypto API misuses and their attack potential. We list the vulnerabilities, the attack types, the respective severity, the novelty, and the affected error types in Table I.

1) **Predictability Through Initialization:** Applications can become insecure if sensitive information like a key is predictable [5], [19], [14]. As an instance of this vulnerability, we consider predictable and constant crypto keys, e.g., a hard-coded key or predictable password used to derive a key from. Furthermore, a cryptographic secure random number in Java requires a non-predictable seed as well as the usage of a dedicated class, e.g., `java.security.SecureRandom`. If the application code misses to fullfill these requirements, the crypto operation becomes predictable. While Rahaman et al. [19] separated predictable keys and PRNGs, our model combines them, as both attack types are due to predictability.
2) **Predictability Through Usage:** In contrast to attacks of the type *Predictability Through Initialization*, the improper usage of an API can render the respective computation predictable. An example of this issue is the usage of crypto APIs which rely on data to be processed, e.g., `MessageDigest`, and fail to process the required data. Thus, essentially resulting in a predictable computation.
3) **MitM attacks on SSL/TLS:** The improper usage of

https://t.me/learningnets

SSL/TLS can enable an attacker to launch a Man-in-the-Middle (MitM) attack to gain sensitive information [6]. This includes improper configuration of connections, e.g., incorrect verification of protocols, as well as insecure cryptographic protocols, e.g., TLS 1.1, which are vulnerable to attacks like the POODLE attack.
4) **Chosen-Plaintext-Attack (CPA):** An encryption algorithm should be provably secure against chosen-plaintext-attacks (CPA) [5]. An example is a static *Initialization Vector* (IV) for the *Cipher Block Chaining* (CBC) mode.
5) **Chosen-Ciphertext-Attack (CCA):** While an encryption scheme should be provably secure against CPA, it should also be safe against chosen-ciphertext-attacks (CCA). Concretely, we cover improper padding schemes, like PKCS5 and PKCS7 in combination with the block cipher mode CBC [23], [12], and the usage of plain RSA [3] as these are all vulnerable to CCA. Note, that a padding attack requires an interaction between the attacker and the program that responds to messages from the attacker. Thus, data at rest is not vulnerable.
6) **Bruteforce Attacks:** Some cryptographic algorithms are vulnerable to repeated, extensive computations, e.g., a feasible collision computation for MD5 and SHA-1 [22]. Further, primitives like DES [19] with a block size of 64 bit or a 64-bit authentication tag for GCM [8] are vulnerable to brute-force attacks to break the encryption.
7) **Credential dumping:** In Java, string values are immutable and therefore cannot be cleared or overwritten from memory, except when the garbage collector runs, which is out of the developer's control. Thus, the JCA enforces the usage of byte arrays, e.g., `PBEKeySpec`, to handle sensitive information. However, developers may use strings to store this information and only convert it to byte arrays before calling the crypto API. Furthermore, classes like `PBEKeySpec` provide the method `clearPassword` to clear the internal copy of the password. However, usages of this class without calling this method render an application vulnerable to credential dumping.
8) **DoS Attacks:** A Denial-of-Service (DoS) attack limits the availability of a service, e.g., by deliberately overloading the memory or consuming an extensive amount of CPU cycles. One possibility to achieve this is by triggering an unintended behavior such as uncaught exceptions [27]. Such exceptions can be raised when contradicting the API contract, e.g., by missing the initialization of a cipher object[4]. Depending on the program, such an exception can prevent further correct control flow, can cause a program crash, or if triggered in rapid succession, it can spam the log file, increase CPU usage or cause IO congestion. A recent example is CVE-2021-23372, where an exception causes the application to crash.

We categorize the severity into high, medium, and low. Our prioritization is based on factors like whether the vulnerability can be exploited remotely, how difficult an attack is to perform, and if the attack leads to a direct gain or needs the combination with other flaws to be of use. Attacks such as MitM which can be triggered remotely and provide direct benefits for an attacker [19] are ranked as high severity. We mark vulnerabilities as medium severity, which lead to compromises of secrets for active, dedicated attackers. These vulnerabilities are of great help in undermining the security of the systems [26]. Examples of this are CPA and CCA attacks. An example of a vulnerability with low severity are credentials passed as a string, as these require prior access to the system or running process to be extracted. Thus, the attacker must have exploited other vulnerabilities before to gain access to the system.

[4]https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/Cipher.html, accessed 19.09.2022

IV. Empirical Study

In this section, we describe the setup of our study and the insights gained from it. First, the data set used for our analysis is described in Section IV-A, followed by the misuses identified in Section IV-B. Our research questions are answered in Section IV-C and IV-D, respectively. We published an artifact[5] including our script for the threat model.

A. Dataset

The dataset created by Spinellis et al. [21] addresses the generalizability problem of open-source studies and consists of 17,264 GitHub projects that are mainly developed or guided by enterprise employees and span multiple languages. *CogniCrypt*$_{SAST}$ works on Java binaries, hence, we filtered out the non-Java projects and attempted to compile the rest automatically for Maven/Gradle build instructions. This resulted in 936 Java enterprise-driven, open-source binaries, which serve as the basis of our study. We applied the *CogniCrypt*$_{SAST}$ version 2.7.2[6] with the corresponding rule set for JCA and BC to them[7]. As the objective of our study was exploratory in nature and was designed to understand cryptographic misuses in the wild and not evaluate the tools themselves, we choose *CogniCrypt*$_{SAST}$ due to its allowlisting approach and the inclusion of rules for JCA and BC. For each successfully analyzed application in our data set, *CogniCrypt*$_{SAST}$ created a report including details of the crypto objects analyzed and the identified misuses.

B. Crypto (Mis)uses - an Overview

Prevalence of JCA and BC. We consider an application to use a crypto API if we found at least one usage of either JCA or BC, without judging the security of the respective usage yet. In total, we found 210 applications that use a crypto API. All of these projects use the standard Java crypto library (JCA), while 7 of these projects also use the BC API. Within these applications, we analyzed 3,294 different crypto objects (hereafter object)[8] for JCA and BC.

[5]https://doi.org/10.6084/m9.figshare.21178243

[6]https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/2.7.2

[7]Previous studies used older *CogniCrypt*$_{SAST}$ versions [13], [9], [19].

[8]*CogniCrypt*$_{SAST}$ uses the term OBJECT to refer to any initialized Java object which interacts with a class covered by a CrySL rule [13]

https://t.me/learningnets

JCA Class	Misuses observed	Projects affected
`Cipher`	563	36
`MessageDigest`	472	113
`SSLContext`	414	68
`SecretKeySpec`	218	53
`Mac`	161	34
`Signature`	143	18
`KeyStore`	120	44

TABLE II: An overview of the observed misuses and affected projects for all JCA classes with more than 100 misuses.

Misuses of JCA and BC. In the 210 projects, we spotted crypto misuses in 185 and out of these 5 projects have misuses of both the BC and JCA library. The remaining 180 projects only use the JCA library for crypto. While previous studies reported 95 % [13] and 96 % [9] of applications with misuses for the JCA, we observe 88.10 % projects with at least one misuse. Thus, the selection criteria of so-called enterprise-driven applications [21] may have a slight positive impact on the number of misuses and thus security of the applications.

Leading causes of misuses (error types): We identified 2,695 misuses and those are mainly distributed over *Required Predicate*, *Incomplete Operation*, and *Constraint* errors. Most of the misuse reports, in total 960, are due to a *Required Predicate*, which means that composing multiple crypto objects seems to be challenging to get right. An example of this misuse is in line 4 of Listing 1 where the passed key is generated insecurely. Thus, the required predicate of the function call `initSign`, namely a secure key, is not fulfilled and causes a misuse. *Incomplete operation* errors arising out of a missing method call contributed to 815 misuses. For example, crypto primitives may require multiple method calls for completion, e.g., initializing, updating, and finally retrieving a hash code. While this is the second most frequent misuse, only 37.80 % of the projects are impacted by such misuses. Most common in applications are misuses due to insecure parameters to functions, e.g., *SHA-1* as a parameter to an initialization of a `MessageDigest` object. Overall, 80 % of the applications contain at least one such *Constraint* error, causing in total 566 misuses. This prevalence can be explained by the fact that the security of algorithms evolves over time, algorithms may have not been updated, and may be used for tasks beside the crypto domain. We observed both cases during the disclosure of our manual analysis (Sec IV-E).

Most frequently misused crypto classes: The error types discussed above describe misuses from the API perspective. Another interesting perspective is investigating which crypto primitives were often misused based on the classes that pertain to their implementation, as certain classes may be more prone to *effective false positives*, e.g., `MessageDigest`, or to high-severity misuses, e.g., `SSLContext`. In the following, we will discuss all classes with more than 100 misuses. Table II presents the total number of misuses and affected projects for these classes. While in total most of the misuses occur due to the class `Cipher`, 80.54 % of the projects, have no misuse of this class at all. Thus, affecting only 36 projects. We observed the second-most misuses due to the class, `MessageDigest` with 472 misuses in total. Thus, more projects (113) have at least one misuse of the class `MessageDigest` than no misuse of this class. This indicates that misuses of the class `MessageDigest` are widespread among many applications. The class `SSLContext` contributes to 414 misuses in total within 68 projects. We observe fewer misuses for the class `SecretKeySpec` with 218 misuses in total distributed among 53 projects. These results reveal that only a few JCA classes are frequently misused in many of the applications. For our data set, we can avoid 61.86 % of the misuses by fixing misuses of these four classes.

C. Manual Analysis of Reports (RQ1)

To gain a more in-depth understanding of common (effective) false positives, we randomly sampled 157 (Confidence: 99 %, margin of error: 10 %) misuses. The first four authors of this paper independently analyzed the sampled misuses, with at least two reviews per misuse. In case of disagreement, the reviewers in question resolved them with further code review until they reached a conclusion. The focus of our analysis is the precision of the report as well as the context of the misuse to answer our first research question.

Previous studies [13] concentrated on measuring precision under the assumption that the rules of the static analysis are correct and based on error types that can be easily verified manually, namely *Constraint* and *Type State* errors. In contrast, we inspected all error types and show that some misuses can occur due to an incomplete or outdated rule-set or occur in a non-security context. Thus, resulting in *effective false positives*. We also consider such issues as a contributing factor to the overall lower precision discussed in this paper.

Undecided Reports. We explicitly marked misuses as *undecided* if it was impossible to judge their validity due to cryptic or confusing error reports. A common reason was that the reported insecure usage could not be identified at the location of the report, nor in the respective callers. In total, we observed 31 misuses which fall into this category. All of these misuses occur for the more complex error types, namely *Incomplete Operation*, *Required Predicate*, and *Type State*. We assume that these error reports will be of little help to the user attempting to make decisions based on them. For the remainder of the discussion, we will focus on the remaining 126 misuses.

True Positives. Our qualitative analysis revealed 93 (roughly 74 %) true positives distributed among all previously discussed JCA classes and error types. By using regex, e.g., with grep, to detect usages such as MD5, 25 true positives can be detected. Analyst or developers may consider such simple and fast tools as a prefilter of more sophisticated analyses such as *CogniCrypt*$_{SAST}$ [13] and *CryptoGuard* [19]. The remaining true positives are due to an incorrect order of method calls (21), which requires a typestate analysis, storing secrets in Java strings (22), using a default value of the JCA (15), or hard-coded credentials (2). The remaining 8 instances that we classified as true positives from the static analysis depends on information that can not be dissolved statically. True

https://t.me/learningnets

```java
14 final MessageDigest md = getMessageDigest(algo);
15 final byte[] buffer = new byte[4096];
16 int count = 0;
17 while ((count = inputStream.read(buffer)) > 0) {
18     md.update(buffer, 0, count);
19 }
20 return md.digest();
```

Listing 2: A `MessageDigest` object which only calls `update` when it has something to read from an `InputStream`.

positives due to incorrect call orders and relying on defaults are to the best of our knowledge not covered in existing denylist approaches. Thus, showcasing the importance of a sophisticated static analysis beyond simple pattern matching.

We observed 8 misuses because a required call is only present in a loop body where the respective loop may not be executed (Listing 2, Line 18). Thus, a misuse may be present and causing a vulnerability. While these cases should be analyzed to identify if a vulnerability can be triggered causing a true positive, most often these cases results into effective false positives indicating a coding smell. Note, that this is a known limitation of analyses such as $CogniCrypt_{SAST}$ [13].

False Positives. Our manual analysis resulted in 35 false positives caused by several reasons. One reason are incorrect API specifications. $CogniCrypt_{SAST}$ relies on the correctness of the API specifications, and in case of incorrect or outdated specifications, the respective report is erroneous and leads to a false positive. In total, we observed 17 misuses due to this reason. For example, the *Optimal Asymmetric Encryption Padding (OAEP)* was encoded with a typo as *OEAP* in the CrySL rule, causing a misuse to be reported, when the correct padding scheme *OAEP* is passed. We fixed this error in a pull request which is already merged into the main branch of the CrySL rules repository. Previous studies have assumed the correctness of the CrySL rules [13] and thus not considered these usages as false positives. With this assumption, our analysis reveals a true positive rate of 84 % that is similar to previous studies [13], [10]. In another example, the call order of functions was modeled incorrectly. After a discussion about this issue, a pull request fixing this problem was opened.

Besides the correctness of the specification, the underlying static analysis can cause false positives due to imprecise modeling of the program flow. Our reviews revealed 11 false positives due imprecise modeling, e.g., wrapping of crypto objects. The remaining false positives are due to further challenges with respect to modeling the program statically.

Effective False Positives. During our analysis, we observed that some misuses discussed previously are *effective false positives* (cf. Section II-C). Thus, the user of the analysis would not take any action, as, while they may acknowledge that it is theoretically a correct finding, it is invalid or irrelevant for the specific application. We provide an overview of the common patterns that we identified in the wild in Figure 1.

Concretely, we identified 9 out of the 126 misuses from a non-security context. One example is the usage of *MD5* in

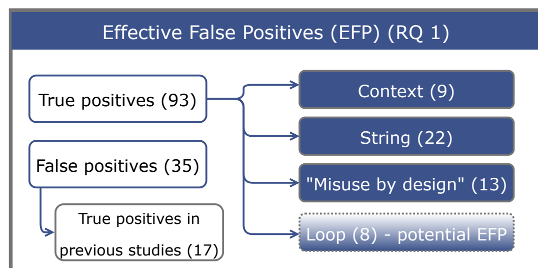

Fig. 1: Common *effective false positives* patterns that we identified during our manual analysis.

deeplearning4j[9]. Here, the *MD5*-digest is used to verify that the input array was not modified during execution. Another example is in the project UAVStack[10] where *MD5*-hashes of a file are used to provide content-aware detection changes.

The number of *effective false positive* misuses due to a non-security context may appear low at first sight. However, one has to consider this number in relation to the overall number of reports for the specific kind, e.g., class `MessageDigest`. It turns out that 22.86 % of the `MessageDigest` misuses in our randomly selected sample fall into the category of *effective false positive* due to non-security context. If we consider (a) that roughly 25% of the reported `MessageDigest` misuses are potentially *effective false positives* and (b) that this class causes by far the most of the misuses reported by previous studies [9], [13], [19], respectively the second most frequent in our study, it becomes clear that previous reports may contain a significant number of *effective false positives*. Further, this may significantly decrease the acceptance of these tools by developers [11], [20].

Besides the context, we observed that the 22 true positives, which use a string instead of a byte-array, may result in *effective false positives*. While it is possible to read user input as a byte-array, some credentials are passed by API design as a string instead of the byte-array. Thus, the developer has no effective and easy way to fix the misuse by hand.

Our analysis revealed another potential source for *effective false positives* not considered by previous studies. Some misuses may be intentionally introduced in the projects contained in our data set. For example, the static analyzer SonarSource[11] includes tests, not following the typical naming scheme for test, for correct API usages and insecure crypto misuses. These misuses, in total 13 in our sample, within the tests are intended.

Research Question 1

A non-security context, the usage of string, and "intentional misuses" are common *effective false positives*.

[9] An open-source deep learning library for JVM-based languages (Stars: 12.2k, Forks: 4.9k), https://github.com/deeplearning4j/deeplearning4j

[10] Graphical tool to monitor and analyze programs running JVM (Stars: 667, Forks: 278), https://github.com/uavorg/uavstack

[11] https://github.com/SonarSource/sonar-java (Stars: 755, Forks: 501)

https://t.me/learningnets

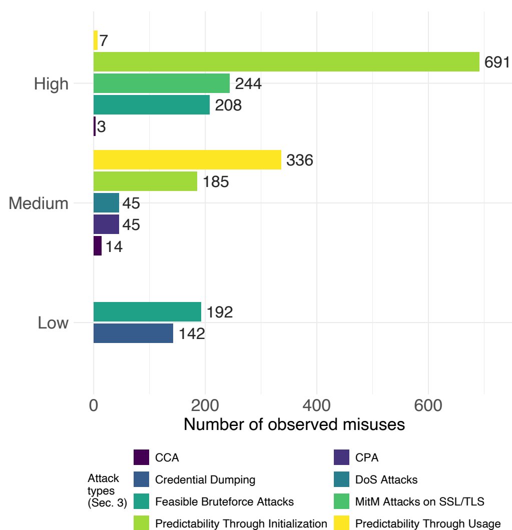

Fig. 2: Number of misuses by severity for the attack types.

D. Vulnerabilities (RQ2)

Based upon our vulnerability model introduced in Section III, we will discuss the most common vulnerabilities and their severity in this section. Most of the misuses are due to the attack types *Predictability Through Initialization* (876), *Bruteforce Attacks* (400), and *Predictability Through Usage* (343). In total, we identified 1,153 high-severity, 643 medium-severity, and 334 low-severity misuses. We present the number of misuses, their severity, and their relationship to the respective attack types in Figure 2.

High-severity vulnerabilities. Vulnerabilities belonging to the attack type *Predictability Through Initialization* are caused by 691 misuses spread over 108 projects. They are due to an insecurely generated key caused by a *Required Predicate* error. Another source of insecurely generated keys are insecure key generation parameters, e.g., *HmacSHA1* or *DES*. These were reported for 84 misuses within 29 projects.

The usage of SSL, TLS 1.0, and TLS 1.1 is insecure and should be avoided. We found 141 such misuses in 61 individual projects. For example, 29 misuses within 20 projects are caused by using SSL.

Medium-severity vulnerabilities. Our analysis detected 336 misuses which can result in *Predictability Through Usage* by observing the crypto function calls, their call order, and inputs. For example, the analysis identifies at least one path which consumes the crypto object without adding any data, e.g., instantiating a message digest without providing an input. Beside the predictable computations, the most relevant medium-severity vulnerabilities are due to insecure PRNGs (total: 185) and DoS attacks caused by exceptions (total: 45).

Low-severity vulnerabilities. Most (192) of the low-severity vulnerabilities are of attack type *brute-force* for ciphers, e.g., using an insecure signature algorithm or a 64-bit block cipher. For the attack type *credential dumping*, we identified 130 low-severity misuses within 47 projects due to the use of a string instead of a byte-array for passing on secrets. If an attacker has control over the system, they might retrieve the secret handled as string from memory as it is immutable and cannot be cleared from memory. Furthermore, we found 12 misuses which use a byte array for their secrets and misses to clear the memory explicitly. Thus, the secret may stay in memory as for the previously discussed secrets handled as strings.

Research Question 2

Nearly half the misuses (42.78 %) are of high-severity and should be prioritized while fixing misuses.

E. Responsible Disclosure.

We informed - when possible - all projects for which we could confirm a true positive from the analysis perspective. To avoid influencing the maintainers, we did not provide information about our judgment w.r.t *effective false positives*. So far, we received feedback for 22 misuses from 55 reported misuses. One project fixed the misuse that we classified as high severity and 15 projects considered the misuse that we reported as an *effective false positive*, with responses ranging over "MD5 is used to generate test data", "the affected program component is not shipped to the customers", "the usage is secure in our setting", or "misuses in dependencies are ignored". One of these projects added documentation to the identified insecure usage, and initiated an internal analysis of their code for crypto misuses. Overall, the disclosure confirms our observation that a more fine-grained assessment of misuses is important, e.g., considering a context. For five misuses, we received no final decision from the maintainers, e.g., they only thanked us for the report, and for one misuse we were asked to provide a fix that requires to change their API. Further, some discussions raised the questions which entity, e.g., the maintainers or the user of an application, is responsible for fixing a misuse.

V. Related Work

Previous studies on crypto misuses have primarily focused on Android applications and identified that 88 % to 99 % of the applications using crypto have at least one misuse [5], [16], [13], [19], [18]. Beside Android applications, Krüger et al. [13] inspected maven artifacts, mostly Java libraries. To demonstrate the scalability of their tool *CryptoGuard*, Rahaman et al. [19] analyzed 46 Apache projects in addition to their set of Android applications. All studies focused on precision from the tool's perspective, rather than understanding false positives from the developer's or security analyzer's perspective. An exception is an exploratory study [10] that opened issues for some identified misuses. The results obtained mostly agree with the feedback we received from the disclosure as well as with our manual analysis. In contrast, we not only rely on the judgement of developers by manually analyzing the misuses.

Beside Java, IoT firmware, iOS applications and Python projects have been analyzed, revealing that nearly one fourth to 82 % of the software is vulnerable [28], [15], [7], [25].

https://t.me/learningnets

Previous research about misuses in code available from Q&A platforms such as Stack Overflow reveals that 71 % to 90 % of the posts have at least one misuse [4]. Thus, 98 % of the Android Apps with code copied from Stack Overflow snippets have at least one misuse [1].

VI. Conclusion

In this paper, we presented a study of Java crypto misuses that focuses on understanding (effective) false positives and the severity of misuses. To understand common *effective false positives* that arise from misuses, we manually reviewed a random sample of misuses and identified several *effective false positives* such as the ones happening in a non-security context. We provide an extensive threat model covering previously undiscussed vulnerabilities such as DoS attacks in the context of crypto API misuses. Our threat model marks nearly half of the misuses as high-severity. For example, 29.05 % of the applications still use SSL, TLS 1.0, or TLS 1.1. and may be exploitable remotely.

Overall, our study reveals several directions for future work. While we got first insights into the differences of misuses between the JCA and BC, future research can answer the question if the BC API supports developers to write more secure code. Further, our manual analysis reveals several sources of *effective false positives* that can be addressed in existing static analysis tools.

Acknowledgments

We want to thank the anonymous reviewers and everyone who supported us on our journey to publish this work.

This research work has been founded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – SFB 1119 – 236615297 and – SFB 1053 – 210487104 – and by the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and the Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE, by the LOEWE initiative (Hesse, Germany) within the emergenCITY center.

References

[1] Acar, Y., Backes, M., Fahl, S., Garfinkel, S., Kim, D., Mazurek, M.L., Stransky, C.: Comparing the Usability of Cryptographic APIs. In: IEEE Symposium on Security and Privacy. SP, USENIX Association (2017)

[2] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. PASTE, ACM (2007)

[3] Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In: Annual International Cryptology Conference. CRYPTO, Springer (1998)

[4] Braga, A., Dahab, R.: Mining Cryptography Misuse in Online Forums. In: IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE (2016)

[5] Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An Empirical Study of Cryptographic Misuse in Android Applications. In: ACM Conference on Computer & Communications Security. CCS, ACM (2013)

[6] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: An analysis of android ssl (in)security. In: ACM Conference on Computer and Communications Security. CCS, ACM (2012)

[7] Feichtner, J., Missmann, D., Spreitzer, R.: Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In: ACM Conference on Security & Privacy in Wireless and Mobile Networks. WiSec, ACM (2018)

[8] Ferguson, N.: (May 2005), https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf

[9] Gao, J., Kong, P., Li, L., Bissyandé, T.F., Klein, J.: Negative results on mining crypto-api usage rules in android apps. In: International Conference on Mining Software Repositories. MSR, IEEE/ACM (2019)

[10] Hazhirpasand, M., Ghafari, M., Nierstrasz, O.: Java cryptography uses in the wild. In: 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM (2020)

[11] Johnson, B., Song, Y., Murphy-Hill, E., Bowdidge, R.: Why don't software developers use static analysis tools to find bugs? In: International Conference on Software Engineering. ICSE, IEEE (2013)

[12] Klima, V., Rosa, T.: Side channel attacks on cbc encrypted messages in the pkcs# 7 format. IACR Cryptol. ePrint Arch. (2003)

[13] Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: Crysl: An extensible approach to validating the correct usage of cryptographic apis. In: IEEE Transactions on Software Engineering. TSE, IEEE (2019)

[14] Lazar, D., Chen, H., Wang, X., Zeldovich, N.: Why Does Cryptographic Software Fail?: A Case Study and Open Problems. In: Asia-Pacific Workshop on Systems. APSys, ACM (2014)

[15] Li, Y., Zhang, Y., Li, J., Gu, D.: icryptotracer: Dynamic analysis on misuse of cryptography functions in ios applications. In: International Conference on Network and System Security. NSS, Springer (2015)

[16] Muslukhov, I., Boshmaf, Y., Beznosov, K.: Source Attribution of Cryptographic API Misuse in Android Applications. In: Asia Conference on Computer and Communications Security. ASIACCS, ACM (2018)

[17] Nadi, S., Krüger, S., Mezini, M., Bodden, E.: Jumping through hoops: why do Java developers struggle with cryptography APIs? In: International Conference on Software Engineering. ICSE, ACM (2016)

[18] Piccolboni, L., Guglielmo, G.D., Carloni, L.P., Sethumadhavan, S.: CRYLOGGER: Detecting Crypto Misuses Dynamically. In: Symposium on Security and Privacy (SP). IEEE (May 2021). https://doi.org/10.1109/SP40001.2021.00010

[19] Rahaman, S., Xiao, Y., Afrose, S., Shaon, F., Tian, K., Frantz, M., Kantarcioglu, M., Yao, D.: CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In: Conference on Computer and Communications Security. CCS, ACM (2019)

[20] Sadowski, C., Van Gogh, J., Jaspan, C., Soderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: International Conference on Software Engineering. ICSE, IEEE (2015)

[21] Spinellis, D., Kotti, Z., Kravvaritis, K., Theodorou, G., Louridas, P.: A Dataset of Enterprise-Driven Open Source Software. In: International Conference on Mining Software Repositories. MSR, ACM (2020)

[22] Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The first collision for full sha-1. In: Annual International Cryptology Conference. CRYPTO, Springer (2017)

[23] Vaudenay, S.: Security flaws induced by cbc padding—applications to ssl, ipsec, wtls... In: Advances in Cryptology. EUROCRYPT, Springer (2002)

[24] Wagner, S., Jürjens, J., Koller, C., Trischberger, P.: Comparing bug finding tools with reviews and tests. In: Testing of Communicating Systems. TestCom, Springer (2005)

[25] Wickert, A.K., Baumgärtner, L., Breitfelder, F., Mezini, M.: Python crypto misuses in the wild. In: International Symposium on Empirical Software Engineering and Measurement. ESEM, ACM (2021)

[26] Wijayarathna, C., Arachchilage, N.A.G.: Using cognitive dimensions to evaluate the usability of security APIs: An empirical investigation. In: Information and Software Technology. IST, Butterworth-Heinemann (2019)

[27] Wu, J., Liu, S., Ji, S., Yang, M., Luo, T., Wu, Y., Wang, Y.: Exception beyond exception: Crashing android system by trapping in "uncaught exception". In: International Conference on Software Engineering: Software Engineering in Practice Track. ICSE-SEIP, IEEE (2017)

[28] Zhang, L., Chen, J., Diao, W., Guo, S., Weng, J., Zhang, K.: Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In: International Symposium on Research in Attacks, Intrusions and Defenses. RAID, USENIX Association (2019)

https://t.me/learningnets