Towards Precise Reporting of Cryptographic Misuses

Yikang Chen*, Yibo Liu†, Ka Lok Wu*, Duc V. Le‡, Sze Yiu Chau*
*The Chinese University of Hong Kong †Arizona State University ‡Visa Research
{ykchen, wkl021, sychau}@ie.cuhk.edu.hk yiboliu@asu.edu levduc112@gmail.com

***Abstract*—In the last decade, a series of papers were published on using static analysis to detect cryptographic API misuse. In each paper, apps are checked against a set of rules to see if violations exist. A common theme among these papers is that rule violations are plentiful, often at the scale of thousands. Interestingly, while much effort went into tackling false negatives, curiously, not much has been said on (1) whether the misuse alarms are indeed correct and meaningful, and (2) what can future work improve upon apart from finding more misuses.**

In this paper, we take a deep dive into the rule violations reported by various academic papers as well as the rules, models and implementations of their detectors, in an attempt to (1) explain the gap between their misuse alarms and actual vulnerabilities, and (2) shed light on possible directions for improving the precision and usability of misuse detectors. Results of our analysis suggest that the small-scale inspections done by previous work had some unfortunate blind-spots, leaving problems in their rules, models, and implementations unnoticed, which in turn led to unnecessary overestimation of misuses (and vulnerabilities). To facilitate future research on the topic, we distill these avoidable false alarms into high-level patterns that capture their root causes, and discuss design, evaluation and reporting strategies that can improve the precision of misuse findings. Furthermore, to demonstrate the generalizability of these false alarm patterns and improvement directions, we also investigate a popular industry detector and a dynamic detector, and discuss how some of the false alarm patterns do and do not apply to them. Our findings suggest that the problem of precisely reporting cryptographic misuses still has much room for future work to improve upon.

I. Introduction

As cryptography plays an indispensable role in securing contemporary systems, many research efforts went into analyzing both the implementation and usage of cryptography. Specifically, a long line of work has been dedicated to the problem of designing and implementing detectors that can catch misuses of cryptographic APIs in various applications [1]–[9]. Common examples of misuses that are caught and reported include the invocation of algorithms perceived to be weak (*e.g.*, AES-ECB), as well as the use of constants (*e.g.*, keys and passwords) and weak parameters (*e.g.*, short key lengths). While previous works did an excellent job in discovering many *potentially* severe findings, curious questions remain on whether the misuse alarms are indeed correct and actionable to developers, and what are the areas that future work can improve upon besides competing to find more potential misuses.

Problem. In this paper, we focus on the problem of *false alarms* from static cryptographic misuse detectors. Existing research suggests a high false positive rate in bug detection tools can lead to developers resisting tool adoption [10], [11]. Interestingly, while previous work investigated the problem of false negatives in detecting cryptographic misuse [8], currently there is no independent study of false alarms. Thus in this paper, we take a technical deep dive into the problem of false alarms in cryptographic misuse detectors, and discuss their root causes as well as possible ways of addressing them. The main objectives of this paper are to present a more nuanced and balanced account of what counts as misuses (and vulnerabilities), investigate why developers write code that detectors consider as misuses, and identify possible improvement directions (IDs) in terms of improving the precision and usability of cryptographic misuse detectors in general.

Scope and methodology. We select 3 recent papers based on *static analysis* that have publicly released their detectors [1]–[3], which constitute the current state-of-the-art academic efforts on using static analysis to detect cryptographic misuses. Isolating false alarms from a corpus of misuse reports is challenging, since no single detector achieves perfect recall and precision, and there is no *misuse oracle* that can programmatically decide if a misuse report is a false alarm. Thus, to shed light on the problem of false alarms, we adopt a 2-phase approach in our analysis. First, we manually sample misuse reports that are either (i) frequently occurring or (ii) likely to be false alarms, and perform root-cause analysis on them. This allows us to demonstrate the existence of false alarms, and distill high-level patterns of their root causes. Then, for some false alarms with root causes that can be corrected, we refine and rerun the detectors to estimate their overall spread and scale beyond our manual samples. Furthermore, to demonstrate the generalizability of the root-cause patterns and IDs, we also investigate a popular open-source industry static detector [12] and an academic dynamic detector [7], and discuss how some of the false alarm patterns do and do not apply to them.

Findings. Interestingly, we find that many misuse alarms might not be very actionable to developers, and that previous works might have overestimated the number of misuses and vulnerabilities. For example, a noticeable portion of constant keys and passwords reported by two academic detectors are actually false positives caused by implementation bugs. We

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.241032
www.ndss-symposium.org

Majority of the work was conducted while the authors were respectively at The Chinese University of Hong Kong† and the University of Bern‡.

find this situation alarming, as a more precise tool that reports less false alarms might appear outperformed by these existing tools. Furthermore, despite the best intentions of helping developers, certain misuse rules and their models capture neither sufficient conditions nor necessary conditions of cryptographic vulnerabilities, and thus the resulting alarms might not be able to really help developers write more secure code.

Root causes. Although static analysis is not expected to be perfect, our results suggest that false alarms in detecting cryptographic misuse can occur not only due to classic challenges, but also because of ① overly conservative misuse rules, ② imprecise modeling, and ③ implementation bugs in detectors themselves. The false alarm patterns due to ① likely also apply to other papers (*e.g.*, [4], [6], [7], [9]) and industrial tools not considered by this work, as they share similar misuse rules.

False alarms due to ③ highlight the need to improve evaluation of detectors to avoid blind-spots, whereas false alarms caused by ① and ② reflect the *fallacy of the converse*. That is, the usage rules and their models are sometimes not designed to capture sufficient conditions of vulnerabilities. As will be explained later, one major contributor to this sufficiency gap is the *context* of usage. While an API method (*e.g.*, AES-ECB) might be insecure in a certain context (*e.g.*, direct encryption of sensitive payloads), usage of the same method in a different context might be acceptable (*e.g.*, implementing other secure AES modes such as GCM) or mandatory by standards (*e.g.*, Adobe PDF [13]). To clarify, the point of this paper is not to promote the use of API methods that are deprecated due to known weaknesses. However, directly reporting all API calls to such methods as misuses or vulnerabilities without considering the context of usage, leads to many false alarms and might not be the most constructive approach.

Additionally, we found the differences in *idiosyncrasies* (*e.g.*, API behaviors and default values) on different Java platforms (*e.g.*, Android) to be another contributor to the sufficiency gap. Interestingly, previous work often directly apply detectors designed for conventional Java on Android apps [1], [2], thus misjudging many apps and underestimating the resulting false alarms. This shows that the subtleties of Android warrant a careful and fine-grained custom handling.

Contributions. In summary, this paper makes the following technical contributions:

1) We revisit academic static detectors of cryptographic API misuse, and demonstrate with empirical evidence that, despite the precision implied by the corresponding papers, many false alarms exist. This suggests there is much room for improvement in terms of precisely detecting and reporting cryptographic misuses.
2) For a *subset* of modeling and implementation issues, we refine the detectors to estimate the scale of their resulting false alarms in the data set. We include the refined versions in our publicly released artifacts, which could be useful to other studies.
3) Based on the false alarms observed, we distill high-level root-cause patterns, and discuss the challenges as well as possible improvements in making misuse detectors more usable to developers.
4) Based on false positives found in actual app code, we prepared a collection of minimal working examples (MWEs) that can illustrate problems in existing detectors. To facilitate future research, these MWEs are also included in our publicly released artifacts.

II. Related Work

Cryptographic API misuse detection. Researchers have put much effort into detecting cryptographic API misuse over the past decade. MalloDroid [14] extended Androguard [15] to perform static analysis for SSL/TLS related API in Android applications. Another static analysis tool based on Androguard, CryptoLint [4], further implemented six common rules of cryptographic API usage for Android applications. CryptoREX [3] applied the same six rules in the context of IoT firmware and reported the prevalence of cryptographic misuses in firmware images based on their static taint analysis. More recently, CogniCrypt$_{SAST}$ [2] and CryptoGuard [1] have been created for detecting cryptographic API misuse in Java and Android applications. Both tools are stated as flow-, context- and field-sensitive, and they both consider a larger rule set than the likes of CryptoLint and CryptoREX. In contrast, Crylogger [7], another recent effort, uses a dynamic analysis to log the usage of cryptographic APIs in Android applications and detect violations by analyzing the log. Crylogger is positioned as a tool that complements other static tools such as CryptoGuard [7]. Apart from devising detectors, researchers have also prepared two benchmarks, CryptoAPI-Bench [16] and CamBench [17]. Both benchmarks only contain manually crafted examples that lack the intricacies of real-world applications code, and thus their corresponding evaluations did not discover many of the false alarms found in this paper.

Meta-studies. In the literature, meta-studies were conducted on topics including fuzzing [18], Website fingerprinting attacks [19] and defenses [20], smart contract vulnerabilities [21] and analysis tools [22], and Android taint-analysis [23]. Back in 2013, [10] conducted a user study to investigate why developers are not widely adopting static analysis tools. The study concluded that a large volume of false alarms significantly reduces developers' tool acceptance. For *general* API misuse detectors based on static analysis, a prior work [24] performed an empirical evaluation based on 90 real-world misuses and 10 hand-crafted misuses, and found that all detectors suffer from low precision and recall in real-world applications. Another prior work [25] focuses on static vulnerability detectors for C code, and found that false negatives could be quite prevalent in realistic settings. On the problem of *cryptographic* API misuse, a recent work used mutators to investigate the problem of false negatives [8]. Concerning false positives, the most relevant works include the manual investigation [26] done on the alarms from CogniCrypt$_{SAST}$, as well a small-scale user study presented in [27], where developers reportedly rejected some of the pull requests, citing acceptable usage in non-security-critical context. Comparing to these prior work, this paper presents an in-depth technical investigation that covers more detectors, reveals more false alarm patterns, and provides more concrete improvement directions.

III. Overview of the Detectors

For this study, we consider three representative papers from the literature that publicly released their detectors, *i.e.*, CryptoGuard [1], CogniCrypt$_{SAST}$ [2], and CryptoREX [3] Both

TABLE I. MISUSE RULES FROM CRYPTOGUARD

Rule 1: Do not use predictable/constant cryptographic keys.
Rule 2: Do not use predictable/constant passwords for PBE.
Rule 3: Do not use predictable/constant passwords for KeyStore.
Rule 4: Do not use custom Hostname verifiers to accept all hosts.
Rule 5: Do not use custom TrustManager to trust all certificates.
Rule 6: Do not use custom SSLSocketFactory without manual Hostname verification.
Rule 7: Do not use HTTP.
Rule 8: Do not use predictable/constant PRNG seeds.
Rule 9: Do not use cryptographically insecure PRNGs (e.g., `java.util.Random`).
Rule 10: Do not use static Salts in PBE.
Rule 11: Do not use ECB mode in symmetric ciphers.
Rule 12: Do not use static IVs in CBC mode symmetric ciphers.
Rule 13: Do not use fewer than 1,000 iterations for PBE.
Rule 14: Do not use 64-bit block ciphers (e.g., DES, IDEA, Blowfish, RC4, RC2).
Rule 15: Do not use insecure asymmetric ciphers (e.g, RSA, ECC).
Rule 16: Do not use insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2).

TABLE II. ERROR TYPES IN COGNICRYPT$_{SAST}$

Error Types#	Description
HardCodedError (H)	object has hardcoded value
ForbiddenMethodError (F)	object calls a forbidden method
RequiredPredicateError (R)	object has a required predicate not satisfied
TypestateError (T)	object typestate not following an expected sequence of events
ConstraintError (C)	object uses a value that is not allowed
IncompleteOperationError (I)	object does not have an expected event
NeverTypeOfError (N)	object is of a forbidden type

TABLE III. MISUSE RULES FROM CRYPTOREX

Rule 1: Don not use electronic code book (ECB) mode for encryption
Rule 2: Do not use a non-random initialization vector (IV) for ciphertext block chaining (CBC) encryption
Rule 3: Do not use constant encryption keys
Rule 4: Do not use constant salts for password-based encryption (PBE)
Rule 5: Do not use fewer than 1000 iterations for PBE
Rule 6: Do not use static seeds for random number generation (RNG) functions

CryptoGuard and CogniCrypt$_{SAST}$ target Java applications, while CryptoREX targets binaries. CryptoLint [4], one of the seminal work on detecting cryptographic API misuses, first defined six misuse rules. Subsequent papers often adopt these same rules, and might expand on the rule set to cover more potential misuses. Each misuse rule can either be modeled as a blacklist or a whitelist.

CryptoGuard. As summarized in Table I, CryptoGuard checks 16 rules. Some rules are quite similar to that of CryptoLint, but CryptoGuard targets more classes in the Java API. According to the CryptoGuard paper, their rules correspond to 5 types of attacks: rules 1-3 cover predictable secrets, rules 4-7 cover man-in-the-middle (MITM), rules 8-9 cover predictability attacks, rules 10-12 cover chosen-plaintext attacks, and rules 13-16 cover brute-force attacks. Under the hood, CryptoGuard performs an on-demand data flow analysis based on the Soot framework [28]. It implements forward and backward slicing in a flow-, context- and field-sensitive manner, and targets specific parameters used by certain classes of the Java API as the slicing criteria. It traces def-use relations among intra-procedural program statements, and the inter-procedural caller-callee relations across different methods.

CogniCrypt$_{SAST}$ with CrySL. CrySL is a domain specific language (DSL) for expressing the misuse rules to be checked by CogniCrypt$_{SAST}$. CrySL captures the expected usage patterns defined for 49 Java classes, which cover 7 *error types* that represent different cryptographic misuses to be reported, as shown in Table II. For example, a Java object of a certain class may be incorrectly used without first calling its `initilize()` method, resulting in a TypestateError. For ease of discussion, we only consider the 11 Java classes that share similarity with the high-level rules of CryptoGuard.

Each CrySL rule is modeled in CogniCrypt$_{SAST}$ as a finite-state machine, which integrates IDE^{al} [29], a flow-, field-, and context-sensitive typestate [30] analysis that models the call sequence of API methods from the corresponding Java classes. CogniCrypt$_{SAST}$ also extends Boomerang [31], an on-demand pointer analysis, for tracking specific objects or values in a call sequence. In the rest of this paper, we use CogniCrypt$_{SAST}$ to refer to the tool and the CrySL rules it uses as a whole. The techniques used in CogniCrypt$_{SAST}$ are more complex than that of CryptoGuard, though the CryptoGuard paper claims it outperforms CogniCrypt$_{SAST}$ in terms of precision and recall on its benchmark [1].

CryptoREX. Comparing to the likes of CryptoGuard and CogniCrypt$_{SAST}$, CryptoREX has a relatively simpler design. As shown in Table III, CryptoREX checks 6 misuse rules that firmware developers should conform to, which are basically adopted from that of CryptoLint. To detect violation of those rules, CryptoREX executes a backward slicing to determine (i) whether certain functions of interests are being called, and (ii) whether some parameters of a particular function call will be set to constant values. Specifically, CryptoREX traverses all backward paths from each call site of a cryptographic function of interests, in an attempt to compute all possible values of the specific parameters. Apart from this, CryptoREX also uses the `nm -D <filename>` command to check whether PRNG functions `rand` and `srand` are imported by a binary, as a part of its rule 6. In CryptoREX, the list of functions and parameters that are of interests can be described in a configurable manner, which by default uses a list made of 107 functions from 7 popular cryptographic libraries.

IV. METHODOLOGY

In this paper, we consider 2 types of false alarms: False Positives (FPs) and Ineffectual True Positives (ITPs). FPs refer to false alarms caused by bugs and other limitations of the static analysis implemented in a detector. In contrast, ITPs are considered true positives from the perspective of *program analysis*, where a misuse detector detected patterns that it is designed to detect, albeit what it detects might not always help developers (*i.e.*, ineffectual), due to the design of their misuse rules and models.

To test the detectors targeting Java (*i.e.*, CryptoGuard and CogniCrypt$_{SAST}$), we run them on a data set of 3489 open-source Android apps collected from F-Droid [32]. Choosing F-Droid over Google Play has the advantage of being able to see the non-obfuscated source code, making it easier for us to understand the context of certain reported misuses. For CryptoREX, we attempted to download 1437 firmware images (1 version per product), and successfully collected 1177 from 6 vendors *(Netgear: 573, D-link: 469, Linksys: 52, Zyxel: 46, TP-Link: 33, Tomato: 4)*. We then run CryptoREX on them. In our experiments, unless stated otherwise, we use CryptoGuard (commit id *92551ee*), CogniCrypt$_{SAST}$ (commit id *1405ebd*) with its CrySL rule set (commit id *6d844ab*), and CryptoREX (commit id *3dc81c9*).

We note that no false-alarm oracles currently exist, and differential testing [33] of misuse detectors is difficult, as they have different targets, rules, and reports that are not directly comparable. Thus, in this paper, we adopt a 2-phase approach in our analysis. First, we sample and manually analyze some

misuses reported by the detectors, and determine the root causes and nature of such reports. Then, for some false alarms, we refine the detector models or implementations, and run the detectors again to estimate their overall spread in the data set.

For manual analysis, given the sheer number of misuse alarms and the labor intensity of root cause analysis, it is not possible to cover all the misuses reported. To facilitate our sampling, we merge misuse alarms of each rule based on the reported misuse data (*e.g.*, certain constant values), together with the class and method names (CryptoGuard and CogniCrypt$_{SAST}$), or the file, function names and addresses (CryptoREX). This helps because many third-party libraries are reused across apps and firmware, and most misuse alarms are triggered by third-party library code [1], [34]. We then rank the merged misuses by their occurrence in apps (and firmware) after merging. Next, we inspect the *top-10* offending methods (functions), because they affect the most number of apps (systems). Then, based on the values of misuse data, we pick and inspect additional alarms that are likely to be false. The root cause analysis for each detector is conducted independently by one of the authors, and the analyzed results are checked and discussed by the other authors. Then the author prepares a representative minimal working example (MWE) by debloating real-world code to demonstrate the existence and root cause of a false positive found in a detector. An overview of the total number of misuses reported and sampled can be found in Appendix A-A. In the following sections, we present the high-level false alarm patterns, grouped by their root causes, with some representative examples. Appendix B explains how to access our publicly released artifacts.

V. False Positives from Static Analysis

We now present the false positives (FPs) observed in our investigation. Here we mainly focus on the root causes and improvement directions (IDs) that are not discussed in the original papers. For instance, we observed 31 FPs due to the depth of orthogonal slicing being 1, which is already acknowledged in the CryptoGuard paper [1], and thus we do not show the details here. Likewise, both CryptoGuard and CogniCrypt$_{SAST}$ [1], [2] explicitly acknowledged their path insensitivity, so we skip discussing the corresponding FPs.

Pattern #1 – Broken def-use chains due to variable reassignment (CryptoGuard). Rules 1–3, 8, 10, 12–13 of CryptoGuard all involve finding constants by backward slicing, which uses def-use relations to track variables of interest. Interestingly, we found that the backward slicing in CryptoGuard erroneously assumes a variable in the Jimple intermediate representation (IR) will only be assigned once, and thus the slicing will sometimes add unrelated statements to its def-use chains, discovering data flows that actually do not exist. To illustrate this problem, we show a rule 3 FP example in Listing 1. This is a typical way of loading a `KeyStore` from a given input stream with a provided password (line 11). The `KeyStore.getDefaultType()` method (line 5) returns a constant string `jks`, which is incorrectly detected by CryptoGuard as a constant password. To see why this happens, look at the Jimple IR of the `loadAppKeyStore` method shown in Listing 2. The `String` variable `$r2` is first assigned the output of `KeyStore.getDefaultType()` (line 7), and then later is reassigned to hold the `pwd` field of `testKeyStore` (line 10). When CryptoGuard uses backward slicing to track `$r5` (line 12), it first adds the code on line 11 since `$r5` is assigned the result of `$r2.toCharArray()`. Then the tracking continues by considering `$r2`. Line 10 is added to the def-use chain, and because the backward slicing algorithm does not kill `$r2` after this, it continues to add line 7 as well. Hence, CryptoGuard sees a non-existent data flow from `KeyStore.getDefaultType()` to `pwd`, thus reports a rule 3 violation claiming `jks` is a constant password. The corresponding simplified intra-procedural def-use chain is shown in the Figure 1(a) with brown color, showing the erroneous def-use relations from `$r5` to `jks`.

The mistaken assumption would have been correct if the Jimple intermediate representation (IR) generated by Soot is in static single assignment (SSA) form [35], but in practice it is not, and the same variable can be assigned multiple times within a method. This problem can be refined by killing off a variable once the nearest assignment has been found by backward slicing, and keeping the union of used variable sets at merge points after backward slicing traverses through branches. This idea is illustrated in Figure 1(b). Keeping the union of used variable sets at merge point of multiple branches makes a safe refinement of CryptoGuard's original def-use chains. We implemented this refinement in CryptoGuard, rerun it on the app data set, and found a large number of FPs for rules 1–3, 8, 10, 12–13 caused by this same root cause, as shown in Table IV. As such, we find this FP pattern particularly important to explain and address, otherwise a new misuse detector that does not exhibit the same FPs could appear outperformed by CryptoGuard.

TABLE IV. CryptoGuard: broken def-use false positives

Rule	Alarms	FPs (%)	Rule	Alarms	FPs (%)
1, 2*	972	599 (61.63%)	10	150	20 (13.33%)
3	364	118 (32.42%)	12	490	82 (16.73%)
8	105	13 (12.38%)	13	1510	1438 (95.23%)

* CryptoGuard combines these two into one type of alarm

```
1  public class testKeyStore {
2    private File keyStoreFile;       private String pwd;
3    KeyStore loadAppKeyStore() throws Exception {
4      KeyStore ks;
5      ks = KeyStore.getInstance(KeyStore.getDefaultType());
6      ...
7      ins = new java.io.FileInputStream(keyStoreFile);
8      ks.load(ins, pwd.toCharArray()); // backward-slice 2nd param
9      return ks;
10   }
11   public void setKey(String pass) { this.pwd = pass; }
12 }
```

Listing 1. An FP example of Pattern #1

```
1  java.security.KeyStore loadAppKeyStore() {
2    testKeyStore r0;              java.io.File $r1;
3    java.lang.String $r2;         java.security.KeyStore $r3;
4    java.io.FileInputStream $r4;  char[] $r5;
5    ...
6    r0 := @this: com.test.testKeyStore;
7    $r2 = staticinvoke <java.security.KeyStore: java.lang.String
         getDefaultType()>();
8    $r3 = staticinvoke <java.security.KeyStore: java.security.
         KeyStore getInstance(java.lang.String)>($r2);
9    ...
10   $r2 = r0.<com.test.testKeyStore: java.lang.String pwd>;
11   $r5 = virtualinvoke $r2.<java.lang.String: char[] toCharArray
         ()>();
```

```
12    virtualinvoke $r3.<java.security.KeyStore: void load(java.io.
          InputStream,char[])>($r4, $r5);
13    ...
14  }
```

Listing 2. Jimple code of Listing 1

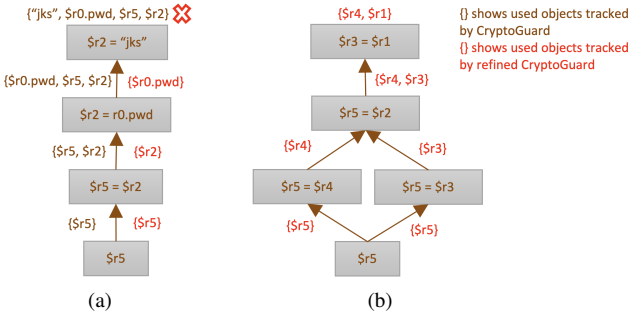

Fig. 1. Simplified intra-procedural def-use chains
(a) shows the def-use chain of Listing 2 targeting $r5
(b) shows the def-use chain for explaining our safe refinement in CryptoGuard

Pattern #2 – Incorrect string matching in data flow analysis (CryptoGuard). Another contributor to FPs in CryptoGuard is the string matching logic used to implement data flow analysis. Variable names, field names, and types of statements are often processed and matched as strings. However, we found that some of the string operations used in CryptoGuard do not actually perform what is needed by the analysis. Such problems can be classified into two cases: (a) mismatching field names; and (b) mismatching array variables.

Problem (a) explains the gap on rule 13 alarms that we observed between the latest stable release (commit id `92551ee`) and an older version of CryptoGuard (commit id `94135c5`). During our experiments, we noticed that the number of rule 13 alarms in our experiment is much larger than that of the original paper (which reported only 312 alarms for rule 13). The main difference between these two is that the latest stable release uses a new version of Soot. After reviewing and debugging the CryptoGuard code, we found that the gap in rule 13 alarms is caused by the problem of mismatching field names. In general, to enable field-sensitive analysis, CryptoGuard needs to match the field name (*e.g.*, `r0.<com.test.Crypto: java.lang.String defaultKey>`) in statements. However, the regular expression used[1] to match field names missed one common case: some variable names in the Jimple IR may begin with a dollar sign [36]. This bug led to many variables fail to match fields, thus also failing the field slicing. With the new version of Soot, less variable names are given the dollar sign prefix, thus CryptoGuard performs its field slicing with more success, which unfortunately outputs many more FPs due to the broken def-use chains (Pattern #1). Fixing the def-use chains mitigated these FPs when field slicing functions as expected.

For (b), CryptoGuard uses `toString().contains()`[2] to determine if the definition of an array variable matches any use of array variables in def-use chains. However, this is a bug that incorrectly expands the matching set. For example, `r1` is also a match for `r10` under the `contains` matching logic, thus some unrelated arrays will also be considered to have def-use relations during slicing. In our experiments, we observed 23 FPs due to this string matching bug.

Pattern #3 – Incorrect detection of hard-coded arrays due to a bug (CogniCrypt$_{SAST}$). To detect hard-coded passwords, CogniCrypt$_{SAST}$ defines a `notHardCoded` constraint for password-related classes such as `PBEKeySpec` and `KeyStore`. Interestingly, we found a bug in how CogniCrypt$_{SAST}$ determines if an array is hard-coded. Listing 3 shows an illustrative FP example, where a `KeyStore` object uses the return value of `getPassword()` as its password parameter in `KeyStore.load()`. Notice that `getPassword()` uses `SecureRandom` to generate a random char array as its return value. Surprisingly, CogniCrypt$_{SAST}$ reports this as a hard-coded constant password. We manually analyze the CogniCrypt$_{SAST}$ code to locate the bug, the details of which can be found in Appendix A-C. In our sampled data, we observed 24 HardCodedError FPs in `KeyStore`, and 28 FPs in `PBEKeySpec` due to this bug.

```
1  public void test() throws Exception {
2    KeyStore kS = KeyStore.getInstance(KeyStore.getDefaultType());
3    kS.load(null, getPassword());
4  }
5  public char[] getPassword() { // returns random char array
6    byte[] pass = new byte[256];
7    SecureRandom sR = new SecureRandom();
8    sR.nextBytes(pass);
9    return bytesToChars(pass);
10 }
```

Listing 3. An FP example of Pattern #3

Pattern #4 – Incorrect handling of call-return edges of CFG (CryptoREX). Although CryptoREX is based on angr [37], it develops its own CFG recovery algorithms instead of using that of angr. In short, CryptoREX sets up the call-return edges of a CFG, while angr sets up the call edges and return edges. The problem is that CryptoREX does not properly handle the call-return edges, *i.e.*, all function calls are ignored by CryptoREX during backward slicing. However, the side effect of function calls are still captured by CryptoREX, *i.e.*, parameters of function calls are stored in registers and stacks. Given that function calls are ignored but instructions for passing parameters are processed, CryptoREX is prone to anomalous results. An illustrative example can be found in Listing 4. In the decompiled code, variable `v1` (register `r0`) holds the return value of function `sub_464F40`, which is obviously not a constant value. However, CryptoREX incorrectly reports a rule 6 alarm, as it mistakes `4096` as the seed, which was passed to function `sub_5A7D4` as parameter `a1` via register `r0` on line 3. Through sampling, we observed that this bug led 1 FP for rule 3, 28 FPs for rule 4, and 113 FPs for rule 6.

```
1  int __fastcall sub_5E3D0(_DWORD *a1) {
2      ...
3      *(_WORD *)(v11 + 4) = sub_5A7D4(4096);
4      ...
5  }
6  __int64 sub_464F0() {
7      struct timeval tv; // [sp+0h] [bp-14h] BYREF
8      gettimeofday(&tv, 0);
9      return tv.tv_usec / 1000 + 1000LL * tv.tv_sec;
10 }
11 int __fastcall sub_5A7D4(int a1) {
12     unsigned int v1; // register r0
```

[1] https://github.com/CryptoGuardOSS/cryptoguard/blob/92551eeb/src/main/java/slicer/backward/method/MethodInstructionSlicer.java#L200

[2] https://github.com/CryptoGuardOSS/cryptoguard/blob/92551eeb/src/main/java/slicer/backward/method/MethodInstructionSlicer.java#L179

```
13      if ( byte_ADCEC != 1 ) {
14          v1 = sub_464F0();   srand(v1);   byte_ADCEC = 1;
15      }
16      ...
17  }
```

Listing 4. An FP example of CryptoREX's Rule 6

> *ID #1:* To implement effective and precise analysis, idiosyncrasies of the IR need to be carefully considered. Detectors also need thorough testing and evaluation, based on real-world datasets with sufficient sample size, to catch their own bugs. Although the bugs presented here are specific to individual detectors, this ID is generally applicable to future crypto-misuse detectors as well as vulnerability detectors in a more general setting.

Pattern #5 – Static typing information available but underutilized (CogniCrypt$_{SAST}$). We found that some of the TypestateError and IncompleteOperationError alarms reported by CogniCrypt$_{SAST}$ are caused by unresolved but statically resolvable polymorphism in typestate analysis. In Java, polymorphism can happen due to the use of abstract classes, inheritances, generics and interfaces. While it is impractical to expect static analysis to perfectly handle all polymorphism, however, some polymorphism can be resolved by utilizing static typing information available in the IR. However, in its typestate analysis, CogniCrypt$_{SAST}$ adopts Class Hierarchy Analysis (CHA) for its call graph construction, which includes all classes that implement the same interface, even when the instantiated class (static type) is already given in the IR. Similar problems also exist for generics. This imprecision of call graphs caused many false positives. Due to space constraints, we provide an illustrative example in Appendix A-D. In our experiment, we found at least 34 FPs for TypestateError and 80 FPs for IncompleteOperationError because of this.

> *ID #2:* We recommend using a more precise call graph construction algorithm with reasonable overhead to reduce false positives. There are other call graph construction algorithms, such as Rapid Type Analysis (RTA) [38] and Variable Type Analysis (VTA) [39], which scale linearly to the number of callsites and can better utilize static typing information. In terms of empirical performance, experiments from prior work suggest the runtime overhead of VTA and RTA can be comparable to that of CHA [38], [40], though the implementation framework of choice can also significantly influence the observed overhead [40].

VI. False Alarms due to Modeling

We now present the ITPs due to modeling issues in detectors, where the misuse rules themselves are reasonable. Unfortunately, false alarms can happen when the models are not designed to capture sufficient conditions of vulnerabilities.

Pattern #6 – Reasonable iteration counts considered insecure (CogniCrypt$_{SAST}$). Interestingly, CogniCrypt$_{SAST}$ requires the PBE iteration count to be at least 10000 in its `PBEKeySpec` and `PBEParameterSpec` rules. However, the acceptable lower bound[3] should actually be 1000 [1], [42]. We found 22 ConstraintError in `PBEKeySpec` (and 2 in `PBEParameterSpec`) to be false alarms because of this.

Pattern #7 – Reasonable key sizes considered insecure (CryptoGuard). Rule 15 of CryptoGuard detects insecure (short) key sizes of public-keys (*e.g.*, 1024-bit RSA). In CryptoGuard, this rule is checked in three steps: (1) find the creation of `java.security.KeyPairGenerator` for RSA or elliptic curve (EC); (2) use intra-procedural forward slicing to find whether each `KeyPairGenerator` object has its `initialize` method explicitly invoked to set the key size (if not, the default key sizes of RSA and EC are regarded as violations); (3) if `initialize` is explicitly called, use backward slicing to determine whether the algorithm and key size to be set are appropriate (CryptoGuard cited [43] for the lower bounds on key sizes). While the steps might seem reasonable, there are several subtle problems.

CryptoGuard requires RSA keys to be at least 2048-bit long, which approximates to 112-bit security [43] and is a reasonable lower bound. However, CryptoGuard requires EC keys to be at least 512-bit long, which is at a much higher security level (roughly equivalent to 15360-bit RSA) [43]. This means generating a 256-bit key for `ED25519` or a 384-bit key for `ECDSA-384` are all reported as vulnerabilities by CryptoGuard. Overall, we found 53.3% (16/30) of all the rule 15 alarms are ITPs because of this.

Furthermore, we tested Android versions 6 to 13 (API levels 23 – 33), and found that the default key sizes for RSA and EC are 2048-bit and 256-bit respectively, which means even the default key sizes for RSA and EC on Android are acceptable by current standard [43]. Thus, equating default key sizes with vulnerabilities, as done by CryptoGuard, could lead to additional false alarms when analyzing apps targeting API levels 23 or above.

> *ID #3:* Future detectors should justify their adopted lower bounds for different cryptographic algorithms by clearly citing the standards/recommendations of the time. Additionally, detectors can also consider reporting the reference documents together with the corresponding alarms, which can help developers evaluate the severity without introducing false negatives. This ID can be applied to any future cryptographic misuse detectors.

Pattern #8 – No key-pair generator equals to insecure public key (CryptoGuard). We found 8 additional rule 15 false alarms from CryptoGuard due to exception handling, a representative example of which can be found in the JJWT library[4]. In CryptoGuard, slicing is performed based on a directed graph (representing a method body) generated by Soot. A method may have more than one exit statement (*e.g.*, `throw`, `return`). However, CryptoGuard always takes the final statement in the directed graph as the slicing result for the entire method. In the context of rule 15, a method could catch the `NoSuchAlgorithmException` from `KeyPairGenerator.getInstance()`, and then exit. Notice that when `NoSuchAlgorithmException` is thrown, no valid `KeyPairGenerator` objects would be created by `getInstance()`, and thus no `initialize()` or

[3] Although the minimum iteration count is likely to increase in near future [41], here we are following the minimum requirements applicable back when the original papers were published.

[4] https://github.com/jwtk/jjwt/blob/f6aa291e/impl/src/main/java/io/jsonwebtoken/impl/crypto/RsaProvider.java#L186

genKeyPair() would be possible. In that case, reporting insecure public key leads to false alarms, because there is no generator to begin with, and no keys could be generated.

ID #4: To track whether an object indeed invokes a specific method, backward slicing should consider all branches, including normal return exits and thrown exceptions, and if the target object is not created due to exception from methods like getInstance(), a misuse alarm is not necessary. Notice that this only removes the cases when no KeyPairGenerator object is initiated due to the exception, and thus will not introduce false negatives. Although we observed false alarms from CryptoGuard, exception branching leading to non-existent objects is a general pattern faced by static vulnerability detectors in other settings as well.

Pattern #9 – Constant seeds assumed to always make outputs of SecureRandom predictable (CryptoGuard). Rule 8 of CryptoGuard concerns the use of constant seed with the java.security.SecureRandom class. While the intention is reasonable, however, whether constant seeds pose a security threat (lead to predictable PRNG outputs) depends upon several nuanced factors, including the target platform, API call sequence, and choice of PRNG implementation.

According to the API documentation [44], [45], once an instance of SecureRandom is already properly seeded (*e.g.*, the self-seeding induced by calling nextBytes() before setSeed()), subsequent calls to the setSeed() method *supplements* rather than replaces the existing seed, and thus do not pose a security threat even if constant seeds are used.

Furthermore, even if an instance of SecureRandom has not been seeded before, the choice of PRNG implementation also affects whether a given constant seed leads to the same PRNG outputs. For example, on typical Unix-like systems that are equipped with /dev/random or /dev/urandom, with the default Sun provider, variants of NativePRNG are available and used by default [46]. When a NativePRNG variant is used, the seed given will not be the sole source of randomness that SecureRandom relies upon. In such cases, even if a constant seed is used, the PRNG outputs are still random. Nevertheless, on typical non-Android Java environments, using the SHA1PRNG or DRBG implementations instead of NativePRNG could lead to the same outputs when constant seeds are set via setSeed() prior to self-seeding.

Things are a little different and confusing on Android, however. Since version 4.2, the SecureRandom class defaults to an OpenSSL PRNG implementation from the AndroidOpenSSL (a.k.a Conscrypt) provider, where a given seed is always used as a supplement and not the sole randomness relied by the PRNG, and thus even constant seeds will not lead to the same PRNG outputs [47]. However, later it was found that the OpenSSL PRNG implementation on Android versions below 4.4 had a bug, and thus applications that directly invoke an instance of OpenSSL PRNG should explicitly initialize it with entropy from /dev/urandom or /dev/random to work around the bug [48]. On Android 6 or above, when the provider is not specified, it defaults to Conscrypt, in which SHA1PRNG is simply an alias of OpenSSL PRNG. The Bouncy Castle-based Crypto provider that provides the old SHA1PRNG implementation (which gives the same outputs under a constant seed) has been deprecated in Android 7 [49]. However, for apps targeting Android 6 or below, one can still request the old SHA1PRNG by specifying the Crypto provider [49]. Table V below summarizes the behavior of SecureRandom under different implementation on various target platforms.

TABLE V. IMPACT OF CONSTANT SEEDS ON SECURERANDOM UNDER DIFFERENT IMPLEMENTATIONS ON VARIOUS PLATFORMS

Java App Target	PRNG Implementation	SecureRandom behavior with constant seeds
Android $\leq$ 4.1	*Default* = SHA1PRNG	same outputs if not properly seeded before [45], [47]
Android 4.1	OpenSSL PRNG	random outputs [47] (but has a bug [48])
Android 4.2 & 4.3	*Default* = OpenSSL PRNG [47]	random outputs [47] (but has a bug [48])
Android $\geq$ 4.4	*Default* = OpenSSL PRNG [47]	random outputs [47]
Android $\leq$ 6	SHA1PRNG (Crypto)	same outputs if not properly seeded before [45], [47], [49]
Android $\geq$ 6	SHA1PRNG (Conscrypt) = OpenSSL PRNG [49]	random outputs [47], [49]
Unix-like	*Default*	depends on configuration [46] (same as one of the below)
Unix-like	variants of NativePRNG	random outputs
Unix-like	SHA1PRNG / DRBG	same outputs if not properly seeded before [44]

In our experiments, we found that CryptoGuard reported 81 apps to have rule 8 violations. To estimate a lower bound of false alarms, we used the minSdkVersion declared in app manifests [50] to detect obvious cases of rule 8 false alarms. With this, we found that 15 of the 81 apps have a minSdkVersion of Android 7[5] or above, where constant seeds never induce predictable outputs. More false alarms might exist due to specific choices of PRNG implementation as well as API call sequences that induce self-seeding.

ID #5: The effect of constant seeds varies due to several factors. Future detectors can utilize the target platform information (*e.g.*, from the Android app manifest) and target specific vulnerable API call sequences. Additionally, detectors can also consider presenting its alarms with labels of high-confidence. For instance, if a rule and its model capture sufficient conditions of a vulnerability (in this case, vulnerable API call sequences on specific platform-implementation combinations), then the misuse alarm can be given a high-confidence label. The other potential misuses (*e.g.*, other constant seeds) can then be reported without that label. This can help developers prioritize misuse alarms without introducing false negatives. This ID can be applied to any static or dynamic misuse detectors, as well as the remaining false alarm patterns discussed in this paper.

Pattern #10 – Narrow whitelist constraints in detecting MITM issues (CryptoGuard). Rules 4–6 of CryptoGuard catch misuses that open door to MITM attacks by checking if the implementation complies with some predefined whitelist constraints. However, we find the whitelists overly narrow, leading to false alarms.

Rule 4 catches hostname verifiers that accept all hosts. Specifically, its whitelist pattern requires any classes implementing the HostnameVerifier interface to have a verify() method where the SSLSession parameter *influences* the return value through *def-use relations*. However, we note that SSLSession can exert its influence via control flow instead of def-use relations. A typical ITP pattern of this can be found in Listing 5. The return value is either true or

[5]We note that Android 7 was released back in 2016, and thus this is not unfair to a detector released in 2019.

`false`, and it does not have any def-use relations with other statements. Nonetheless, the `session` parameter influences the `return` value via the `if` statement. We found 7 false alarms that follow this pattern in our sampling exercise.

```
public class Verifier implements HostnameVerifier {
  ...
  @Override
  public boolean verify(String hname, SSLSession session) {
    HostnameVerifier dHNV = HttpsURLConnection.
        getDefaultHostnameVerifier();
    if (dHNV.verify(hname, session)) { return true; }
    return false;
  }
}
```

Listing 5. A false alarm example of CryptoGuard's rule 4

Rule 5 detects custom `TrustManager` that accepts all certificates. The `X509TrustManager` interface has three methods to implement: `checkClientTrusted`, `checkServerTrusted`, and `getAcceptedIssuers`. In CryptoGuard, rule 5 is made of 3 subrules [1], which we refer to as rule 5-1, 5-2, and 5-3. Violating any of the 3 subrules would lead to an alarm. Rule 5-2 is a blacklist rule that detects unpinned self-signed certificate with an expiration check, but it does not raise any alarms due to a bug, which is a false negative that will be discussed further in Section X. Here we focus on rule 5-1 and 5-3.

Rule 5-1's model requires the `CertificateException` to be thrown by *both* the `checkClientTrusted()` method and the `checkServerTrusted()` method. This is at odds with the original CryptoGuard paper [1], which only mentions checking `checkServerTrusted()`, but the detector actually reports an alarm when any of `checkClientTrusted()` and `checkServerTrusted()` never throw the expected exception. Because of this, we found many ITPs concerning dummy `checkClientTrusted()` methods, which is common in typical TLS deployments where only one-way certificate validation (client validating server certificate) is needed.

In an attempt to gauge the scale of rule 5-1 false alarms, we modified CryptoGuard to output different alarms for `checkClientTrusted()` and `checkServerTrusted()`. During this exercise, we found another implementation bug[6] that makes the `checkServerTrusted()` method never getting checked. Because CryptoGuard uses a `Map` to hold the slicing criteria, the rule 5-2 criteria actually overwrites that of rule 5-1 due to a duplicate key, thus the exception check for `checkServerTrusted()` never happens. Thus, all rule 5-1 alarms are caused by `checkClientTrusted()` alone.

Rule 5-3's model requires the `getAcceptedIssuers()` method of a *custom* `X509TrustManager` implementation to call and return the `getAcceptedIssuers()` method of a *built-in* instance of `X509TrustManager`. However, this captures neither a sufficient nor a necessary condition of the target vulnerability (*i.e.*, accepting all certificates):

Case 1: even if the *custom* `getAcceptedIssuers()` method returns null or returns an empty array, the *custom* `checkServerTrusted()` method, which actually determines the trustworthiness of the server certificate, can still implement proper certificate validation without calling the custom `getAcceptedIssuers()`, by either (i) loading trusted CA certificates directly from `KeyStore`, or (ii) pinning specific issuer certificates directly in its code, or (iii) calling methods (*e.g.*, `getAcceptedIssuers()` or `checkServerTrusted()`) of a built-in `X509TrustManager`. Options (i) and (ii) can also be implemented in the custom `getAcceptedIssuers()`, but would still violate rule 5-3.

Case 2: even if the custom `getAcceptedIssuers()` method invokes and returns the `getAcceptedIssuers()` of a built-in `X509TrustManager` (thus satisfying the rule 5-3 whitelist), the custom `checkServerTrusted()` can still blindly accepts any certificates.

We found that all 125 rule 5-3 violations we sampled are false alarms due to the aforementioned Case 1.

```
SSLSocketFactory fac = (SSLSocketFactory) SSLSocketFactory.
    getDefault();
SSLSocket so = (SSLSocket) fac.createSocket("g.co", 443);
HostnameVerifier ver = HttpsURLConnection.
    getDefaultHostnameVerifier();
if(!ver.verify(so.getSession().getPeerHost(), so.getSession()))
  { threw CertificateException("Hostname mismatch!"); }
```

Listing 6. Sample code that satisfies rule 6 whitelist pattern

Rule 6 defines a whitelist constraint of hostname verification for `SSLSocket`. Listing 6 shows an example that complies with this constraint: after creating an `SSLSocket` object through the `createSocket()` method, the `verify()` method of a `HostnameVerifier` object must be invoked, with the second parameter being the `SSLSession` derived from the `SSLSocket` object, and the return value of `verify()` must be used in the condition of an `if` statement. However, false alarms can happen due to various reasons: (1) there are other ways to enable hostname verification; (2) server identity can be confirmed without checking hostname (*e.g.*, by pinning certificate of a specific server or private CA). In the end, we found 15 false alarms due to (1), but did not observe examples of (2). A typical example we observed in app code is to perform hostname verification in other places, for example, as a part of the custom implementation of `checkServerTrusted()`, effectively verifying *both* the certificate chain and hostname in that method. Another example concerns the use of a different API method to enable hostname verification. Since Android 7 (API level 24), the `setEndpointIdentificationAlgorithm()` method [51] was introduced in the `SSLParameters` class. One can use this to set up an SSL socket with hostname verification enabled.

Pattern #11 – All IVs must come from SecureRandom, even for decryption (CogniCrypt$_{SAST}$). This is an interesting case of CogniCrypt$_{SAST}$'s RequiredPredicateError, which detects constant IVs and salts, among other misuses. The corresponding whitelist rules for `PBEKeySpec`, `PBEParameterSpec`, `IvParameterSpec` require the IVs and salts to be generated from `SecureRandom`. While this might make sense for encryption, it is obviously not possible for decryption to freshly generate new random IVs. We observed at least 280 false alarms because of this. Likewise, in AES-SIV, a synthetic IV is meant to be generated from input instead of `SecureRandom`. We observed at least 37 false alarms caused by this.

Pattern #12 – Legitimate origins of key materials prohibited

[6] https://github.com/CryptoGuardOSS/cryptoguard/blob/92551eeb/src/main/java/rule/CustomTrustManagerFinder.java#L40

(CogniCrypt$_{SAST}$). The rule `SecretKeySpec.crysl` used by CogniCrypt$_{SAST}$ has a predicate that requires the byte array given to the constructor of `SecretKeySpec` to be obtained from an existing `Key` or `SecretKey` object. This is an overly restrictive whitelist rule, because it can be violated by loading the bytes from a key file, or by generating a byte array with `SecureRandom`. We also observed that libraries such as Google Tink [52] define some custom classes for holding keys, and constructing `SecretKeySpec` with byte arrays from them also violates the whitelist rule. We observed 549 false alarms of `RequiredPredicateError` due to this pattern.

Pattern #13 – Whitelist constraints ignore idiosyncrasies of Android (CogniCrypt$_{SAST}$). CogniCrypt$_{SAST}$ has a set of rules designed for Java Cryptography Architecture (JCA), and also uses it to analyze Android apps [2]. We found many ITPs in Android apps due to the rule set not considering idiosyncrasies of the Android API. For example, the whitelist constraints for the `KeyStore` class omitted some common key store names on Android, such as, `AndroidKeyStore`, `BKS`, and `AndroidCAStore`. Such omissions caused 96.2% (331/344) of the `KeyStore` constraint errors reported by CogniCrypt$_{SAST}$ to be ITPs. Similarly, the constraints for `TrustManagerFactory` omitted a common factory name on Android (*i.e.*, `X509`), thus all 46 of the reported constraint errors for `TrustManagerFactory` are ITPs. Moreover, `PKCS7Padding` is an alias of `PKCS5Padding` on Android, but the former is excluded in the whitelist constraints for the `Cipher` class, thus 20.9% (150/716) of all the ConstraintError for `Cipher` are ITPs.

Another interesting example concerns the protocol string given to `SSLContext.getInstance()`. CogniCrypt$_{SAST}$ requires the string to be either `TLSv1.2` or `TLSv1.3`. Although the intention of disabling older versions of TLS is understandable, however, based on the results of our experiments (Table VIII in Appendix A-B), many other protocol strings enable the same set of TLS versions. We found that 92.6% (1704/1840) of all the ConstraintError reported for `SSLContext` are ITPs because of this. To actually disable specific versions of TLS, the constraints should instead be imposed on `SSLSocket.setEnabledProtocols()`.

Additionally, we find that on Android, the `init` method of both `SSLContext` and `TrustManagerFactory` can take `null` parameters, in which case the call will fall back to secure defaults supported by the platform, but nonetheless trigger the RequiredPredicateError. This led to 720 ITPs for `TrustManagerFactory` and 1130 ITPs for `SSLContext`.

Moreover, CogniCrypt$_{SAST}$ requires passwords to never take the `String` type, otherwise it reports the NeverTypeOfError. Likewise, some IncompleteOperationError rules are designed to catch `PBEKeySpec` objects that do not call the `clearPassword()` method. While these might be important on conventional platforms to prevent password compromise through memory dumping attacks, Android has application sandbox [53] to isolate app resources such as memory. Thus, it is not clear under which threat model would the 722 NeverTypeOfError alarms and the 69 `PBEKeySpec` IncompleteOperationError due to not calling `clearPassword()` constitute misuses that need to be fixed in Android apps.

Pattern #14 – Seeding with srandom not allowed (CryptoREX). Rule 6 of CryptoREX uses the `nm -D <filename>` command to check if `rand` and `srand` are both used by a binary. If not, then it reports a violation, as calling `rand` without seeding gives the same outputs. Interestingly, among the 22 binaries sampled for rule 6, we found `hostapd`, `wpa_supplicant`, and two closed-source binaries using `srandom` instead of `srand` to seed the PRNG. CryptoREX reports them as misuses, despite `srandom` being functionally equivalent to `srand`. After modifying CryptoREX to consider `srandom`, the reported number of rule 6 violations dropped from 133 to 116.

> *ID #6:* Whitelists require careful curation to capture common legitimate programming patterns. These patterns can be discovered by mining and testing code repositories. Also, due to the subtle differences between Android and conventional Java, we advocate curating a whitelist tailor-made for Android, instead of directly reusing the one for conventional Java. When the whitelist only contains patterns that capture sufficient conditions of non-vulnerability, which can be learned from repository mining, false alarms can be avoided without introducing false negatives. This ID is generally applicable to any static or dynamic detectors.

VII. False Alarms due to Usage Contexts

Now we discuss ITPs due to overly conservative misuse rules. Despite the best intentions, some of the misuse rules also do not capture sufficient conditions of vulnerabilities, resulting in various false alarms.

Pattern #15 – All usage of AES-ECB considered insecure (CryptoGuard, CogniCrypt$_{SAST}$, CryptoREX). For CryptoGuard's rule 11, we found 57 alarms concern the use of `AES/ECB/NOPADDING`. Upon closer inspection, 22 (38%) of them are in fact ITPs. The reason is that AES-ECB can be used as the raw AES block cipher for implementing other secure modes of operation. This is often necessary because the standard Java API does not have a separate method for the raw AES block cipher. We observed this in one application that uses AES-ECB to implement AES-OCB (2 false alarms). Another application contains Google's Tink library [52], which uses AES-ECB to implement AES-EAX (3 false alarms). In fact, this usage of AES-ECB was also detected and reported by an industrial tool (CodeSafe from Qianxin), and got dismissed by the Tink developers[7]. Additionally, we also observed Bitcoin wallet implementations use AES-ECB as the raw AES block cipher to implement the BIP38 [54] standard, which is a method for encrypting Bitcoin private keys (17 false alarms). We exchanged emails with a developer of the open-source bitcoinj library, who opined that this usage is intentional and not a vulnerability. CogniCrypt$_{SAST}$ reports similar false alarms, as it also detects all usages of AES-ECB.

Likewise, CryptoREX's rule 1 also prohibits any use of AES-ECB. Interestingly, it also considers any use of the raw block cipher function `AES_encrypt` from OpenSSL to be insecure. Out of its 566 rule 1 alarms, 140 are found to be ITPs. An example can be found in a recent version of `hostap`[8], where the AES-ECB functions from the OpenSSL EVP interface are mapped to its internal interface, which

[7] https://github.com/google/tink/issues/246

[8] https://w1.fi/cgit/hostap/tree/src/crypto/crypto_openssl.c?h=hostap_2_10#n372

will be used to implement other AES modes. The raw block cipher `AES_encrypt` is used by older versions of `hostap`[9] for the same purpose, as well as in OpenSSH[10], which uses `AES_encrypt` to implement the UMAC algorithm [55].

Pattern #16 – All usages of non-CSPRNGs considered vulnerabilities (CryptoGuard, CryptoREX). With a total of 9042 alarms, CryptoGuard's rule 9, which detects any usage of `java.util.Random`, has the highest number of alarms among its 16 rules. Upon closer inspection, we found 2602 cases in the top-10 offending methods to be ITPs. Cryptographically secure pseudo-random number generators (CSPRNGs) are required when implementing algorithms that expect both the *uniformity* and *unpredictability* properties from the underlying PRNG. However, non-CSPRNGs are often used where *unpredictiability* is not needed. For instance, a top contributor of rule 9 alarms is the use of non-CSPRNG in the `Explode` transition (UI animation) on Android[11]. Another example include picking a random track in a media player (*e.g.*, ExoPlayer[12]). Additionally, for some problems (e.g., finding square root and verifying matrix multiplications), there is no efficient deterministic algorithm known. Therefore, developers utilize probabilistic algorithms to solve these problems. In many such cases, *unpredictability* is not necessary, and the *uniformity* of non-CSPRNGs suffices. For example, a non-CSPRNG can be used in the Tonelli and Shanks (T&S) algorithm [56] for finding the square root modulo p. Similarly, the uniformity of non-CSPRNG is sufficient in solving quadratic equation in $\mathbb{F}_{2^n}$ [57]. Both of these are prominent examples of false alarms found in Bouncy Castle (and Spongy Castle) cryptographic library, with 112 ITPs due to the T&S algorithm, and 246 ITPs due to solving quadratic equations in $\mathbb{F}_{2^n}$.

Furthermore, we found apps written in Kotlin triggered many rule 9 alarms. A closer examination revealed that this is due to the implementations of certain Kotlin features either map to or use the non-CSPRNG from Java. For instance, the `PlatformRandom.kt` from the Kotlin standard library (stdlib) maps Kotlin's `Random` to Java's `java.util.Random`. Thus, any apps written in Kotlin that use the stdlib will be reported by CryptoGuard to have rule 9 vulnerabilities.

CryptoREX's rule 6 has similar problems, as the non-CSPRNG function `rand()` can be used in diverse contexts, many of which are unlikely to be vulnerabilities. For example, in a closed-source binary `funjsq_dl`, `rand()` is used to generate a random rollback time for different log files to avoid too many simultaneous rollback operations. Through manual sampling, we observed 272 false alarms due to `rand()` being used in non-security-critical contexts, which could be grouped into 4 high-level cases. Due to space constraints, we give a more detailed account in Appendix A-E.

In contrast, CogniCrypt$_{SAST}$ does not follow the same approach of detecting all calls to non-CSPRNGs, and thus avoided many false alarms. Instead, its whitelist rules focus on specific scenarios where a CSPRNG might be needed.

Pattern #17 – All `http://` considered vulnerabilities (CryptoGuard). CryptoGuard's rule 7 catches URLs that start with `http://`. While the motivation is clear, many resulting alarms are difficult for developers to act upon. There are in total 1645 alarms for rule 7, and the 610 alarms from the top-10 contributing methods all appear to be ITPs, where the URLs are constructed for local loopback (`localhost`), other debugging purposes, or as placeholders. Prominent examples of debug URLs can be found in the `DevServerHelper` class of the React Native framework from Facebook, where various methods construct URLs of the debug server (the host machine of an Android emulator) with `http://` for debugging purposes, contributing 122 alarms. Additionally, we found placeholder URLs such as `http://undefined/` and `http://example.com/`, which do not seem to be endpoints of meaningful communication. In fact, across all the rule 7 alarms, the `localhost` case alone contributed 477 (29.0%) alarms. It is not clear how an attacker can sniff such traffic without first compromising the device, which is much stronger than the typical threat model of a network MITM.

Pattern #18 – All usage of collision-prone hash functions considered vulnerabilities (CryptoGuard). Rule 16 of CryptoGuard catches the use of hash functions that are perceived as weak. The use of collision-prone hash functions, however, does not always constitute vulnerabilities, so long as the usage scenario is not affected by potential collisions, or that a collision resolution mechanism is already in place (*e.g.*, separate chaining in a hash table). Through manual sampling, we found 72 false alarms because of that. Specifically, the Facebook library uses MD5 to hash files to be reported in its error messages, which were identified as misuses in 31 apps. Similarly, some libraries also employ MD5 as a checksum to detect unintentional corruption while fetching an image (16 alarms), mp3 file (8 alarms) or general files (7 alarms). Additionally, another library utilizes MD5 digests as the lookup index in a local cache, resulting in 10 alarms.

ID #7: AES-ECB, `http://`, non-CSPRNG, and collision-prone hash functions have legitimate usages where they provide sufficient guarantees and desirable performance. We recommend detectors to target specific usages that are indeed vulnerable under a well-defined threat model. If concerns over non-CSPRNGs are mainly on key generation, then detectors should target keys that might be generated by non-CSPRNGs. This is doable, as shown by CogniCrypt$_{SAST}$. Likewise, SHA-1 is believed to be second-preimage resistant despite not collision-resistant [58], [59], which can still be used to build other secure cryptographic constructs. For example, the use of SHA-1 in hash-based message authentication code (HMAC) continued to be approved by NIST in 2020 (Table 3 of [60]). If concerns over collision-prone hash functions are mainly on their usage in digital signatures, then detectors could target API methods for signing. This principle of focusing more on sufficient conditions and tightening the decision boundary can also apply to other vulnerability detectors, dynamic or static.

[9] https://w1.fi/cgit/hostap/tree/src/crypto/crypto_openssl.c?h=hostap_0_7_2#n226

[10] https://github.com/openssh/openssh-portable/blob/5796bf8c/umac.c#L165

[11] https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/transition/Explode.java#150

[12] https://github.com/google/ExoPlayer/blob/03569f9e/library/core/src/main/java/com/google/android/exoplayer2/trackselection/RandomTrackSelection.java

[13] https://github.com/google/tink/blob/8f6316b9/java_src/src/main/java/com/google/crypto/tink/subtle/AesEaxJce.java#L103

VIII. False alarms due to standard mandates

We now discuss the ITPs caused by protocol standard mandates. In our investigations of the corresponding standard documents, we only consider keywords such as `MUST` and `SHALL` [61] as mandatory. Other keywords such as `SHOULD` and `MAY` are treated as options and are thus not used to classify alarms as ITPs. We note that some of the mandated usages might constitute more realistic threats than the others, nevertheless, they help to explain why developers write and keep code that looks like misuses in the apps and libraries.

Pattern #19 – Protocol standards mandate the use of weak algorithms, modes, and constants. Another cause of ITPs concerning CryptoGuard's rule 1, 10–14, and 16, is that some protocol standards actually require the use of algorithms, modes, and constants that are perceived to be weak, and thus even standard-compliant implementations will trigger the alarms. We consider such alarms ineffectual, as the developers (app or library) did not make a mistake or misuse on their own, and they are not in a position to arbitrarily change the design choices stipulated in protocol standards.

Examples of this are plentiful. For instance, DNSSEC signers are not recommended to use SHA-1 in signing [62], yet implementations must continue to support validation of `RSASHA1` signatures [62]. Because of this, we observed 30 ITPs for CryptoGuard's rule 16 contributed by apps that contain DNSSEC libraries such as `org.minidns.dnssec` and `org.xbill.dns`. Likewise, to implement support for Bitcoin Script [63], wallet apps have to use SHA-1, as it is one of the opcodes. This led to 8 ITPs for CryptoGuard's rule 16. Moreover, we found that `AES/ECB/NOPADDING` is also needed for decrypting and validating the permission string in encrypted Adobe PDF (Algorithm 3.13 of [13]). This led to 2 additional ITPs for CryptoGuard's rule 11.

For CryptoGuard's rule 13, Bouncy Castle (and Spongy Castle) has code that does password-based encryption (PBE) with iteration count set to one[14], triggering 31 alarms. We found that this is to support reading encrypted private keys from the old OpenSSL format [64]. One might argue this format should be designed better, but developers are not in a position to unilaterally increase the iteration count on their own, without breaking compatibility with this legacy format.

Some alarms of CryptoGuard's rule 12 are also caused by standard mandates. For example, we observed 32 ITPs from 3 apps that implement the Machine Readable Travel Documents (MRTDs) standard [65], which includes a Password Authenticated Connection Establishment (PACE) protocol. The PACE definition requires a zero IV to be use with 3DES-CBC, and an IV of -1 to be used with AES-CBC [65], which triggered the false alarms. Similar problems also affect CryptoREX's rule 2, which detects constant IVs in CBC mode of block ciphers. An example is the Apple Filing Protocol (AFP) [66]. AFP supports several user authentication modules, one of which is called DHX2. In DHX2, the IVs are defined as fixed values [67], thus triggering 6 alarms of CryptoREX's rule 2. In any case, developers cannot unilaterally replace these constant IVs with random numbers, otherwise interoperability will be broken.

[14] https://github.com/bcgit/bc-java/blob/40678655/pkix/src/main/java/org/bouncycastle/openssl/jcajce/PEMUtilities.java#L326

Another large contributor to CryptoGuard's rule 11, 14 and 16 alarms, is the Apache HttpClient library which implements support for the New Technology LAN Manager (NTLM) protocol suite [68], which uses MD5, RC4 and DES-ECB[15], and provides single sign-on (SSO) in Microsoft environments. Overall, this contributes to 56 alarms of CryptoGuard's rule 2, and 32.5% (233/716) of its rule 11 and 14 alarms[16]. Likewise, for CryptoREX, NTLM also contributed 478 rule 1 alarms, due to its use of DES-ECB. Although the cryptographic algorithms used in NTLM are known to be weak, they cannot be unilaterally changed by developers. A more meaningful but orthogonal discussion is whether the NTLM protocol suite should be deprecated, though the exploitability of NTLM with HTTP over TLS depends upon the threat model, and protocol deprecation efforts require vendor buy-in and coordination, which might explain why the likes of NTLM are kept in some libraries for compatibility reasons.

ID #8: Developers are sometimes bound by standard mandates to use certain algorithms and constants. As a partial refinement, one can extract class/method names known to be implementing such standards, and incorporate them in a misuse alarm filter. If done at the level of targeting specific classes/methods that implement known standards (instead of whitelisting all usage of algorithms/constants in an undiscriminating manner), this should not lead to false negatives. Alternatively, detectors can also consider reporting those classes/methods with a low-confidence label (see ID #5). Protocol/standard mandates can be discovered in a manner similar to repository mining (see ID #6), to better understand why developers' hands are tied. This ID should be generally applicable to other misuse detectors.

IX. Generalizability of false alarm patterns

While the false alarm patterns are distilled from three academic static detectors, in this section, we demonstrate and discuss their generalizability in other tools.

Due to the labor intensity of conducting root cause analysis of false alarms, we evaluate one popular industrial tool, SpotBugs [69] with the Find Security Bugs plugin (hereafter referred to as FindSecBugs, commit id `4760fea`) [12], which is open-source. FindSecBugs is an actively maintained industrial tool, which also targets security bugs related to cryptographic APIs, making it ideal for investigating the generalizability of the false alarm patterns discussed in previous sections. While FindSecBugs has a list of 141 bug patterns for security weaknesses [70], for ease of comparison, we only consider the ones that overlap with what we investigated in the three academic detectors. The results are shown in Table VI.

Similar to CryptoGuard (Pattern #10), we found that FindSecBugs also does not capture sufficient or necessary condition of vulnerability in its detection of insecure `TrustManager` and `HostnameVerifier`. Specifically, FindSecBugs does not use any data flow analysis to model the condition of return value in the methods of `TrustManager` and `HostnameVerifier`.

[15] The use of ECB mode DES in NTLM was classified as *functional false positives* by a previous work [34].

[16] In CryptoGuard, violations of rule 11 and 14 are reported together. Also, CryptoGuard does not perform reachability analysis, thus support of NTLM in a 3rd-party library does not imply the apps are indeed using NTLM.

Instead, the condition is whether the targeted method (e.g., `verify()`) is empty, which is checked by verifying that there are no statements of invocation or field loading[17]. In this case, a simple `verify()` method that always returns `false`, which rejects any hostnames, would be identified as an insecure implementation of `HostnameVerifier`.

Unsurprisingly, other rules in FindSecBugs, such as prohibiting all ECB, SHA-1, and MD5, also do not consider any contexts, akin to the three academic detectors we evaluated (see Pattern #15, #18), and thus result in similar ITPs. However, due to the absence of data flow analysis in FindSecBugs, none of the false alarms on hard-coded keys and short cryptographic keys discussed above are found in FindSecBugs.

An interesting surprise, however, is that FindSecBugs actually escapes the decryption context when detecting static IVs[18], which reduces false alarms. Although FindSecBugs does not use data flow analysis to discover static IV values (and can thus have false negatives), this approach of considering decryption is in sharp contrast to the CogniCrypt$_{SAST}$ model which requires fresh random IVs in both encryption and decryption (Pattern #11). This in turn shows the feasibility of ID #7.

TABLE VI. FALSE ALARMS PATTERNS IN FINDSECBUGS

Original FindSecBugs Patterns [70]	How is the pattern being modeled?	False Alarm Patterns
TrustManager that accept any certificates	report bug when there are no statements of invocation or field loading in the methods `checkClientTrusted()`, `checkServerTrusted()` or `getAcceptedIssuers()` of a class implementing `X509TrustManager`.	Pattern #10 applies
HostnameVerifier that accept any signed certificates	report bug when there are no statements of invocation or field loading in the method `verify()` of a class implementing `HostnameVerifier`.	Pattern #10 applies
SHA-1 is a weak hash function	report any `MessageDigest.getInstance("SHA-1");`	Pattern #18 applies
MD5 are weak hash functions	report any `MessageDigest.getInstance("MD5");`	Pattern #18 applies
Weak SSLContext	report any `SSLContext.getInstance("SSL");`	Pattern #13 applies
DES is insecure	report any `Cipher.getInstance("DES/...");`	Pattern #19 applies
DESede is insecure	report any `Cipher.getInstance("DESede/...");`	Pattern #19 applies
Hard-coded key	mark constant values as hardcoded in a method and report bug when methods `PBEKeySpec` or `SecretKeySpec` initialized using the marked values.	/
Static IV	report static IV when there is no `Cipher.getIV()`, no `SecureRandom.nextBytes()`, and the cipher mode is not decryption in a method initializing `IvParameterSpec`.	Avoided Pattern #11
ECB mode is insecure	report any `Cipher.getInstance("AES/ECB/NoPadding");`	Pattern #15 applies
RSA usage with short key	report any constant `key_size` $<$ 2048 in `KeyPairGenerator.initialize(key_size)`	/

Furthermore, we looked into a recent work, Crylogger [7], a dynamic detector that analyzes execution logs collected in an instrumented environment. Running it through the same F-Droid app dataset, we noticed that the problem of not distinguishing decryption from encryption (Pattern #11) also applies to Crylogger's rule 5 and 7, which shows that this is indeed a rule modeling problem not limited only to static analysis. On the other hand, Crylogger avoids the AES-EAX false alarms (Pattern #15) because it checks the number of encryption blocks, which can be seen as an example of the refinement proposed in ID #7. We note that there are additional false alarms from Crylogger due to how it detects predictable keys, for which we give a detailed account in Appendix A-F.

Finally, some of the false alarm patterns discussed in Sections VII and VIII likely also apply to other academic and industry detectors. A recent work that uses a hybrid approach to find vulnerabilities in binaries [71] also detects any constant seeds used in PRNG regardless of context, and we conjecture that some legitimate usages are also reported as vulnerabilities. Likewise, some overly conservative rules discussed in Section VII are also enforced by a recent work [9].

X. DISCUSSIONS

Threats to validity. We acknowledge our manual sampling is a best-effort investigation, and a different group of researchers might not arrive at the exact same labeling as ours. As such, in this paper, we focus mostly on the false alarm patterns with root causes that tend to be stable and reproducible (*e.g.*, erroneous data flows, narrow whitelists, legitimate usages, and standard mandates), and put less emphasis on the overall percentage of false alarms in the entire data set. Given the lack of a false alarm oracle (or a detector with perfect precision and recall), we consider this an acceptable compromise in exploring the nuances of cryptographic misuses.

False negatives. While sampling misuse reports, we also observed some interesting false negative patterns. An obvious example is CryptoGuard's rule 5-1 discussed in Pattern #10. Additionally, we found that the functionality of its rule 5-2 also becomes ineffective due to a bug in string matching the slicing criteria. The slicing algorithm for rule 5 uses the `String.startsWith()` method to decide whether it met a statement that matches the slicing criteria. It works correctly when the slicing criteria are `throw` in rule 5-1 or `return` in rule 5-3. However, for the slicing criteria `checkValidity()` in rule 5-2, it can never be matched by `String.startsWith()` because the Jimple IR code[19] does not start with the slicing criteria. Thus, rule 5-2 actually never gets enforced, leading to false negatives.

Another bug we found in CryptoGuard while investigating false alarm Pattern #2 is that the variable names in Jimple IR are assumed to always contain only a single digit, since CryptoGuard's extraction of fields uses `substring(3)`[20]. However, when the variable name contains two digits (*e.g.*, `r10.<class field>`), the field slicing will also fail, which could in turn lead to false negatives.

Another interesting false negative is that CryptoGuard only raises alarms for key sizes of even number[21]. Thus, one can use RSA with a short key size in odd number (*e.g.*, 1023-bit, which is functional in practice) without triggering alarms.

Finally, since static detectors heavily rely on matching class names when applying various rules, one can potentially evade

[17] https://github.com/find-sec-bugs/find-sec-bugs/blob/3dd2f25/findsecbugs-plugin/src/main/java/com/h3xstream/findsecbugs/crypto/WeakTrustManagerDetector.java

[18] https://github.com/find-sec-bugs/find-sec-bugs/blob/3dd2f25/findsecbugs-plugin/src/main/java/com/h3xstream/findsecbugs/crypto/StaticIvDetector.java

[19] `virtualinvoke $r3.<java.security.cert.X509Certificate: void checkValidity()>()`

[20] https://github.com/CryptoGuardOSS/cryptoguard/blob/92551eeb/src/main/java/slicer/backward/method/MethodInstructionSlicer.java#L203

[21] https://github.com/CryptoGuardOSS/cryptoguard/blob/92551eeb/src/main/java/rule/ExportGradeKeyInitializationFinder.java#L208

detection by using custom wrapper classes that `extends` the standard classes without overriding any methods. This can be seen as a dual of the false alarms caused by the custom key classes of Google Tink discussed in Pattern #12.

Usability. In addition to the precision problem, root cause analysis is another major usability challenge, which can also render misuse alarms less actionable to developers. In our investigation, we often encounter cases where the offending method reported is not where the perceived misuse actually happened. Consider a simple example where method `a()` has a constant string `x`, which is then passed to method `b()`, and eventually passed to method `c()`, where it is used as a secret key. In many cases, the detectors would report either `c()` or `a()` as the sole offending method of having a constant key of `x`. However, when only `c()` is reported, it becomes difficult to identify and fix the root cause (*i.e.*, go modify `a()` to replace the constant key). Often times we find ourselves searching for `x` in a code base with `grep`, and if `x` is a common constant, it becomes difficult to pinpoint `a()`. Conversely, when only `a()` is reported, it is easy to locate `x`, but it becomes difficult to comprehend where `x` is used and what are the impacts. Likewise in CogniCrypt$_{SAST}$, the required and ensured predicates can propagate across many different classes, triggering a chain of errors being reported. We find it challenging sometimes to pinpoint the offending source behind the first violated predicate, which is needed to understand and fix a reported misuse. Based on this experience, we conjecture that the root cause analysis can be made easier if the detectors can also output (or even visualize) the data flows found in the underlying static analysis. This information should already be available in the detectors. Being able to see how the data flows (and predicates) propagate can also help in deciding whether a misuse report is a false alarm or not.

XI. Lessons and Recommendations

Lessons learned. Despite the best intentions of helping developers, many alarms reported by the detectors do not seem to be actual misuses or vulnerabilities. Thus, when measuring the security of apps (and firmware), these detectors must be used with care. This in turn suggests that the problem of precisely reporting cryptographic misuses is yet to be completely solved.

Although academic tool are not expected to be perfect, the false alarms discussed in Sections V and VI suggest the small-scale inspections performed by the corresponding papers are inadequate in validating the correctness of the detectors. Specifically, CryptoGuard manually inspected 1295 alarms in Apache projects but none in Android apps. CogniCrypt$_{SAST}$ manually inspected 50 (out of 10000) Android apps, but did not investigate any `RequiredPredicateError` (which includes the fresh random IV alarms discussed in Pattern #11). CryptoREX manually checked 30 (out of 679) alarms. To some extent, those papers were victims of their own success, in the sense that static analysis enables a scalable scanning of many applications, resulting in a large number of alarms that disincentivizes a thorough manual inspection.

On the other hand, the patterns discussed in Sections VII and VIII show that there are many subtle nuances in defining what constitutes a cryptographic misuse. Some API methods that are perceived to be weak might have many legitimate usages, and thus outright prohibiting them do not always lead to meaningful findings for developers. As such, it might not be the most constructive approach to report all those alarms as misuses or vulnerabilities.

Assumptions are important. At the end of the day, many of the false alarms are caused by assumptions that might not hold in reality. This includes assumptions on what are the correct ways of writing code for a specific task (which affects the whitelist approach), as well as assumptions on where and how will a certain function be used (which affects the blacklist approach). Going forward, large-scale measurement studies of code repositories might help to justify the assumptions taken by the misuse detectors. For instance, if functions like AES-ECB are overwhelmingly used in vulnerable manners rather than in legitimate contexts such as implementing other secure modes of AES, then it makes sense to blacklist all usages of AES-ECB. Likewise, the whitelisting approach can also benefit from the common programming patterns and practices revealed by measurement studies of code repositories.

Recommendations for future work. In terms of design, we recommend future work to focus on targeting and modeling *sufficient conditions* of vulnerabilities. As discussed in Pattern #16, the CogniCrypt$_{SAST}$ approach of focusing on specific scenarios that require CSPRNGs avoids many false alarms and certainly has merits. This design philosophy should be applied to other rules and their models as well. Furthermore, it would also help to have a more fine-grained set of API models. Despite their similarity, the APIs of Android and conventional Java have subtle but non-negligible differences that warrant careful considerations and refinements. For Android apps, one can further refine the API models based on the target API version, which is available to detectors in the app manifest file. The app manifest has already been used by other analysis (*e.g.*, [72]), and was used by CryptoGuard in its post-static-analysis origin attribution [1].

For evaluation, we recommend future work to first test their detectors on open-source data sets, which improves the explainability of misuse alarms. Furthermore, we recommend addressing evaluation blind-spots at two different levels. To test for problems in rule modeling and implementation (cf. Sections V and VI), we recommend grouping the offending methods per misuse rule (see Section IV for details), and then evaluate the security threats induced by the top-10 offending methods under a well-defined threat model. In our experience, this gives good coverage and helps to uncover many false alarms due to rule modeling or bugs. To identify problems in high-level misuse rules, it might be better to enlist the help of independent third parties, otherwise the original rule designer might not see problems in the misuse rules. One way is to contact the corresponding app or library developers and seek their opinions on the resulting misuse alarms, similar to what was done in [27]. Increased contact with human stakeholders (developers), however, might require additional ethical considerations and procedures to avoid causing harms (*e.g.*, confusing developers into patching non-bugs).

XII. Conclusion

In this paper, we revisit the problem of using static analysis to detect cryptographic API misuse. Our empirical evaluation

shows that many false alarms exist. We further dissect the root causes of false alarms, which include overly conservative rules and models, as well as bugs in detectors. We present a more nuanced view of what constitute a misuse, and discuss improvement directions for making misuse detectors more precise and usable for developers. To facilitate future research on this topic, we publicly released all our artifacts.

Acknowledgments

We thank the anonymous reviewers and shepherd for helping us improve our paper. This work was supported in part by a grant from the Research Grants Council (RGC) of Hong Kong (Project No.: CUHK 24205021), and various grants from CUHK and its Department of Information Engineering.

References

[1] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. Yao, "Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2455–2472.

[2] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2382–2400, 2019.

[3] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "{CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 151–164.

[4] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 73–84.

[5] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: Supporting developers in using cryptography," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[6] A.-K. Wickert, L. Baumgärtner, F. Breitfelder, and M. Mezini, "Python crypto misuses in the wild," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–6.

[7] L. Piccolboni, G. Di Guglielmo, L. P. Carloni, and S. Sethumadhavan, "Crylogger: Detecting crypto misuses dynamically," in *IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1972–1989.

[8] A. S. Ami, N. Cooper, K. Kafle, K. Moran, D. Poshyvanyk, and A. Nadkarni, "Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 614–631.

[9] S. Rahaman, H. Cai, O. Chowdhury, and D. Yao, "From theory to code: Identifying logical flaws in cryptographic implementations in c/c++," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3790–3803, 2021.

[10] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[12] "The SpotBugs plugin for security audits of Java web applications and Android applications." Jun. 2023. [Online]. Available: https://github.com/find-sec-bugs/find-sec-bugs

[13] Adobe Acrobat SDK version 9.0, "Adobe Supplement to the ISO 32000," 2008, https://web.archive.org/web/20200621050243/https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/adobe_supplement_iso32000.pdf.

[14] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 50–61.

[15] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications," Jan. 2020. [Online]. Available: https://github.com/androguard/androguard

[16] S. Afrose, S. Rahaman, and D. Yao, "Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 49–61.

[17] M. Schlichtig, A.-K. Wickert, S. Krüger, E. Bodden, and M. Mezini, "Cambench–cryptographic api misuse detection tool benchmark suite," *arXiv preprint arXiv:2204.06447*, 2022.

[18] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[19] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt, "A critical evaluation of website fingerprinting attacks," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 263–274.

[20] N. Mathews, J. K. Holland, S. E. Oh, M. S. Rahman, N. Hopper, and M. Wright, "Sok: A critical evaluation of efficient website fingerprinting defenses," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 344–361.

[21] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1325–1341.

[22] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[23] L. Luo, E. Bodden, and J. Späth, "A qualitative analysis of android taint-analysis results," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 102–114.

[24] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[25] S. Lipp, S. Banescu, and A. Pretschner, "An empirical study on the effectiveness of static c code analyzers for vulnerability detection," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22), July 18–22, 2022, Virtual, South Korea*, 2022.

[26] A.-K. Wickert, L. Baumgärtner, M. Schlichtig, K. Narasimhan, and M. Mezini, "To fix or not to fix: A critical study of crypto-misuses in the wild," in *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2022.

[27] Y. Zhang, M. M. A. Kabir, Y. Xiao, D. D. Yao, and N. Meng, "Automatic detection of java cryptographic api misuses: Are we there yet," *IEEE Transactions on Software Engineering*, 2022.

[28] "Soot - a java optimization framework," Jul. 2022. [Online]. Available: https://github.com/soot-oss/soot

[29] J. Späth, K. Ali, and E. Bodden, "Ide al: Efficient and precise alias-aware dataflow analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, 2017.

[30] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, no. 1, pp. 157–171, 1986.

[31] J. Späth, K. Ali, and E. Bodden, "Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[32] "F-droid: Free and open source android app repository," Jul. 2022. [Online]. Available: https://f-droid.org/

[33] R. B. Evans and A. Savoia, "Differential testing: a new approach to change detection," in *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, 2007, pp. 549–552.

[34] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source attribution of cryptographic api misuse in android applications," in *Proceedings of the*

2018 on Asia Conference on Computer and Communications Security, 2018, pp. 133–146.

[35] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988, pp. 12–27.

[36] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[37] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[38] D. F. Bacon and P. F. Sweeney, "Fast static analysis of c++ virtual function calls," in *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1996, pp. 324–341.

[39] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallée-Rai, P. Lam, E. Gagnon, and C. Godin, "Practical virtual method call resolution for java," *ACM SIGPLAN Notices*, vol. 35, no. 10, pp. 264–280, 2000.

[40] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 251–261.

[41] NIST, "Announcement of proposal to revise nist sp 800-132, recommendation for password-based key derivation: Part 1: Storage applications," 2023. [Online]. Available: https://csrc.nist.gov/News/2023/proposal-to-revise-nist-sp-800-132-pbkdf

[42] K. Moriarty (Ed.), B. Kaliski, and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1," RFC 8018 (Informational), RFC Editor, Fremont, CA, USA, Jan. 2017. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8018.txt

[43] E. Barker, "NIST Special Publication 800-57 Part 1 Revision 5 Recommendation for Key Management: Part 1 – General," https://www.keylength.com/en/4.

[44] "SecureRandom (Java Platform SE 8)," https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html.

[45] "SecureRandom — Android Developers," https://developer.android.com/reference/java/security/SecureRandom.html.

[46] "Java Cryptography Architecture Oracle Providers Documentation," https://docs.oracle.com/javase/8/docs/technotes/guides/security/SunProviders.html#SecureRandomImp.

[47] "Android Developers Blog: Using Cryptography to Store Credentials Safely," https://android-developers.googleblog.com/2013/02/using-cryptography-to-store-credentials.html.

[48] "Android Developers Blog: Some SecureRandom Thoughts," https://android-developers.googleblog.com/2013/08/some-securerandom-thoughts.html.

[49] "Android Developers Blog: Security "Crypto" provider deprecated in Android N," https://android-developers.googleblog.com/2016/06/security-crypto-provider-deprecated-in.html.

[50] "⟨uses-sdk⟩ — Android Developers," https://developer.android.com/guide/topics/manifest/uses-sdk-element.

[51] "SSLParameters — Android Developers," Nov. 2022. [Online]. Available: https://developer.android.com/reference/javax/net/ssl/SSLParameters#setEndpointIdentificationAlgorithm(java.lang.String)

[52] "Google tink library that provides cryptographic apis," Jul. 2022. [Online]. Available: https://github.com/google/tink

[53] "Application sandbox in android," Jul. 2022. [Online]. Available: https://source.android.com/docs/security/app-sandbox

[54] "Bip38 for bitcoin," Jul. 2022. [Online]. Available: https://en.bitcoin.it/wiki/BIP_0038

[55] T. Krovetz (Ed.), "UMAC: Message Authentication Code using Universal Hashing," RFC 4418 (Informational), RFC Editor, Fremont, CA, USA, Mar. 2006. [Online]. Available: https://www.rfc-editor.org/rfc/rfc4418.txt

[56] H. Cohen, *A Course in Computational Algebraic Number Theory*, ser. Graduate Texts in Mathematics. Springer Berlin Heidelberg, vol. 138. [Online]. Available: http://link.springer.com/10.1007/978-3-662-02945-9

[57] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*, ser. London Mathematical Society Lecture Note Series. Cambridge university press, no. 265.

[58] NIST, *SHA-3 standard: Permutation-based hash and extendable-output functions*. FIPS 202, 2015, http://dx.doi.org/10.6028/NIST.FIPS.202.

[59] D. Boneh and V. Shoup, *A graduate course in applied cryptography*, 2023.

[60] NIST, *Recommendation for Key Management: Part 1 – General*. NIST Special Publication 800-57 Part 1 Revision 5, 2020, https://doi.org/10.6028/NIST.SP.800-57pt1r5.

[61] S. Bradner, "Rfc2119: Key words for use in rfcs to indicate requirement levels," 1997. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2119

[62] P. Wouters and O. Sury, "Algorithm Implementation Requirements and Usage Guidance for DNSSEC," RFC 8624 (Proposed Standard), RFC Editor, Fremont, CA, USA, Jun. 2019. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8624.txt

[63] "Bitcoin script system for transactions," Nov. 2022. [Online]. Available: https://en.bitcoin.it/wiki/Script

[64] "/docs/manmaster/man3/PEM_read_PrivateKey.html," 2022, https://www.openssl.org/docs/manmaster/man3/PEM_read_PrivateKey.html#PEM-ENCRYPTION-FORMAT.

[65] International Civil Aviation Organization, "Doc 9303 Machine Readable Travel Documents Eighth Edition Part 11: Security Mechanisms for MRTDs," 2021.

[66] "Apple Filing Protocol Concepts," 2012. [Online]. Available: https://developer.apple.com/library/archive/documentation/Networking/Conceptual/AFP/Concepts/Concepts.html

[67] "AFP File Server Security," 2012. [Online]. Available: https://developer.apple.com/library/archive/documentation/Networking/Conceptual/AFP/AFPSecurity/AFPSecurity.html

[68] "Ntlm support in apache httpclient library," Jul. 2022. [Online]. Available: https://hc.apache.org/httpcomponents-client-4.5.x/ntlm.html

[69] "SpotBugs: A tool for static analysis to look for bugs in Java code." Oct. 2023. [Online]. Available: https://spotbugs.github.io/index.html

[70] "The list of bug patterns with descriptions in Find Security Bugs." Jun. 2023. [Online]. Available: https://find-sec-bugs.github.io/bugs.htm

[71] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang *et al.*, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," 2022.

[72] M. Oltrogge, N. Huaman, S. Amft, Y. Acar, M. Backes, and S. Fahl, "Why eve and mallory still love android: Revisiting tls (in) security in android applications." in *USENIX Security Symposium*, 2021.

[73] "Samba - opening windows to a wider world," Nov. 2022. [Online]. Available: https://www.samba.org/

[74] "avahi - mDNS/DNS-SD," Nov. 2022. [Online]. Available: https://www.avahi.org/

[75] "Linux WPA/WPA2/IEEE 802.1X Supplicant," Nov. 2022. [Online]. Available: https://w1.fi/wpa_supplicant/

[76] "OpenL2TP - Advanced crypto casino systems," Nov. 2022. [Online]. Available: https://openl2tp.org/

[77] "winbindd — Name Service Switch daemon for resolving names from NT servers," Nov. 2022. [Online]. Available: https://www.samba.org/samba/docs/current/man-html/winbindd.8.html

[78] "RAND_seed," Nov. 2022. [Online]. Available: https://www.openssl.org/docs/man3.0/man3/RAND_seed.html

[79] L. E. Bassham III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks *et al.*, *Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications*. National Institute of Standards & Technology, 2010.

APPENDIX A

A. Misuses reported and sampled

Table VII shows the overview of the total number of misuses and methods reported, as well as the coverage of our sampling effort for each detector. A long tail can be seen by comparing the misuses reported and the offending methods. This is often due to the use of lesser known libraries, or obfuscation of app code, which breaks the method grouping.

TABLE VII. MISUSES AND METHODS REPORTED AND SAMPLED

Rule	Reported Misuses	Sampled (percentage)	Offending Methods	Sampled (percentage)
CryptoGuard				
1,2	972	29.0%	190	6.8%
3	364	51.1%	105	9.5%
4	322	57.8%	121	13.2%
5	948	26.4%	463	2.2%
6	151	60.9%	57	19.3%
7	1645	41.2%	683	2.2%
8	105	88.6%	20	50.0%
9	9042	38.0%	3088	1.2%
10	150	53.3%	24	41.7%
11,14	716	54.5%	266	15.8%
12	490	24.3%	61	16.4%
13	1510	34.4%	86	14.0%
15	30	100.0%	11	100.0%
16	5804	42.8%	1249	1.9%
CogniCrypt$_{SAST}$				
Cipher	4457	7.9%	1317	1.3%
MessageDigest	5031	24.1%	1790	0.6%
SecretKeySpec	2216	10.3%	1037	1.0%
PBEKeySpec	302	37.1%	74	13.5%
KeyPairGenerator	814	17.4%	357	2.8%
PBEParameterSpec	28	60.7%	16	62.5%
IvParameterSpec	861	26.5%	414	2.4%
SecureRandom	629	38.5%	135	7.4%
KeyStore	769	23.9%	362	2.8%
SSLContext	5164	39.3%	825	1.2%
TrustManagerFactory	1209	59.6%	327	3.1%
CryptoREX				
1	623	100.0%	23	100.0%
2	7	100.0%	3	100.0%
3	68	100.0%	8	100.0%
4	172	100.0%	10	100.0%
5	6	100.0%	1	100.0%
6	1045	70.1%	145	20.0%

B. Protocol strings and TLS versions

Table VIII shows the actual TLS versions that are enabled on Android, when the code provides different protocol strings to the `SSLContext.getInstance()` method.

TABLE VIII. TLS VERSIONS AND PROTOCOL REQUESTED

Protocol requested via `SSLContext.getInstance()`	Android version	Available protocols
TLS	Android 10 - 13 (level 29 - 33)	TLSv1.3 TLSv1.2 TLSv1.1 TLSv1
	Android 6 - 9 (level 23 - 28)	TLSv1.2 TLSv1.1 TLSv1
	Android 5 (level 21 - 22)	TLSv1.2 TLSv1.1 TLSv1 SSLv3
	Android 4 (level 15 - 19)	TLSv1 SSLv3
SSL	Android 10 - 13 (level 29 - 33)	TLSv1.3 TLSv1.2 TLSv1.1 TLSv1
	Android 8 - 9 (level 26 - 28)	TLSv1.2 TLSv1.1 TLSv1
	Android 5 - 7 (level 21 - 25)	TLSv1.2 TLSv1.1 TLSv1 SSLv3
	Android 4 (level 15 - 19)	TLSv1 SSLv3
TLSv1	Android 6 - 13 (level 23 - 33)	TLSv1.2 TLSv1.1 TLSv1
	Android 5 (level 21 - 22)	TLSv1.2 TLSv1.1 TLSv1 SSLv3
	Android 4 (level 15 - 19)	TLSv1 SSLv3
TLSv1.2	Android 6 - 13 (level 23 - 33)	TLSv1.2 TLSv1.1 TLSv1
	Android 5 (level 21 - 22)	TLSv1.2 TLSv1.1 TLSv1 SSLv3
	Android 4 (level 16 - 19)	TLSv1 SSLv3
TLSv1.3	Android 10 - 13 (level 29 - 33)	TLSv1.3 TLSv1.2 TLSv1.1 TLSv1

C. CogniCrypt$_{SAST}$ bug in detecting hard-coded arrays

```
/***
 * Function that finds the values assigned to a soot array.
 * @param callSite call site at which sootValue is involved
 * @param allocSite allocation site at which sootValue is
      involved
 * @param arrayLocal soot array local variable for which values
      are to be found
 * @return extracted array values
 */
protected Map<String, CallSiteWithExtractedValue>
     extractSootArray(CallSiteWithParamIndex callSite,
     ExtractedValue allocSite){
  Value arrayLocal = allocSite.getValue();
  Body methodBody = allocSite.stmt().getMethod().getActiveBody()
       ;
  Map<String, CallSiteWithExtractedValue> arrVal = Maps.
       newHashMap();
  if (methodBody != null) {
    Iterator<Unit> unitIterator = methodBody.getUnits().
         snapshotIterator();
    while (unitIterator.hasNext()) {
      final Unit unit = unitIterator.next();
      if (unit instanceof AssignStmt) {
        AssignStmt uStmt = (AssignStmt) (unit);
        Value leftValue = uStmt.getLeftOp();
        Value rightValue = uStmt.getRightOp();
        if (leftValue.toString().contains(arrayLocal.toString()
             ) && !rightValue.toString().contains("newarray"))
              {
          arrVal.put(retrieveConstantFromValue(rightValue), new
               CallSiteWithExtractedValue(callSite, allocSite
              ));
        }
      }
    }
  }
 return arrVal;

/***
 * Function that decides if an array is hardcoded.
 */
private boolean isHardCodedArray(Map<String,
     CallSiteWithExtractedValue> extractSootArray) {
     return !(extractSootArray.keySet().size() == 1 &&
          extractSootArray.containsKey(""));
  }
}
```

Listing 7. Source code for deciding hard-coded array in CogniCrypt$_{SAST}$

The main bug can be found in line 9 of Listing 7: the method `getValue()` of the `allocSite` variable returns to `arrayLocal` the right-hand-side (RHS) value of an allocation site. However, it should actually return the left-hand-side (LHS) value of an allocation site. As an example, consider an `allocSite` of local array `r1 = newarray (char)[]`. The RHS value of it (`newarray (char)[]`) cannot match any variable names in the subsequent matching operation on line 20. `arrayLocal` should instead take `r1` as its value so that the matching operation in line 20 would be feasible. Due to this bug, the `extractSootArray()` method always returns an empty `Map`. Then the `isHardCodedArray()` method always returns true due to a zero sized key set in the empty `Map`.

D. Example of statically resolvable polymorphism

In Listing 8, there are two classes that implement the `Hasher` interface: the `DoUpdateHasher` class and the `NoUpdateHasher` class. In `MainActivity`, a `DoUpdateHasher` object is fed into the `HasherWrapper` (line 36). The `DoUpdateHasher` object indeed invokes the `MessageDigest.update()` method and returns the hash value of "data". However, due to the interface class as parameter type in `HasherWrapper.getHash(Hasher hash)`, CogniCrypt$_{SAST}$ regards the parameter type as `Hasher` and considers all classes that implement this interface to be possible during inter-procedural analysis, without using the static type information from `MainActivity`. As the result, CogniCrypt$_{SAST}$ will report a TypestateError for the `MessageDigest` object, which is an FP.

```
1  public interface Hasher { // interface defintion
2    void updateData(MessageDigest mD) throws Exception;
3  }
4
5  public class DoUpdateHasher implements Hasher {
6    private String data = "data";
7    @Override
8    public void updateData(MessageDigest mD) throws Exception {
9      byte[] data = this.data.getBytes("UTF-8");
10     mD.update(data);
11   }
12 }
13
14 public class NoUpdateHasher implements Hasher {
15   @Override
16   public void updateData(MessageDigest mD) throws Exception
17   { /* do nothing */ }
18 }
19
20 public class HasherWrapper {
21   public byte[] getHash(Hasher hash) throws Exception {
22     byte[] hashdata = new byte[] {};
23     MessageDigest mD = MessageDigest.getInstance("SHA-256");
24     hash.updateData(mD);
25     hashdata = mD.digest();
26     return hashdata;
27   }
28 }
29
30 public class MainActivity extends AppCompatActivity {
31     @Override
32     protected void onCreate(Bundle savedInstanceState) {
33         ...
34         DoUpdateHasher doUpdateHasher = new DoUpdateHasher();
35         HasherWrapper hasherWrapper = new HasherWrapper();
36         hasherWrapper.getHash(doUpdateHasher);
37     }
38 }
```

Listing 8. FP case of polymorphism in CogniCrypt$_{SAST}$

E. Non-cryptographic usage of PRNG in firmware

For CryptoREX's rule 6 alarms, we specifically investigated 22 binaries which covers 24 alarms. We found that 13 alarms are actually ITPs due to usage of `rand()` in non-security-critical contexts. Here are the 4 high-level cases:

(i) *`rand()` used to implement a cache mechanism.* (5/13) In a modified version of Samba [73] that is used by multiple Netgear firmware images, the function `rand()` is used to implement a simple cache mechanism. We suppose the Samba in Netgear firmware was modified by vendor as we cannot find the corresponding source code on Samba's GitHub repository, so we relied on the Netgear GPL Open Source Code repository[22] to retrieve the source code of the firmware, and we attach the relevant code as an example in Listing 9. The function first tries to retrieve the corresponding `passwd*` from the buffer `pwnam_cache`. If the desired `passwd*` is not cached in the buffer, it then calls the function `sys_getpwnam()` (equivalent to `getpwnam()`) and tries to find an unused part of the buffer to cache this. If the buffer has no room for the new `passwd*` to be cached, function `rand` is called to select a random cached entry to discard. In this case, whether the generated random number is predictable does not matter much.

```
1  static struct passwd **pwnam_cache = NULL;
2  ...
3  struct passwd *getpwnam_alloc(TALLOC_CTX *mem_ctx,
4  const char *name) {
5      int i;
6      struct passwd *temp;
7      init_pwnam_cache();
8      for (i=0; i<PWNAMCACHE_SIZE; i++) {
9          if ((pwnam_cache[i] != NULL) &&
10             (strcmp(name, pwnam_cache[i]->pw_name) == 0)) {
11             DEBUG(10, ("Got %s from pwnam_cachen", name));
12             return talloc_reference(mem_ctx, pwnam_cache[i]);
13         }
14     }
15     temp = sys_getpwnam(name);
16     if (!temp) {
17         return NULL;
18     }
19     for (i=0; i<PWNAMCACHE_SIZE; i++) {
20         if (pwnam_cache[i] == NULL)
21             break;
22     }
23     if (i == PWNAMCACHE_SIZE)
24         i = rand() % PWNAMCACHE_SIZE;
25     if (pwnam_cache[i] != NULL) {
26         TALLOC_FREE(pwnam_cache[i]);
27     }
28     pwnam_cache[i] = tcopy_passwd(pwnam_cache, temp);
29     if (pwnam_cache[i]!= NULL && mem_ctx != NULL) {
30         return talloc_reference(mem_ctx, pwnam_cache[i]);
31     }
32     return tcopy_passwd(NULL, pwnam_cache[i]);
33 }
```

Listing 9. An ITP example of Rule 6-2 (no seed) from Samba

(ii) *`rand()` used to generate random time interval.* (5/13) In Samba [73], avahi [74], wpa_supplicant [75], OpenL2TP [76], and two closed-source binaries, `rand()` is called to generate a random delay for various purposes. For example, in function `winbind_named_pipe_sock()` of `libwbclient.so.0`, a shared library wrapper around winbindd [77] requests, `rand()` is called to generate a random delay between 1 second and 3

[22] https://kb.netgear.com/2649/NETGEAR-Open-Source-Code-for-Programmers-GPL

seconds when receiving `EAGAIN` (try again) response from a non-blocking socket.

(iii) *`rand()` used to generate service cookie.* (1/13) In avahi-core, `rand()` is used to generate different service cookies, which are used to distinguish different services with the same names created by avahi-publish. The usage of `rand()` in this situation appears non-security-critical.

(iv) *`rand()` is called but the generated number does not have any actual effects.* (2/13) In a closed-source binary netgear_ntp, `rand()` is called but its return value is not used. In another closed-source binary email, the return value `rand()` is used to add entropy to OpenSSL's PRNG. According to the official OpenSSL's documentation [78], manual (re-)seeding the default OpenSSL random generator is not necessary (but allowed), thus we consider it an ITP.

F. NIST SP 800-22 Statistical Test misuse in Crylogger

Crylogger's rule 6 and 8 detect "badly derived" keys and IVs respectively. For that, Crylogger logs all the values generated by `Util.Random` and `SecureRandom` as well as all the key materials and IV values. If any of the key or IV values came from `Util.Random`, they are reported as insecure. Otherwise, if they came from `SecureRandom`, they are considered secure. For keys and IVs that are not from either of those two PRNGs (*e.g.*, loaded from a key store or generated by other PRNGs), Crylogger will then put them to the NIST SP 800-22 statistical test to see if they are "badly derived" or not.

Several problems exist in the design and implementation of this detection. First, to determine whether keys and IVs came from `Util.Random` or `SecureRandom`, Crylogger requires an exact match of the entire string. In other words, it assumes that for each invocation of the PRNG, all outputs will be used as one key or IV. However, this assumption does not always hold in practice. In our experiments, we observed that Google Tink's implementation of AES-SIV first generates a 64-byte key, which is then split into two 32-byte sub-keys, one for CTR mode encryption, one for its CMAC[23]. Because of this, Crylogger will fail to recognize that the CTR key came from `SecureRandom`, thus putting it into the NIST SP 800-22 test.

The NIST SP 800-22 test is a suite of 15 statistical tests meant to test whether a binary sequence appear random. For these tests to deliver meaningful results, there are 3 critical criteria: (1) there are enough runs, (2) for each run, there are long enough input bits, and (3) each run of each test tests the same PRNG. Interestingly, Crylogger breaks all 3 criteria.

First, we observed that although Crylogger separately tests the randomness of keys and IVs, when Crylogger aggregates inputs to NIST SP 800-22, it disregards the originating PRNGs. That is, for keys (and separately, IVs) that Crylogger deems necessary to test, it aggregates all of them as inputs to one run of NIST SP 800-22, even if they were generated by different PRNGs. As the result of which, it is not clear how should one interpret the end results of NIST SP 800-22, as multiple PRNGs could have contributed to one input binary sequence.

Second, in our experiments, we found that 4 of the 15 NIST SP 800-22 tests always fail. Upon a closer look, we noticed those 4 tests require a large number of input bits (ranging from 10^4 to 10^6 bits). This is due to the fact that those tests use the Central Limit Theorem to approximate the asymptotic distribution of the binary stream, so these tests are not valid when inputs are too short. According to the NIST SP 800-22 document [79], each test has a *minimum requirement* on its input size. We found that the Python library of NIST SP 800-22 used by Crylogger returns a failure for a particular test if the input size is less than its minimum requirement. However, since Crylogger considers keys (and IVs) to be badly derived when any one of the 15 tests failed[24], false alarms can easily happen due to the log file collected by Crylogger having not enough bits of keys (and IVs). In order to satisfy the 10^6 bits input size requirement, one execution of an app would need to have more than 7810 distinct 128-bit AES keys that are not directly from `Util.Random` and `SecureRandom`. Thus, securely derived keys that were loaded from key store are often reported as insecure by Crylogger. And together with its exact match heuristics, the Google Tink implementation of AES-SIV, despite generating keys from `SecureRandom`[25], was reported as insecure by Crylogger.

Moreover, statistical tests themselves can also have false positives, and it is recommended to run the NIST SP 800-22 test suite multiple times in order to achieve statistical significance. As an experiment, we collected 1000 sufficiently long sequences from `SecureRandom` for 1000 runs of NIST SP 800-22. Using the original C implementation of SP 800-22 from NIST, we found that out of the 1000 runs, the Discrete Fourier Transform (Spectral) Test failed for 18 runs, and the Frequency (Monobit) Test failed for 8 runs. This highlights the fact that a single run of NIST SP 800-22 cannot directly distinguish a CSPRNG from a non-CSPRNG. Thus, by requiring a PRNG to always not fail any single run of NIST SP 800-22, Crylogger puts an unnecessarily high expectation on the PRNG. In general, one would usually first fix the acceptable probability α (also known as the significance level) of getting a false positive (the case where the test says the data is non-random when the data is actually random). In other words, one expects even a (truly) random sequence would fail the test with probability α. Then, when the same test is run for n times for a particular PRNG, if the proportion of sequences that did not fail the statistical test falls outside of the confidence interval $(1-\alpha \pm 3\sqrt{\frac{\alpha(1-\alpha)}{n}}$, computed with the normal approximation of the binomial distribution), then one can say with $(1-\alpha)$ certainty that the PRNG failed the statistical test. For instance, when $\alpha = 0.01$, $n = 1000$, confidence interval $= 0.99 \pm 0.0094392$.

Finally, we note that in April 2022, NIST decided to revise NIST SP 800-22 Rev. 1a[26]. In particular, it will clarify the purpose of the SP 800-22 test suite, and reject its use for assessing cryptographic random number generators. Future detectors should thus avoid using NIST SP 800-22 as the basis of deciding whether keys and IVs are bad.

[23] https://github.com/google/tink/blob/ca852750/java_src/src/main/java/com/google/crypto/tink/subtle/AesSiv.java#L53C3-L62C4

[24] The original Crylogger paper acknowledges the possibilities of skipping some tests due to insufficient input bits, but the tool does not implement it.

[25] https://github.com/google/tink/blob/8f6316b9/java_src/src/main/java/com/google/crypto/tink/internal/Random.java#L26

[26] https://csrc.nist.gov/news/2022/decision-to-revise-nist-sp-800-22-rev-1a

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

Our artifacts contain all reported alarms from evaluated detectors, analysis results of labeled alarms, and refined CryptoGuard as well as false positive examples.

1) How to access: The DOI link is https://doi.org/10.5281/zenodo.10158303. Our artifacts are hosted at https://github.com/kynehc/crypto-detector-evaluation-artifacts.

2) Hardware dependencies: Our experiments of evaluating detectors were ran on a server machine with two 10-core Intel(R) Xeon(R) Silver 4210R CPUs @ 2.40GHz, with a total of 256 GB of RAM.

The detectors have a time-out mechanism. Thus, the total number of alarms emitted by a detector on unfinished apps might vary depending on the hardware performance.

3) Software dependencies: All software dependencies are listed in our artifact's readme files.

4) Benchmarks: None.

B. Artifact Installation & Configuration

Our artifacts contain the instructions to prepare the environment for the evaluation.

C. Major Claims

- (C1): We analyzed sampled false alarms from evaluated detectors, and show various false alarm patterns exist. This is validated by the experiment (E1).
- (C2): We implemented this refinement in CryptoGuard, rerun it on the app data set, and found a large number of FPs for rules 1– 3, 8, 10, 12–13 because of Pattern #1, as shown in Table IV. This is validated by the experiment (E2).

D. Evaluation

1) Experiment (E1): Our readme file in artifacts lists the labeled alarms and corresponding analysis results that manifest various false alarm patterns.

2) Experiment (E2): To validate the functionality of the refined CryptoGuard in reducing false positives caused by Pattern #1, we provide a scaled-down version of the experiment we conducted on refined CryptoGuard. Our artifacts include detailed instructions on the experiment.

E. Notes

The claims about Crylogger were added during paper revision, and thus they were not covered by the the artifacts we submitted to artifact evaluation. Additionally, due to copyright issues, we cannot provide licenses or copes of IDA Pro to reproduce the experiments on CryptoREX.