Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques

Amit Seal Ami*, Nathan Cooper*, Kaushal Kafle*, Kevin Moran†, Denys Poshyvanyk*, and Adwait Nadkarni*
*William & Mary, Williamsburg, VA, USA
{aami@email., nacooper01@email., kkafle@email., denys@cs., nadkarni@cs.}wm.edu
†George Mason University, Fairfax, VA, USA
kpmoran@gmu.edu

***Abstract*—The *correct* use of cryptography is central to ensuring data security in modern software systems. Hence, several academic and commercial static analysis tools have been developed for detecting and mitigating crypto-API misuse. While developers are optimistically adopting these crypto-API misuse detectors (or crypto-detectors) in their software development cycles, this momentum must be accompanied by a *rigorous understanding of their effectiveness at finding crypto-API misuse in practice*. This paper presents the MASC framework, which enables a systematic and data-driven evaluation of crypto-detectors using mutation testing. We ground MASC in a comprehensive view of the problem space by developing a data-driven taxonomy of existing crypto-API misuse, containing 105 misuse cases organized among nine semantic clusters. We develop 12 generalizable *usage-based mutation operators* and three *mutation scopes* that can expressively instantiate thousands of compilable variants of the misuse cases for thoroughly evaluating crypto-detectors. Using MASC, we evaluate *nine* major crypto-detectors and discover 19 unique, undocumented flaws that severely impact the ability of crypto-detectors to discover misuses in practice. We conclude with a discussion on the diverse perspectives that influence the design of crypto-detectors and future directions towards building security-focused crypto-detectors by design.**

I. Introduction

Effective cryptography is critical in ensuring the security of confidential data in modern software. However, ensuring the *correct use* of cryptographic primitives has historically been a hard problem, whether we consider the vulnerable banking systems from Anderson's seminal work [1], or the widespread misuse of cryptographic APIs (*i.e., crypto-APIs*) in mobile and Web apps that can lead to the compromise of confidential financial or medical data and even the integrity of IoT devices [2]–[8]. In response, security researchers have developed a wide array of techniques and tools for detecting crypto-API misuse [2]–[4], [9]–[17] that can be integrated into the software development cycle, thereby preventing vulnerabilities at the source. These crypto-API misuse detectors, or *crypto-detectors*, play a crucial role in the security of end-user software.

Crypto-detectors have been independently used by developers for decades [18]. They are integrated into IDEs (*e.g.,* the CogniCrypt plugin for Eclipse [19]), incorporated in the internal testing suites of organizations (*e.g.,* Cryptoguard [3], integrated into Oracle's testing suite [20]), or are currently targeted for commercialization and widespread deployment [3], [21]. In fact, several crypto-detectors are also being formally provisioned by code hosting services as a way of allowing developers to ensure compliance with data security standards and security best-practices (*e.g.,* Github's CodeScan initiative [22]). Thus, the importance of crypto-detectors in ensuring data security in modern Web and mobile software cannot be overstated, as key stakeholders (*i.e.,* researchers, code-hosting services, app markets, and developers) are increasingly reliant on them. However, what is concerning is that while stakeholders are optimistically adopting crypto-detectors, *we know very little regarding their actual effectiveness at finding crypto-API misuse*. That is, beyond manually-curated benchmarks, there is no approach for *systematically* evaluating crypto-detectors. This example in Listing 1 illustrates the gravity of this problem:

```
String algorithm = "DES";
Cipher cipher = Cipher.getInstance(algorithm);
```

Listing 1. Instantiating "DES" as a cipher instance.

In this example, we define DES as our algorithm of choice, and instantiate it using the Cipher.getInstance(<parameter>) API. Given that DES is not secure, one would expect any crypto-detector to detect this relatively straightforward misuse. However, two very popular crypto-detectors, *i.e.,* Tool$_X$[1] (used by over 3k+ open source Java projects), and QARK [23] (promoted by LinkedIn and recommended in security testing books [24]–[26]), are unable to detect this trivial misuse case as we discuss later in the paper. Further, one might consider manually-curated benchmarks (*e.g.,* CryptoAPIBench [27], or the OWASP Benchmark [28]) as practical and sufficient for evaluating crypto-detectors to uncover such issues. However, given the scale and diversity of crypto protocols, APIs, and their potential misuse, benchmarks may be incomplete, incorrect, and impractical to maintain; *e.g.,* the OWASP benchmark considered using ECB mode with AES as secure until it was reported in March 2020 [29]. Thus, it is imperative to address this problem through a reliable and evolving evaluation technique that scales to the volume and diversity of crypto-API misuse.

In this paper, we propose the first systematic, data-driven framework that leverages the well-founded approach of Mutation Analysis for evaluating Static Crypto-API misuse detectors – the MASC framework, pronounced as *mask*. Stakeholders can use MASC in a manner similar to the typical use of mutation analysis in software testing: MASC *mutates* Android/Java apps by seeding them with *mutants*, *i.e.,* code snippets exhibiting

[1]We have anonymized this tool in the paper as requested by its developers.

crypto-API misuse. These mutated apps are then analyzed with the crypto-detector that is the target of the evaluation, resulting in mutants that are *undetected*, which when analyzed further reveal design or implementation-level flaws in the crypto-detector. To enable this workflow for practical and effective evaluation of crypto-detectors, MASC addresses three key ***research challenges*** (**RCs**) arising from the unique scale and complexity of the problem domain of crypto-API misuse:

RC$_1$: *Taming the Complexity of Crypto-API Misuse* - An approach that effectively evaluates crypto-detectors must comprehensively express (*i.e.,* test with) relevant misuse cases *across all existing crypto-APIs*, which is challenging as crypto-APIs are as vast as the primitives they enable. For instance, APIs express the initialization of secure random numbers, creation of ciphers for encryption/decryption, computing message authentication codes (MACs), and higher-level abstractions such as certificate and hostname verification for SSL/TLS.

RC$_2$: *Instantiating Realistic Misuse Case Variations* - To evaluate crypto-detectors, code *instances* of crypto-API misuse must be seeded into apps for analysis. However, simply injecting misuse identified in the wild *verbatim* may not lead to a robust analysis, as it does not express the variations with which developers may use such APIs. Strategic and expressive *instantiation* of misuse cases is critical for an effective evaluation, as even subtle variations may evade detection, and hence lead to the discovery of flaws (*e.g.,* passing DES as a variable instead of a constant in Listing 1).

RC$_3$: *Scaling the Analysis* - Efficiently creating and seeding large numbers of *compilable* mutants *without significant manual intervention* is critical for identifying as many flaws in crypto-detectors as possible. Thus, the resultant framework must efficiently scale to thousands of tests (*i.e.,* mutants).

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

- **Crypto-API Misuse Taxonomy:** We construct the first comprehensive taxonomy of crypto-API misuse cases (105 cases, grouped into nine clusters), using a data-driven process that systematically identifies, studies, and extracts misuse cases from academic and industrial sources published over the last 20 years. The taxonomy provides a broad view of the problem space, and forms the core building block for MASC's approach, enabling it to be grounded in real misuse cases observed in the wild (**RC$_1$**).
- ***Crypto*-Mutation Operators and Scopes:** We contextualize mutation testing for evaluating crypto-detectors by designing abstractions that allow us to instantiate the misuse cases from the taxonomy to create a diverse array of feasible (*i.e., compilable*) mutants. We begin by formulating a threat model consisting of 3 adversary-types that represent the threat conditions that crypto-detectors may face in practice. We then design *usage-based mutation operators*, *i.e.,* general operators that leverage the common usage characteristics of diverse crypto-APIs, to expressively instantiate misuse cases from the taxonomy (addresses **RC$_2$**). Similarly, we also design the novel abstraction of *mutation scopes* for seeding mutants of variable fidelity to realistic API-use and threats.
- **The MASC Framework:** We implement the MASC framework for evaluating Java-based crypto-detectors, including 12 mutation operators that can express a majority of the cases in our taxonomy, and 3 mutation scopes. We implement the underlying static analysis to automatically instantiate thousands of *compilable* mutants, with manual effort limited to configuring the mutation operators with values signifying the misuse (**RC$_3$**).
- **Empirical Evaluation of Crypto-Detectors:** We evaluate 9 major crypto-detectors using $20,303$ mutants generated by MASC, and reveal 19 previously unknown flaws (several of which are design-level). A majority of these discoveries of flaws in individual detectors (*i.e.,* 45/76 or 59.2%) are due to mutation (vs. being unable to detect the base/verbatim instantiations of the misuse case). Through the study of open source apps, we *demonstrate that the flaws uncovered by* MASC *are serious and would impact real systems*. Finally, we disclose our findings to the designers/maintainers of the affected crypto-detectors, and further leverage these communication channels to obtain *their perspectives* on the flaws. These perspectives allow us to present a balanced discussion on the factors influencing the current design and testing of crypto-detectors, as well as a path forward towards more robust tool.

Artifact Release: To foster further research in the evaluation and development of effective cryptographic misuse detection techniques, and in turn, more secure software, we have released all code and data associated with this paper [30].

II. Motivation and Background

Insecure use of cryptographic APIs is the second most common cause of software vulnerabilities after data leaks [31]. To preempt vulnerabilities before software release, non-experts such as software developers or quality assurance teams are likely to use crypto-API misuse detectors (or *crypto-detectors*) as a part of the Continuous Integration/Continuous Delivery (CI/CD) pipeline (*e.g.,* Xanitizer [12] and ShiftLeft [16] used in GitHub Code Scan [22]), quality assurance suites (*e.g.,* SWAMP [32]) or IDEs (*e.g.,* CogniCrypt [9]). Thus, the inability of a crypto-detector to flag an instance of a misuse that it claims to detect directly impacts the security of end-user software. We illustrate this problem with a motivating example, followed by a threat model that describes the potential adversarial conditions a crypto-detector may face in the wild.

A. Motivating Example

Consider Alice, a Java developer who uses CryptoGuard [3], a state-of-the-art crypto-detector, for identifying cryptographic vulnerabilities in her software before release. In one of her apps, Alice decides to use the DES cipher, as follows:

```
Cipher cipher = Cipher.getInstance("des");
```

Listing 2. Instantiating DES as a cipher instance in lower case.

This is another instance of the misuse previously shown in Listing 1, *i.e.,* using the vulnerable `DES` cipher. *CryptoGuard* is unable to detect this vulnerability as Alice uses "`des`" instead of "`DES`" as the parameter (see Section X). However, this is a problem, because the lowercase parameter makes no functional difference as Java officially supports both parameter choices. As CryptoGuard does not detect this vulnerability, Alice will assume that her app is secure and release it to end-users. Thus, we need to systematically identify such flaws, which would allow the maintainers of crypto-detectors such as CryptoGuard to promptly fix them, enabling holistic security improvements.

B. Threat Model

To evaluate crypto-detectors, we first define the *scope* of our evaluation, for which we leverage the documentation of popular crypto-detectors to understand how they position their tools, *i.e.,* what use cases they target (see [30] for all quotes). For example, Tool_X's documentation states that it may be used to *"ensure compliance with security and coding standards"* . Similarly, SpotBugs's *Find Security Bugs* plugin is expected to be used for *"security audits"* [33]. Further, CogniCrypt states that its analyses *"ensure that all usages of cryptographic APIs remain secure"* [34], which may suggest the ability to detect vulnerabilities in code not produced by the developer, but originating in a third-party source (*e.g.,* external library, or a contractor), whose developer may not be entirely "virtuous". In fact, 8/9 crypto-detectors evaluated in this paper claim similar cases that demand strong guarantees, *i.e.,* for tasks such as compliance auditing or security assurance that are generally expected to be performed by an *independent* third party that assumes the worst, including bad coding practices or malpractice [35]. As aptly stated by Anderson [35], *"When you really want a protection property to hold, it's vital that the design and implementation be subjected to hostile review"*.

Thus, given that crypto-detectors claim to be useful for tasks such as compliance audits, it is likely for them to be deployed in adversarial circumstances, *i.e.,* where there is tension between the party that uses a crypto-detector for evaluating software for secure crypto-use (*e.g.,* markets such as Google Play, compliance certifiers such as Underwriters Laboratories (UL) [36]), and the party implementing the software (*e.g.,* a third-party developer). With this intuition, we define a threat model consisting of three types of adversaries (**T**1 – **T**3), which guides/scopes our evaluation according to the conditions crypto-detectors are likely to face in the wild:

- **T**1 ***Benign developer, accidental misuse –*** This scenario assumes a benign developer, such as Alice, who accidentally misuses crypto-API, but attempts to detect and address such vulnerabilities using a crypto-detector before releasing the software.
- **T**2 ***Benign developer, harmful fix –*** This scenario also assumes a benign developer such as Alice who is trying to address a vulnerability identified by a crypto-detector in good faith, but ends up introducing a new vulnerability instead. For instance, a developer may not fully understand the problem identified by a crypto-detector, such as missing certificate verification (*e.g.,* an empty `checkServerTrusted` method in a custom `TrustManager`), and address it with an inadequate Stack Overflow fix [37].
- **T**3 ***Evasive developer, harmful fix –*** This scenario assumes a developer whose goal is to finish a task as quickly or with low effort (*e.g.,* a third-party contractor), and is hence attempting to purposefully evade a crypto-detector. Upon receiving a vulnerability alert from a crypto-detector, such a developer may try quick-fixes that do not address the problem, but simply hide it (*e.g.,* hiding the vulnerable code in a class that the crypto-detector does not analyze).

  For example, Google Play evaluates apps by third-party developers to ensure compliance with its crypto-use policies, but there is ample evidence of developers seeking to actively violate these policies [38], [39]. In fact, as Oltrogge et al. [40] recently discovered that developers have been using Android's Network Security Configurations (NSCs) to *circumvent safe defaults* (*e.g.,* to permit cleartext traffic that is disabled by default).

This threat model, which guides `MASC`'s design (Section IV), represents that adversarial conditions under which crypto-detectors may have to operate in practice, and hence, motivates an evaluation based on what crypto-detectors *should be* detecting. However, we note that there may be a gap between *what should be* and *what is*, *i.e.,* while crypto-detectors may want to be relevant in strong deployment scenarios such as compliance checking, their actual design may not account for adversarial use cases (*i.e.,* **T**3). Therefore, we balance our evaluation that uses this threat model with a discussion that acknowledges all views related to this argument, and especially the tool designer's perspective (Sec. XII).

III. Related Work

Security researchers have recently shown significant interest in the external validation of static analysis tools [41]–[45]. Particularly, there is a growing realization that static analysis security tools are sound in theory, but *soundy in practice*, *i.e.,* consisting of a core set of sound decisions, as well as certain strategic unsound choices made for practical reasons such as performance or precision [46]. Soundy tools are desirable for security analysis as their sound core ensures sufficient detection of targeted behavior, while also being *practical*, *i.e.,* without incurring too many false alarms. However, given the lack of oversight and evaluation they have faced so far, *crypto-detectors* may violate this basic assumption behind sound*i*ness and may in fact be *unsound*, *i.e.,* have fundamental flaws that prevent them from detecting even straightforward instances of crypto-API misuse observed in apps. This intuition drives our approach for systematically evaluating crypto-detectors, leading to novel contributions that deviate from related work.

To the best of our knowledge, `MASC` is the first framework to use mutation testing, combined with a large-scale data-driven taxonomy of crypto-API misuse, for comprehensively evaluating the detection ability of crypto-detectors to find design/implementation flaws. However, in a more general sense, Bonett et al. [45] were the first to leverage the intuition

behind mutation testing for evaluating Java/Android security tools, and developed the μSE framework for evaluating data leaks detectors (*e.g.,* FlowDroid [47] and Argus [48]). MASC significantly deviates from μSE in terms of its design focus, in order to address the unique challenges imposed by the problem domain of crypto-misuse detection (*i.e.,*$\mathbf{RC}_1$ – $\mathbf{RC}_3$ in Sec. I). Particularly, μSE assumes that for finding flaws, it is sufficient to *manually* define "a" security operator and strategically place it at hard-to-reach locations in source code. This assumption does not hold when evaluating crypto-detectors as it is improbable to cast cryptographic misuse as a single mutation, given that cryptographic misuse cases are diverse ($\mathbf{RC}_1$), and developers may express the same type of misuse in different ways ($\mathbf{RC}_2$). For example, consider three well-known types of misuse that would require unique mutation operators: (1) using DES for encryption (operator inserts prohibited parameter names, *e.g.,* DES), (2) trusting all SSL/TLS certificates (operator creates a malformed TrustManager), and (3) using a predictable initialization vector (IV) (operator derives predictable values for the IV). In fact, developers may even express the same misuse in different ways, necessitating unique operators to express such distinct *instances*, *e.g.,* the DES misuse expressed differently in Listing 1 and Listing 2. Thus, instead of adopting μSE's single-operator approach, MASC designs general usage-based mutation operators that can expressively instantiate misuses from our taxonomy of 105 misuses. In a similar manner, MASC's contextualized mutation abstractions (*i.e.,* for evaluating crypto-detectors) distinguish it from other systems that perform vulnerability injection for C programs [49], discover API misuse using mutation [50], [51], or evaluate static analysis tools for precision using handcrafted benchmarks or user-defined policies [41], [44].

Finally, the goal behind MASC is to assist the designers of crypto-detectors [2], [9], [10], [19], [34] in identifying design and implementation gaps in their tools, and hence, MASC is complementary to the large body of work in this area. Particularly, prior work provides rule-sets or benchmarks [52], [53] consisting of a limited set of cryptographic "bad practices" [54], or taxonomies of smaller subsets (*e.g.,* SSL/TLS misuse taxonomy by Vasan et al. [55]). However, we believe that ours is the first systematically-driven and comprehensive taxonomy of *crypto-API* misuse, which captures 105 cases that are further expanded upon into numerous unique misuse instances through MASC's operators. Thus, relative to prior handcrafted benchmarks, MASC can thoroughly test detectors with a far more comprehensive set of crypto-misuse instances.

IV. THE MASC FRAMEWORK

We propose a framework for Mutation-based Analysis of Static Crypto-misuse detection techniques (or MASC). Fig. 1 provides an overview of the MASC framework. As described previously ($\mathbf{RC}_1$), cryptographic libraries contain a sizable, diverse set of APIs, each with different potential misuse cases, leading to an exponentially large design space. Therefore, we initialize MASC by developing a *data-driven taxonomy of crypto-API misuse*, which grounds our evaluation in a unified collection of misuse cases observed in practice (Sec. V).

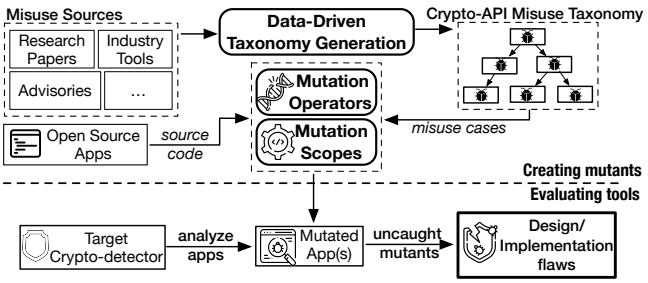

Fig. 1. A conceptual overview of the MASC framework.

The misuse cases in the taxonomy must be *instantiated* in an *expressive manner* to account for the diverse ways for expressing a misuse, *i.e., misuse instances*, that crypto-detectors may face in practice. For example, we previously described two ways of encrypting with DES: (1) providing DES as a variable in Cipher.getInstance(<parameter>) (Listing 1), or (2) using it in lowercase (Listing 2), which both represent something a benign developer might do (*i.e.,* threat **T**1). To represent all such instances without having to hard-code instantiations for every misuse case, we identify *usage-characteristics* of cryptographic APIs (particularly, in JCA), and leverage them to define *general*, *usage-based mutation operators*, *i.e.,* functions that can create misuse instances (*i.e., mutants*) by instantiating one or more misuse cases from the taxonomy (Sec. VI).

Upon instantiating mutants by applying our mutation operators to the misuse cases from the taxonomy, MASC *seeds*, *i.e.,* injects, the mutants into real Java/Android applications. The challenge here is to seed the mutants at specific locations that reflect the threat scenarios described in Sec. II-B, because crypto-detectors may not only face various instances of misuse cases, but also variations in *where* the misuse instances appear, *e.g.,* evasive (**T**3) developers may attempt to actively hide code to evade analysis. Thus, we define the abstraction of *mutation scopes* that place the instantiated mutants at strategic locations within code, emulating practical threat scenarios (Sec. VII). Finally, we analyze these *mutated apps* with the crypto-detector that is targeted for evaluation (Sec. IX), which results in *undetected/unkilled mutants* that can be then inspected to uncover design or implementation flaws (Sec. X).

V. TAXONOMY OF CRYPTOGRAPHIC MISUSE

To ground MASC in real cases of crypto API misuses, we systematically developed a taxonomy that provides a unified perspective of previously-known crypto-API misuse. In particular, we focus on identifying instances of crypto-API misuses in the Java ecosystem, due to its popularity and ubiquity, and because most crypto-detectors that we seek to evaluate were designed to analyze Java/Android apps.

As crypto-API misuse has been widely studied, it is likely that a majority of the misuse cases that we are interested in codifying are already present in existing literature. Therefore, our methodology identifies crypto-API misuses in existing artifacts sourced from both industry and academia, following Kitchenham et al.'s [56] guidelines for identifying relevant

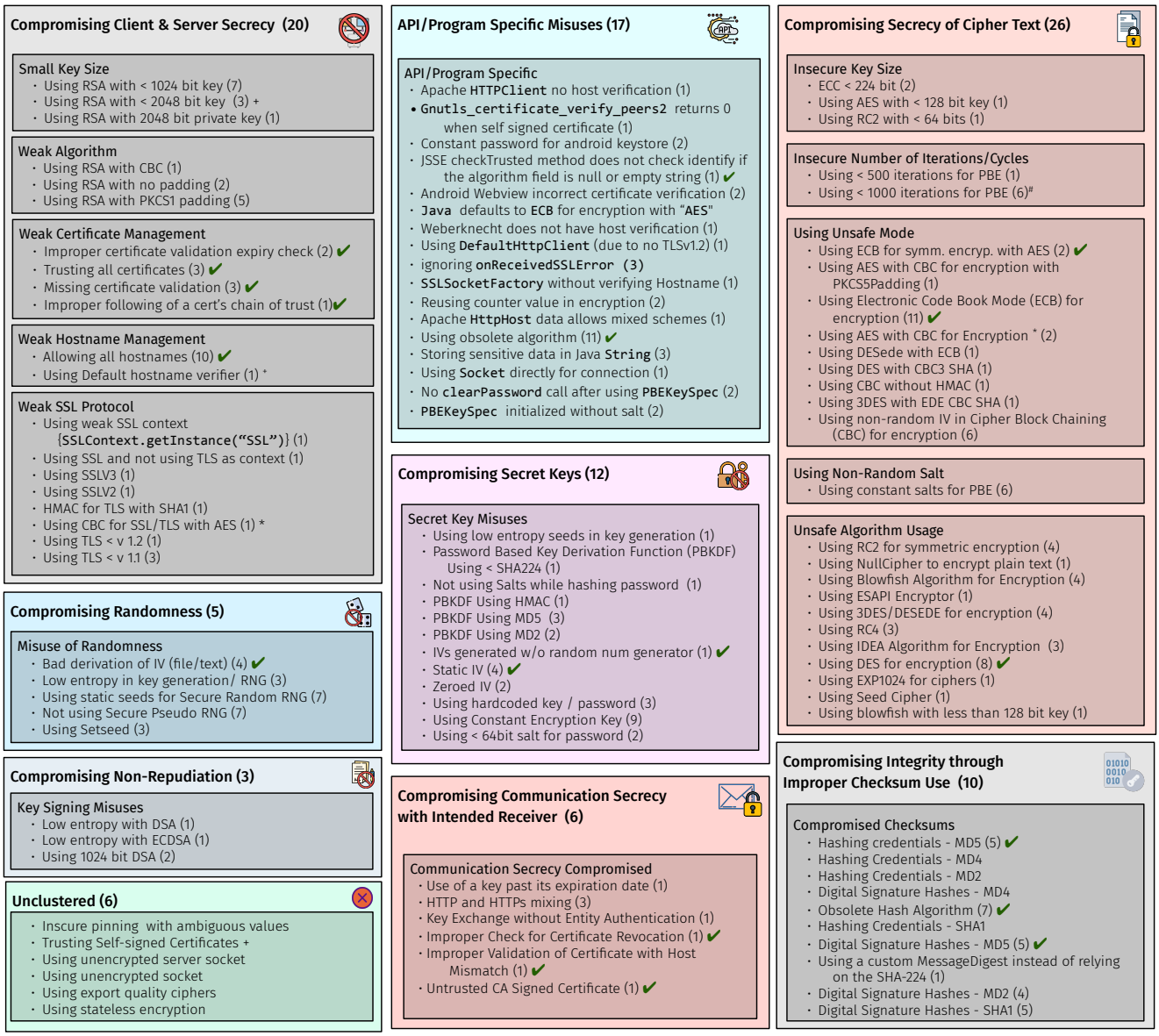

Fig. 2. The derived taxonomy of cryptographic misuses. (n) indicates misuse was present across n artifacts. A ✓indicates that the specific misuse case was instantiated with MASC's mutation operators for our evaluation (Sec. IX).

artifacts, as well as Petersen et al.'s [57] recommendations for constructing a systematic mapping, in three main steps: (1) identifying information sources, (2) defining the search, inclusion, and exclusion criteria for identifying artifacts, and (3) extracting misuse cases from artifacts and clustering them for ease of representation and extensibility. Two authors executed this methodology and created the taxonomy illustrated in Fig. 2. Data from each step is provided in the online appendix [30].

A. Identifying Information Sources

We considered information sources from both academia and industry. More specifically, we considered the proceedings of top-tier venues in security and software engineering (*i.e.,* USENIX Security, ACM CCS, IEEE S&P, NDSS, ICSE, ASE, FSE), *published after 1999, i.e., in the last 20 years*. Moreover, we also performed a thorough search for relevant keywords (Sec. V-B) in digital libraries, *i.e.,* the ACM Digital Library, IEEE Explore, and Google Scholar, which aided in identifying artifacts that may have fallen outside the top conferences. Finally, to incorporate sources outside academia, we studied guidelines from the Open Web Application Security Project (OWASP) [58], and documentation of several industry tools.

B. Search, Inclusion, and Exclusion Criteria

We select relevant artifacts from the identified sources using a keyword-based search with precise inclusion/exclusion criteria. We defined 3 classes of keyword phrases and enumerated several keyword combinations for each class, drawing from domain expertise and a random sample of artifacts.

To decide whether to consider an artifact for further analysis, we defined a simple ***inclusion criterion***, that the artifact should discuss crypto API misuse or its detection. We also defined an

exclusion criterion, *i.e.,* that the crypto-API misuse described by the artifact relates to a programming environment outside the Java ecosystem, was published prior to 1999, or does not contain relevant information. Following this methodology, we short-listed 40 artifacts for misuse extraction, *i.e.,* 35 from academia and 5 from industry. Note that we count multiple documents for a single industry tool as one artifact.

C. Misuse Extraction and Clustering

Two authors independently extracted, described, and grouped individual misuse cases from the 40 artifacts. More specifically, each identified misuse case was labeled using a specifically designed data extraction form (see online appendix for figure [30]). The two authors met and resolved disagreements, to eventually identify 105 *unique misuse cases*.

Such a large set of misuse cases could prove intractable for direct analysis or extension. Hence, we constructed a *categorized* taxonomy by grouping the discovered misuse cases into *semantically meaningful clusters*. Each author constructed the clusters as per two differentiating criteria: (1) the *security goal/property* represented by the misuse cases (*e.g.,* secrecy, integrity, non-repudiation), and (2) its *level of abstraction* (*i.e.,* specific context) within the communication/computing stack (*e.g.,* confidentiality in general, or confidentiality with respect to SSL/TLS). The two authors met and reached agreement on a taxonomy consisting of 105 *misuse cases* grouped into *nine semantic clusters*, as shown in Fig. 2. The entire process of taxonomy generation took over *two person-months* in effort.

VI. Usage-based Mutation Operators

In designing our mutation operators, we must balance the dichotomous tradeoff between representing as many misuse cases (and their corresponding variations) as possible, while also creating a tractable number of operators that can be reasonably maintained in the future. Thus, building a large set of hard-coded operators that are tightly coupled with specific misuse cases would be infeasible from an engineering/maintenance perspective. Further, to discover new issues in crypto-detectors, these operators should not exploit general soundiness-related [46], [59]) limitations, such as dynamic code execution and implicit calls. Therefore, we seek to build operators that are general enough to be maintainable, but which also provide expressive instantiation of several misuse cases, guided by the threat model in Section II-B, and without focusing on any specific static analysis technique or soundiness issue.

We define the abstraction of *usage-based mutation operators*, inspired by a key observation: misuse cases that are unrelated in terms of the security problem may still be related in terms of *how the crypto APIs corresponding to the misuse cases are expected to be used*. Thus, characterizing the common usage of crypto APIs would allow us to mutate that characterization and define operators that apply to multiple misuse cases, while remaining independent of the specifics of each misuse.

Common Crypto-API Usage Characteristics: We identified two common patterns in crypto-API usage by examining crypto-API documentation from JCA, and our taxonomy, which we term as (1) *restrictive* and (2) *flexible* invocation. To elaborate, a developer can only instantiate certain objects by providing values from a predefined set, hence the name *restrictive* invocation; *e.g.,* for symmetric encryption with the `Cipher.getInstance(<parameter>)` method, JCA *only* accepts predefined configuration values for the algorithm name, mode, and padding, in String form. Conversely, JCA allows significant extensibility for certain crypto APIs, which we term as *flexible* invocation; *e.g.,* developers can customize the hostname verification component of the SSL/TLS handshake by creating a class extending the `HostnameVerifier`, and overriding its `verify` method, with *any content*. We leverage these notions of *restrictive* & *flexible* usage to define our operator types.

A. Operators based on Restrictive Crypto API Invocation

Our derived taxonomy indicates that several parameter values used in restrictive API invocations may not be secure (*e.g.,* `DES`, or `MD5`). Therefore, we designed six mutation operator types (**OPs**) that apply a diverse array of transformations to such values and generate API invocations that are acceptable/-compilable as per JCA syntax and conventions, but not secure.

OP_1: *Atypical case* – This operator changes the case of algorithm specification misuse to an atypical form (*e.g.,* lowercase), and represents accidental misuse/typos by developers (*i.e.,* **T**1). For example, as previously shown in Listing 1, this operator would instantiate the `DES` misuse by specifying "des" (lowercase) in the `Cipher.getInstance(<parameter>)` API.

OP_2: *Weak Algorithm in Variable* – This operator represents the relatively common usage of *expressing API arguments in variables* before passing them to an API, and can be applied to instantiate all misuse cases that are represented by restrictive API invocations (*e.g.,* `DES` instantiation in Listing 2).

OP_3: *Explicit case fix* – This operator instantiates misuse cases by using the atypical case for an argument (*e.g.,* algorithm name) in a restrictive API invocation, as seen in OP_1, but also explicitly *fixes* the case, emulating a developer attempting to follow conventions by explicitly invoking String manipulation methods (*i.e.,* **T**2); *e.g.,*`SSLContext.getInstance("ssl".toUpperCase())` is one instantiation of the misuse of using `SSL2 or SSLv3` protocols.

OP_4: *Removing noise* – This operator extends OP_3 by defining transformations that are more complex than simple case changes, such as removing extra characters or "noise" present in the arguments, which is likely if the arguments are acquired from a properties/database file; *e.g.,* `Cipher.getInstance("DES//".replace("//",""))`.

OP_5: *Method chains* – This operator performs arbitrary transformations (*e.g.,* among those in OP_1 – OP_4) on the restricted argument string, but splits the transformation into a chain of procedure calls, thereby hiding the eventual value (see Listing 3 in Appendix A). Such behavior can be attributed to an evasive developer (**T**3).

OP_6: *Predictable/Non-Random Derivation* – This operator emulates a benign developer (**T**1) attempting to *derive a random value* (*i.e.,* instead of using a cryptographically-secure random number generator), by performing seemingly complex

operations resulting in a predictable value, and then using the derived value to obtain other cryptographic parameters, such as IVs. For example, Listing 4 in Appendix A shows such an instantiation of the "Bad derivation of IV" misuse from the taxonomy that uses the system time to derive the IV.

B. Operators based on Flexible Crypto API Invocations

In contrast with restrictive APIs, Java allows several types of flexible extensions to crypto APIs represented by interfaces or abstract classes, *only enforcing typical syntactic rules*, with little control over what semantics developers express. Thus, we consider three particular types of flexible extensions developers may make, and hence, our **OPs** may emulate, in the context of API misuse cases that involve flexible invocations: (1) *method overriding*, (2) *class extension*, and (3) *object instantiation*.

1) Method Overriding: Crypto APIs are often declared as interfaces or abstract classes containing abstract methods, to facilitate customization. These abstract classes provide a fertile ground for defining mutation operators with a propensity for circumventing detectors (*i.e.,* considering threats **T**3).

OP$_7$: *Ineffective Exceptions* – If the correct behavior of a method is to throw an exception, such as invalid certificate, this operator creates misuse instances of two types: (1) not throwing any exception, and (2) throwing an exception within a conditional block that is only executed when a highly unlikely condition is true. For example, as shown in Listing 5 in Appendix A, this operator instantiates a weak `TrustManager` by implementing a `checkServerTrusted` method containing a condition block is unlikely to be executed.

OP$_8$: *Ineffective Return Values* – If the correct behavior of a method is to return a specific value to denote security failure, this operator modifies the return value to create two misuse instances: (1) if the return type is boolean, return a predefined boolean value that weakens the crypto feature, or return *null* otherwise, and (2) introduce a condition block that will always, prematurely return a misuse-inducing boolean/null value before the secure return statement is reached. Contrary to **OP$_7$**, this operator ensures that the condition will always return the value resulting in misuse (see Listing 6 in Appendix A).

OP$_9$: *Irrelevant Loop* – This operator adds loops that *seem to perform some work*, or security checks, before returning a value, but in reality *do nothing to change the outcome*, emulating an evasive (**T**3) developer.

2) Class Extension: Creating abstractions on top of previously-defined on crypto classes is fairly common; *e.g.,* the abstract class `X509ExtendedTrustManager` implements the interface `X509TrustManager` in the JCA, and developers can be expected to extend/implement both these constructs in order to customize certificate verification for their use-cases. Similarly, developers may create abstract subtypes of an interface or abstract class; *e.g.,* as shown in Listing 7 (Appendix A), the `X509TrustManager` can be both extended and implemented by an abstract type interface and an abstract class respectively. Our next set of operators is motivated by this observation:

OP$_{10}$: *Abstract Extension* with *Override* – This mutation operator creates an abstract sub-type of the parent class (*e.g.,* the `X509TrustManager` as shown in Listing 7), but this time, overrides the methods *incorrectly*, *i.e.,* adapting the techniques in **OP$_7$** – **OP$_9$** for creating various instances of misuse.

OP$_{11}$: *Concrete Extension* with *Override* – This mutation operator creates a concrete class based on a crypto API, *incorrectly* overriding methods similar to **OP$_{10}$**.

3) Object Instantiation Operators: In Java, objects can be created by calling either the default or a parametrized constructor, and moreover, it may also be possible to override the properties of the object through Inner class object declarations. We leverage these properties to create **OP$_{12}$** as follows:

OP$_{12}$: *Anonymous Inner Object* – Creating an instance of a flexible crypto API through constructor or anonymous inner class object is fairly common, as seen for `HostnameVerifier` in Oracle Reference Guide [60] and Android developer documentation [61], respectively. Similarly, this operator creates anonymous inner class objects from abstract crypto APIs, and instantiates misuse cases by overriding the abstract methods using **OP$_7$** – **OP$_9$**, as shown in Listing 8 (Appendix A), where the misuse is introduced through **OP$_8$**.

`MASC's` 12 operators are capable of instantiating **69/105** (65.71%), misuse cases distributed across *all* 9 *semantic clusters*. This indicates that `MASC's` operators can express a *diverse majority* of misuse cases, signaling a reasonable trade-off between the number of operators and their expressivity. Of the remaining 36 cases that our operators do not instantiate, 16 are trivial to implement (*e.g.,* using AES with a < 128 bit key, see Listing 14, Appendix A). Finally, 20 cases (19.01%) would require a non-trivial amount of additional engineering effort; *e.g.,* designing an operator that uses a custom `MessageDigest` algorithm instead of a known standard such as `SHA-3`, which would require a custom hashing algorithm.

VII. Threat-based Mutation Scopes

We seek to emulate the typical placement of vulnerable code by benign (**T**1, **T**2), and evasive (**T**3) developers, for which we design three *mutation scopes*:

1. *Main* Scope: The *main scope* is the simplest of the three, and seeds mutants at the beginning of the main method of a simple Java or Android template app developed by the authors (*i.e.,* instead of the real, third-party applications mutated by the other two scopes). This specific seeding strategy ensures that the mutant would always be reachable and executable, emulating basic placement by a benign developer (**T**1, **T**2).

2. *Similarity* Scope: The *similarity scope* seeds security operators at locations in a target application's source code where a similar API is already being used, *i.e.,* akin to modifying existing API usages and making them vulnerable. Hence, this scope emulates code placement by a typical, well-intentioned, developer (**T**1, **T**2), assuming that the target app we mutate is also written by a benign developer. This helps us evaluate if the crypto-detector is able to detect misuse at *realistic* locations.

3. *Exhaustive* Scope: As the name suggests, this scope *exhaustively* seeds mutants at *all locations* in the target app's code, *i.e.,* class definitions, conditional segments, method bodies as well as anonymous inner class object declarations. Note that

some mutants may not be seeded in all of these locations; *e.g.,* a `Cipher.getInstance(<parameter>)` is generally surrounded by `try-catch` block, which cannot be placed at class scope. This scope emulates placement by an evasive developer (**T**3).

VIII. Implementation

The implementation of MASC involved three components, namely, (1) selecting misuse cases from the taxonomy for mutation, (2) implementing mutation operators that instantiate the misuse cases, and (3) seeding/inserting the instantiated mutants in Java/Android source code at targeted locations.

1. Selecting misuse cases from the Taxonomy: We chose 19 misuse cases from the taxonomy for mutation with MASC's 12 operators (indicated by a ✓ in Fig. 2), focusing on ensuring broad *coverage* across our taxonomy as well as on their *prevalence*, *i.e.,* prioritizing misuse cases that are discussed more frequently in the artifacts used to construct the taxonomy, and which most crypto-detectors can be expected to target. We expand on some of these choices in Appendix B.

2. Implementing mutants: The mutation operators described in Sec. VI are designed to be applied to one or more crypto APIs, for instantiating specific misuse cases. To ensure compilable mutants by design, MASC carefully considers the syntactic requirements of the API being implemented (*e.g.,* the requirement of a surrounding try-catch block with appropriate exception handling), as well as the semantic requirements of a particular misuse being instantiated, such as the need to throw an exception only under a truly improbable condition (*e.g.,* as expressed in $\mathbf{OP}_7$). MASC uses Java Reflection to determine all the "syntactic glue" for automatically instantiating a compilable mutant, *i.e.,* exceptions thrown, requirements of being surrounded by a try-catch block, the need to implement a certain abstract class and certain methods, etc. MASC then combines this automatically-derived and compilable syntactic glue with parameters, *i.e.,* values to be used in arguments, return statements, or conditions, which we manually define for specific operators (and misuse cases), to create mutants.

To further ensure compilability and evaluation using only compilable mutants, we take two steps: (1) We use Eclipse JDT's AST-based checks for identifying syntactic anomalies in the generated mutated apps, and (2) compile the mutated app automatically using build/test scripts *provided with the original app*. In the end, every single mutant analyzed by the target crypto-detector is compilable and accurately expresses the particular misuse case that is instantiated. This level of automation allows MASC to create thousands of mutants with very little manual effort, and makes MASC extensible to future evolution in Java cryptographic APIs (addressing $\mathbf{RC}_3$).

3. Identifying Target Locations and Seeding Mutants: To identify target locations for the similarity scope, we extended the MDroid+ [62], [63] mutation analysis framework, by retargeting its procedure for identifying suitable mutant locations, adding support for dependencies that crypto-based mutations may introduce, and enabling identification of anonymous inner class objects as mutant-seeding locations, resulting in 10 additional, custom AST- and string-based location detectors. Further, MASC extends μSE [45] to implement the *exhaustive* scope, *i.e.,* to identify locations where crypto-APIs can be feasibly inserted to instantiate compilable mutants (*e.g.,* a `Cipher.getInstance(<parameter>)` has to be contained in a `try-catch` block, making it infeasible to insert at class-level).

Finally, although we have heavily tested MASC, there may be corner cases due to our implementation that result in compilation errors. We observed ≈ 20 uncompilable mutants during our evaluation with over $20,303$ mutants, *i.e.,* in 0.098% cases. These errors do not affect the soundness of the evaluation, as these non-compilable mutants are simply discarded.

IX. Evaluation Overview and Methodology

The two main goals of our evaluation are to (1) measure the effectiveness of MASC at uncovering flaws in crypto-detectors, and (2) learn the characteristics of the flaws and their real-world impact, in terms of the security of end-user applications. Therefore, we formulate the following research questions:

- $\mathbf{RQ}_1$**:** *Can* MASC *discover flaws in crypto-detectors?*
- $\mathbf{RQ}_2$**:** *What are the characteristics of these flaws?*
- $\mathbf{RQ}_3$**:** *What is the impact of the flaws on the effectiveness of crypto-detectors in practice?*

To answer $\mathbf{RQ}_1$ – $\mathbf{RQ}_3$, we first used MASC to evaluate a set of *nine* major crypto-detectors, namely CryptoGuard, CogniCrypt, Xanitizer, Tool$_X$, SpotBugs with FindSecBugs, QARK, LGTM, Github Code Security (GCS), and ShiftLeft Scan, prioritizing practically relevant tools that were recent and under active maintenance. As MASC's usefulness is in systematically evaluating individual crypto-detectors by characterizing their detection-ability, with the goal of enabling improvement in the tools, and hence, the results of our evaluation indicate gaps in individual tools, and not comparative advantages.

Step 1 – Selecting and mutating apps: We use MASC to mutate 13 open source Android apps from F-Droid [64] and Github [65], and four sub-systems of Apache Qpid Broker-J [66], a large Java Apache project (for list see [30]). Our selection criteria was biased towards popular projects that did not contain obsolete dependencies (*i.e.,* compilable using Android Studio 4/Java LTS 1.8). Moreover, we specifically used the similarity scope on 3/13 Android apps, and all 4 Qpid sub-systems, as they contained several types of crypto API usage (*e.g.,* `Cipher`, `MessageDigest`, `X509TrustManager`). In total, we generated $2,515$ mutants using the 13 Android apps (`src` & `apk`) and $17,788$ mutants using the 4 Java programs (`src` & `jar`), totaling $20,303$ mutants. We confirmed that each mutated app was compilable and runnable. Generating these 20k mutants took MASC roughly 15 minutes, and did not require any human intervention, addressing $\mathbf{RC}_3$. As the cost to generate this volume of mutants is feasible, MASC may not benefit from generating a focused subset of mutants (see Appendix B).

Step 2 – Evaluating crypto-detectors and identifying *unkilled/undetected* mutants: To evaluate a crypto-detector, we analyzed the mutants using the crypto-detector, and identified the mutants that were *not killed* by it, *i.e., undetected* as misuse. To facilitate accurate identification of killed mutants, we compare the mutation log MASC generates when inserting

mutants (which describes the precise location and type of mutant injected, for each mutant), and the reports from crypto-detectors, which for all the tools contained precise location features such as the class/method names, line numbers, and other details such as associated variables. To elaborate, we use the following methodology: We first compare the analysis report generated by the crypto-detector/target on a mutated app, with its analysis report on the original (*i.e.,* unmutated) version of the same app. Any difference in the two reports can be attributed to mutants inserted by MASC, *i.e.,* denotes an instance of a "mutant being killed". To identify the mutant that was killed, we obtain location features (*i.e.,* type, file, class, method, line number and/or variables associated) of the specific "killed mutant" from the crypto-detectors report on the mutated app, and use them to search for a unique mutant in MASC's mutation log. If a match is found, we mark that specific mutant as killed. We additionally confirm the location of each killed mutant by referring to the mutated app's source code. Once all the killed mutants are identified, the remaining mutants (*i.e.,* inserted in MASC's mutation but not killed) are flagged as unkilled.

This approach ensures that alarms by the crypto-detector for anomalies not inserted by MASC are not considered in the evaluation, and all mutants inserted by MASC are identified as either killed or unkilled. Our semi-automated implementation of this methodology, which adapts to the disparate report formats of the studied crypto-detectors, can be found in Appendix B. The evaluation resulted in $7,540$ undetected mutants on average across all detectors (see Table III in the Appendix).

Step 3 – Identifying *flaws* ($\mathbf{RQ}_1$): We analyzed 350 random undetected mutants to discover *flaws*, wherein a flaw is defined as a misuse case that a particular crypto-detector *claims* to detect in its documentation, but fails to detect in practice. We took care to also exempt the exceptions/limitations explicitly stated in the crypto-detector's documentation, ensuring that all of our identified flaws are *novel*. On a similar note, while a crypto-detector may seem flawed because it does not detect a newer, more recently identified misuse pattern, we confirm that all the flaws we report are due to misuse cases that are older than the tools in which we find them. This can be attributed to two features of our evaluation: our choice of the most frequently discussed misuse cases in a taxonomy going back 20 years, and, our choice of tools that were recently built or maintained. Finally, to confirm the flaw without the intricacies of the mutated app, we created a corresponding minimal app that contained only the undetected misuse instance (*i.e.,* the flaw), and re-analyzed it with the detector.

Step 4 – Characterizing flaws ($\mathbf{RQ}_2$): We characterized the flaws by grouping them by their most likely cause, into *flaw classes*. We also tested each of the nine tools with the minimal examples for all the flaws, allowing us to further understand each flaw given its presence/absence in certain detectors, and their documented capabilities/limitations. We reported the flaws to the respective tool maintainers, and contributed 3 patches to CryptoGuard that were all integrated [30].

Step 5 – Understanding the practical *impact* of flaws ($\mathbf{RQ}_3$): To gauge the impact of the flaws, we studied publicly available applications *after* investigating $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$. We first tried to determine if the misuse instances similar to the ones that led to the flaws were also found in real-world apps (*i.e.,* public GitHub repositories) using GitHub Code Search [67], followed by manual confirmation. Additionally, we manually searched Stack Overflow [68] and Cryptography Stack Exchange [69] for keywords such as "unsafe hostnameverifier" and "unsafe x509trustmanager". Finally, we narrowed our search space to identify the impact on apps *that were in fact analyzed with a crypto-detector*. As only LGTM showcases the repos that it scans, we manually explored the top 11 Java repositories from this set (prioritized by the number of contributors), and discovered several misuse instances that LGTM tool may have failed to detect due to the flaws discovered by MASC.

Step 6 – Attributing flaws to mutation vs. base instantiation: To determine whether a flaw detected by MASC can be attributed to mutation, versus the crypto-detector's inability to handle even a base case (*i.e.,* the most literal instantiations of the misuse case from the taxonomy), we additionally evaluated each crypto-detector with base instantiations for each misuse that led to a flaw exhibited by it.

X. Results and Findings

Our manual analysis of undetected mutants revealed 19 flaws across our 9 crypto-detectors that we resolved to both design and implementation-gaps ($\mathbf{RQ}_1$). We organize these flaws into five *flaw classes* ($\mathbf{RQ}_2$), representing the shared limitations that caused them. Table I provides the complete list of the flaws, categorized along flaw classes, while Table II provides a mapping of the flaws to the crypto-detectors that exhibit them.

As shown in Table II, a majority of the total flaws (computed by adding ✗ and Ø instances) identified in crypto-detectors, *i.e.,* 45/76 or 59.21% can be *solely* attributed to our mutation-based approach, whereas only 31/76, *i.e.,* 40.79% could *also* be found using base instantiations of the corresponding misuse cases. Further, all flaws in 6/9 crypto-detectors were only identified using MASC. This demonstrates the advantage of using mutation, over simply evaluating with base instantiations of misuses from the taxonomy. GCS, LGTM, and QARK fail to detect base cases due to incomplete rule sets (see Appendix B for discussion).

At the initial stage of our evaluation (*i.e.,* before we analyzed uncaught mutants), we discovered that certain crypto-detectors [3], [9] analyze only a limited portion of the target applications, which greatly affects the reliability of their results. As these gaps were not detected using MASC's evaluation, we do not count these in our flaws or flaw classes. However, due to their impact on the basic guarantees provided by crypto-detectors, we believe that such gaps must be discussed, addressed, and avoided by future crypto-detectors, which is why we discuss them under a flaw class *zero*.

Flaw Class Zero (FC0) – *Incomplete analysis of target code*: Starting from Android API level 21 (Android 5.0), apps with over 64k reference methods are automatically split to multiple Dalvik Executable (DEX) byte code files in the form of `classes<N>.dex`. Most apps built after 2014 would fall into

TABLE I
DESCRIPTIONS OF FLAWS DISCOVERED BY ANALYZING CRYPTO-DETECTORS.

ID	Flaw Name (Operator)	Description of Flaws
	FLAW CLASS 1 (FC1): STRING CASE MISHANDLING +	
F1	smallCaseParameter ($\mathbf{OP}_1$)	Not detecting an insecure algorithm provided in lower case; *e.g.,*`Cipher.getInstance("des");`
	FLAW CLASS 2 (FC2): INCORRECT VALUE RESOLUTION +	
F2	valueInVariable ($\mathbf{OP}_2$)	Not resolving values passed through variables. *e.g.,*`String value = "DES"; Cipher.getInstance(value);`
F3*	secureParameterReplaceInsecure ($\mathbf{OP}_4$)	Not resolving parameter replacement; *e.g.,* `MessageDigest.getInstance("SHA-256".replace("SHA-256", "MD5"));`
F4*	insecureParameterReplaceInsecure ($\mathbf{OP}_4$)	Not resolving an *insecure* parameter's replacement with another *insecure* parameter *e.g.,* `Cipher.getInstance("AES".replace("A", "D"));` (*i.e.,* where "AES" by itself is insecure as it defaults to using ECB).
F5*	stringCaseTransform ($\mathbf{OP}_3$)	Not resolving the case after transformation for analysis; *e.g.,*`Cipher.getInstance("des".toUpperCase(Locale.English));`
F6*	noiseReplace ($\mathbf{OP}_4$)	Not resolving noisy versions of insecure parameters, when noise is removed through a transformation; *e.g.,* `Cipher.getInstance("DE$S".replace("$", ""));`
F7	parameterFromMethodChaining ($\mathbf{OP}_5$)	Not resolving insecure parameters that are passed through method chaining, *i.e.,* from a class that contains both secure and insecure values; *e.g.,*`Cipher.getInstance(obj.A().B().getValue());` where obj.A().getValue() returns the secure value, but obj.A().B().getValue(), and obj.B().getValue() return the insecure value.
F8*	deterministicByteFromCharacters ($\mathbf{OP}_6$)	Not detecting *constant IVs*, if created using complex loops, casting, and string transformations; *e.g.,* a `new IvParameterSpec(v.getBytes(),0,8)`, which uses a `String v=""; for(int i=65; i<75; i++){ v+=(char)i;}`
F9	predictableByteFromSystemAPI ($\mathbf{OP}_6$)	Not detecting *predictable IVs* that are created using a predictable source (*e.g.,* system time), converted to bytes; *e.g.,*`new IvParameterSpec(val.getBytes(),0,8);`, such that `val = new Date(System.currentTimeMillis()).toString();`
	FLAW CLASS 3 (FC3): INCORRECT RESOLUTION OF COMPLEX INHERITANCE AND ANONYMOUS OBJECTS	
F10	X509ExtendedTrustManager ($\mathbf{OP}_{12}$)	Not detecting *vulnerable SSL verification* in *anonymous inner class objects* created from the `X509ExtendedTrustManager` class from JCA; *e.g.,* see Listing 9 in Appendix).
F11	X509TrustManagerSubType ($\mathbf{OP}_{12}$)	Not detecting *vulnerable SSL verification* in anonymous inner class objects *created from an empty abstract class* which implements the `X509TrustManager` interface; *e.g.,* see Listing 12).
F12	IntHostnameVerifier ($\mathbf{OP}_{12}$)	Not detecting *vulnerable hostname verification* in an anonymous inner class object that is created from an *interface that extends* the `HostnameVerifier` interface from JCA; *e.g.,* see Listing 13 in Appendix.
F13	AbcHostnameVerifier ($\mathbf{OP}_{12}$)	Not detecting *vulnerable hostname verification* in an anonymous inner class object that is created from an *empty abstract class* that *implements* the `HostnameVerifier` interface from JCA; *e.g.,* see Listing 11 in Appendix.
	FLAW CLASS 4 (FC4): INSUFFICIENT ANALYSIS OF GENERIC CONDITIONS IN EXTENSIBLE CRYPTO-APIS	
F14	X509TrustManagerGenericConditions ($\mathbf{OP}_7$, $\mathbf{OP}_9$, $\mathbf{OP}_{12}$)	Insecure validation of a overridden `checkServerTrusted` method created within an anonymous inner class (constructed similarly as in **F13**), due to the failure to detect *security exceptions thrown under impossible conditions*; *e.g.,* `if(!(true||arg0 == null||arg1 == null)) throw new CertificateException();`
F15	IntHostnameVerifierGenericCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.,* the `verify()` method within an anonymous inner class (constructed similarly as in **F14**), due to the failure to detect *an always-true condition block that returns* `true`; *e.g.,*`if(true || session == null) return true; return false;`
F16	AbcHostnameVerifierGenericCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.,* the `verify()` method within an anonymous inner class (constructed similarly as in **F15**), due to the failure to detect *an always-true condition block that returns* `true`; *e.g.,*`if(true || session == null) return true; return false;`
	FLAW CLASS 5 (FC5): INSUFFICIENT ANALYSIS OF CONTEXT-SPECIFIC, CONDITIONS IN EXTENSIBLE CRYPTO-APIS	
F17	X509TrustManagerSpecificConditions ($\mathbf{OP}_7$, $\mathbf{OP}_{12}$)	Insecure validation of a overridden `checkServerTrusted` method created within an anonymous inner class created from the `X509TrustManager`, due to the failure to detect *security exceptions thrown under impossible but context-specific conditions, i.e., conditions that seem to be relevant due to specific variable use, but are actually not*; *e.g.,*`if (!(null != s || s.equalsIgnoreCase("RSA")|| certs.length >= 314))throw new CertificateException("RSA");`
F18	IntHostnameVerifierSpecificCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.,* the `verify()` method within an anonymous inner class (constructed similarly as in **F14**), due to the failure to detect *a context-specific always-true condition block that returns* `true`; *e.g.,*`if(true || session.getCipherSuite().length()>=0) return true; return false;`
F19	AbcHostnameVerifierSpecificCondition ($\mathbf{OP}_8$, $\mathbf{OP}_{12}$)	Insecure analysis of vulnerable hostname verification, *i.e.,* the `verify()` method within an anonymous inner class (constructed similarly as in **F15**), due to the failure to detect *a context-specific always-true condition block that returns* `true`; *e.g.,*`if(true || session.getCipherSuite().length()>=0)return true; return false;`

+ flaws were observed for mulitple API misuse cases
*Certain seemingly-unrealistic flaws may be seen in or outside a crypto-detector's "scope", depending on the perspective; see Section XII for a broader treatment of this caveat.

this category, as the Android Studio IDE automatically uses *multidex* packaging for any app built for Android 5.0 or higher [70]. However, we discovered that CryptoGuard and CogniCrypt do not handle multiple dex files (see Table II), despite being *released 4 and 5 years after Android 21, respectively*. Given that this flaw affected CryptoGuard's ability to analyze a majority of Android mutants (*i.e.,* detecting only only $871/2515$), we developed a patch that fixes this issue, and used the patched version of CryptoGuard for our evaluation. The CogniCrypt maintainers confirmed that they are working on this issue in their upcoming release, which is why we did not create a patch, but used non-multidex minimal examples to confirm that each flaw discovered in CogniCrypt is not due to the multidex issue. Similarly, we discovered that CryptoGuard ignores any class whose package name contains "`android.`" or ends in "`android`", which prevents it from analyzing important apps such as LastPass (`com.lastpass.lpandroid`) and LinkedIn (`com.linkedin.android`), which indicates the severity of even trivial implementation flaws. We submitted a patch to address this flaw and evaluated the patched version [30].

The remainder of this section discusses each flaw class with representative examples as they manifest in specific crypto-detectors, as well as the *impact* of the flaws in terms of examples found in real software.

FC1: String Case Mishandling (F1): As discussed in the motivating example (and seen in **F1** in Table I), a developer may use `des` or `dEs` (instead of `DES`) in `Cipher.getInstance(<parameter>)` without JCA raising exceptions. CryptoGuard does not detect such misuse. We submitted a patch to CryptoGuard to address this flaw, which was accepted, and demonstrates that this flaw was recognized as

TABLE II
FLAWS OBSERVED IN DIFFERENT STATIC CRYPTO-DETECTORS

Class	ID	CG	CC	SB	XT	TX	QA	SL	GCS	LGTM
FC1	F1	✗	✓	✓	✓	✓	Ø	✓	✓	✓
FC2	F2	✓	✓	✓	✓	✗	Ø	✓	◐	◐
	F3*	✗	✗	✗	✓	✗	Ø	✗	✓	✓
	F4*	✗	✗	✗	✓	✗	Ø	✗	✗	✗
	F5*	✗	✗	◐	✓	✗	Ø	◐	✓	✓
	F6*	✗	✗	✗	✓	✗	Ø	✗	✗	✗
	F7	✗	✓	✗	✓	✗	Ø	✗	✓	✓
	F8	✗	✓	✓	✓	-	Ø	✓	Ø	✗
	F9	✗	✓	✓	✓	-	Ø	✓	Ø	✗
FC3	F10	✗	-	✓	✓	◐	◐	✓	Ø	Ø
	F11	✗	-	✓	✓	◐	◐	✓	Ø	Ø
	F12	✗	-	✓	✓	✓	-	✓	Ø	Ø
	F13	✗	-	✓	✓	✓	-	✓	Ø	Ø
FC4	F14	✗	-	✓	✓	✓	✗	✓	Ø	Ø
	F15	✗	-	✓	✓	✓	-	✓	Ø	Ø
	F16	✗	-	✓	✓	✓	-	✓	Ø	Ø
FC5	F17	✗	-	✗	✗	✓	✗	✗	Ø	Ø
	F18	✗	-	✓	✗	✓	-	✓	Ø	Ø
	F19	✗	-	✓	✓	✓	-	✓	Ø	Ø

✗ = Flaw Present, ✓ = Flaw Absent, ◐ = Flaw partially present, -= detector does not claim to handle the misuse associated with the flaw, Ø= detector claims to handle but did not detect base version of misuse;
CG = CryptoGuard, CC = CogniCrypt, SB = SpotBugs, XT = Xanitizer, TX = Tool$_X$, QA = QARK, SL = ShiftLeft, GCS = Github Code Security.
*Certain seemingly-unrealistic flaws may be seen in/outside a crypto-detector's "scope", depending on the perspective; see Section XII for a broader treatment of this caveat.

an *implementation gap* by the CryptoGuard developers [30].

FC2: Incorrect Value Resolution (F2 – F9): The flaws in this class occur due to the inability of 8/9 of the crypto-detectors to resolve parameters passed to crypto-APIs. For example, consider **F2**, previously discussed in Listing 1 in Section I, where the string value of an algorithm type (*e.g.,* `DES`) is passed through a variable to `Cipher.getInstance(<parameter>)`. Tool$_X$ was not able to detect this (mis)use instance, hence exhibiting **F2** (Table II), and in fact, demonstrated a consistent inability to resolve parameter values (flaws **F2** – **F7**), indicating a *design gap*. On the contrary, as SpotBugs partially detects the type of misuse represented in **F5**, *i.e.,* when it is instantiated using the `Cipher.getInstance(<parameter>)` API, but not using the `MessageDigest.getInstance(<parameter>)` API, which indicates an *implementation gap*.

Further, LGTM and GCS are partially susceptible to F2 because of an intricate problem in their rulesets. That is, both tools are capable of tracking values passed from variables, and generally detect mutations similar to the one in F2 (*i.e.,* and also the one in Listing 1, created using **OP**$_2$). However, one of the mutant instances that we created using **OP**$_2$ used `AES` in `Cipher.getInstance(<parameter>)`, which may seem correct but is actually a misuse, since specifying `AES` alone initializes the cipher to use the highly vulnerable `ECB` mode as the block chaining mode. Unfortunately, both LGTM and GCS use the same CODEQL ruleset [71] which doesn't consider this nuance, leading both tools to ignore this misuse.

Finally, we observe that CogniCrypt detects some of the more complex behaviors represented by flaws in **FC2** (*i.e.,* **F7** – **F9**), but does not detect the simpler ones (**F3** – **F6**). From our interaction with CogniCrypt's maintainers, we have discovered that CogniCrypt should be able to detect such transformations by design, as they deviate from CrySL rules. However, in practice, CogniCrypt cannot reason about certain transformations at present (but could be modified to do so in the future), and produces an ambiguous output that neither flags such instances as misuse, nor as warnings for manual inspection, due to an implementation gap. The developers agree that CogniCrypt should clearly flag the API-use that it does not handle in the report, and refer such use to manual inspection.

Impact (FC2): We found misuse similar to the instance in **F2** in Apache Druid, an app with $10.3K$ stars and 400 contributors on Github (see Listing 17 in Appendix A). Further, we found real apps that convert the case of algorithm values before using them in a restrictive crypto API [72] (**F5**, instantiated using **OP**$_3$), or process values to replace "noise" [73] (**F3**, **F4**, **F6**, instantiated using **OP**$_4$). We observed that ExoPlayer, a media player from Google with over 16.8K stars on GitHub, used the predictable *Random* API for creating `IvParameterSpec` objects [74] until 2019, similar in nature to **F9**. Developers also use constants for IVs (**F8**), as seen in UltimateAndroid [75] (2.1K stars), and JeeSuite (570 stars) [76].

We also found instances of these flaws in apps that were analyzed with a crypto-detector, specifically, LGTM. Apache Ignite [77] (360 contributors, 3.5K stars) contains a misuse instance similar to one that led to **F2**, where only the name of the cipher is passed to the `Cipher.getInstance(<parameter>)` API [78] which causes it to default to "ECB" mode. LGTM does not report this as it considers `ECB` use as insecure for Java (but oddly secure for JavaScript). We found similar instances of `ECB` misuse in Apache-Hive [79] (250 contributors, 3.4K stars), Azure SDK for Java [80] (328 contributors, 857 stars), which LGTM would not detect. Finally, in Apache Ignite [77], we found a `Cipher.getInstance(<parameter>)` invocation that contained a method call in place of the cipher argument (*i.e.,* a chain of length 1, a basic instance of **F7**) [78].

FC3: Incorrect Resolution of Complex Inheritance and Anonymous Objects (F10 – F13): The flaws in this class occur due to the inability of 3/9 crypto-detectors to resolve complex inheritance relationships among classes, generally resulting from applying *flexible* mutation operators (Sec. VI-B) to certain misuse cases. For example, consider **F11** in Table I, also illustrated in Listing 12 in Appendix A. Further, we find that Xanitizer & SpotBugs are immune to these flaws, which indicates that traversing intricate inheritance relationships is a *design consideration for some crypto-detectors*, and a *design gap* in others such as CryptoGuard and QARK. Moreover, such indirect relationships can not only be expected from evasive developers (*i.e.,* **T**3) but is also found in real apps investigated by the crypto-detectors, as described below.

Impact (FC3): We found an exact instance of the misuse representing **F10** (generated from **OP**$_{12}$) in the class `TrustAllSSLSocketFactory` in Apache JMeter [81] (4.7K stars in GitHub). **F11** is the generic version of **F10**, and fairly common in repositories and libraries (*e.g.,* BountyCastle [82], [83]). **F12** an **F13** were also generated using **OP**$_{12}$, but with the

HostnameVerifier-related misuse, and we did not find similar instances in the wild in our limited search.

FC4: Insufficient Analysis of Generic Conditions in Extensible Crypto-APIs (F14 – F16): The flaws in this class represent the inability of certain crypto-detectors to identify fake conditions within overridden methods, *i.e.,* unrealistic conditions, or always true condition blocks (*e.g.,* as Listing 16 in Appendix A shows for **F14**). Flaws in this class represent the behavior of an evasive developer (**T**3). Xanitizer and SpotBugs can identify such spurious conditions.

FC5: Insufficient Analysis of Context-specific Conditions in Extensible Crypto-APIs (F17 – F19): The flaws in this class represent misuse similar to **FC4**, except that the fake conditions used here are contextualized to the overridden function, *i.e.,* they check *context-specific* attributes (*e.g.,* the length of the certificate chain passed into the method, **F17**). An evasive developer may attempt this to add further realism to fake conditions to evade tools such as Xanitizer that are capable of detecting generic conditions. Indeed, we observe that Xanitizer fails to detect misuse when context-specific conditions are used, for both **F17** and **F18**. Our suspicion is that this weakness is due to an optimization, which exempts conditions from analysis if they seem realistic.

Particularly, we observe that Xanitizer correctly detects the fake condition in **F19**, and that the only difference between **F19** and **F18** is that the instances of misuse they represent occur under slightly different class hierarchies. Hence, our speculation is that this an *accidental benefit*, *i.e.,* the difference could be the result of an *incomplete implementation of the unnecessary optimization* across different class hierarchies. SpotBugs shows a similar trend, potentially because Xanitizer uses SpotBugs for several SSL-related analyses. Finally, we observe that Tool$_X$ is immune to both generic **FC4**) and context-specific fake conditions **FC5**).

Impact (FC4, FC5): In this Stack Overflow post [37], the developer describes several ways in which they tried to get Google Play to accept their faulty TrustManager implementation, one of which is exactly the same as the misuse instance that led to **F17** (generated using **OP**$_7$ and **OP**$_{12}$), which is a more specific variant of **F14** (generated **OP**$_7$, **OP**$_9$ and **OP**$_{12}$), as illustrated in Listing 10 in Appendix A. We observe similar evasive attempts towards vulnerable hostname verification [84] which are similar in nature to **F15** and **F16**, and could be instantiated using **OP**$_8$ and **OP**$_{10}$. We also found developers trying to evade detection by applying context-specific conditions in the hostname verifier [85], similar to **F18** and **F19**.

XI. Limitations

MASC does not attempt to replace formal verification, and hence, does not guarantee that all flaws in a crypto-detector will be found. Instead, it enables systematic evaluation of crypto-detectors, which is an advancement over manually curated benchmarks. Aside from this general design-choice, our approach has the following limitations:

1. Completeness of the Taxonomy: To ensure a taxonomy that is as comprehensive as possible, we meticulously follow best-practices learned from prior work [56], [86], and also ensure labeling by two authors. However, the fact remains that regardless of how systematic our approach is, due to the manual and generalized of the SLR, we may miss certain subtle contexts during the information extraction phase (see specific examples in Appendix C). Thus, while not necessarily complete, this taxonomy, generated through over *2 person months of manual effort*, is, to the best of our knowledge, the most comprehensive in recent literature.

2. Focus on Generic Mutation Operators: We have currently constructed generic operators based on typical *usage conventions*, *i.e.,* to apply to as many misuse instances from the taxonomy as possible. However, currently, MASC does not incorporate operators that may fall outside of usage-based conventions, *i.e.,* which may be more tightly coupled with specific misuse cases, such as an operator for calling create and clearPassword in a specific order for PBEKeySpec. We plan to incorporate such operators into MASC in the future.

3. Focus on Java and JCA: MASC's approach is largely informed by JCA and Java. Additional mutation operators and adjustments will be required to adapt MASC to JCA-equivalent frameworks in other languages, particularly when adapting our usage-based mutation operators to non-JCA conventions.

4. Evolution of APIs: Future, tangential changes in how JCA operates might require changing the implementation of MASC's mutation operators. Furthermore, incremental effort will be required to incorporate new misuse cases that are discovered with the evolution of crypto APIs. We have designed MASC to be as flexible as possible by means of reflection and automated code generation for mutation operators, which should make adapting to such changes easier.

5. Relative Effectiveness of Individual Operators: This paper demonstrates the key claims of MASC, and its overall effectiveness at finding flaws, but does not evaluate/claim the *relative usefulness of each operator* individually. A comprehensive investigation of relative usefulness would require the mutation of all/most misuse cases from the taxonomy, with every possible operator and scope, a broader set of apps to mutate, and a complete set of crypto-detectors, which is outside the scope of MASC, but a direction for future work.

XII. Discussion and Conclusion

Designing crypto-detectors is in no way a simple task; tool designers have to balance several orthogonal requirements such as detecting as many vulnerabilities as possible without introducing false positives, while also scaling to large code-bases. Yet, the fact remains that there is significant room for improvement in how crypto-detectors are built and evaluated, as evidenced by the flaws discovered by MASC.

To move forward, we need to understand the divergent perspectives regarding the design of crypto-detectors, and reach a consensus (or at least an agreeable minima) in terms of what is expected from crypto-detectors and how we will design and evaluate them to satisfy the expectations. We seek to begin this discourse within the security community by integrating several views on the design decisions behind crypto-detectors, informed

by our results and conversations with tool designers (quoted with consent) during the vulnerability reporting process.

A. Security-centric Evaluation vs. Technique-centric Design

Determining what misuse is within or outside the scope for a crypto-detector is a complex question that yields several different viewpoints. This paper's view is *security-centric*, *i.e.,* even if some misuse instances may seem unlikely or evasive, crypto-detectors that target security-focused use cases (*e.g.,* compliance, auditing) should attempt to account for them. However, we observe that tool designers typically adhere to a *technique-centric* perspective, *i.e.,* the design of crypto-detectors is not influenced by a threat model, but mainly by what static analysis can and cannot accomplish (while implicitly assuming a benign developer). This quote from the maintainers of CryptoGuard highlights this view, wherein they state that the "lines" between what is within/outside scope *"seen so far were technically motivated – not use-case motivated..should we use alias analysis?..."*. This gap in perspective does not mean that crypto-detectors may not detect any of the mutants generated by MASC using operators based on the **T**3 threat model; rather, it only means that detection (or lack thereof) may not be *caused* by a security-centric design.

B. Defining "Scope" for the Technique-centric Design

We observe that even within crypto-detectors that take a *technique-centric* approach, there is little agreement on the appropriate scope of detection. For instance, Xanitizer focuses on catching every possible misuse instance, regardless of any external factors such as whether that kind of misuse is observed in the wild, or a threat model, as the designers believe that *"the distinction should not be between 'common' and 'uncommon', but instead between 'can be (easily) computed statically' and 'can not be computed'."*. This makes it possible for Xanitizer to detect unknown or rare problems, but may also result in it not detecting a commonly observed misuse that is hard to compute statically, although we did not observe such cases.

In contrast, CryptoGuard, CogniCrypt, and GCS/LGTM (same developers) would *consider seemingly-unlikely/evasive flaws within scope* (*e.g.,* **F3** – **F6**, **F8**), because they were found in the wild (unlike Xanitizer, for which this is not a consideration). This view aligns with our perspective, that regardless of how it was generated, if a misuse instance (representing a flaw) is discovered in real apps (which is *true for all flaws except* **F12** *and* **F13**, it should be within the detection scope. However, GCS/LGTM maintainers extend this definition with the condition that the observations in the wild be frequent, to motivate change. These divergent perspectives motivate the need to clearly define the expectations from crypto-detectors.

C. Utility of Seemingly-Uncommon or Evasive Tests

As Bessey et al. state from practical deployment experience in 2010 [18], *"No bug is too foolish to check for"*, and that *"Given enough code, developers will write almost anything you can think of..."*. The results from our evaluation and the impact study corroborate this sentiment, *i.e.,* **F3** – **F6** and **F8** were all obtained using operators ($\mathbf{OP}_3$, $\mathbf{OP}_4$, and $\mathbf{OP}_6$) modeled to emulate threats **T**1 and **T**2, *i.e.,* representing *benign* behavior (however unlikely); and indeed, these flaws were later found in supposedly benign applications. This suggests that the experience of Bessey et al. is valid a decade later, making it important to evaluate crypto-detectors with "more-than-trivial" cases to not only test their detection prowess, but to also account for real problems that may exist in the wild.

D. The Need to Strengthen Crypto-Detectors

We argue that it is not only justified for tools to detect uncommon cases (*e.g.,* given that even benign developers write seemingly-unlikely code), but also critical for their sustained relevance. As the designers of Coverity found [18], false negatives matter from a commercial perspective, because *"Potential customers intentionally introduced bugs into the system, asking 'Why didn't you find it?'"*.

Perhaps more significantly, the importance of automated crypto-detectors with the ability to guarantee *assurance* is rising with the advent of new *compliance* legislation such as the IoT Cybersecurity Improvement Act of 2020 [87], which seeks to rein in vulnerabilities in billions of IoT systems that include vulnerable server-side/mobile components. Vulnerabilities found *after* a compliance certification may result in penalties for the developers, and financial consequences for the compliance checkers/labs and crypto-detectors used. Complementing static detection with manual or dynamic analysis may be infeasible at this scale, as tool designers noted: *e.g., "...review an entire codebase at once, manual review can be difficult."* (LGTM) and *"Existing dynamic analysis tools will be able to detect them only if the code is triggered (which can be notoriously difficult)"* (CryptoGuard). Thus, static crypto-detectors will need to become more robust, and capable of detecting hard-to-detect misuse instances.

E. Towards Crypto-Detectors Strengthened by a Security-Centric Evaluation

Fortunately, we observe that there is support among tool designers for moving towards stronger security guarantees. For instance, CogniCrypt designers see a future research direction in expressing evasive scenarios in the CrySL language *i.e., "...what would be a nice future direction is to tweak the SAST with such optimizations/ more analysis but still allow the CrySL developer to decide if he wants to switch these 'evasive user' checks..."*, but indicate the caveat that developers may not use such additional configuration options [88]. However, we believe that such options will benefit *independent evaluators*, as well as developers who are unsure of the quality of their own supply chain, to perform a hostile review of the third-party code at their disposal. Similarly, the CryptoGuard designers state that *"...this insight of evasive developers is very interesting and timely, which immediately opens up new directions."*

This potential paradigm-shift towards a *security-focused* design of crypto-detectors is timely, and MASC's rigorous evaluation with expressive test cases can play an effective role in it, by enabling tool designers to proactively address gaps in detection during the design phase itself, rather than

reactively (*i.e.,* after a vulnerability is discovered in the wild). More importantly, further large-scale evaluations using MASC, and the flaws discovered therein, will enable the community to continually examine the design choices made by crypto-detectors and reach consensus on what assurances we can feasibly expect. We envision that such development aided by MASC will lead to a mature ecosystem of crypto-detectors with a well-defined and strong security posture, and which can hold their own in adversarial situations (*e.g.,* compliance assessments) within certain *known* bounds, which will eventually lead to long-lasting security benefits for end-user software.

ACKNOWLEDGMENT

We thank the developers of the evaluated tools for making their tools available to the community, and for being open to discussion, suggestions, and improvements. We would like to thank Sazzadur Rahaman for his constructive feedback, which enabled insights that shaped the discussion in this paper. We would also like to thank the anonymous reviewers for their constructive and detailed feedback on the paper. Finally, the authors have been supported in part by the NSF-1815336, NSF-1815186 and NSF-1955853 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

REFERENCES

[1] R. Anderson, "Why Cryptosystems Fail," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*, ser. CCS '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 215227. [Online]. Available: https://doi-org.proxy.wm.edu/10.1145/168588.168615

[2] 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*, ser. CCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 5061. [Online]. Available: https://doi.org/10.1145/2382196.2382205

[3] S. Rahaman, Y. Xiao, S. Afrose, F. Shaon, K. Tian, M. Frantz, M. Kantarcioglu, and D. 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 - CCS '19*. London, United Kingdom: ACM Press, Nov. 2019, pp. 2455–2472.

[4] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps," in *Proceedings 2014 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, Feb. 2014.

[5] J. Tang, J. Li, R. Li, H. Han, X. Gu, and Z. Xu, "SSLDetecter: Detecting SSL Security Vulnerabilities of Android Applications Based on a Novel Automatic Traversal Method," *Security and Communication Networks*, vol. 2019, pp. 1–20, Oct. 2019.

[6] C. Zuo, J. Wu, and S. Guo, "Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '15. New York, NY, USA: ACM, 2015, pp. 591–596.

[7] 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)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 151–164.

[8] K. Kafle, K. Moran, S. Manandhar, A. Nadkarni, and D. Poshyvanyk, "A Study of Data Store-based Home Automation," in *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY)*, Mar. 2019.

[9] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 10:1–10:27.

[10] 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 - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 73–84.

[11] "Find Security Bugs," https://find-sec-bugs.github.io/, accessed May, 2020.

[12] "Xanitizer by RIGS IT - Because Security Matters," https://www.rigs-it.com/xanitizer/, accessed May, 2020.

[13] "Coverity SAST Software — Synopsys," https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html, accessed May, 2020.

[14] "Code Quality and Security — SonarQube," https://www.sonarqube.org/, accessed May, 2020.

[15] "GitHub Security Lab," https://securitylab.github.com/, accessed Nov, 2020.

[16] "ShiftLeft Scan," https://shiftleft.io/scan, accessed Nov, 2020.

[17] "LGTM - Continuous security analysis," https://lgtm.com/, accessed Nov, 2020.

[18] 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, Feb. 2010.

[19] "CogniCrypt - Secure Integration of Cryptographic Software — CogniCrypt," https://www.eclipse.org/cognicrypt/, accessed June, 2020.

[20] "Oracle - Industrial Experience of Finding Cryptographic Vulnerabilities in Large-scale Codebases," https://labs.oracle.com/pls/apex/f?p=94065:40150:0::::P40150_PUBLICATION_ID:6629, Jul. 2020, accessed July, 2020.

[21] "NSF Award Search: Award#1929701 - SaTC: TTP: Medium: Collaborative: Deployment-quality and Accessible Solutions for Cryptography Code Development," https://www.nsf.gov/awardsearch/showAward?AWD_ID=1929701&HistoricalAwards=false, Sep. 2019, accessed May, 2020.

[22] "Announcing third-party code scanning tools: static analysis & developer security training - The GitHub Blog," https://github.blog/2020-10-05-announcing-third-party-code-scanning-tools-static-analysis-and-developer-security-training/, Oct. 2020, accessed Nov, 2020.

[23] "Introducing QARK: An Open Source Tool to Improve Android Application Security — LinkedIn Engineering," https://engineering.linkedin.com/blog/2015/08/introducing-qark, Aug. 2015, accessed May, 2020.

[24] S. R. Kotipalli and M. A. Imran, *Hacking Android*. Packt Publishing, Jul. 2016, ch. 4. Overview of Attacking Android Apps - QARK(Quick Android Review Kit), pp. 126–138.

[25] W. Halton, B. Weaver, J. Ansari, S. Kotipalli, and M. A. Imran, *Penetration testing : a complete pentesting guide facilitating smooth backtracking for working hackers : a course in three modules*, 1st ed. Packt Publishing, Nov. 2016, ch. Overview of Attacking Android Apps - QARK (Quick Android Review Kit).

[26] T. H.-C. Hsu, *Practical Security Automation and Testing: Tools and Techniques for Automated Security Scanning and Testing in DevSecOps*. Birmingham: Packt Publishing, Feb. 2019, ch. Android Security Testing - Static secure code scanning with QARK.

[27] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A comprehensive benchmark on java cryptographic API misuses," in *2019 IEEE Cybersecurity Development (SecDev)*, Sep. 2019, pp. 49–61.

[28] "OWASP Benchmark," https://owasp.org/www-project-benchmark/, accessed May, 2020.

[29] "Test cases for risky or broken cryptographic algorithm erroneously labeled as not vulnerable · Issue #92 · OWASP/Benchmark," https://github.com/OWASP/Benchmark/issues/92, Jan. 2020, accessed Nov, 2020.

[30] "MASC Artifact," https://github.com/Secure-Platforms-Lab-W-M/MASC-Artifact, accessed Aug, 2021.

[31] Veracode, "Veracode's 10th State of Software Security Report Finds Organizations Reduce Rising 'Security Debt' via DevSecOps, Special Sprints," https://www.veracode.com/veracodes-10th-state-software-

security-report-finds-organizations-reduce-rising-security-debt, Oct. 2019, accessed May, 2020.
[32] "Software Assurance Marketplace," https://continuousassurance.org/, accessed Jun, 2020.
[33] "find-sec-bugs/find-sec-bugs: The SpotBugs plugin for security audits of Java web applications and Android applications. (Also work with Kotlin, Groovy and Scala projects)," https://github.com/find-sec-bugs/find-sec-bugs, accessed Apr, 2020.
[34] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: Supporting Developers in Using Cryptography," in *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 931–936.
[35] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 3rd ed. New York: Wiley, Dec. 2020, ch. 28.
[36] "IoT Security Rating — Identity Management & Security," https://ims.ul.com/IoT-security-rating, accessed Apr, 2021.
[37] "Java - an Unsafe Implementation of the Interface X509TrustManager from Google," https://stackoverflow.com/questions/35545126/an-unsafe-implementation-of-the-interface-x509trustmanager-from-google, Feb. 2016, accessed Apr, 2021.
[38] "How to bypass SSL certificate validation in Android app?" https://stackoverflow.com/questions/35548162/how-to-bypass-ssl-certificate-validation-in-android-app, Feb. 2016, accessed Apr, 2021.
[39] "Bypass SSL Warning from google play," https://stackoverflow.com/questions/36913633/bypass-ssl-warning-from-google-play, Apr. 2016, accessed Apr, 2021.
[40] 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 *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/oltrogge
[41] L. Qiu, Y. Wang, and J. Rubin, "Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*. Amsterdam, Netherlands: ACM Press, 2018, pp. 176–186.
[42] "secure-software-engineering/DroidBench: A micro-benchmark suite to assess the stability of taint-analysis tools for Android," https://github.com/secure-software-engineering/DroidBench, accessed Jun, 2020.
[43] "fgwei/ICC-Bench: Benchmark apps for static analyzing inter-component data leakage problem of Android apps." https://github.com/fgwei/ICC-Bench, accessed Jun, 2020.
[44] F. Pauck, E. Bodden, and H. Wehrheim, "Do Android taint analysis tools keep their promises?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, Oct. 2018, pp. 331–341.
[45] R. Bonett, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Discovering flaws in security-focused static analysis tools for Android using systematic mutation," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1263–1280. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/bonett
[46] B. Livshits, D. Vardoulakis, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, and A. Mø ller, "In defense of soundiness: A manifesto," *Communications of the ACM*, vol. 58, no. 2, pp. 44–46, Jan. 2015.
[47] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
[48] X. O. Fengguo Wei, Sankardas Roy and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Nov. 2014.
[49] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, May 2016.
[50] M. Wen, Y. Liu, R. Wu, X. Xie, S.-C. Cheung, and Z. Su, "Exposing Library API Misuses Via Mutation Analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 866–877.
[51] J. Gao, P. Kong, L. Li, T. F. Bissyande, and J. Klein, "Negative Results on Mining Crypto-API Usage Rules in Android Apps," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada: IEEE, May 2019, pp. 388–398.
[52] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira, "Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study," in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct. 2017, pp. 170–181.
[53] ——, "Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software," *IEEE Transactions on Reliability*, vol. 68, no. 4, pp. 1384–1403, Dec. 2019.
[54] A. Braga and R. Dahab, "Mining Cryptography Misuse in Online Forums," in *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Aug. 2016, pp. 143–150.
[55] National Institute of Technology (NIT) Puducherry, Karaikal, India, K. Vasan K., and A. R. Kumar P., "Taxonomy of SSL/TLS Attacks," *International Journal of Computer Network and Information Security*, vol. 8, no. 2, pp. 15–24, Feb. 2016.
[56] B. Kitchenham, O. P. Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, "Systematic literature reviews in software engineering – A systematic literature review," *Information and Software Technology*, vol. 51, no. 1, pp. 7–15, 2009.
[57] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, Aug. 2015.
[58] "OWASP Foundation — Open Source Foundation for Application Security," https://www.owasp.org/, accessed May, 2020.
[59] "Soundiness Home Page," http://soundiness.org/, accessed May, 2020.
[60] "Java Secure Socket Extension (JSSE) Reference Guide - Oracle Help Center," https://docs.oracle.com/en/java/javase/11/security/java-secure-socket-extension-jsse-reference-guide.html, accessed Nov, 2020.
[61] "Security with HTTPS and SSL - Android Developers," https://developer.android.com/training/articles/security-ssl, accessed Nov, 2020.
[62] K. Moran, M. Tufano, C. Bernal-Cárdenas, M. Linares-Vásquez, G. Bavota, C. Vendome, M. Di Penta, and D. Poshyvanyk, "MDroid+: A Mutation Testing Framework for Android," *Proceedings of the 40th International Conference on Software Engineering Companion Proceeedings - ICSE '18*, pp. 33–36, 2018.
[63] M. Linares-Vásquez, G. Bavota, M. Tufano, K. Moran, M. Di Penta, C. Vendome, C. Bernal-Cárdenas, and D. Poshyvanyk, "Enabling Mutation Testing for Android Apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 233–244.
[64] "F-Droid - Free and Open Source Android App Repository," https://f-droid.org/, accessed Jun, 2020.
[65] "GitHub: Where the world builds software GitHub," https://github.com, accessed Jul, 2021.
[66] "Broker-J - Apache Qpid," https://qpid.apache.org/components/broker-j/, accessed May, 2020.
[67] "GitHub - Where software is built — Advanced search ," https://github.com/search/advanced, accessed Jul, 2021.
[68] "Stack Overflow - Where Developers Learn, Share, & Build Careers," https://stackoverflow.com, accessed Jun, 2020.
[69] "Cryptography Stack Exchange," https://crypto.stackexchange.com, accessed Jun, 2020.
[70] A. Developers, "Enable multidex for apps with over 64K methods — android developers," https://developer.android.com/studio/build/multidex, accessed May, 2020.
[71] "codeql/Encryption.qll at github/codeql," https://github.com/github/codeql/blob/768e5190a1c9d40a4acc7143c461c3b114e7fd59/java/ql/src/semmle/code/java/security/Encryption.qll#L142, Aug. 2018, accessed Jul, 2021.
[72] "pdf-service/md5util.java at elainrd/pdf-servic," https://github.com/elainrd/pdf-service/blob/243588e446ed875e13a99ea08e2baa3e0806c346/src/main/java/com/hhd/pdf/util/Md5Util.java#L76, Apr. 2019, accessed Jul, 2021.
[73] "encryption-machine/asymencrpmachine.java at mrdrivingduck/encryption-machine," https://github.com/mrdrivingduck/encryption-machine/blob/74c5679c86cf11f74409cfc63e1385b906734099/src/iot/zjt/encrypt/machine/AsymEncrpMachine.java#L95, May 2019, accessed Jul, 2021.

[74] "ExoPlayer/CachedContentIndex.java at google/ExoPlayer," https://github.com/google/ExoPlayer/blob/f182c0c1169cba7c22280058368127c24609054f/library/core/src/main/java/com/google/android/exoplayer2/upstream/cache/CachedContentIndex.java#L332, Nov. 2016, accessed Jul, 2021.
[75] "UltimateAndroid/TripleDES.java at cymcsg/UltimateAndroid," https://github.com/cymcsg/UltimateAndroid/blob/678afdda49d1e7c91a36830946a85e0fda541971/UltimateAndroid/ultimateandroid/src/main/java/com/marshalchen/ua/common/commonUtils/urlUtils/TripleDES.java#L144, May 2016, accessed Jul, 2021.
[76] "jeesuite-libs/DES.java at vakinge/jeesuite-libs," https://github.com/vakinge/jeesuite-libs/blob/master/jeesuite-common/src/main/java/com/jeesuite/common/crypt/DES.java#L37, Jul. 2017, accessed Jul, 2021.
[77] "Apache Ignite," https://github.com/apache/ignite, accessed Jul, 2021.
[78] "ignite/IgniteUtils.java at apache/ignite," https://github.com/apache/ignite/blob/1a3fd112b02133892c7c95d4be607079ffa83211/modules/core/src/main/java/org/apache/ignite/internal/util/IgniteUtils.java#L11714, Jan. 2015, accessed Jul, 2021.
[79] "hive/GenericUDFAesBase.java at master - apache/hive," https://github.com/apache/hive/blob/526bd87e9103375f0ddb8064dcfd8c31342b4c08/ql/src/java/org/apache/hadoop/hive/ql/udf/generic/GenericUDFAesBase.java#L108, Aug. 2015, accessed Jul, 2021.
[80] "Azure/azure-sdk-for-java: This repository is for active development of the Azure SDK for Java," https://github.com/Azure/azure-sdk-for-java, accessed Jul, 2021.
[81] "jmeter/TrustAllSSLSocketFactory.java at apache/jmeter," https://github.com/apache/jmeter/blob/704adb91f7f967402b9b709e89f5b73f0a466283/src/core/src/main/java/org/apache/jmeter/util/TrustAllSSLSocketFactory.java, Mar. 2019, accessed Jul, 2021.
[82] "BCX509ExtendedTrustManager (Bouncy Castle Library 1.66 API Specification)," https://www.bouncycastle.org/docs/tlsdocs1.5on/org/bouncycastle/jsse/BCX509ExtendedTrustManager.html, accessed Jul, 2021.
[83] "JGDMS/FilterX509TrustManager.java at pfirmstone/JGDMS," https://github.com/pfirmstone/JGDMS/blob/a44b96809783199b5fd69ffb803e0c4ceb9fad67/JGDMS/jgdms-jeri/src/main/java/net/jini/jeri/ssl/FilterX509TrustManager.java, Dec. 2018, accessed Jul, 2021.
[84] "android - Google Play Warning: How to fix incorrect implementation of HostnameVerifier? - Stack Overflow," https://stackoverflow.com/a/41330005, Dec. 2016, accessed Jul, 2021.
[85] "android - unsafe implementation of the HostnameVerifier interface - Stack Overflow," https://stackoverflow.com/questions/47069277/unsafe-implementation-of-the-hostnameverifier-interface, Mar. 2018, accessed Jul, 2021.
[86] J. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.
[87] "H.r.1668 - 116th congress (2019-2020): Internet of things cybersecurity improvement act of 2020 — congress.gov — library of congress," https://www.congress.gov/bill/116th-congress/house-bill/1668, Dec. 2020, accessed Jul, 2021.
[88] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 672–681.
[89] F. Fischer, K. Bottinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security," in *2017 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA, USA: IEEE, May 2017, pp. 121–136.
[90] OASIS, "The Static Analysis Results Interchange Format (SARIF)," https://sarifweb.azurewebsites.net/, accessed Jul, 2021.
[91] "Sarif Viewer - Visual Studio Marketplace," https://marketplace.visualstudio.com/items?itemName=MS-SarifVSCode.sarif-viewer, accessed Jul, 2021.
[92] A. S. Ami, K. Kafle, K. Moran, A. Nadkarni, and D. Poshyvanyk, "Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques," *ACM Transactions on Privacy and Security*, vol. 24, no. 3, pp. 15:1–15:37, Feb. 2021.
[93] "codeql/UnsafeCertTrust.ql at github/codeql," https://github.com/github/codeql/blob/768e5190a1c9d40a4acc7143c461c3b114e7fd59/java/ql/src/experimental/Security/CWE/CWE-273/UnsafeCertTrust.ql, May 2020, accessed Jul, 2021.
[94] "Java: CWE-273 Unsafe certificate trust by luchua-bc Pull Request #3550 - github/codeql," https://github.com/github/codeql/pull/3550, Jun. 2020, accessed Jul, 2021.
[95] "qark/ecb_cipher_usage.py at master linkedin/qark," https://github.com/linkedin/qark/blob/master/qark/plugins/crypto/ecb_cipher_usage.py#L31, Jan. 2018, accessed Jul, 2021.
[96] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative Data Analysis: A Methods Sourcebook*, 3rd ed. Thousand Oaks, Califorinia: SAGE Publications, Inc, 2014.

APPENDIX A
CODE SNIPPETS

```
Class T { String algo="AES/CBC/PKCS5Padding";
T mthd1(){ algo = "AES";  return this;} T mthd2(){
     algo="DES"; return this;} }
Cipher.getInstance(new T().mthd1().mthd2());
```

Listing 3. Method Chaining ($\mathbf{OP}_5$).

```
val = new Date(System.currentTimeMillis()).toString();
new IvParameterSpec(val.getBytes(),0,8);}
```

Listing 4. Predictable/Non-Random Derivation of Value ($\mathbf{OP}_6$)

```
void checkServerTrusted(X509Certificate[] x, String s)
throws CertificateException {
if (!(null != s && s.equalsIgnoreCase("RSA"))) {
     throw new CertificateException("not RSA");}
```

Listing 5. Exception in an *always-false* condition block ($\mathbf{OP}_7$).

```
public boolean verify(String host, SSLSession s) {
  if(true || s.getCipherSuite().length()>=0)}
    return true;} return false;}
```

Listing 6. False return within an *always true* condition block ($\mathbf{OP}_8$).

```
interface ITM extends X509TrustManager { }
abstract class ATM implements X509TrustManager { }
```

Listing 7. Implementing an Interface with no overridden methods.

```
new HostnameVerifier(){
  public boolean verify(String h, SSLSession s) {
    return true; } };
```

Listing 8. Inner class object from Abstract type ($\mathbf{OP}_{12}$)

```
new X509ExtendedTrustManager(){
  public void checkClientTrusted(X509Certificate[] chain,
       String a) throws CertificateException {}
  public void checkServerTrusted(X509Certificate[] chain,
       String authType)throws CertificateException {}
  public X509Certificate[] getAcceptedIssuers() {return
       null;} ...};
```

Listing 9. Anonymous Inner Class Object of `X509ExtendedTrustManager` (F10)

```
void checkServerTrusted(X509Certificate[] certs, String s)
   throws CertificateException {
 if (!(null != s || s.equalsIgnoreCase("RSA") ||
      certs.length >= 314)) {
   throw new CertificateException("Error");}}
```

Listing 10. Specific Condition in `checkServerTrusted` method (F17)

```
abstract class AHV implements HostnameVerifier{} new AHV(){
  public boolean verify(String h, SSLSession s)
       return true;}};
```

Listing 11. Anonymous Inner Class Object of An Empty Abstract Class that implements `HostnameVerifier`

```
abstract class AbstractTM implements X509TrustManager{}
     new AbstractTM(){
  public void checkServerTrusted(X509Certificate[] chain,
       String authType) throws CertificateException {}
    public X509Certificate[] getAcceptedIssuers() {return
        null;}}};
```

Listing 12. Anonymous inner class object with a vulnerable `checkServerTrusted` method (F13)

```
interface IHV extends HostnameVerifier{} new IHV(){
  public boolean verify(String h, SSLSession s) return
      true;}};
```

Listing 13. Anonymous Inner Class Object of an Interface that extends `HostnameVerifier`

```
KeyGenerator keyGen = KeyGenerator.getInstance("AES");
keyGen.init(128); SecretKey secretKey=keyGen.generateKey();
```

Listing 14. Misuse case requiring a trivial new operator

```
if (!className.contains("android."))
    classNames.add(className.substring(1,
        className.length() - 1)); return classNames;
```

Listing 15. CryptoGuard's code ignoring names with "android"

```
if(!(true || arg0==null || arg1==null)) {
    throw new CertificateException();}
```

Listing 16. Generic Conditions in `checkServerTrusted`

```
this.name = name == null ? "AES" : name;
this.mode = mode == null ? "CBC" : mode;
this.pad = pad == null ? "PKCS5Padding" : pad;
this.string = StringUtils.format(
    "%s/%s/%s", this.name, this.mode, this.pad);
```

Listing 17. Transformation String formation in Apache Druid similar to **F2** which uses `AES` in `CBC` mode with `PKCS5Padding`, a configuration that is known to be a misuse [29], [89].

APPENDIX B
ADDITIONAL IMPLEMENTATION AND EVALUATION DETAILS

A. Expanded rationale for choosing certain operators

We prioritized misuse cases for inclusion in `MASC` that are discussed more frequently in the artifacts. For instance, when implementing restrictive operators (Sec. VI-A), we chose the misuse of using `AES` with `ECB` mode, or using `ECB` mode in general, as both misuse cases were frequently mentioned in our artifacts (*i.e.,* in 2 and 11 artifacts respectively). Similarly, we chose the misuse cases of using `MD5` algorithm with the `MessageDigest` API for hashing (5 artifacts), and digital signatures (5 artifacts). When implementing flexible mutation operators (Sec. VI-B), we observed that the majority of the misuse cases relate to improper SSL/TLS verification and error handling, and hence chose to mutate the `X509TrustManager` and `HostnameVerifier` APIs with $\mathbf{OP}_7$ – $\mathbf{OP}_{13}$.

B. Do we need to optimize the number of mutants generated?

`MASC` generates thousands of mutants to evaluate crypto-detectors, which may prompt the question: should we determine exactly *how many* mutants to generate or optimize them? The answer to this question is no, for two main reasons.

First, `MASC` generates mutants as per the mutation-scope applied, *i.e.,* for the exhaustive scope, it is natural for `MASC` to seed an instance of the same mutation at every possible/-compilable entry point (including internal methods) in the mutated application. Similarly, for the similarity scope, we seed a mutant besides every similar "usage" in the mutated application. Therefore, in all of these cases, every mutant seeded is justified/necessitated by the mutation scope being instantiated. Any reduction in mutants would require `MASC` to sacrifice the goals of its mutation scopes, which may not be in the interest of a best-effort comprehensive evaluation. Second, in our experience, the number of mutants does not significantly affect the time to seed taken by `MASC`. That is, `MASC` took just 15 minutes to seed over 20000 mutants as a part of our evaluation (see Section X). Moreover, once the target tools analysis is complete, we only have to analyze the unkilled mutants, which is a far smaller number than those originally seeded (Section X). Therefore, in our experience, there is little to gain (and much to lose) by reducing the number of mutants seeded; *i.e.,* we want to evaluate the tools as thoroughly as we can, even if it means evaluating them with certain mutation instances/mutants that may be effectively similar.

That said, from an analysis perspective, it may be interesting to dive deeper into the relative effectiveness of individual features (*i.e.,* operators as well as scopes), even if they are all individually necessary, as each mutation operator exploits a unique API use characteristic, and scopes exploit unique code-placement opportunities, and any combination of these may appear in real programs. However, it would be premature to determine relative advantages among scopes/operators using the existing evaluation sample (i.e., 9 detectors evaluated, 13 open-source apps mutated, 19 misuse cases instantiated, with 12 operators). For instance, mutating other misuse cases, or evaluating another tool, or using a different set of open source apps to mutate, may all result in additional/different success at the feature-level (although overall, MASC would still find flaws, and satisfy its claims). We defer such an evaluation to determine the relative advantages of different mutation features to future work, as described in Section XI.

C. Further details regarding confirming killed mutants

Matching the mutation log generated by `MASC` with the reports generated by crypto-detectors is challenging because crypto-detectors often generate reports in heterogeneous and often mutually incompatible ways; *i.e.,* GCS, LGTM, ShiftLeft generate text files following the recently introduced Static Analysis Results Interchange Format (SARIF) [90] format, CogniCrypt, CryptoGuard, Tool_X, SpotBugs and QARK generate reports in custom report formats, downloadable as HTML, CSV, or text, and finally, Xanitizer generates PDFs with source code annotations. We developed a semi-automated implementation that allows us to systematically identify uncaught mutants given these disparate formats. For QARK and CryptoGuard, we wrote custom scripts to parse and summarize their reports into a more manageable format, which we then manually reviewed and matched against `MASC's` mutation logs. For SARIF formatted reports, we used a VSCode based SARIF viewer [91] that allows iterative searching of logs and tool reports by location. For CogniCrypt, SpotBugs, and Xanitizer, we performed the matching manually since even though they used custom Text or

TABLE III
MUTANTS ANALYZED VS DETECTED BY CRYPTO-DETECTORS

Tool	Input Type	Analyzed	Detected
CryptoGuard	`apk` or `jar`	20,303	18,616/19,759
CogniCrypt	`apk` or `jar`	20,303	475
Xanitizer	Java Src Code & `jar`	17,788	17,774
Tool$_X$	Android or Java Src Code	20,303	48
SpotBugs	`jar`	17,788	17,715
QARK	Java Src Code or `apk`	20,303	7
LGTM	Java Src Code	20,303	16,929
GCS	Java Src Code	20,303	16,929
ShiftLeft	Java Src Code	20,303	20,199

PDF formats, they were generated in such a way that manual checking was trivial. This process is in line with prior work that faces similar challenges [92]. As more tools move to standard formats such as SARIF (which is being promoted by analysis suites such as Github Code Scan) and being adopted by crypto-detectors (*e.g.,* Xanitizer adopted SARIF after our study concluded), we expect the methodology to be fully automated.

D. Why GCS, LGTM, and QARK fail to detect base cases

We observe that GCS and LGTM fail to detect base cases (*i.e.,* Ø in Table II) for **FC3** – **FC5**, although they claim to find SSL vulnerabilities in Java, due to incomplete rulesets (*i.e.,* the absence of several SSL-related rules) [71]. However, we noticed that there was an SSL-related experimental pull request for GCS's ruleset [93], [94] and even upon integrating it into GCS and LGTM, we found both tools to still be vulnerable to the base cases. Similarly, QARK fails to detect base cases for all flaws in **FC1** and **FC2**, because of its incomplete ruleset [95].

APPENDIX C
TYPES OF CASES THAT OUR SLR APPROACH MAY MISS

Our SLR approach involves manually analyzing each document in an attempt to include all misuse cases, but this extraction of misuse cases is often affected by the context in which they are expressed. For instance, CogniCrypt's core philosophy is whitelisting, which is reflected throughout its papers and documentation. However, there are two ways in which whitelisting is expressed in the paper, one concerning functionality, and another security, *i.e.,* cases of desired behavior expressed in the paper may not always indicate a security best-practice. For instance, the ORDER keyword in the CrySL language initially caused us to miss the PBEKeySpec misuse (now included in the taxonomy), because as defined in the paper, ORDER keyword allows defining "*usage patterns*" that will not break functionality. Thus, as the "usage" patterns were not security misuses (or desired behaviors for security), we did not include them as misuse cases in the taxonomy.However, in a later part of the paper, the ORDER keyword is used to express a security-sensitive usage, for PBEKeySpec, but the difference in connotation is not made explicit. This implicit and subtle context-switch was missed by both our annotators in the initial SLR, but fixed in a later iteration, and misuse cases related to the ORDER keyword were added to the taxonomy.

Similarly when labeling for misuse extraction (Sec. V-C) we marked each misuse found in a document using common terminology (*i.e.,* labels) across all documents. Thus, if a misuse found in the current document was previously discovered and annotated with a particular label, we would simply apply the same label to the newly found instance.This standard, best-practice approach [56], [96] makes it feasible to extract a common taxonomy from a variety of documents written by different authors, who may use inconsistent terminology. However, a limitation of this generalization is that in a rare case wherein a particular example may be interpreted as two different kinds of misuse, our approach may lose context and label it as only one type of misuse. For instance, based on how the misuse of a "password stored in String" was described in most of the documents we studied, the misuse label of "using a hardcoded password" was applied to identify it across the documents. However, this results in the loss of the additional, semantically different misuse that may still be expressed in terms of a "password stored in String", that passwords should not be stored/used in a String data construct for garbage collection-related reasons. Note that this problem would only occur in rare instances wherein (1) there are multiple contexts/interpretations of the same misuse example, and (2) only one or few document(s) use the additional context. This misuse has also been included in the taxonomy.