CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects

Sazzadur Rahaman[1], Ya Xiao[1], Sharmin Afrose[1], Fahad Shaon[2], Ke Tian[1], Miles Frantz[1],
Murat Kantarcioglu[2], Danfeng (Daphne) Yao[1]
[1]Computer Science, Virginia Tech, Blacksburg, VA
[2]Computer Science, University of Texas at Dallas, Dallas, TX
{sazzad14,yax99,sharminafrose,ketian,frantzme,danfeng}@vt.edu,{fahad.shaon,muratk}@utdallas.edu

ABSTRACT

Cryptographic API misuses, such as exposed secrets, predictable random numbers, and vulnerable certificate verification, seriously threaten software security. The vision of automatically screening cryptographic API calls in massive-sized (e.g., millions of LoC) programs is not new. However, hindered by the practical difficulty of reducing false positives without compromising analysis quality, this goal has not been accomplished. CryptoGuard is a set of detection algorithms that refine program slices by identifying language-specific irrelevant elements. The refinements reduce false alerts by 76% to 80% in our experiments. Running our tool, CryptoGuard, on 46 high-impact large-scale Apache projects and 6,181 Android apps generated many security insights. Our findings helped multiple popular Apache projects to harden their code, including Spark, Ranger, and Ofbiz. We also have made progress towards the science of analysis in this space, including manually analyzing 1,295 Apache alerts, confirming 1,277 true positives (98.61% precision), and in-depth comparison with leading solutions including CrySL, SpotBugs, and Coverity.

CCS CONCEPTS

• **Security and privacy → Software and application security**;

KEYWORDS

Accuracy; cryptographic API misuses; static program analysis; false positive; false negative; benchmark; Java;

ACM Reference Format:
Sazzadur Rahaman[1], Ya Xiao[1], Sharmin Afrose[1], Fahad Shaon[2], Ke Tian[1], Miles Frantz[1], Murat Kantarcioglu[2], Danfeng (Daphne) Yao[1]. 2019. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3345659

1 INTRODUCTION

Cryptographic algorithms offer provable security guarantees in the presence of adversaries. However, vulnerabilities and deficiencies in low-level cryptographic implementations seriously reduce the guarantees in practice [16, 25, 28, 30, 37, 38]. Researchers also found misusing cryptographic APIs is not unusual in application-level code [33]. Causes of these vulnerabilities are multi-fold, which include complex APIs [13, 58], the lack of cybersecurity training [55], the lack of tools [15], and insecure and misleading forum posts (such as on StackOverflow) [14, 55]. Some aspects of security libraries (such as JCA, JCE, and JSSE[1]) are difficult for developers to use correctly, e.g., certificate verification [39] and cross-language encryption and decryption [55].

In this work, we focus on the goal of screening massive-sized Java projects for cryptographic API misuses. Specifically, we aim to design a static analysis tool that has no or few false positives (i.e., false alarms) and can be routinely used by developers.

Efforts to screen cryptographic APIs have been previously reported in the literature, including static analysis (e.g., CrySL [47], FixDroid [60], CogniCrypt [46], CryptoLint [33]) and dynamic analysis (e.g., SMV-Hunter [68], and AndroSSL [36]), as well as manual code inspection [39]. Static and dynamic analyses have their respective pros and cons. Static methods do not require the execution of programs. They scale up to a large number of programs, cover a wide range of security rules, and are unlikely to have false negatives (i.e., missed detections). Dynamic methods, in comparison, require one to trigger and detect specific misuse symptoms at runtime (e.g., misconfigurations of SSL/TLS). The advantage of dynamic approaches is that they tend to produce fewer false positives (i.e., false alarms) than static analysis. Deployment-grade code screening tools need to be scalable with wide coverage. Thus, static program analysis approach is favorable. However, existing static analysis-based tools (e.g., [33, 46, 47, 60]) are not optimized to operate on the scale of massive-sized Java projects (e.g., millions of LoC).

Existing static analysis tools are also limited in detecting SSL/TLS API misuses and are not designed to detect complex misuse scenarios. For example, MalloDroid [35] uses a list of known insecure implementations of `HostnameVerifier` and `TrustManager` to screen apps. Google Play recently deployed an automatic app checking mechanism for SSL/TLS hostname verifier and certificate verification vulnerabilities [12]. However, the inspection appears to only target obvious misuse scenarios, e.g., `return true` in `verify` method or an empty body in `checkServerTrusted` [5].

CCS '19 November 11–15, 2019 London, United Kingdom
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3345659

[1] JCA, JCE, and JSSE stand for Java Cryptography Architecture, Java Cryptography Extension, and Java Secure Socket Extension, respectively.

We made substantial progress toward building a high accuracy and low runtime static analysis solution for detecting cryptographic and SSL/TLS API misuse vulnerabilities. Our tool,CryptoGuard, is built on specialized forward and backward program slicing techniques. These slicing algorithms are implemented by using flow-, context- and field-sensitive data-flow analysis.

Although program slicing is a well-known technique for identifying the set of instructions that influence or are influenced by a program variable, its direct application to screening cryptographic implementations has several problems, which are explained next.

Detection accuracy. A challenging problem is the excessive number of false positives that basic static analysis (including slicing) generates. Several types of detection require one to search for constants or values from predictable APIs, e.g., passwords, seeds, or initialization vectors (IVs). However, benign constants or irrelevant parameters may be mistaken as violations (e.g., array/collection bookkeeping constants). Another source of detection inaccuracy comes from the assumption that all the system and runtime libraries are present during the analysis. This assumption holds for Android apps, but not necessarily for Java projects.

CryptoGuard addresses the false positive problem with a set of refinement algorithms derived from empirical observations of common programming idioms and language restrictions. The refinements remove irrelevant resource identifiers, arguments about states of operations, constants on infeasible paths, and bookkeeping values. For eight of our rules, these refinement algorithms reduce the total number of alerts by 76% in Apache and 80% in Android (Figure 3). Our manual analysis shows that CryptoGuard has a precision of 98.61% on Apache.

Efficiency and coverage. Analysis techniques that build a super control-flow graph of the entire program would incur significant memory and runtime overhead. In contrast, our on-demand slicing algorithms are lightweight, which start from the slicing criteria and only propagate to the methods that have the potential to impact security. Hence, a large portion of the code base is not touched.

Our technical contributions are summarized as follows.

- We designed and implemented a set of analysis algorithms for detecting cryptographic and SSL/TLS API misuses. Our static code checking tool, CryptoGuard, is designed for developers to use routinely on large Java projects. Besides open-sourcing CryptoGuard[2], we are currently integrating it with the Software Assurance Marketplace (SWAMP) [32], a well-known free software security analysis platform.
- We gained numerous security insights from screening 46 Apache projects. For 15 of our rules, we observed violations in Apache projects (Table 9). 39 out of the 46 projects have at least one type of cryptographic misuses, and 33 projects have at least two. We reported our security findings to Apache, some of which have been promptly fixed. In Section 7, we share our experience of disclosing to the Apache teams and their pragmatic constraints e.g., backward compatibility.
- Our evaluation on 6,181 Android apps shows that around 95% of the total vulnerabilities come from libraries that are packaged with the application code. Some libraries are from Google, Facebook, Apache, and Tencent (Table 5). We observe violations in most of the categories, including hardcoded keyStore passwords, e.g., `notasecret` is used in multiple Google libraries (Table 4). We also detected multiple SSL/TLS (MitM) vulnerabilities that Google Play's automatic screening seemed to have missed.
- We created a benchmark named CryptoApi-Bench with 112 unit test cases.[3] CryptoApi-Bench contains basic intra-procedural instances, inter-procedural cases, field sensitive cases, false positive tests, and correct API uses.

2 THREAT MODEL AND OVERVIEW

We describe our threat model and discuss the technical challenges associated with detecting these threats with static program analysis. For each challenge, we briefly overview our solution.

2.1 Threat Model

We summarize the vulnerabilities that CryptoGuard aims to detect below and in Table 1. We also rank their severity.

1. Vulnerabilities due to predictable secrets. Software with predictable cryptographic keys and passwords are inherently insecure [33]. Here, we consider the use of any constants, as well as values that are derived from constants or API calls with predictable outputs (e.g., DeviceID, Timestamps) to be insecure.

2. Vulnerabilities from MitM attacks on SSL/TLS. Improper customization of Java Secure Socket Extension (JSSE) APIs may result in man-in-the-middle (MitM) vulnerabilities [35, 39]. CryptoLint [33] does not detect these vulnerabilities.

3. Vulnerabilities from predictable PRNGs. Predictable pseudorandom number generators (PRNGs) are a major source of vulnerabilities [21, 40, 42]. Using `java.util.Random` as a PRNG is insecure [7, 45]. In addition, seeds for `java.security.SecureRandom` [8] should not be predictable.

4. Vulnerabilities from CPA. Ciphertexts should be indistinguishable under chosen plaintext attacks (CPA) [33]. Static salts make dictionary attacks easier on password-based encryption (PBE). In addition, static initialization vectors (IVs) in cipher block chaining (CBC) and electronic codebook (ECB) modes are insecure [20, 49].

5. Vulnerabilities from feasible bruteforce attacks. MD5 and SHA1 are susceptible to hash collision [69, 70] and pre-image [9, 27] attacks. In addition, bruteforce attacks are feasible for 64-bit symmetric ciphers (e.g., DES, 3DES, IDEA, Blowfish) [22]. 1024-bit RSA/DSA/DH and 160-bit ECC are also weak [4]. RFC 8018 recommends at least 1000 iterations for PBE [56].

How severe are these vulnerabilities? Each case has specific attack scenarios documented in the literature. To prioritize alerts, we categorize their severity into high, medium, and low, based on *i)* attacker's gain and *ii)* attack difficulty. Vulnerabilities from predictable secrets and SSL/TLS MitM are immediately exploitable and substantially benefit attackers. In Android, an application can only access its own KeyStore. Hence, hard-coded passwords are less harmful in Android. However, privilege escalation attacks bypassing this restriction have been demonstrated [72]. Commercially available rainbow tables allow attackers to easily obtain pre-images of MD5 and SHA1 hashes for typical passwords [10]. Hash collisions for these algorithms enable attackers to forge digital signatures

[2] Available at https://github.com/CryptoGuardOSS/cryptoguard under GPL v3.0.

[3] Our benchmark is available at https://github.com/CryptoGuardOSS/cryptoapi-bench.

```
1 class PasswordEncryptor {
2
3  Crypto crypto;
4
5  public PasswordEncryptor(){
6   String passKey = PasswordEncryptor
                                  .getKey("pass.key");
7    crypto = new Crypto(passKey); ⓟ
8  }
9
10 byte[] encPass(String [] arg){
11  return crypto.encrypt(arg[0], arg[1]); ⓟ
12 }
13
14 static String getKey(String src){
15  String key = Context.getProperty(src);
16  if (key == null) {
17    key = "defaultkey";
18   }
19  return key;
20   }
21 }
```

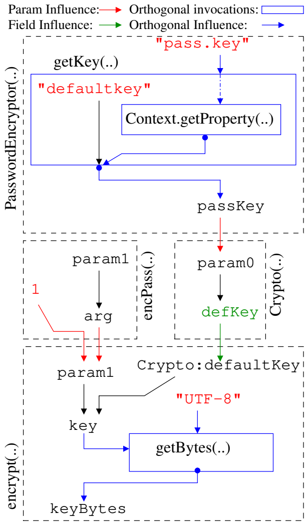
```
22 class Crypto {
23
24  String ALGO = "AES";
25  String ALGO_SPEC = "AES/CBC/NoPadding";
26  String defaultKey;
27  Cipher cipher;
28
29  public Crypto(String defKey){
30   cipher = Cipher.getInstance(ALGO_SPEC);
31   defaultKey = defKey; // assigning field
32  }
33
34  byte[] encrypt(String txt,String key){
35   if (key == null){
36    key = defaultKey; ⓕ
37   }
38   byte[] keyBytes = key.getBytes("UTF-8");
39   byte[] txtBytes = txt.getBytes();
40   SecretKeySpec keySpc =

   new SecretKeySpec(keyBytes, ALGO);

41   cipher.init(Cipher.ENCRYPT_MODE,keySpc);

42   return cipher.doFinal(txtBytes);}}
```

(a)



(b)

Figure 1: (a) An example demonstrating various features of CryptoGuard. `Crypto` class is used for generic AES encryption and `PasswordEncryptor` class uses `Crypto` for password encryption. ⓕ indicates influence through the fields and ⓟ indicates influence through the method parameters. (b) Partial data dependency graph for `keyBytes` variable.

or break the integrity of any messages [23, 69]. Therefore, these vulnerabilities are classified as high risks. Vulnerabilities from predictability and CPA provide substantial advantages to attackers by significantly reducing attack efforts. They are medium-level risks. Brute-forcing ciphers, requiring non-trivial effort, is low risk.

2.2 Technical Challenges and Solution Overview

The task of screening millions of lines of code for cryptographic API misuses poses a set of technical challenges.

Technical Challenge I: False positives.

1. False positives due to phantom methods. A method is phantom if its body is not available during analysis. Unlike Android, Java web applications have phantom libraries. A non-system library that is not packaged with the project binaries is referred to as a phantom library. Existing cryptographic misuse vulnerability solutions (e.g., CryptoLint [33], CrySL [47]) are not designed to handle phantom libraries, which may cause false positives. For example, in Figure 1(a) if the class `Context` is a member of a phantom library, then `getProperty` method (Line 15) is a phantom method. The data-flow diagram in Figure 1(b) shows that a straightforward def-use analysis would likely report `pass.key` as a hard-coded key, since it cannot explore `getProperty` method at Line 15.

Our solution is a set of crypto-specific methods to refine slicing outputs (Section 5). For example, examining the context reveals that `pass.key` is used as an identifier of a key and has no security influence on `keyBytes`. Thus, it can be safely discarded.

2. False positives due to data structures. Constants for bookkeeping data structures are another major source of false positives that are largely uncovered in the existing literature (e.g., [33, 47]). Most frequently used data structures include lists, maps, and arrays. For example, a data-structure-unaware analysis would likely report "1" from Line 11 (Figure 1(a)) as a hard-coded key, as it influences the `key` parameter of `encrypt` method (Figure 1(b)). Our refinement algorithms track and discard any kinds of data-structure-bookkeeping constants (Section 5).

Technical Challenge II: precision vs. runtime tradeoff. For a large project with millions LoC, building a super-CFG is costly and unnecessary. Cryptographic functionality is often confined within a small fraction of the project. However, most flow-, context- and field-sensitive analysis based tools (e.g., [33, 47]) appear to build a super control-flow graph, e.g., by superimposing the project's call graph over control-flow graphs of methods, adding call edges between *invoke* instructions, method entries, and exits.

In contrast, we adopt the following more scalable approaches.

1. Control the depth of orthogonal explorations. Most of our cryptographic vulnerabilities involve finding constants. A distinguishing feature of constants is that they require no or few processing before use. Generally, processing is done by orthogonal method invocations. The clipping of orthogonal explorations may impact the detection accuracy and runtime. Based on our experiments in Section 5.4, we set the depth of orthogonal exploration to 1 in our detection. We use similar techniques as in phantom methods handling to reduce the false positives introduced by clipping.

2. Demand-driven analysis. Our flow- and context- sensitive analysis is demand driven. We perform on-demand inter-procedural backward data flow analysis to perform backward slicing, where the analysis starts from the slicing criteria and propagates *upward* and *orthogonally* on-demand. For example, in Figure 1(a), a propagation from `encrypt` method to `encPass` method, is an upward propagation. A propagation to orthogonal method invocations at Line 6 and 38 are orthogonal propagation. Our on-demand field sensitivity is applied to a field only if it is used in our inter-procedural backward

Table 1: Cryptographic vulnerabilities, properties, and static analysis methods used. High, medium, and low risk levels are denoted by H/M/L, respectively. CPA stands for chosen ciphertext attack, MitM for man-in-the-middle, C/I/A for confidentiality, integrity, and authenticity, respectively. ↑ means backward slicing and ↓ means forward slicing. Slicing is inter-procedural unless otherwise specified (e.g., intra, both). Refinement insights are applied for all the inter-procedural backward slicing.

No	Vulnerabilities	Attack Type	Crypto Property	Severity	Our Analysis Method
1	Predictable/constant cryptographic keys.	Predictable Secrets	Confidentiality	H	↑ slicing & ↓ slicing
2	Predictable/constant passwords for PBE		Confidentiality	H	↑ slicing & ↓ slicing
3	Predictable/constant passwords for KeyStore		Confidentiality	H	↑ slicing & ↓ slicing
4	Custom Hostname verifiers to accept all hosts	SSL/TLS MitM	C/I/A	H	↑ slicing (intra)
5	Custom TrustManager to trust all certificates		C/I/A	H	↑ slicing (intra)
6	Custom SSLSocketFactory w/o manual Hostname verification		C/I/A	H	↓ slicing (intra)
7	Occasional use of HTTP		C/I/A	H	↑ slicing
8	Predictable/constant PRNG seeds	Predictability	Randomness	M	↑ slicing & ↓ slicing
9	Cryptographically insecure PRNGs (e.g., java.util.Random)		Randomness	M	Search
10	Static Salts in PBE	CPA	Confidentiality	M	↑ slicing & ↓ slicing
11	ECB mode in symmetric ciphers		Confidentiality	M	↑ slicing
12	Static IVs in CBC mode symmetric ciphers		Confidentiality	M	↑ slicing & ↓ slicing
13	Fewer than 1,000 iterations for PBE	Brute-force	Confidentiality	L	↑ slicing & ↓ slicing
14	64-bit block ciphers (e.g., DES, IDEA, Blowfish, RC4, RC2)		Confidentiality	L	↑ slicing
15	Insecure asymmetric ciphers (e.g, RSA, ECC)		C/A	L	↑ slicing & ↓ slicing (both)
16	Insecure cryptographic hash (e.g., SHA1, MD5, MD4, MD2)		Integrity	H	↑ slicing

slices. A field's influence is considered indirect, if the field is accessed using orthogonal method invocations (i.e., getter methods). We refer to this field sensitivity as *data-only class field-sensitivity.*

3. Subproject awareness. Code in large projects is usually organized into subprojects, packaged as separate `.jars`. CryptoGuard creates and consults a directed acyclic graph (DAG) representing subproject dependencies. This approach *i)* excludes unnecessary subprojects and *ii)* analyzes independent sub-projects concurrently.

3 MAP VULNERABILITIES TO ANALYSIS

It is important to map cryptographic properties to concrete Java programming elements that can be statically enforced. We break down the detection plan into one or more abstract steps so that each step can be mapped to a single round of static analysis.

In this section, we illustrate the process of mapping cryptographic vulnerabilities to concrete program analysis tasks. This mapping process is manual and only needs to be performed once for each vulnerability. In what follows, we use rule i to refer to the detection of vulnerability i in Table 1. For example, in Rule 4, we detect the abuse of `HostnameVerifier` interface. Ideally, an implementation of `HostnameVerifier` must use the `javax.net.ssl.SSLSession` parameter `verify` method to verify the hostname. Using the `return` statement as the slicing criterion, we perform intra-procedural backward slicing of `verify` method to implement this rule.

Rule 5 is to detect the abuse of the `X509TrustManager` interface. We reduce the task to detecting 3 concrete cases: *i)* throwing no exception after validating a certificate in `checkServerTrusted`, *ii)* unpinned self-signed certificate with an expiration check, and *iii)* not providing a valid list of certificates in `getAcceptedIssuers`. For Case *i)*, intuitively, our program analysis needs to search for the occurrences of `throw` or propagated exception. `throw` is the slicing criterion in the (intra-procedural) backward slicing. Simple parsing is inadequate, as the analysis needs to learn the type of the thrown exception.

Rule 6 is to detect whether any method uses `SSLSocket` directly without performing hostname verification. Intuitively, to detect this vulnerability, we need to track whether an `SSLSocket` created from `SSLSocketFactory` influences the `SSLSession` parameter of a `verify` method (of a `HostnameVerifier`) invocation. In addition, we also need to check whether the return value of the `verify` method is used in a condition checking statement (e.g., `if`). For detection, we use forward program slicing to identify all the instructions that are influenced by the `SSLSocketFactory` instance. Among these instructions, we examine three cases *i)* an `SSLSocket` is created, *ii)* an `SSLSession` is created and used in `verify`, and *iii)* the return value of `verify` method is used to make decisions. These three cases represent a correct use of `SSLSocket` with proper hostname verification.

Rule 15 is to detect insecure asymmetric cipher configurations (e.g., 1024-bit RSA). A more concrete goal is to detect an insecure default key size use and an explicit definition of insecure key size. The tasks of program analysis are to determine *a)* whether the key size is defined explicitly or by default, *b)* the statically defined key size, and *c)* the key generation algorithm. For Task *a)*, our analysis uses forward slicing to determine whether the `initialize` method is invoked to set the key size of a key-pair generator. For Tasks *b)* and *c)*, we use two rounds of backward program slicing to determine the key size and algorithm, respectively. We also employ on-demand field sensitivity for data-only classes in Task *b).* The analyses for Rule 15 are the most complex in CryptoGuard.

Mappings for other rules can be deduced from Table 1. For example, ↑ in Rules 1 & 2 means these rules are implemented using inter-procedural backward slicing and ↓ indicates inter-procedural forward slicing is used for on-demand data-only class field sensitivity. We list the slicing criteria in Tables 11, 12 and 13 in the appendix.

4 CRYPTO-SPECIFIC SLICING

We specialize static def-use analysis [74] and forward and backward program slicings [52] for detecting Java cryptographic API misuses.

We break down the detection strategy into one or more steps, so that a step can be realized with a single round of program slicing. After performing the slicing, each program slice is analyzed to find the presence of a vulnerability. Our 16 categories of vulnerabilities require different program analysis methods for detection. Table 1 summarizes slicing techniques to detect each of the vulnerabilities. General-purpose slicing alone is inadequate. Thus, we explain our solution for overcoming the accuracy challenge in Section 5.

A definition of variable v is a statement that modifies v (e.g., declaration, assignment). A use of variable v is a statement that reads v (e.g., a method call with v as an argument). Def-use data-flow analysis or def-use analysis identifies the definition and use statements and describes their dependency relations. Given a slicing criterion, which is a statement or a variable in a statement (e.g., a parameter of an API), backward program slicing is to compute a set of program statements that affect the slicing criterion in terms of data flow. Given a slicing criterion, forward program slicing is to compute a set of program statements that are affected by the slicing criterion in terms of data flow. Given a program and a slicing criterion, a program slicer returns a list of program slices. Intra-procedural program slicing mechanisms use def-use analysis to compute slices.

To confine inter-procedural backward slicing within security code regions, the analysis starts from cryptographic APIs and follows their influences recursively. This approach effectively skips the bulk of the functional code and substantially speeds up the analysis.

Slicing Criteria The choice of slicing criterion directly impacts the analysis outcomes. We choose slicing criteria based on several factors, including relevance to the vulnerability, simplicity of checking rules, shared across multiple projects. Our slicing criteria and corresponding APIs are shown in Tables 11, 12, and 13 in the appendix.

Backward Slicing For inter-procedural backward slicing, the slicing criteria are defined as the parameters of a target method's invocation. For example, to find predictable secrets (in Rules 1-3), we use the key parameter of the constructors of `SecretKeySpec` as the slicing criterion. For intra-procedural backward slicing, we define three types of slicing criteria: *i)* parameters of a method, *ii)* assignments, and *iii)* `throw` and `return`. For example, to detect insecure hostname verifiers that accept all hosts (in Rule 4), we use the `return` statement in the `verify` method as the slicing criterion.

Intra-procedural backward slicing. The purpose of intra-procedural backward slicing is two-fold. It is used independently to enforce security as well as a building block of inter-procedural back program slicing. The intra-procedural program slicing utilizes the def-use property of a statement to decide whether a statement should be included in a slice or not. Our implementation utilizes the worklist algorithm from the intra-procedural data-flow analysis framework of Soot. During this process, if any orthogonal method invocations are encountered, it recursively slices them to collect the arguments and statements that influence any field or return statements within that orthogonal methods. To reduce runtime overhead, such orthogonal method explorations are clipped at a pre-configurable depth. We use refinement insights in Section 5 to exclude security irrelevant instructions that basic use-def analysis cannot identify.

```
$r1.setText("mytext");
$r1.setKey("mykey");
...
key = $r1.getKey();
```

Figure 2: Indirect field access using orthogonal invocations on *data-only class* object `$r1`.

On-demand Inter-procedural backward slicing. This algorithm performs the upward propagation of the analysis. Our inter-procedural backward slicing builds on intra-procedural backward slicing. Major steps of the algorithm are as follows. *i)* We build a caller-callee relationship graph of all the methods of the program. The call-graph construction uses class-hierarchy analysis. *ii)* We identify all the callsites of the method specified in the slicing criterion. A callsite refers to a method invocation. *iii)* For all the callsites, we obtain all the inter-procedural backward slices by invoking intra-procedural slicing recursively to follow the caller chain. *iv)* Our procedure is field sensitive. Typical field initialization statements are assignments. After encountering a field assignment, the analysis follows the influences through fields, recursively.

Forward Slicing Some of our analysis demands forward slicing, which inspects the statements occurring after the slicing criterion.

Intra-procedural forward slicing. We design intra-procedural forward slicing for Rules 6 (SSLSocketFactory w/o Hostname verification) and 15 (Weak asymmetric crypto). The operation of intra-procedural forward slicing is similar to that of intra-procedural backward slicing. In forward slicing, we choose assignments as the slicing criteria. The traversal follows the order of the execution, i.e., going forward. Because problematic code regions for Rules 6 and 15 are confined within a method, their forward slicing analyses do not need to be inter-procedural.

Inter-procedural forward slicing. Given an assign instruction or a constant as the slicing criterion, we perform the inter-procedural forward slicing to identify instructions that are influenced by the slicing criterion in terms of def-use relations. Our inter-procedural forward slicing operates on the slices obtained from inter-procedural backward slicing. The latter produces an ordered collection of instructions combined from all visited methods.

We define a class as a data-only class, if the fields of the class are only visible within orthogonal method invocations. We use inter-procedural forward slicing for on-demand field sensitivity of data-only classes, as the field sensitivity during upward propagation (inter-procedural backward slicing) does not cover them. In Figure 2, `$r1` is an object of data-only class, where its fields are accessed indirectly with an orthogonal method (i.e, *getKey*) invocation. Given a constant, using inter-procedural forward slicing, CryptoGuard determines whether the constant influences any field of a data-only class object and records it. Later on, when it encounters an assign invocation on the same object and observes that the previously recorded field influences the return statement, then it reports the constant. Through this on-demand field sensitivity for data-only class, CryptoGuard knows that constant `mytext` (Figure 2) is not a hard-coded key. ↓ in Table 1 represents the use of forward slicing for on-demand data-only class field sensitivity [4].

[4] Current prototype uses this field sensitivity for 8 rules.

5 REFINEMENT FOR FP REDUCTION

We design a set of refinement algorithms to exclude security irrelevant instructions to reduce false alarms. These *refinement insights (RI)* are deduced by observing common programming idioms and language restrictions. We also discuss the possibility of false negatives (i.e., missed detection).

5.1 Overview of Refinement Insights (RI)

Eight of our rules (1, 2, 3, 8, 10, 12, 13 and 15) require identifying constants/predictable values in a program slice. The purpose is to ensure that no data (e.g., cryptographic keys, passwords, IVs, and seeds) is hardcoded or solely derived from any hardcoded values. Use of any predictable values (e.g., Timestamp, DeviceID) is also insecure for Rules 1, 2, 3 and 8. However, many constants do not impact security. We refer to them as *pseudo-influences.* Pseudo-influences are a major source of false positives. Based on empirical observations of common programming idioms and language restrictions, we have five strategies to systematically remove irrelevant constants/predictable values from slices and reduce pseudo-influences, which are summarized next.

- *RI-I: Removal of state indicators.* We discard constants/predictable values that are used to describe the state of a variable during an orthogonal method invocation.
- *RI-II: Removal of resource identifiers.* We discard constants/predictable values that are used as the identifier of a value source during an orthogonal method invocation.
- *RI-III: Removal of bookkeeping indices.* We discard constants/predictable values that are used as the index or size of any data structures. Specifically, RI-III discards any influences on *i)* size parameter of an array or a collection instantiation, *ii)* indices of an array, *iii)* indices of a collection.
- *RI-IV: Removal of contextually incompatible constants.* We discard constants/predictable values, if their types are incompatible with the analysis context. For example, a boolean variable cannot be used as a key, IV, or salt.
- *RI-V: Removal of constants in infeasible paths.* Some constant initializations are updated along the path to the slicing criterion. We need to discard the initializations that do not have a valid path of influence to the criterion.

RI-I, RI-II and RI-IV are used to handle the clipping orthogonal method explorations, which can occur due to phantom method invocations or pre-configured clipping at a certain depth. RI-III is used to achieve data structure awareness and RI-V are used to compensate path insensitivity. Next, we highlight the details of two refinement insights based on removing state indicators and resource identifiers. Details for other RIs can be found in the appendix.

5.2 RI-I: Removal of State Indicators

Clipping orthogonal method exploration can cause false positives, if the arguments of method is used to describe the state of a variable. Consider UTF-8 in Line 38 of Figure 1(a). Its Jimple[5] representation is as follows, where $r2 represents variable key, $r4 represents keyBytes, and virtualinvoke is for invoking the non-static method of a class.

```
$r4 = virtualinvoke $r2.<java.lang.String: byte[]
getBytes(java.lang.String)>("UTF-8")
```

If the analysis is clipped so that it cannot explore the getBytes method, then a def-use analysis shows that constant UTF-8 influences the value of $r4 (i.e., keyBytes). Thus, a straightforward detection method would report UTF-8 as a hardcoded key. However, UTF-8 is for describing the encoding of $r2 and can be safely ignored. We refer to this type of constants as *state indicator pseudo-influence.*

The use of refinement insights has direct impact on analysis outcomes. For example, discarding arguments of virtualinvoke may generate false negatives. Suppose virtualinvoke is used to set a key in a KeyHolder instance with some constant: virtualinvoke $r5.<KeyHolder: void setKey(java.lang.String)>("abcd"). Constant abcd needs to be flagged. On the contrary, we observe that arguments of virtualinvoke appearing in assign statements are typically used to describe the state of a variable and can be ignored. Thus, RI-I states that *i)* arguments of any virtualinvoke method invocation in an assignment instruction can be regarded as pseudo-influences, and *ii)* any constants that influence these arguments can also be discarded.

5.3 RI-II: Removal of Source Identifiers

Another type of pseudo-influences due to the clipping of orthogonal method exploration is the identifiers of value sources. We use an example to illustrate the importance of this insight. For the code below, a straightforward analysis would flag constant ENCRYPT_KEY. However, it is an identifier for retrieving a value from a Java Map data structure, and thus a false positive.

```
$r30 = interfaceinvoke r29.<java.util.Map: java.lang.Object
get(java.lang.Object)>("ENCRYPT_KEY")
```

i) Retrieving values from an external source. Static method invocations (staticinvoke in Jimple) in assign statements are typically used to read values from external sources, e.g., Line 15 in Figure 1(a):

```
$r4 = staticinvoke <Context: java.lang.String
getProperty(java.lang.String)>(src)
```

Variable src refers to the identifier, not the actual value of the key. Thus, it is a pseudo influence. To avoid such pseudo-influences, RI-II discards any arguments of staticinvoke that appear in an assignment. Although staticinvoke may be used to transform a value from one representation to another, it is unlikely to use staticinvoke to transform a constant.

5.4 Evaluation of Refinement Methods

We compared the numbers of reported alerts before and after employing the five refinement algorithms for 46 Apache projects and 6,181 Android apps. Our experiments show that refinement algorithms reduce the total alerts by 76% in Apache and 80% in Android. For Apache projects, we manually confirmed that all the removed alerts are indeed false positives[6]. All constant-related rules (including 1, 2, 3, and 12) greatly benefit from the refinements and have significant reduction of irrelevant alerts. Results for top six rules with maximum reductions are shown in Figure 3. The detailed breakdown is shown in Figure 7 in the appendix. The most effective

[5] Jimple is an intermediate representation (IR) of a Java program.

[6] Regarding the validity of the manual analysis, the manual confirmation of alerts was conducted by a second-year Ph.D. student with a prior Master degree in cybersecurity (the second author), under the close guidance of a professor and a senior Ph.D. student (the first author).

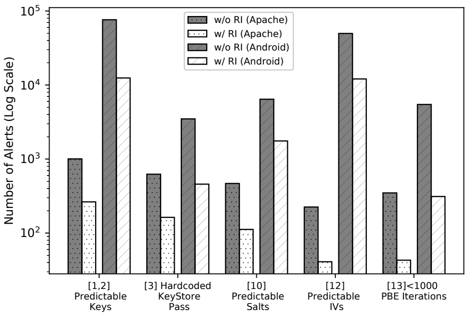


Figure 3: Reduction of false positives with refinement insights in 46 Apache projects (94 root-subprojects) and 6,181 Android apps. Top 6 rules with maximum reductions are shown.

refinement insight for Apache and Android are RI-III (removal of array/collection bookkeeping information).

With refinements enabled, there are a total of 1,295 alerts for the 46 Apache projects. Our careful manual source-code analysis confirms that 1,277 alerts are true positives, resulting in a precision of 98.61%. Out of the 18 false positives, 1 case is due to path insensitivity and 17 to clipping orthogonal explorations (discussed in Section 7). All experiments reported in the next section were conducted with refinements enabled. Refinements may cause false negatives, which we discuss in Section 7.

Impact of Orthogonal Exploration Depth. To measure the impact of the orthogonal exploration depth, we conducted an experiment with 30 Apache root-subprojects and varied the clipping from depth 1 to 10. The depth of orthogonal exploration refers to the distance of an orthogonal method from the main slice. An orthogonal method at depth 1 is a method invoked by the main slice.

The results are shown in Figure 4. The total number of discovered constants across all projects increases slightly with the depth (Figure 4(a) right Y-axis). However, our manual analysis revealed that none of the new constants is a true positive, i.e., the new constants are false positives. Thus, the increase of the orthogonal exploration depth does not improve the recall in this specific experiment, causing a decrease in the F1 score (Figure 4(a) left Y-axis). Interestingly, the analysis runtime does not increase with the increasing depth (Figure 4(b)). The average runtime for the 30 root-subprojects is presented in Table 7 in the appendix. Figure 4(c) shows that the number of inter-procedural slices and their average sizes are drastically reduced when the depth increases from 1 to 2. When the analysis explores inside a method, influences on an argument of an orthogonal invocation might become irrelevant, causing this drastic reduction. Given these observations, we set the orthogonal exploration depth to 1 for the rest of our experiments, as it returns the fewest number of irrelevant constants.

6 SECURITY FINDINGS AND EVALUATION

Our experimental evaluation answers the following questions.

- What are the security findings in Apache Projects? Do Apache projects have any high-risk vulnerabilities such as hardcoded secrets or MitM vulnerabilities? (Section 6.1)
- What are the security findings in Android Apps? Do third-party libraries have any high-risk vulnerabilities? (Section 6.2)
- How does CryptoGuard compare with CrySL, SpotBugs, and the free trial version of Coverity on benchmarks or real-world projects? (Section 6.3)

Table 2: Breakdown of accuracy in Apache projects. Duplicates are handled at root-subproject level (total 82 root-subprojects) level. For Rules 1, 2, 3, 8, 10, 12, each constant/predictable value of an array/collection is considered as an individual violation.

Rules	Total Alerts	# True Positives	Precision
(1,2) Predictable Keys	264	248	94.14 %
(3) Hardcoded Store Pass	148	148	100 %
(4) Dummy Hostname Verifier	12	12	100 %
(5) Dummy Cert. Validation	30	30	100 %
(6) Used Improper Socket	4	4	100 %
(7) Used HTTP	222	222	100 %
(8) Predictable Seeds	0	0	0%
(9) Untrusted PRNG	142	142	100 %
(10) Static Salts	112	112	100 %
(11) ECB mode for Symm. Crypto	41	41	100 %
(12) Static IV	41	40	97.56 %
(13) <1000 PBE iterations	43	42	97.67 %
(14) Broken Symm. Crypto Algorithm	86	86	100 %
(15) Insecure Asymm. Crypto	12	12	100 %
(16) Broken Hash	138	138	100 %
Total	**1,295**	**1,277**	**98.61 %**

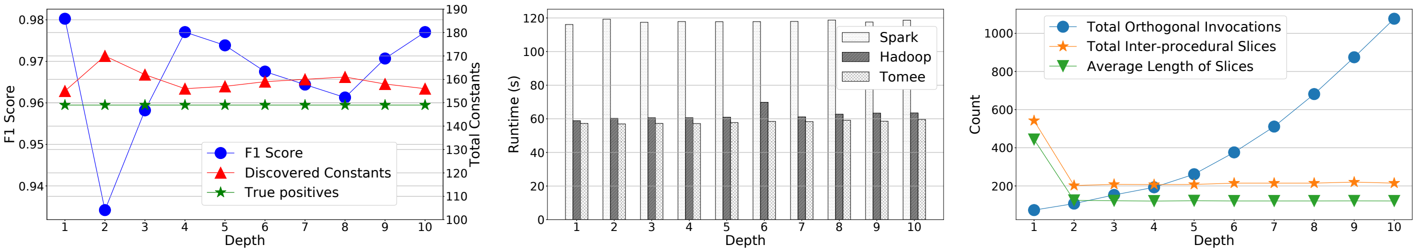
Selection and pre-processing of programs. We selected 46 popular Apache projects that have crypto API uses. The popularity is measured with the numbers of stars and forks in Github. The maximum, minimum and average Line of Code (LoC) are around 2, 571K (Hadoop), 1.1K (Commons Crypto) and 402K, respectively. We perform subproject dependency analysis to build DAGs by parsing build scripts. Subproject dependency analysis was automated for *gradle* and *maven*, and was manual for *Ant*. We identified the root-subprojects, which are sub-projects that have no incoming edges on the subproject dependency DAG. We analyzed 94 root-subprojects in total[7]. We downloaded 6, 181 high popularity Android apps from the Google app market covering 58 categories. The median value of the number of apps per category is 120. We used Soot to decompile .apk files to Java bytecode in order to interface with CryptoGuard. We use online APK decompiler to obtain human-readable source code for manual verification.

We ran 4 concurrent instances of CryptoGuard in an Intel Xeon(R) X5650 server (2.67GHz CPU and 32GB RAM). For Apache, the average runtime was 3.3 minutes with a median of around 1 minute. For Android, we terminated unfinished analysis after 10 minutes. The average runtime was 3.2 minutes with a median of 2.85 minutes, including the cutoff ones. 552 (9%) of 6,181 app's analysis did not finish within 10 minutes, on which CryptoGuard generated partial results. Most of them missed results from Rule 7, which CryptoGuard runs the last.

6.1 Security Findings in Apache Projects

Out of the 46 Apache projects, 39 projects have at least one type of cryptographic misuses and 33 projects have at least two types. Table 9 summarizes our security findings in screening Apache

[7] We exclude 15 test root-subprojects.



(a) (b) (c)

Figure 4: The impact of the orthogonal exploration depth on F1 scores and the number of discovered constants in (a), runtime in (b), and analysis properties in (c) for 8 rules.

projects. Predictable keys (Rules 1 and 2), HTTP URL (Rule 7), insecure hash functions (Rule 16), and the insecure PRNGs (Rule 9) are the most common types of vulnerabilities in Apache. As predictable values, we only observed constants for all these rules. We did not observe any predictable seeds under Rule 8.

```
<http:tlsClientParameters disableCNCheck="true">
      ...
</http:tlsClientParameters>
```

(a) A portion of `https-cfg-client.xml`

```
...
} else if (tlsClientParameters.isDisableCNCheck()) {
  verifier = new AllowAllHostnameVerifier();
}
```

(b) A portion of `SSLUtils.java`

Figure 5: Disabled hostname verification in Apache Cxf.

```
public static String sendUpsRequest(...) {
  ...
  http.setAllowUntrusted(true);
  ... }
```

(a) A portion of `UpsServices.java`

```
SSLContext getSSLContext(String alias, boolean trustAny) {
  ...
  TrustManager[] tm;
  if (trustAny) {
    tm = SSLUtil.getTrustAnyManagers(); } ... }
```

(b) A portion of `SSLUtil.java`

Figure 6: Trusting all certificates in Apache Ofbiz.

6.1.1 Vulnerabilities from Predictable Secrets. 16 Apache projects (37 sub-rootprojects) have hardcoded keys (Rule 1, 2). Three (Meecrowave, Kylin, and Cloudstack) of them use hardcoded symmetric keys (Rule 1). Meecrowave uses *DESede* (i.e., Triple DES[8]) for obfuscation purpose. Unfortunately, deterministic keys make it trivial to break the obfuscation. Kylin (635 Forks, 1325 Stars) uses *AES* to encrypt user passwords. However, using hardcoded keys makes these passwords vulnerable. In Apache Cloudstack, it appears that hardcoded keys are used in the test code, which is accidentally packaged with the production code.

For Rule 2, most of the hardcoded passwords in PBE serve as the default. The most common default password for PBE is `masterpassphrase` (e.g., Ambari and Knox). Manifoldcf uses `NowIsTheTime`. Setting PBE to take the default hardcoded passwords without sufficient warnings are risky. Distributions using the default configuration are susceptible to the recovery of the plaintext password by an attacker who has the access to the PBE ciphertext. Apache Ranger (165 forks, 155 stars) uses a hardcoded password as default for PBE for all distributions. Its installation Wiki does not mention about it. System administrators unaware of this setup are likely not to change the default. This coding practice significantly weakens the security guarantee of PBE.

For Rule 3, most common hardcoded passwords for KeyStores (for storing private keys) are `changeit` (e.g., Tomcat, Knox, Judi, Ofbiz and Wss4j) and none (e.g., Knox, Hive and Hadoop). Most of them are set as default. There are 9 projects that have both predictable keys (Rules 1 and 2) and hardcoded KeyStore passwords (Rule 3), indicating persistent insecure coding styles.

Insecure common practices. During manual analysis, we found three types of insecure common practices in Apache projects for storing secrets: *i)* hard-coding default keys or passwords in the source code, *ii)* storing plaintext keys or passwords in configuration files, and *iii)* storing encrypted passwords in configuration files with decryption keys in plaintext in source code or configuration. Java provides a special security APIs (e.g., `Callback` and `CallbackHandler`) to prompt users for secrets (e.g., passwords). However, none of these projects support this option.

Sysadmins are forced to store plaintext passwords in the filesystem unless they personally modify the code. The biggest danger that these insecure secret-storage practices bring to users is probably the inflated sense of security and not being able to know the actual risks.

6.1.2 Vulnerabilities from SSL/TLS MitM. Man-in-the-Middle (MitM) vulnerabilities are high risk in our threat model. 5 Apache projects (8 root-subprojects) have dummy hostname verifiers that accept any hostnames (Rule 4), including Spark (15086 forks, 16324 stars), Ambari (814 forks, 778 stars), Cxf (706 forks, 398 stars), Ofbiz, and Meecrowave. 6 Apache projects have dummy trust managers that trust any certificates (Rule 5), including Spark, Ambari, Cloudstack, Qpid-broker, Jclouds, and Ofbiz. It appears that most projects offer them as an additional connectivity option.

[8] Triple DES itself is considered insecure. OpenSSL removed the support of Triple DES. NIST recommended moving to AES as soon as possible [71].

Our manual analysis reveals that some projects set this insecure implementation as default (e.g., Figure 5 and Figure 6). In Figure 6, we see that Ofbiz uses insecure SSL/TLS configurations by default while using UPS (a shipping company) service. When plain sockets are used, it is recommended to verify the hostname manually. We found 3 projects that do not follow this rule and accept any arbitrary hostnames. We also found 7 projects (24 root-subprojects) that occasionally use the HTTP protocol for communication.

6.1.3 Medium and Low Severity Vulnerabilities.

It is important to be aware of the medium and low-risk vulnerabilities in the system and to recognize that the risk levels may increase under different adversarial models.

We found hardcoded salts in 4 projects including Apache Ranger, Manifoldcf, Juddi, and Wicket. We also observe the use of ECB mode in AES in 5 projects and predictable IVs in 2 projects with a total of 40 occurrences. We found 5 projects that use PBE with less than 1,000 iterations (Rule 13). Ranger and Wicket projects use 17 iterations for PBE; and Incubator-Taverna-Workbench and Juddi projects use 20 iterations, much fewer than the required 1,000.

Listing 1: A vulnerable code snippet from Apache Ranger

```
1 PBEKeySpec getPBEParameterSpec(String password) throws Throwable {
2    MessageDigest md = MessageDigest.getInstance(MD_ALGO); // MD5
3    byte[] saltGen = md.digest(password.getBytes());
4    byte[] salt = new byte[SALT_SIZE];
5    System.arraycopy(saltGen, 0, salt, 0, SALT_SIZE);
6    int iteration = password.toCharArray().length + 1;
7    return new PBEKeySpec(password.toCharArray(), salt, iteration); }
```

Listing 1 shows a code snippet from Ranger, which has multiple issues. The number of iterations is proportional to the password size (Line 6), which is far less than the required $1,000$. In addition, this code offers a timing side-channel. An adversary capable of measuring PBE execution time (e.g., in multi-tenant environments) may learn the length of the password. This information can substantially decrease the difficulty of dictionary attacks. Another issue is that the salt is computed as the MD5 hash of the password (Lines 2-3). An adversary obtaining the salt may quickly recover the password. The salt's dependence on the password itself also breaks the indistinguishability requirement of PBE under chosen plaintext attack.

Listing 2: Only checking the expiration (`checkValidity`) of self-signed certificates in *Yahoo Finance (TWStock)* app, due to (`com.softmobile`) library.

```
void checkServerTrusted(X509Certificate[] chain, String str){
 if (chain == null || chain.length != 1) {
     this.f7654a.checkServerTrusted(chain, str);
 } else {
    //Lack of signature verification and others
    chain[0].checkValidity();}}
```

Listing 3: Ignoring exceptions in `checkServerTrusted` in *Sina Finance* app.

```
void checkServerTrusted(X509Certificate[] chain, String str){
  try {
    this.f7427a.checkServerTrusted(chain, str);
  } catch (CertificateException e) {}} //Ignores exception
```

We found various occurrences of Blowfish, DES, and RC4 ciphers for Rule 14. Under Rule 15, we found 3 occurrences of using default key size of 1024 and 9 other occurrences that explicitly initialize the key size to 1024. 23 projects use `java.util.Random` as a PRNG (Rule 9), where two of them set static seeds to `java.util.Random`. We do not observe any deterministic seed to a `java.security.SecureRandom` (Rule 8).

Listing 4: `SSLSocket` without manual hostname verification in *ProTaxi Driver* app.

```
try {
 SSLContext instance = SSLContext.getInstance("TLS");
 ...
 this.webSocketClient
     .setSocket(instance.getSocketFactory().createSocket());
} catch (Throwable e) { ... }
this.webSocketClient.connect();
```

6.2 Security Findings in Android Apps

Violations in apps or in libraries? We distinguished app's own code from libraries by using the package information from `AndroidManifest.xml`.[9] Android also uses it during `R.java` file generation (robust against obfuscation). We found that on average **95% of the detected vulnerabilities come from libraries** (Table 4). This result extends the observation from 7 types of vulnerabilities (reported in [19]) to 16.

Table 4 shows the distribution of vulnerability sources for each rule. For hardcoded KeyStore passwords (Rule 3), all violations come from libraries. Most frequent hardcoded KeyStore password is `notasecret`, which is used to access certificates and keys in Google libraries (e.g., `*.googleapis.GoogleUtils`, `*.googleapis.*.GoogleCredential`).

Besides Google, other high-profile library sources include Facebook, Apache, Umeng, and Tencent (Table 5). These libraries frequently appear in different applications. We distinguished these libraries using base packages. CryptoGuard can detect API misuses in obfuscated packages, i.e., any violations from within the obfuscated code are also reported. However, we are unable to report the vendors of obfuscated libraries. Pinpointing the source of an obfuscated package is an active area of research [19].

Overview of other Android findings. We found exposed secrets, similar to Apache projects. Table 9 summarizes the discovered vulnerabilities in Android applications. The categories of untrusted PRNG (Rule 9) and broken hash (Rule 16) have the most violations. Interestingly, we observed 544 cases of predictable seeds (Rule 8). 13 cases of them used time-stamps from `<java.lang.System: long currentTimeMillis()>` API calls.

Compared with Apache projects, Android apps have higher percentages of SSL/TLS API misuses (Rules 4, 5 and 6) and HTTP use (Rule 7). For example, 25.3% of Android apps have dummy trust manager (Rule 5), which is more than twice the number in Apache (11.7%) as shown in Table 9 in the appendix.

Our analysis can detect sophisticated cases that Google Play's built-in screening is likely to miss. We give code snippets for such cases (Listing 2, 3, 4). CryptoGuard detects a case where developers allow unpinned self-signed certificates with a mere expiration check, as shown in Listing 2. Another case is where developers ignore the exception in `checkServerTrusted` method as shown in Listing 3. In addition, CryptoGuard detects 271 occurrences

[9] An .apk contains both the app code and the libraries.

Table 3: Experimental results on the CryptoApi-Bench basic and CryptoApi-Bench advanced benchmarks (as of April 2019) with CrySL, Coverity, SpotBugs and CryptoGuard. GTP stands for the ground truth positives. TP, FP, and FN are the number of true positives, false positives, false negatives in a tool's output, respectively. Pre. and Rec. represent precision and recall, respectively. Tools are evaluated on 6 common rules (out of our 16 rules), i.e., the maximum common subset of all tools. For these 6 rules, there are 6 correct cases (i.e., true negatives) in basic and 3 correct cases in advanced, which are used for computing FPRs. Total alerts = TP + FP.

Tools	CryptoApi-Bench: Basic							CryptoApi-Bench: Advanced															
	GT:14			Summary				Inter-Pro. (Two) GT: 13			Inter-Pro. (Multiple) GT: 13			Field Sensitive GT: 13			False Positive GT: 3			Summary			
	TP	FP	FN	FPR	FNR	Pre.	Rec.	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	FPR	FNR	Pre.	Rec.
CrySL[47]	10	6	4	50.00	28.57	**62.50**	**71.43**	10	3	3	0	2	13	10	2	3	0	6	3	81.25	52.38	**60.61**	**47.62**
Coverity[2]	13	0	1	0.00	7.14	**100.0**	**92.86**	3	0	10	3	0	10	1	0	12	0	0	3	0.00	83.33	**100.0**	**16.67**
SpotBugs[3]	13	0	1	0.00	7.14	**100.0**	**92.86**	0	0	13	0	12	13	0	0	13	0	0	3	80.00	100.0	**0.00**	**0.00**
CryptoGuard	14	0	0	0.00	0.00	**100.0**	**100.0**	12	0	1	12	0	1	13	0	0	3	0	0	0.00	4.76	**100.0**	**95.24**

Table 4: Distribution of vulnerabilities in Android apps.

	Library (Total)	Library (Unique)	App Itself	Total
(1,2) Predictable Keys	11,634 (93.4%)	5,940	823 (6.6%)	12,457
(3) Hardcoded Store Password	431 (94.1%)	170	27 (5.8%)	458
(4) Dummy Hostname Verifier	1,148 (99.3%)	51	7 (0.7%)	1,155
(5) Dummy Cert. Validation	3,715 (96.3%)	1,317	141 (3.7%)	3,856
(6) Used Improper Socket	270 (99.6.4%)	13	1 (0.4%)	271
(7) Used HTTP	7,687 (92.5%)	2,105	623 (7.5%)	8,321
(8) Predictable Seeds	522 (96.0%)	101	22 (4.0%)	544
(9) Untrusted PRNG	26,312 (91.7%)	8,679	2,393 (8.3%)	36,223
(10) Predictable Salts	1,638 (93.2%)	774	119 (6.8%)	1,757
(11) ECB in Symm. Crypto	1657 (93.1%)	682	123 (6.9%)	1,780
(12) Predictable IVs	11,357 (94.2%)	6,048	692 (5.8%)	12,089
(13) <1000 PBE iterations	294 (94.2%)	129	18 (57.8%)	312
(14) Broken Symm. Crypto	1,668 (95.8%)	753	74 (4.2%)	1,742
(15) Insecure Asymm. Crypto	4 (3.6%)	3	107 (96.4%)	111
(16) Broken Hash	49,257 (99.0%)	7509	496 (1.0%)	49,769
Total	**117,594 (95.40%)**	**34,274**	**5,666 (4.60%)**	**130,845**

Table 5: Violations in 5 popular libs (manually confirmed).

Package name	Violated rules
com.google.api	3, 4, 5, 7
com.umeng.analytics	7, 9, 12, 16
com.facebook.ads	5, 9, 16
org.apache.commons	5, 9, 16
com.tencent.open	2, 7, 9

of improper use of `SSLSocket` without manual Hostname verification in 210 apps. One such example is shown in Listing 4, where `SSLSocket` is used in `WebSocketClient` without manually verifying the hostname [10]. In comparison, Google Play's inspection appears to only detect obvious misuses [5].

Grouping security violations by app popularity or category did not show substantial differences across groups.

6.3 Comparison with Existing Tools

We compare the accuracy and runtime of CryptoGuard with three existing tools, i.e., CrySL [47], Coverity [2], and SpotBugs [3][11]. We use CryptoGuard 03.06.00 (commit id *ea75a45*), SpotBugs 3.1.0 (from SWAMP). Results from Coverity online were obtained before March 30, 2019. For CrySL, we analyze Apache projects with CrySL 2.0 (commit id *5f531d1*) and Android applications with CrySL-Android 1.0.0 (commit id *856b1da*) [1].

Benchmark preparation. First, we[12] had to construct CryptoApi-Bench, a comprehensive benchmark for comparing the quality of cryptographic vulnerability detection tools. Regarding the existing benchmark DroidBench [18], *i)* DroidBench does not cover cryptographic APIs, *ii)* the free web version of Coverity requires source code, however DroidBench only contains APK binaries.

CryptoApi-Bench covers all 16 cryptographic rules specified in Table 1. As of April 2019, there are 38 basic test cases and 74 advanced test cases. The basic benchmark contains 25 straightforward API misuses and 13 correct API uses (i.e., true negative cases). The advanced cases have more complex scenarios, including 42 inter-procedural cases[13], 20 field-sensitive cases, 9 false positive test cases (for evaluating the ability of recognizing irrelevant elements), and 3 correct API uses (i.e., true negative cases). Figures 8 and 9 in the appendix show the distributions of test cases per rule and per API, respectively. A more recent version of the benchmark with more diverse test cases can be found in [17]. See Github for the most updated version `https://github.com/CryptoGuardOSS/cryptoapi-bench`.

Benchmark comparison. To maintain fairness in our comparison, we only report the benchmark results for the six shared rules (1, 2, 3, 11, 14, 16) that are covered by all the tools, CrySL [47], Coverity [2], SpotBugs [3], and ours. Due to the lack of documentation, we had to infer a tool's coverage based on whether or not it ever generates any alert in that category. We show the results in Table 3. SpotBugs, Coverity, and CryptoGuard perform well on the basic benchmark. For CrySL, its errors are partly due to their rule definitions being very specific. For example, CrySL raises an alert if a cryptographic key is not directly obtained from the key generator. However, in some cases, a previously generated cryptographic key can be used securely in the code without a key generator. For cryptographic passwords, CrySL raises an alert if it is derived from a String, likely because Java recommends using char[] so that a password can be wiped after use. However, this String-based policy would miss hardcoded passwords defined in char[], generating false negatives. For the advance benchmark, both CrySL and SpotBugs generate false positives, when a variable is passed through multiple methods. For all cases, Coverity has zero false positives, likely because of the use

[10] Guide for the correct use can be found at https://developer.android.com/training/articles/security-ssl#WarningsSslSocket.

[11] CryptoLint's code is unavailable.

[12] The person (third author) who led the benchmark design is different from the person (first author) who implemented CryptoGuard.

[13] 21 cases involve two methods and 21 cases involve more than two methods.

Table 6: Summary of average runtime (in seconds) across all completed runs for CrySL and CryptoGuard. We evaluated 30 Apache root-subprojects and 30 Android apps, each with 3 runs. Incmpl stands for the number of incomplete analyses. Standard deviations (std) are computed across projects/apps. Variations across runs are negligible.

Tool	Apache root-subprojects			Android applications		
	Incmpl.	Avg. (std)	Median	Incmpl.	Avg. (std)	Median
CrySL	18	16.5 (18.0)	6.9	4	15.7 (44.1)	5.2
Ours	0	12.7 (14.2)	6.4	0	187.8 (488.3)	52.4

of symbolic execution and/or path-sensitive analysis[14]. However, Coverity misses multiple advanced vulnerability scenarios (for rules that it does cover in the basic benchmark).

Table 10 in the appendix presents the comparison for all 16 rules (not just the 6 common rules). When testing all 16 rules, CryptoGuard failed to report 11 misuses (i.e., false negatives). We discuss the causes in Section 7.

Runtime comparison. We compare CrySL and CryptoGuard on 30 randomly selected Apache root-subprojects (LoC ranging from 471K to 1K) and 30 Android applications (LoC ranging from 1,453K to 0.4K), with 3 runs each. The results are summarized in Table 6 with full runtime details sorted by LoC in Figure 10 and LoC Table 8 in the appendix[15]. CryptoGuard completed all tasks, demonstrating robust and efficient performance. CrySL exited prematurely for 18 Apache projects and 4 Android apps due to various errors (e.g., memory errors[16]). For Apache, CryptoGuard exhibits better overall runtime performance than CrySL.

For Android, CrySL is faster, partly because CrySL only analyzes the code that is reachable by an app's life-cycle. In comparison, CryptoGuard also covers third-party libraries (regardless of life-cycle reachability) and produces more valid alerts. For example, for Card_Maker_for_Pokemon, CryptoGuard and CrySL generated 2 and 0 alerts, respectively. For Cartoon_Avatar_Maker, CryptoGuard and CrySL generated 5 and 1 alerts, respectively. The 8 alerts are distinct true positives in libraries, which means CryptoGuard has 1 false negative and CrySL has 7 false negatives. CryptoGuard's false negative (an MD5 use) comes from the Android core library *com.google.android*, which CryptoGuard currently does not analyze.

For the free web version of Coverity, we are unable to obtain its runtime. We choose not to compare the runtime with SpotBugs. The comparison would not be meaningful, as its analysis is mostly based on the syntactical matching of source code to known bug patterns [43, 65].

Summary of findings. Refinements bring a 76% reduction in alarms for Apache projects and an 80% reduction for Android applications. For Apache projects, we manually confirmed that all the removed alerts are indeed false positives. Manually examining the remaining 1,295 Apache alerts (after refinements) confirms our precision of 98.61%. 39 out of the 46 Apache projects have at least one type of cryptographic misuses and 33 have at least two types. There is a widespread insecure practice of storing plaintext passwords in code or in configuration files. Insecure uses of SSL/TLS APIs are set as the default configuration in some cases. 5,596 (91%) out of the 6,181 Android apps have at least one type of cryptographic misuses and 4,884 (79%) apps have at least two types. 95% of the vulnerabilities come from the libraries that are packaged with the applications. Some libraries are from large software firms. CryptoGuard's detection for SSL/TLS API misuses is more comprehensive than the built-in screening offered by Google Play.

7 LIMITATIONS AND DISCUSSION

No static analysis tool is perfect. CryptoGuard is no exception. We discuss the detection limitations of CryptoGuard and future improvements.

CryptoGuard runs the intra-procedural forward slicing for Rules 6 and 15, where an inter-procedural forward slicing could potentially improve the coverage. For Rule 15, this change might not make much difference, as `KeyPairGenerator` creation and its initialization usually occur in the same method. For Rule 6, our current implementation ignores the direct sub-classes of `SSLSocketFactory` to avoid false positives. Inter-procedural slicing could extend the analysis to the sub-classes.

False positives. One source of false positives comes from the path insensitivity. For example, CryptoGuard raises an alert if the variable `iteration` is assigned with a value of 0 for the following code snippet (from project *jackrabbit-oak*). However, this alert is a false positive, since this assignment is on an infeasible path.

```
int iteration = 0;
...
if (iteration < NO_ITERATION) { // NO_ITERATION = 1
    iteration = DEFAULT_ITERATION;
}
```

CryptoGuard detects the existence of API misuses in a code base but does not verify that the vulnerable code will be triggered at runtime. This issue is a general limitation of static program analysis. Apache Spark confirmed insecure PRNG uses, but stated that the affected code regions are not security critical.[17] However, eliminating this type of alerts is difficult as the analysis needs to be aware of custom defined security criteria (e.g., what constitutes critical security) with in-depth knowledge about project semantics.

Another source of false positives is clipping orthogonal exploration. However, deeper exploration has impacts on both ends – eliminating some false positives while increasing the overall number of (irrelevant) constants discovered. As our experiment shows (in Figure 4), the net result of increasing the depth appears to be discovering more irrelevant constants (as opposed to reducing them).

False negatives due to refinements. Refinements may cause false negatives. For the full benchmark evaluation in Table 10 in Appendix, CryptoGuard has 11 false negatives (i.e., missed detection). All these cases are due to our refinements after clipping orthogonal explorations. For example, RI-II would ignore 6A5B7C8A as a pseudo-influence from the following instruction, if orthogonal explorations to explore *parseHexBinary* are clipped.

`byte[] key = DatatypeConverter.parseHexBinary("6A5B7C8A").` However, these conversions are mostly required to process values from external sources (e.g., file system, network). Any such conversions

[14] Coverity is close sourced, so we are unable to confirm.

[15] LoC is obtained using online Java and APK decompilers and cloc command.

[16] We increased the heap size to 10GB for CrySL, while CryptoGuard ran with the default 4GB heap memory.

[17] It is unclear why Spark chose to use insecure PRNG, even for non-security purposes.

of static values under the rules of Table 1 are highly unlikely. Indeed, outside the benchmark, we did not observe any such cases during our manual investigation of Apache alerts. Additionally, vulnerabilities originated from a clipped orthogonal method may also be missed by CryptoGuard.

Conceptually, all such false negatives could be avoided by increasing the depth of the orthogonal exploration (default depth is 1). Our results in Figure 4 on 30 Apache root-subprojects with varying depths of orthogonal explorations show that the increase of the depth does not necessarily discover new true positive cases or increase the F1 score.

Vulnerability disclosure and feedback. We have heard back from a number of Apache projects regarding our vulnerability disclosure, including Tomcat, Hadoop, Hive, Spark, Ofbiz, Ambari, and Ranger. Apache Spark removed the support of dummy hostname verifier and dummy trust store. Apache Ranger fixed constant default values for PBE [11] and insecure cryptographic primitives [6]. Ofbiz promised to fix the reported issues of constant IVs and KeyStore passwords. Regarding MD5, Apache Hadoop justifies that its MD5 use is for the per-block checksums for Hadoop file systems (HDFS)'s consistency and the setup does not assume the presence of active adversaries. For Android libraries, we have submitted vulnerability reports to Google. Google closed our issue with 4 misuses in Google libraries citing the lack of concrete exploit demonstrations. Facebook, and Tencent have similar requirements. We also received similar feedback from an Apache software foundation's administrator demanding concrete exploit demonstrations before more reported issues can be examined.

Some developers explained that certain operational constraints (e.g., backward compatibility for clients) prevent them from fixing the problems. For example, Apache Tomcat server has to use MD5 in its digest authentication code, because major browsers do not support secure hash functions (as defined in RFC 7616). However, digest authentication is rarely used in the wild[18]. The thorniest issue is secret storage. One justification for developers' choice of storing plaintext passwords or keys in file systems is for supporting humanless environments (e.g., automated scripts to manage services). However, not all deployment scenarios are server farms in a humanless environment. Projects should provide the secure option by prompt human operators for passwords that can be used to unlock/generate other passwords or keys on the fly. Not properly disclosing and documenting the insecure configurations does a great disservice to the project's users.

8 RELATED WORK

Tools to detect cryptographic misuses. Cryptographic misuse detection tools are typically constructed into two broad groups, i.e., static analysis (e.g., CryptoLint [33], MalloDroid [35], FixDroid [60], CogniCrypt [46] and CrySL [47]) and dynamic analysis (e.g., SMV-Hunter [68], AndroSSL [36] and K-Hunt [50]). For example, MalloDroid [35] uses a list of known insecure implementations of `HostnameVerifier` and `TrustManager` to screen Android apps. In [44], authors showed that generating false positives is one of the most significant barrier to adopt static analysis tools. This problem also exists in anomaly and intrusion detection systems [51, 75]. When screening large projects, virtually all static slicing solutions in this space (e.g., [33]) might generate a non-negligible amount of false positives. Contextual refinements similar to CryptoGuard's is necessary to achieve high precision in practice.

CrySL and CryptoGuard have different-but-overlapping security capabilities. Based on CrySL code and documentation, we identified rules that CrySL supports, but CryptoGuard does not cover. For example, CrySL covers rules to verify the correctness of Signature and MAC generation procedures. CrySL also reports non-crypto issues, e.g., variables not being used. On the flip side, CryptoGuard supports rules that CrySL does not cover (Rules 4, 5, 7, 8, and 9), including dummy hostname verifier, dummy certificate validation, use of HTTP, predictable seeds, and untrusted PRNG. For fairness, we only compare the intersected portion of capabilities.

Other misuse detection tools (e.g., FixDroid [60] and CogniCrypt [46]) were mainly built for the user-experience study with the goal of making detection tools developer-friendly, as opposed to a deployment-quality screening solution. For example, FixDroid focuses on providing real-time feedback to developers. CogniCrypt's [46] focus is on code generation (in Eclipse IDE) for several common cryptographic tasks (e.g., data encryption). Some dynamic analysis tools use a simple static analysis to first narrow down the number of potential apps for dynamic analysis. For example, SMV-Hunter [68] looks for apps that contain any custom implementation of `X509TrustManager` or `HostNameVerifier` for the initial screening.

Other tools. TaintCrypt [64] uses static taint analysis to discover library-level cryptographic implementation issues in C/C++ cryptographic libraries (e.g., OpenSSL). It uses symbolic execution based path exploration to reduce false alarms, which is usually costly. SSLLint [41] uses graph mining techniques to detect SSL/TLS API misuses in C/C++, where a program is represented using program dependence graphs. Researchers found that misusing non-cryptographic APIs in Android also have serious security implications. These APIs include APIs to access sensitive information (such as location, IMEI, and passwords) [59], APIs for fingerprint protection [24], and cloud service APIs for information storage [77]. Data driven techniques to identify API misuses have been proposed [61, 76], which use lightweight static analysis to infer detection rules from examples. In [57], authors proposed a Bayesian framework for automatically learning correct API uses. Efforts on automatically repairing insecure code have also been reported [53, 54, 63]. Static code analysis has been extensively used for other related software problems as well, including malware analysis and detection [34, 62, 73], vulnerability discoveries [24, 48], and data-leak detection [26]. In [29], Chi *et al.* presented a system to infer client behaviors by leveraging symbolic executions of client-side code. They used such knowledge to filter anomalous traffic. Fuzzing has been demonstrated to automatically discover software vulnerabilities [31, 66, 67]. These techniques aim to find input guided vulnerabilities that result in observable behaviors (e.g., triggering program crashes [67] or anomalous protocol states [31, 66]). It is unclear how to use fuzzing to detect cryptographic vulnerabilities (e.g., predictable IVs/secrets, legacy primitives) that do not exhibit easily observable anomalous behaviors.

[18] https://security.stackexchange.com/questions/152935/why-is-there-no-adoption-of-rfc-7616-http-digest-auth

9 CONCLUSIONS AND AN OPEN PROBLEM

We described our effort of producing a deployment-quality static analysis tool CryptoGuard to detect cryptographic misuses in Java programs that developers can routinely use. This effort led to several crypto-specific contributions, including language-specific contextual refinements for FP reduction, on-demand flow-sensitive, context-sensitive, and field-sensitive program slicing, and benchmark comparisons of leading solutions. We also obtained a trove of security insights into Java secure coding practices.

An **open research problem** is designing a compiler that automatically transforms a cryptographic vulnerability or rule into a static-analysis-based code-screening algorithm, similar to what CrySL provides, but with much higher expressiveness, precision, and recall. Enabling stateful analysis capturing the lifecycle of cryptographic objects in CryptoGuard is another useful future direction. Unlike CrySL, CryptoGuard does not perform type-state analysis, i.e., it does not check code against type-based state machines. The lack of type-state analysis may cause an overestimation of vulnerabilities. For example, our detection of predictable keys, based on insecure instantiation of keys, cannot determine whether or not the key will be used.

10 ACKNOWLEDGMENT

Authors would like to thank the anonymous reviewers and VT Systems Reading Group for their insightful feedback. This work has been partly supported by the Office of Naval Research under Grant ONR-N00014-17-1-2498, National Science Foundation under Grant SBIR-1647681, SBIR-1758628.

REFERENCES

[1] CogniCrypt_SAST for Android.

[2] Coverity Static Application Security Testing (SAST).

[3] Spotbugs: Find Bugs in Java Programs.

[4] Cryptographic Key Length Recommendation. https://www.keylength.com/en/4/, 2016. [Online; accessed 29-Jan-2018].

[5] Google Play Warning: How to fix incorrect implementation of HostnameVerifier? https://stackoverflow.com/questions/41312795/google-play-warning-how-to-fix-incorrect-implementation-of-hostnameverifier, 2016. [Online; accessed 29-Jan-2018].

[6] Change the default Crypt Algo to use stronger cryptographic algo. "https://issues.apache.org/jira/browse/RANGER-1644", 2017. [Online; accessed Jan 26, 2018].

[7] Class Random. https://docs.oracle.com/javase/8/docs/api/java/util/Random.html, 2017. [Online; accessed 29-Jan-2018].

[8] Class SecureRandom. https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html, 2017. [Online; accessed 29-Jan-2018].

[9] Lifetimes of cryptographic hash functions. http://valerieaurora.org/hash.html, 2017. [Online; accessed 29-Jan-2018].

[10] List of Rainbow Tables. http://project-rainbowcrack.com/table.htm, 2017. [Online; accessed 29-Jan-2018].

[11] Update Doc/Wiki to provide details on using custom encryption key and salt for encryption of credentials. https://issues.apache.org/jira/browse/RANGER-1645, 2017. [Online; accessed Jan 26, 2018].

[12] Google rejected app because of HostnameVerifier issue. "https://stackoverflow.com/questions/48420530/google-rejected-app-because-of-hostnameverifier-issue", 2018. [Online; accessed Jan 26, 2018].

[13] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *IEEE S&P'17*, pages 154–171, 2017.

[14] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *IEEE S&P'16*, pages 289–305, 2016.

[15] Y. Acar et al. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *IEEE Secure Development Conference SecDev*, 2017.

[16] D. Adrian et al. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *ACM CCS'15*, pages 5–17, 2015.

[17] S. Afrose, S. Rahaman, and D. D. Yao. CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. In *IEEE Secure Development Conference (SecDev)*, September 2019.

[18] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, pages 259–269, 2014.

[19] M. Backes, S. Bugiel, and E. Derr. Reliable Third-Party Library Detection in Android and its Security Applications. In *ACM CCS'16*, pages 356–367, 2016.

[20] G. V. Bard. The Vulnerability of SSL to Chosen Plaintext Attack. *IACR Cryptology ePrint Archive*, 2004:111, 2004.

[21] D. J. Bernstein, Y. Chang, C. Cheng, L. Chou, N. Heninger, T. Lange, and N. van Someren. Factoring RSA Keys from Certified Smart Cards: Coppersmith in the Wild. In *ASIACRYPT'13*, pages 341–360, 2013.

[22] K. Bhargavan and G. Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *ACM CCS'16*, pages 456–467, 2016.

[23] K. Bhargavan and G. Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH. In *NDSS'16*, 2016.

[24] A. Bianchi, Y. Fratantonio, A. Machiry, C. Kruegel, G. Vigna, S. P. H. Chung, and W. Lee. Broken Fingers: On the Usage of the Fingerprint API in Android. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[25] D. Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *NOTICES OF THE AMS*, 46(2), 1999.

[26] A. Bosu, F. Liu, D. D. Yao, and G. Wang. Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications. In *ACM AsiaCCS'17*, pages 71–85, 2017.

[27] D. Chang, A. Jati, S. Mishra, and S. K. Sanadhya. Cryptanalytic Time-Memory Tradeoff for Password Hashing Schemes. *IACR Cryptology ePrint Archive*, 2017:603, 2017.

[28] S. Checkoway, J. Maskiewicz, C. Garman, J. Fried, S. Cohney, M. Green, N. Heninger, R. Weinmann, E. Rescorla, and H. Shacham. A Systematic Analysis of the Juniper Dual EC Incident. In *ACM CCS'16*, pages 468–479, 2016.

[29] A. Chi, R. A. Cochran, M. Nesfield, M. K. Reiter, and C. Sturton. A System to Verify Network Behavior of Known Cryptographic Clients. In *USENIX NSDI'17*, pages 177–195, 2017.

[30] J. Clark and P. C. van Oorschot. SoK: SSL and HTTPS: revisiting past challenges and evaluating certificate trust model enhancements. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 511–525, 2013.

[31] J. de Ruiter and E. Poll. Protocol State Fuzzing of TLS Implementations. In *USENIX Security'15*, pages 193–206, 2015.

[32] Welcome to the SWAMP. https://continuousassurance.org, 2018.

[33] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An empirical study of cryptographic misuse in Android applications. In *ACM CCS'13*, pages 73–84, 2013.

[34] K. O. Elish, X. Shu, D. D. Yao, B. G. Ryder, and X. Jiang. Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49:255–273, 2015.

[35] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory love Android: an analysis of Android SSL (in)Security. In *ACM CCS'12*, pages 50–61, 2012.

[36] F. Gagnon, M. Ferland, M. Fortier, S. Desloges, J. Ouellet, and C. Boileau. AndroSSL: A Platform to Test Android Applications Connection Security. In *FPS'15*, pages 294–302, 2015.

[37] C. P. García, B. B. Brumley, and Y. Yarom. "Make Sure DSA Signing Exponentiations Really are Constant-Time". In *ACM CCS'16*, pages 1639–1650, 2016.

[38] C. Garman, M. Green, G. Kaptchuk, I. Miers, and M. Rushanan. Dancing on the Lip of the Volcano: Chosen Ciphertext Attacks on Apple iMessage. In *USENIX Security'16*, pages 655–672, 2016.

[39] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM CCS'12*, 2012.

[40] I. Goldberg and D. Wagner. Randomness and the Netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21(1):66–71, 1996.

[41] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 519–534, 2015.

[42] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security'12*, pages 205–220, 2012.

[43] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[44] B. Johnson et al. Why don't software developers use static analysis tools to find bugs? In *ICSE'13*, pages 672–681, 2013.

[45] H. Krawczyk. How to Predict Congruential Generators. In *CRYPTO'89*, pages 138–153, 1989.

[46] S. Krüger et al. CogniCrypt: supporting developers in using cryptography. In *IEEE/ACM ASE'17*, pages 931–936, 2017.

[47] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *ECOOP'18*, pages 10:1–10:27, 2018.

[48] Y. Kwon, B. Saltaformaggio, I. L. Kim, K. H. Lee, X. Zhang, and D. Xu. A2C: Self destructing exploit executions via input perturbation. In *NDSS'17*, 2017.

[49] D. Lazar, H. Chen, X. Wang, and N. Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *APSys'14*, 2014.

[50] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 412–425, 2018.

[51] R. Lippmann and R. K. Cunningham. Improving intrusion detection performance using keyword selection and neural networks. *Computer Networks*, 34(4):597–603, 2000.

[52] A. D. Lucia. Program Slicing: Methods and Applications. In *IEEE International Workshop on Source Code Analysis and Manipulation SCAM'01*, pages 144–151, 2001.

[53] S. Ma, D. Lo, T. Li, and R. H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016*, pages 711–722, 2016.

[54] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security*, pages 229–246, 2017.

[55] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *ACM ICSE'18*, Gothenburg, Sweden, May 2018.

[56] K. Moriarty, B. Kaliski, and A. Rusch. PKCS# 5: Password-Based Cryptography Specification Version 2.1. 2017.

[57] V. Murali, S. Chaudhuri, and C. Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 151–162, 2017.

[58] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *ICSE'16*, pages 935–946, 2016.

[59] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[60] D. C. Nguyen et al. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *ACM CCS'17*, pages 1065–1077, 2017.

[61] R. Paletov, P. Tsankov, V. Raychev, and M. T. Vechev. Inferring crypto API rules from code changes. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018*, pages 450–464, 2018.

[62] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. In *NDSS'17*, 2017.

[63] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering*, pages 447–456, 2010.

[64] S. Rahaman and D. Yao. Program Analysis of Cryptographic Implementations for Security. In *IEEE Secure Development Conference (SecDev), 2017*, pages 61–68, 2017.

[65] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 245–256, 2004.

[66] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. HVLearn: Automated Black-Box Analysis of Hostname Verification in SSL/TLS Implementations. In *IEEE S&P'17*, pages 521–538, 2017.

[67] J. Somorovsky. Systematic Fuzzing and Testing of TLS Libraries. In *ACM CCS'16*, pages 1492–1504, 2016.

[68] 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 *NDSS'14*, 2014.

[69] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. The First Collision for Full SHA-1. In *CRYPTO'17*, 2017.

[70] M. Stevens, A. K. Lenstra, and B. de Weger. Chosen-Prefix Collisions for MD5 and Colliding X.509 Certificates for Different Identities. In *EUROCRYPT'07*, pages 1–22, 2007.

[71] Update to Current Use and Deprecation of TDEA, 2017. https://csrc.nist.gov/news/2017/update-to-current-use-and-deprecation-of-tdea.

[72] V. van der Veen et al. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM CCS'16*, pages 1675–1689, 2016.

[73] K. Xu, D. Yao, B. Ryder, and K. Tian. Probabilistic Program Modeling for High-Precision Anomaly Classification. In *CSF'15*, July 2015.

[74] H. Y. Yang, E. D. Tempero, and H. Melton. An Empirical Study into Use of Dependency Injection in Java. In *Australian Software Engineering Conference ASWEC'08*, pages 239–247, 2008.

[75] D. Yao, X. Shu, L. Cheng, and S. J. Stolfo. *Anomaly Detection as a Service: Challenges, Advances, and Opportunities.* In Information Security, Privacy, and Trust Series. Morgan & Claypool., 2017.

[76] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*, pages 886–896, 2018.

[77] C. Zuo, Z. Lin, and Y. Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *IEEE S&P'16*, 2019.

11 APPENDIX

11.1 Other Refinement insights

RI-III: Removal of bookkeeping indices.

```
byte[] iv = new byte[] {0x0, 0x0, 0x0,
  0x0, 0x0, 0x0, 0x0, 0x0}
```

Consider the Java statement above. After transforming into jimple representation, this statement looks like the following list of instructions.

```
  $r15 = newarray (byte)[8]
  $r15[0] = 0
  $r15[1] = 0
  $r15[2] = 0
  $r15[3] = 0
  $r15[4] = 0
  $r15[5] = 0
  $r15[6] = 0

  $r2 = $r15
```

The hard coded size and the indices of an array can be regarded as pseudo-influences. To address this false positives, we discard all the constants that influences an array index. Also, any constant that influences the size or the index parameter of a collection can also be regarded as pseudo-influences. We regard `List`, `Set` as collections.

RI-IV: Removal of contextually incompatible constants.

Clipping of orthogonal invocations that doesn't appear in an assign statement can also cause false positives. To reduce false alarms further, we also discard some constants constants based on its type and context. Let's consider, a class named PBEInfo is used to store iteration count and salt and the analysis cannot explore PBEInfo class. A basic use-def analysis will report 5 as a salt from the following invoke instruction: `specialinvoke r1.<KeyHolder: void <init>(Integer, String)>(5, "5341453")`. However, a standalone Boolean or Integer constant is unlikely to be used as a key, IV or salt, since their corresponding APIs only allow byte arrays. Also, any hard-coded size parameter (e.g., number of iterations in PBE (Rule 13), key size for insecure asymmetric crypto (Rule 15)) is unlikely to have any type other than *Integer*. Therefore, it is possible to discard some of the pseudo-influences by considering the types of a constant based on its context.

RI-V: Removal of constants in infeasible paths.

Some constant initializations are overwritten along the path to the point of interest. Counting such constants with infeasible influences will result in false positives. Since, empty strings and `null`s are used for initialization purpose and most often, these

Depth	Runtime in Sec. (STD)
1	37.23 (49.75)
2	35.5 (39.03)
3	35.75 (39.09)
4	35.82 (39.23)
5	36.1 (38.97)
6	36.7 (39.63)
7	36.83 (39.64)
8	37.99 (40.34)
9	38.54 (41.22)
10	38.87 (41.94)

Table 7: The impact of clipping orthogonal explorations at various depth on runtime across 30 Apache root-subprojects. STD is computed across projects. Variations across multiple runs (3 runs) are negligible.

initialization are replaced with other values. To avoid false positive for this case, depending on rules and the slicing criteria we discard `null` and empty strings. For example, `SecretKeySpec` prohibits keys to be `null` or empty. `IvParameterSpec` does not allow `null` as IV. Also, `PBEParameterSpec` does not allow the salt to be `null`.

11.2 Other Evaluation Results

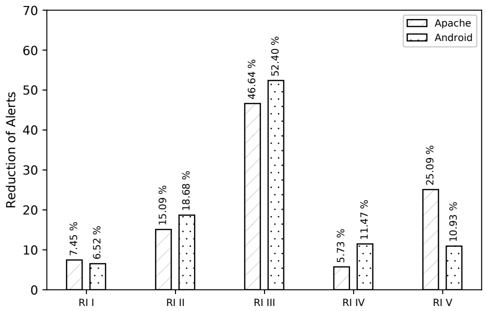


Figure 7: Breakdown of the reduction of false positives due to five of our refinement insights.

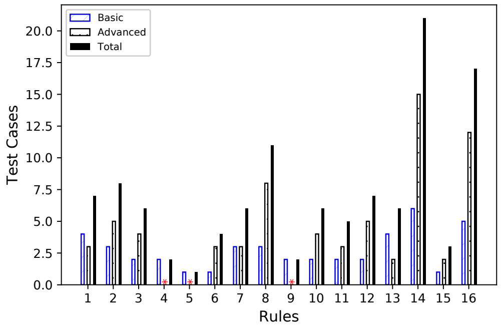


Figure 8: Test cases per Rule in CryptoApi-Bench (as of April 2019).

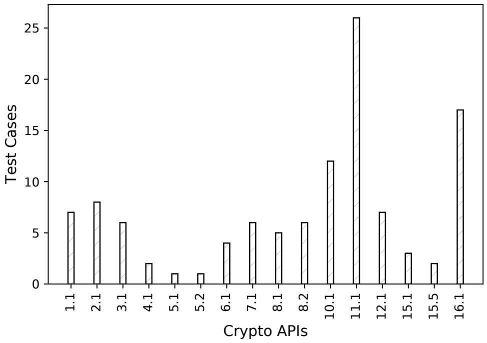


Figure 9: Test cases per API in CryptoApi-Bench (as of April 2019). A test case can cover one ore more APIs (e.g., test cases for Rule 15). APIs corresponding to the labels can be found in Tables 13, 11, and 12.

Table 8: Lines of code (LoC) of 30 Apache root-subprojects with their dependencies and 30 Android applications.

No.	Apache Project	LoC	Android Applications	LoC
1	hive	471k	Square_Point_of_Sale	1,453k
2	meecrowave-runner	395k	Alipay	1,400k
3	fop	185k	Amazon_Kindle	1,256k
4	spark	155k	Perfect365_Makeover	1,231k
5	hadoop	151k	Manga_Reader	1,032k
6	kylin	151k	Free_Bitcoin_Spinner	872k
7	tomee	118k	AICoin	790k
8	jackrabbit-oak	102k	LINE_WEBTOON	749k
9	airavata	89k	Sephora_Shop_Makeup	726k
10	nifi	76k	Audiobooks_from_Audible	711k
11	qpid-jms-amqp-0-x	66k	Mint_Budget_Bills_Finance	673k
12	juddi	63k	Money_Lover	627k
13	wss4j	42k	Ulta_Beauty	570k
14	santuario-java	41k	Daily_Bible_Journey	449k
15	plugin-yarn	35k	iqboxyinc	429k
16	embeddedwebserver	34k	Facebook_Pages_Manager	424k
17	abdera	31k	Dictionary_Merriam_Webster	398k
18	cloudstack	28k	Tiny_Scanner	371k
19	directory-server	23k	misa.sothuchi	365k
20	manifoldcf	19k	receipts	353k
21	tika	17k	Auto_Makeup	338k
22	wicket	16k	JW_Library	320k
23	taverna-workbench	10k	Card_Maker_for_Pokemon	296k
24	shindig	10k	Cartoon_Avatar_Maker	277k
25	activemq-artemis	7k	Clairol_MyShade	227k
26	deltaspike	6k	UPS_Mobile	220k
27	knox	6k	ADP_Mobile_Solutions	203k
28	shiro	2k	ebook_Renta	105k
29	meecrowave-core	1k	jits.mobile.aya	3k
30	geronimo-gshell	1k	zhangdan	0.4k

Table 9: The number of alerts in Apache (total 94 root-subprojects) and Android applications (6,181). For Rules 1, 2, 3, 8, 10, 12, each constant/predictable value of an array/collection is considered as an individual violation.

Rules	Apache		Android	
	# of Root-subprojects	# of Alerts Per Rule	# of Applications	#of Alerts Per Rule
(1,2) Predictable Keys	37 (39.36%)	264	1,617 (26.16%)	12,457
(3) Hardcoded Store Password	29 (30.85%)	148	218 (3.52%)	458
(4) Dummy Hostname Verifier	8 (8.51%)	12	800 (12.94%)	1,155
(5) Dummy Cert. Validation	11 (11.70%)	30	1,564 (25.30%)	3,856
(6) Used Improper Socket	4 (4.25%)	4	210 (3.39%)	271
(7) Used HTTP	24 (29.62%)	222	2,486 (40.22%)	8,321
(8) Predictable Seeds	0 (0%)	0	80 (1.29%)	544
(9) Untrusted PRNG	33 (35.10%)	142	5,194 (84.03%)	36,223
(10) Static Salts	21 (22.34%)	112	199 (3.21%)	1,757
(11) ECB mode for Symm. Crypto	16 (17.02 %)	41	882 (14.26%)	1,780
(12) Static IVs	4 (4.25 %)	41	913 (14.77%)	12,089
(13) <1000 PBE Iterations	25 (26.59 %)	43	151 (2.44%)	312
(14) Broken Symm. Crypto Algorithms	29 (30.85 %)	86	701 (11.34%)	1,742
(15) Insecure Asymm. Crypto	9 (10.98 %)	12	108 (1.74%)	111
(16) Broken Hash	42 (44.68 %)	138	5,272 (85.29%)	49,769

Table 10: Benchmark comparison of CrySL, Coverity, SpotBugs, and CryptoGuard on all 16 rules with CryptoApi-Bench's 112 test cases (as of April 2019). There are 16 secure API use cases (13 in basic and 3 in advanced), which a tool should not raise any alerts on. CryptoGuard successfully passed these 16 test cases. GTP stands for ground truth positive, which is the number of positives in the benchmark. CryptoGuard has 11 false negatives, which we reported in Section 6 and discussed in Section 7.

No.	Rules	GTP	CrySL		Coverity		SpotBugs		CryptoGuard	
			TP	FP	TP	FP	TP	FP	TP	FP
1	Predictable Cryptographic Key	5	0	4	3	0	2	0	5	0
2	Predictable Password for PBE	6	0	2	5	0	3	0	6	0
3	Predictable Password for KeyStore	5	0	5	3	0	2	0	5	0
4	Dummy Hostname Verifier	1	–	–	1	0	1	0	1	0
5	Dummy Cert. Validation	1	–	–	1	0	1	0	1	0
6	Used Improper Socket	4	–	–	4	0	–	–	4	0
7	Use of HTTP	4	–	–	–	–	–	–	4	0
8	Predictable Seed	10	–	–	1	0	–	–	5	0
9	Untrusted PRNG	1	–	–	–	–	1	0	1	0
10	Static Salt	5	5	1	–	–	–	–	3	0
11	ECB in Symm. Crypto	4	2	1	1	0	1	1	4	0
12	Static IV	6	0	6	–	–	6	0	4	0
13	<1000 PBE Iteration	5	2	1	–	–	–	–	4	0
14	Broken Symm. Crypto	20	10	5	4	0	5	5	20	0
15	Insecure Asymm. Crypto	3	2	1	–	–	0	1	2	0
16	Broken Hash	16	8	4	4	0	4	4	16	0
	Total	**96**	**29**	**30**	**27**	**0**	**26**	**11**	**85**	**0**

Table 11: Rules that use intra-procedural backward program slicing to slice implemented methods of standard Java APIs and their corresponding slicing criteria.

No.	Method to Slice	Rule	Criterion
4.1	`javax.net.ssl.HostnameVerifier: boolean verify(String,SSLSession)`	4	`return`
5.1	`void checkServerTrusted(X509Certificate[],String)`	5	`checkValidity()`
5.2	`void checkServerTrusted(X509Certificate[],String)`	5	`throw`
5.3	`java.security.cert.X509Certificate[] getAcceptedIssuers()`	5	`return`

Table 12: Java APIs used as slicing criteria in our intra-procedural forward program slicing and their corresponding security rules.

No.	Slicing Criterion for Intra Procedural Forward Program Slicing	Rule	Semantic
6.1	`javax.net.ssl.SSLSocketFactory: SocketFactory getDefault()`	6	Create SocketFactory
6.2	`javax.net.ssl.SSLContext: SSLSocketFactory getSocketFactory()`	6	Create SocketFactory
15.1	`java.security.KeyPairGenerator: KeyPairGenerator getInstance(java.lang.String)`	15	Create KeyPairGenerator
15.2	`java.security.KeyPairGenerator: KeyPairGenerator getInstance(String,String)>`	15	Create KeyPairGenerator
15.3	`java.security.KeyPairGenerator: KeyPairGenerator getInstance(String,Provider)`	15	Create KeyPairGenerator

Table 13: Java APIs used as slicing criteria in our inter-procedural backward slicing and their corresponding security rules. Boldface indicates the parameter of interest.

No.	API	Rule	Semantic
1.1	javax.crypto.spec.SecretKeySpec: void <init>(**byte[]**,String)	1	Set key
1.2	javax.crypto.spec.SecretKeySpec: void <init>(**byte[]**,int,int,String)	1	Set key
2.1	javax.crypto.spec.PBEKeySpec: void <init>(**char[]**)	2	Set password
2.2	javax.crypto.spec.PBEKeySpec: void <init>(**char[]**,byte[],int,int)	2	Set password
2.3	javax.crypto.spec.PBEKeySpec: void <init>(**char[]**,byte[],int)	2	Set password
3.1	java.security.KeyStore: void load(InputStream,**char[]**)	3	Set password
3.2	java.security.KeyStore: void store(OutputStream,**char[]**)	3	Set password
3.3	java.security.KeyStore: void setKeyEntry(String,Key,**char[]**,Certificate[])	3	Set password
3.4	java.security.KeyStore: Key getKey(String,**char[]**)	3	Set password
7.1	java.net.URL: void <init>(**String**)	7	Set URL
7.2	java.net.URL: void <init>(**String**,String,String)	7	Set URL
7.3	java.net.URL: void <init>(**String**,String,int,String)	7	Set URL
7.4	okhttp3.Request$Builder: Request$Builder url(**String**)	7	Set URL
7.5	retrofit2.Retrofit$Builder: Retrofit$Builder baseUrl(**String**)	7	Set URL
8.1	java.security.SecureRandom: void <init>(**byte[]**)	8	Set seed
8.2	java.security.SecureRandom: void setSeed(**byte[]**)	8	Set seed
8.3	java.security.SecureRandom: void setSeed(**long**)	8	Set seed
10.1	javax.crypto.spec.PBEParameterSpec: void <init>(**byte[]**,int)	10	Set salt
10.2	javax.crypto.spec.PBEParameterSpec: void <init>(**byte[]**,int,AlgorithmParameterSpec)	10	Set salt
10.3	javax.crypto.spec.PBEKeySpec: void <init>(char[],**byte[]**,int,int)	10	Set salt
10.4	javax.crypto.spec.PBEKeySpec: void <init>(char[],**byte[]**,int)	10	Set salt
11.1	javax.crypto.Cipher: Cipher getInstance(**String**)	11, 14	Select cipher
11.2	javax.crypto.Cipher: Cipher getInstance(**String**, String)	11, 14	Select cipher
11.3	javax.crypto.Cipher: Cipher getInstance(**String**, Provider)	11, 14	Select cipher
12.1	javax.crypto.spec.IvParameterSpec: void <init>(**byte[]**)	12	Set IV
12.2	javax.crypto.spec.IvParameterSpec: void <init>(**byte[]**,int,int)	12	Set IV
13.1	javax.crypto.spec.PBEParameterSpec: void <init>(byte[],**int**)	13	Set iterations
13.2	javax.crypto.spec.PBEParameterSpec: void <init>(byte[],**int**,AlgorithmParameterSpec)	13	Set iterations
13.3	javax.crypto.spec.PBEKeySpec: void <init>(char[],byte[],**int**,int)	13	Set iterations
13.4	javax.crypto.spec.PBEKeySpec: void <init>(char[],byte[],**int**)	13	Set iterations
15.1	java.security.KeyPairGenerator: KeyPairGenerator getInstance(**String**)	15	Select generator
15.2	java.security.KeyPairGenerator: KeyPairGenerator getInstance(**String**,String)>	15	Select generator
15.3	java.security.KeyPairGenerator: KeyPairGenerator getInstance(**String**,Provider)	15	Select generator
15.4	java.security.KeyPairGenerator: void initialize(**int**)	15	Set key size
15.5	java.security.KeyPairGenerator: void initialize(**int**,java.security.SecureRandom)	15	Set key size
15.6	java.security.KeyPairGenerator: void initialize(**AlgorithmParameterSpec**)	15	Set key size
15.7	java.security.KeyPairGenerator: void initialize(**AlgorithmParameterSpec**,SecureRandom)	15	Set key size
16.1	java.security.MessageDigest: MessageDigest getInstance(**String**)	16	Select hash
16.2	java.security.MessageDigest: MessageDigest getInstance(**String**, String)	16	Select hash
16.3	java.security.MessageDigest: MessageDigest getInstance(**String**, Provider)	16	Select hash

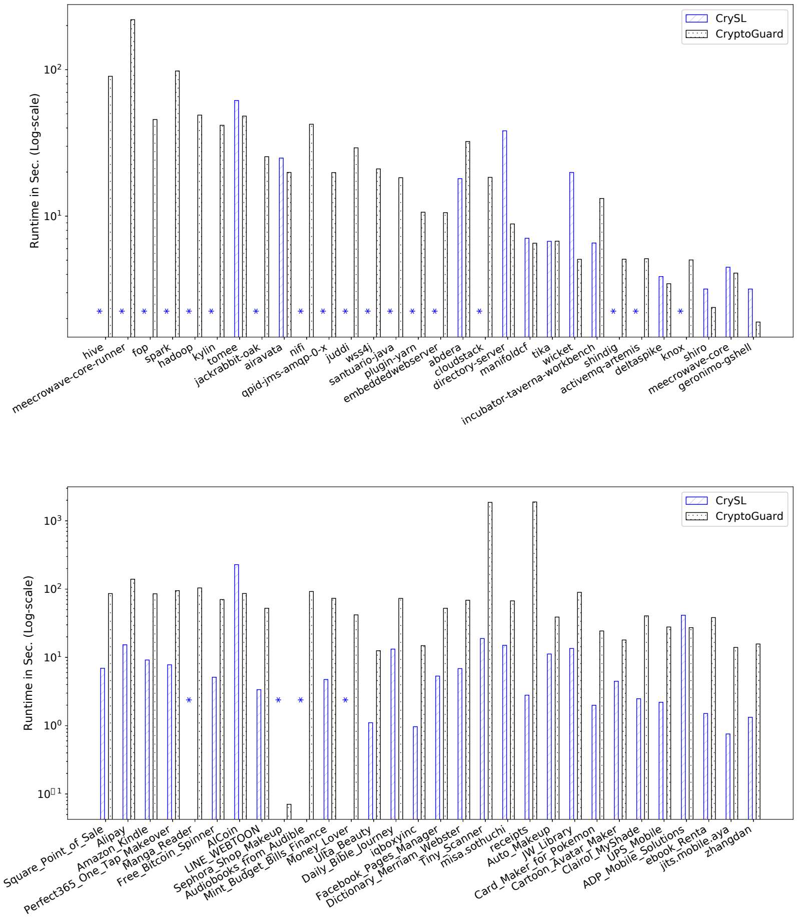


Figure 10: Runtime comparison in log scale of CryptoGuard and CrySL on 30 Apache root-subprojects in (a) and 30 Android applications in (b), ordered by decreasing lines of code (LoC). * indicates crash. CryptoGuard successfully completed all tasks. The LoCs are shown in Table 8.