CRYLOGGER:
Detecting Crypto Misuses Dynamically

Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, Simha Sethumadhavan
{piccolboni, giuseppe, luca, simha}@cs.columbia.edu
Columbia University, New York, NY, USA

Abstract—**Cryptographic (crypto) algorithms are the essential ingredients of all secure systems: crypto hash functions and encryption algorithms, for example, can guarantee properties such as integrity and confidentiality. Developers, however, can misuse the application programming interfaces (API) of such algorithms by using constant keys and weak passwords. This paper presents CRYLOGGER, the first open-source tool to detect crypto misuses dynamically. CRYLOGGER logs the parameters that are passed to the crypto APIs during the execution and checks their legitimacy offline by using a list of crypto rules. We compare CRYLOGGER with CryptoGuard, one of the most effective static tools to detect crypto misuses. We show that our tool complements the results of CryptoGuard, making the case for combining static and dynamic approaches. We analyze 1780 popular Android apps downloaded from the Google Play Store to show that CRYLOGGER can detect crypto misuses on thousands of apps dynamically and automatically. We reverse-engineer 28 Android apps and confirm the issues flagged by CRYLOGGER. We also disclose the most critical vulnerabilities to app developers and collect their feedback.**

Index Terms—**Android, Cryptography, Security, Misuses.**

Repository—**https://github.com/lucapiccolboni/crylogger [1]**

I. INTRODUCTION

Cryptographic (crypto) algorithms are the key ingredients of all secure systems [2]. Crypto algorithms can guarantee that the communication between two entities satisfies strong properties such as data confidentiality (with encryption) and data integrity (with hashing). While the crypto theory can formally guarantee that those properties are satisfied, in practice poor implementations of the crypto algorithms [3] can jeopardize communication security. For instance, Brumley et al. [4] showed how to obtain the entire private key of an encryption algorithm, which is based on elliptic curves, by exploiting an arithmetic bug in OpenSSL. Unfortunately, ensuring that the actual implementation of the crypto algorithms is correct as well as secure is not sufficient. The crypto algorithms can be, in fact, *misused*. Egele et al. [5] showed that 88% of the Android apps they downloaded from the Google Play Store had at least one crypto misuse. For example, thousands of apps used hard-coded keys for encryption instead of truly-random keys, thus compromising data confidentiality. Similarly, Rahaman et al. [6] showed that 86% of the Android apps they analyzed used broken hash functions, e.g., SHA1, for which collisions can be produced [7], threatening data integrity.

Recently, researchers analyzed the causes of crypto misuses in many contexts. Fischer et al. [8] found that many Android apps included snippets of code taken from Stack Overflow and 98% of these snippets included several crypto issues. Nadi et al. [9] claimed that the complexity of application programming interfaces (APIs) is the main origin of crypto misuses in Java. Developers have to take low-level decisions, e.g., select the type of padding of an encryption algorithm, instead of focusing on high-level tasks. Acar et al. [10] compared 5 crypto libraries for Python and argued that poor documentation, lack of code examples and bad choices of default values in the APIs are the main causes of crypto misuses. Muslukhov et al. [11] showed that 90% of the misuses in Android originated from third-party libraries, a result that was later confirmed by Rahaman et al. [6].

At the same time, researchers started to implement tools to automatically detect crypto misuses, e.g., [5], [6]. The idea is to define a set of *crypto rules* and check if an application respects them by verifying the parameters passed to the crypto APIs. The rules usually come from (i) papers that show the vulnerabilities caused by some crypto algorithms or their misconfigurations, e.g., [12], and (ii) organizations and agencies, e.g., NIST and IETF, that define crypto-related standards to prevent attacks. Examples of crypto rules are setting (i) a minimum key size for encryption, e.g., 2048 bits for RSA [13] or (ii) a minimum number of iterations for key derivation, e.g., 1000 for PKCS#5 [14].

To check the crypto rules, researchers developed static as well as dynamic solutions. Static approaches, e.g., CrySL [15], CryptoLint [5], CryptoGuard [6], MalloDroid [16], CogniCrypt [17] and CMA [18], examine the code with program slicing [19] to check the values of the parameters that are passed to the APIs of the crypto algorithms. Static analysis has the benefit that the code is analyzed entirely without the need of executing it. Also, it can scale up to a large number of applications. Static analysis produces, however, false positives, i.e., alarms can be raised on legit calls to crypto algorithms. Some static approaches, e.g. CryptoGuard, suffer also from false negatives, i.e., some misuses escape detection, because the exploration is pruned prematurely to improve scalability on complex programs. It is also possible that static analysis misses some crypto misuses in the code that is loaded dynamically [20]. Most of the recent research efforts focused on static approaches [21], while little has been done to bring dynamic approaches to the same level of completeness and effectiveness. Few approaches have been proposed towards this direction, e.g., SMV-Hunter [22], AndroSSL [23], K-Hunt [24], and iCryptoTracer [25]. Dynamic approaches are usually more difficult to use since they require to trigger the crypto APIs at

runtime to expose the misuses, but they do not usually produce false positives. Unfortunately, these dynamic approaches do not support as many crypto rules as the current static approaches. SMV-Hunter and AndroSSL consider only rules for SSL/TLS, and K-Hunt focuses on crypto keys. iCryptoTracer attacks the hard problem of detecting misuses in iOS apps. iCryptoTracer supports few rules as it needs to rely on API hooking techniques.

A. Contributions

In this paper, we present *CRYLOGGER*, an open-source tool to detect crypto misuses dynamically. It consists of (i) a *logger* that monitors the APIs of the crypto algorithms and stores the values of the relevant parameters in a log file, and (ii) a *checker* that analyzes the log file and reports the crypto rules that have been violated. The key insights of this work are: (1) we log the relevant parameters of the crypto API calls by instrumenting few classes that are used by a large number of applications; (2) we log the values of the parameters of the crypto APIs at runtime, while we check the rules offline to reduce the impact on the applications performance; (3) we show that, for most Android apps, the calls to the crypto APIs can be easily triggered at runtime, and thus a dynamic approach can be effective in detecting misuses even if the code of the applications has not been explored entirely; (4) we show that, for Android apps, it is sufficient to execute an application for a relatively short amount of time to find many of the crypto misuses that are reported by the current static tools.

We envision two main uses of *CRYLOGGER*. (1) Developers can use it to find crypto misuses in their applications as well as in the third-party libraries they include. *CRYLOGGER* can exploit the input sequences that are defined by developers for verification purposes to detect the misuses. *CRYLOGGER* can be used alongside static tools as it complements their analysis (Section VIII). Using *CRYLOGGER* also helps to reduce the false positives reported by static tools. (2) *CRYLOGGER* can be used to check the apps submitted to app stores, e.g., the Google Play Store. Using a dynamic tool on a large number of apps is hard, but *CRYLOGGER* can refine the misuses identified with static analysis because, typically, many of them are false positives that cannot be discarded manually on such a large number of apps.

We make the following contributions:

1. we describe *CRYLOGGER*, the first open-source tool to detect crypto misuses dynamically; the tool is available at: **https://github.com/lucapiccolboni/crylogger** [1];

2. we implement *CRYLOGGER* for Android and Java apps; we support 26 crypto rules, and we decouple the logging and the checking mechanisms so that new rules can be easily added and checked with *CRYLOGGER*;

3. we compare *CRYLOGGER* against CryptoGuard [6], one of the most effective static tools to detect misuses: we use 150 popular Android apps of the Google Play Store for the comparison; we show that *CRYLOGGER* reports misuses that CryptoGuard misses, but we show that the opposite is also possible, thus making the case for combining static and dynamic approaches;
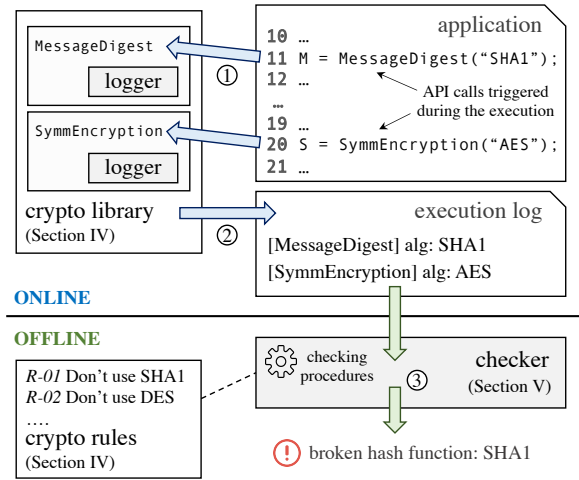

Fig. 1. Overview of *CRYLOGGER*. ① We run the application with an instrumented crypto library. ② We generate a log containing the parameters of the crypto API calls. ③ We check the crypto rules and report all the violations.

4. we reverse engineer 150 Android apps to evaluate the false positives of CryptoGuard; we show that for some rules many false positives are reported due to insecure, but untriggerable, code included in the apps;

5. we compare *CRYLOGGER* against CryptoGuard by using the CryptoAPI-Bench [26], a set of Java programs that include misuses; we also extend the CryptoAPI-Bench with tests cases suited for dynamic tools;

6. we use *CRYLOGGER* to analyze 1780 Android apps downloaded from the Google Play Store (the dataset was collected between September and October 2019). These are the most popular apps of 33 different categories. We confirm the results previously reported with static tools [5], [6] and report new misuses;

7. we disclose the vulnerabilities we found to 306 app and library developers and we report the feedback we received from the 10 who replied; we manually reverse-engineer 28 apps to determine if the vulnerabilities reported by *CRYLOGGER* can actually be exploited.

II. OVERVIEW

Fig. 1 provides an overview of *CRYLOGGER*. It consists of:

1. *logger*: the logger extends a crypto library, for example the Java crypto library, to trace the API calls to crypto algorithms; for each of these calls, it logs the relevant parameters that must be used to check the crypto rules; for example, in Fig. 1, the logger saves the names of the algorithms chosen by the application for message digest (SHA1) and symmetric encryption (AES);

2. *checker*: the checker analyzes the log offline, after the application has been executed, and it produces a list of all the crypto rules violated by the application. To check the rules it uses a set of checking procedures, each of which covers many crypto rules; for instance, in Fig. 1, the checker finds that the application uses the broken algorithm SHA1 as message digest algorithm.

We decouple logging from checking for 4 main reasons: (1) the parameters of interest of the crypto library are more stable, i.e., it is unlikely that new parameters are added; for example, the main parameters of an algorithm for key derivation are the salt, the password and the number of iterations, (2) the crypto rules are likely to change: for example, new rules can be added when new vulnerabilities are found as well as current rules may need to be updated (for example the minimum key size of RSA), (3) crypto rules are context-dependent: some rules may be not relevant for certain applications or contexts, and (4) checking rules offline does not affect the application performance, which is important, for instance, when the application response is critical (Android).

Similarly to most of the current static solutions, we developed *CRYLOGGER* primarily to check Java and Android applications. Our ideas, however, could be adapted to other contexts. In the next sections, we describe our tool in more detail. In Section III, we discuss the related work. In Section IV, we describe a generic crypto library that we use to define the crypto rules and the API parameters that must be logged. In Section V, we explain how *CRYLOGGER* checks the rules. In Section VI, we present an implementation of *CRYLOGGER* for Java and Android [1], by explaining which APIs we instrumented and how we analyzed a large number of Android apps. In Section VII, we describe the dataset of apps we use for the evaluation. In Section VIII, we perform a comparison of *CRYLOGGER* against CryptoGuard by using 150 Android apps and the CryptoAPI-Bench [26]. In Section IX, we present an analysis of 1780 apps from the Google Play Store. We also report the feedback received for disclosing the vulnerabilities and our reverse-engineering analysis of the vulnerabilities found in 28 apps. In Section X, we discuss the limitations of our approach before concluding in Section XI.

III. RELATED WORK

A. Detection of Crypto Misuses

Several tools exist to detect crypto misuses. Most of them are based on *static analysis*, e.g., CryptoLint [5], CryptoGuard [6], CrySL [15], MalloDroid [16], CogniCrypt [17] and CMA [18]. These tools differ in the crypto rules that they support and in the slicing algorithms [19] that they adopt for analysis. Among them, CryptoGuard covers the highest number of crypto rules. As discussed in [27], the main problem with static analysis is the high number of false positives, which requires the users to manually examine the results and determine the true positives. Recent studies [6], [26] showed that CryptoGuard is one of the most effective tools in reducing the false positives, thanks to rule-specific algorithms that refine the results of the static analysis. We show, however, that CryptoGuard still produces many false positives in practice by reporting crypto misuses that can never be triggered at runtime (Section VIII). To achieve scalability on complex apps, some tools "cut off" some branches of the static explorations, e.g., CryptoGuard clips orthogonal explorations. This causes false negatives in addition to false positives. False negatives are also caused by code that is loaded at runtime [20].

Other tools identify crypto misuses by employing *dynamic analysis*. SMV-Hunter [22] and AndroSSL [23], for example,

detect misuses of the SSL/TLS protocol. K-Hunt [24] detects badly-generated keys, insecurely-negotiated keys and recoverable keys by analyzing execution traces of Java programs. iCryptoTracer [25] detects misuses in iOS apps, which is a complex task that must be implemented through API hooking techniques. To the best of our knowledge, there are no approaches that are as exhaustive and effective as static approaches and cover many crypto tasks, e.g., encryption, authentication, and SSL/TLS. This motivated us to develop *CRYLOGGER*, a tool that supports more crypto rules than current static approaches and covers several crypto tasks. The main disadvantage of all dynamic tools is the possibility of missing vulnerabilities due to poor coverage [28]. Some misuses can remain undetected if the application are not explored thoroughly. We show, however, that *CRYLOGGER* is capable of finding most of the crypto misuses that CryptoGuard reports even if the apps are not fully explored (Section VIII).

B. Other Related Research

The problem of crypto misuses has been studied from many different perspectives. Fischer et al. [8] analyzed security-related code snippets taken from Stack Overflow. They found that $>15\%$ of the apps of the Google Play Store contained snippets of code directly taken from Stack Overflow and $\sim98\%$ of these had at least one misuse. In a more recent work [29], they showed that nudges [30] significantly helped developers in making better decisions when crypto tasks need to be implemented. Nadi et al. [9] showed that the main cause of misuses lies in the complexity of the APIs rather than in the lack of security knowledge in developers. Acar et al. [10] showed that poor documentation, lack of code examples and bad choices of default values in the crypto APIs contribute to many of the crypto misuses. Green et al. [31] made the case for developing security-friendly APIs that help developers to avoid common mistakes. Many recent works, e.g., [6], [11] showed that third-party libraries cause most of the crypto misuses in Android, up to 90% in some cases. To simplify the work for developers, several approaches display security tips or warnings in an integrated development environment. For example, CogniCrypt [17] generates code snippets in Eclipse, which can be used when crypto tasks need to be implemented. Similarly, FixDroid [32] provides suggestions to developers on how to fix crypto-related issues in Android Studio. To remove the burden of fixing misuses from developers, some approaches repair problematic code snippets automatically [33]–[36].

C. Testing Android Apps

Analyzing Android apps dynamically and automatically is considered a hard problem [37], [38]. The common solution to verify the apps correctness is Monkey[1]. Monkey generates pseudo-random events that interact with the GUI of the emulator or the real device. Monkey often obtains low code coverage because the events are completely random [39], but it is quite efficient in terms of execution time. Other approaches try to exploit some information about the app to improve coverage. For example, SmartDroid [28] exploits a combination of static and dynamic techniques to trigger the APIs of interest. DroidBot [40]

---

[1]Monkey UI Exerciser: https://developer.android.com/studio/test/monkey.

is a test generator based on control-flow graphs that can be extended to support custom exploration strategies. Dynodroid [41] monitors the app to guide the generation of the next input event. These approaches have a significant overhead on the execution of the app because to generate useful events they require either to (i) rely on static analysis of the code [28] or (ii) create a model at runtime that helps the exploration [40]. In *CRYLOGGER*, we use Monkey as it is lightweight and common among developers.

IV. CRYPTO LIBRARY AND CRYPTO RULES

A typical crypto library (e.g., Java Cryptography Architecture) includes 7 classes of tasks: (1) message digest, (2) symmetric encryption, (3) asymmetric encryption, (4) key derivation/generation, (5) random number generation, (6) key storage, and (7) SSL/TLS and certificates. Fig. 2 shows the parameters used by *CRYLOGGER*. The parameters of Fig. 2 are logged and used to check the rules. We do not claim that this library is complete. We include the classes that are used by current static tools and those that have a corresponding implementation in Java and Android. These are the classes with the highest number of misuses in Android and Java [5], [6], [16]. Extensions are possible, e.g., HKDF [42] can be added to the key derivation class.

(1) MessageDigest implements crypto hash functions [43]. These functions take as input an arbitrary amount of data and produce fixed-length hash values, called digests. They are used to check data integrity. For this class, the most important parameter is the algorithm (alg) that is used as hash function, for example, SHA1, SHA256. Different libraries support different algorithms.

(2) SymmEncryption contains block ciphers that are used for symmetric encryption [43]. A block cipher takes as input a block of data with fixed size (e.g., 128 bits) and a key (whose size is defined by the algorithm) and it generates the corresponding output block (encrypted or decrypted). A decrypted block of data is called plaintext, while an encrypted block is the ciphertext. In addition to the algorithm (alg), e.g., AES, used for encryption and decryption, we log the key (key) and some other parameters. Block ciphers work on a fixed-size data block. Therefore, to work on multiple blocks of data (#blocks) they need to support some operation modes (mode). For example, by using electronic code book (ECB) each block is decrypted / encrypted independently from the other blocks. With cipher block chaining (CBC), each block of plaintext is xored with the previous block of ciphertext. The initialization vector (IV) is a parameter (iv) that defines the block that is xored with the very first block. Other common operation modes are cipher feedback (CFB), output feedback (OFB), and Galois/counter (GCM). Another important parameter is the padding algorithm (pad), which is the algorithm used to fill the last block of data if the input is not a multiple of the block size. Example of padding algorithms are ZEROPADDING, where the last block is filled with zeros, PKCS#5 [14] and PKCS#7 [44].

(3) AsymmEncryption implements algorithms for public-key cryptography [2]. These algorithms use a key pair (key): a public key and a private key. They can be used for (i) encryption and decryption as well as (ii) signature and verification. For (i), the message is encrypted with the public key of the receiver. It can
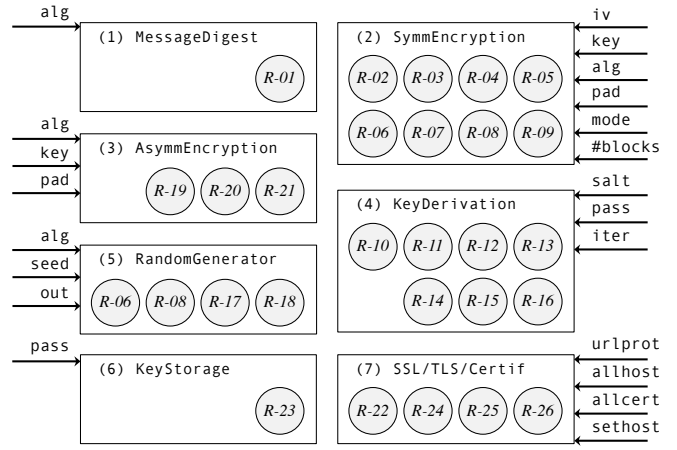

Fig. 2. Classes of a typical crypto library with their parameters (arrows entering in the class). For each class we report the crypto rules of TABLE I that need parameters of that specific class.

be then decrypted only with the private key of the receiver. For (ii), a message is signed with the private key of the sender and verified with the corresponding public key. The parameters of this class are the algorithm (alg) used for encryption, e.g., RSA, elliptic curves (EC) or digital signature algorithm (DSA), and the padding (pad), e.g., NOPADDING, PKCS1-v1.5 and PSS [45].

(4) KeyDerivation implements algorithms to derive crypto keys [43]. A key derivation function takes as input a password or a passphrase (pass) and generates a key by using a salt (salt), i.e., a random value, and by applying a function, e.g., hashing, for a fixed number of iterations (iter). The larger is the number of iterations the harder is to implement brute-force attacks [14].

(5) RandomGenerator implements algorithms for generating random numbers. The relevant parameters are the algorithm (alg) used for generating the numbers, the bytes of the generated number (out), and the seed (seed) for the generation. In this paper we assume that there are only two categories of algorithms: Secure and NotSecure. The parameter alg is Secure if it generates numbers suited for crypto, otherwise it is NotSecure.

(6) KeyStorage implements algorithms to store crypto keys, certificates and other sensitive content. Usually, it takes as input a password or a passphrase (pass) to store contents securely.

(7) SSL/TLS/Certif is a class including multiple functions for SSL/TLS and certificates: (1) connections that can be HTTP or HTTPS (urlprot), (2) host name verification that can accept all the host names or not (allhost), (3) certificate validation, which can trust all certificates or not (allcert), and (4) host name verification for SSL/TLS connections (sethost) [16].

A. Threat Model and Crypto Rules

TABLE I reports the rules that are supported by *CRYLOGGER*. We collected them from (i) papers ad (ii) documents published by NIST as well as IETF. Fig. 2 shows how the rules relate to the crypto classes. Some rules use parameters from more than one class (e.g., *R-06* and *R-08*). We use the same threat model

ID	Rule Description	Ref.	ID	Rule Description	Ref.
R-01	Don't use broken hash functions (SHA1, MD2, MD5, ..)	[8]	R-14 †	Don't use a weak password (score < 3)	[47]
R-02	Don't use broken encryption alg. (RC2, DES, IDEA ..)	[8]	R-15 †	Don't use a NIST-black-listed password	[48]
R-03	Don't use the operation mode ECB with > 1 data block	[5]	R-16	Don't reuse a password multiple times	[48]
R-04 †	Don't use the operation mode CBC (client/server scenarios)	[12]	R-17	Don't use a static (= constant) seed for PRNG	[49]
R-05	Don't use a static (= constant) key for encryption	[5]	R-18	Don't use an unsafe PRNG (java.util.Random)	[49]
R-06 †	Don't use a "badly-derived" key for encryption	[5]	R-19	Don't use a short key (< 2048 bits) for RSA	[13]
R-07	Don't use a static (= constant) initialization vector (IV)	[5]	R-20 †	Don't use the textbook (raw) algorithm for RSA	[50]
R-08 †	Don't use a "badly-derived" initialization vector (IV)	[5]	R-21 †	Don't use the padding PKCS1-v1.5 for RSA	[51]
R-09 †	Don't reuse the initialization vector (IV) and key pairs	[46]	R-22	Don't use HTTP URL connections (use HTTPS)	[16]
R-10	Don't use a static (= constant) salt for key derivation	[5]	R-23	Don't use a static (= constant) password for store	[48]
R-11 †	Don't use a short salt (< 64 bits) for key derivation	[14]	R-24	Don't verify host names in SSL in trivial ways	[16]
R-12 †	Don't use the same salt for different purposes	[46]	R-25	Don't verify certificates in SSL in trivial ways	[16]
R-13	Don't use < 1000 iterations for key derivation	[14]	R-26	Don't manually change the hostname verifier	[16]

TABLE I
Crypto rules that are considered in this paper. The symbol † indicates the rules that are not covered by other approaches (we used [6] as reference).

of the current static tools. We briefly describe the crypto rules below. The severity of most of these rules is discussed in [6].

R-01 does not let applications use broken hash functions, e.g., those for which we can generate collisions, like SHA1 [7]. *R-02* forbids the use of some broken algorithms for symmetric encryption, for example, Blowfish, DES, etc. *R-03* and *R-04* do not allow applications to use the operation modes ECB and CBC, respectively. ECB is well known to be vulnerable since identical blocks of plaintext are encrypted to identical blocks of ciphertext. This breaks the property of semantic security [52]. CBC is instead vulnerable to padding oracle attacks in client-server scenarios [12]. *R-05* and *R-06* put restrictions on how to generate keys. *R-05* requires that the keys for symmetric encryption are randomly generated by the application instead of being hard-coded in the app as constants. *R-06* requires the keys to have enough randomness, i.e., they should be generated by using a random generator that is considered secure for crypto. *R-07* and *R-08* are similar to *R-05* and *R-06*, but they consider the IVs that are used in symmetric encryption instead of the keys. The IVs, in fact, should always be random and non-constant to strengthen data confidentiality when they are paired with some operation modes, e.g., GCM. *R-09* requires that the same pair (key, IV) is never reused to encrypt different messages. Reusing the same pair (key, IV) makes the encryption predictable. *R-10* is the same as *R-05*: it is, however, applied to the salt used in key generation instead of the keys used in symmetric encryption. *R-11* requires the salt to be large enough (≥ 64 bits) to protect the password used for key generation. *R-12* prohibits the reuse of the same salt because it defeats the purpose of adding randomness to the corresponding password. *R-13* requires to use a sufficient number of iterations to generate the key so that brute-force attacks become infeasible. *R-14* and *R-15* require to use a password that has not been black-listed and that is "hard" enough for password-based encryption, respectively. *R-16* forbids using the same password multiple times (e.g., constant passwords). *R-17* requires to use a random value as seed instead of a constant value for pseudo-random number generation (PRNG). Using a constant seed defeats the purpose of generating random number as the sequence of numbers that is generated becomes predictable. *R-18* does not allow applications

to use PRNGs that are not approved for crypto operations, for example java.util.Random [6]. *R-19*, *R-20* and *R-21* forbid some configurations of the RSA algorithm. In particular, the key should be ≥ 2048 bits and a padding algorithm different from NOPADDING (*R-20*) and PKCS1-v1.5 (*R-21*) must be used for encryption / decryption. *R-22* forbids the use of HTTP and requires the use of the more secure alternative HTTPS. *R-23* forbids the use of static passwords for key storage. *R-24* and *R-25* require to properly verify host names and certificates. For example, accepting all host names or all certificates should not be allowed. *R-26* forbids to modify the standard host name verifier, which can lead to insecure communication over SSL/TLS.

V. CHECKING CRYPTO RULES DYNAMICALLY

We define four checking procedures to cover the crypto rules reported in TABLE I. Each checking procedure covers multiple rules, while each rule is verified by only one checking procedure. These checking procedures are shown graphically in Fig. 3 and explained in detail in the next sections. These procedures are generic: they can be applied to new crypto rules if needed.

A. Unacceptable Values

The checking procedure of Fig. 3 (a) extracts from the log all the values of a parameter or a combination of parameters and verifies that they can be used to configure the corresponding crypto class. All the values that are collected from the log are sent to a rule-specific function that says 'yes' if the values are allowed by the rule or 'no' otherwise. For *R-01*, for instance, we need to ensure that the parameter `alg` of `MessageDigest` never takes one of the following values: SHA1, MD2, MD5, etc. This is the most basic checking procedure and it is used to check the highest number of crypto rules. We describe how we check the crypto rules that fall under this type below. For each rule, we report which property must be satisfied by all the values that are collected for that rule.

```
R-01: MessageDigest.alg ∉ {'SHA1', ..}
R-02: SymmEncryption.alg ∉ {'DES', ..}
```

For rules *R-01* and *R-02* we simply check that broken algorithms are not used for message digest and encryption, respectively.
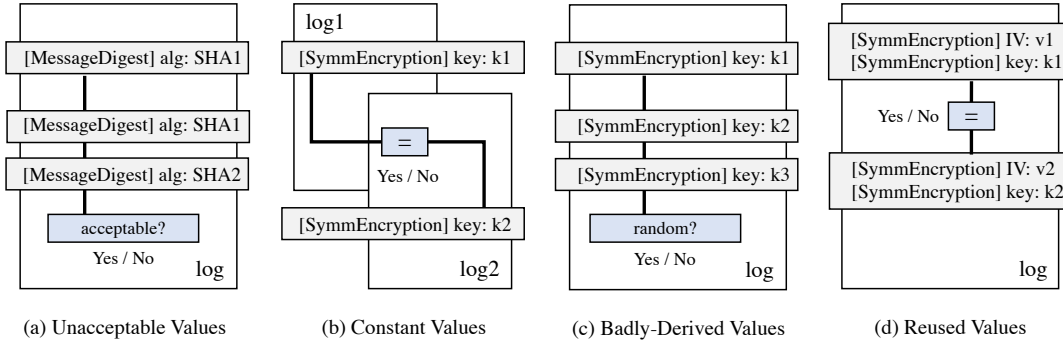
Fig. 3. We define four checking procedures to cover all the crypto rules of TABLE I. (a) We check if some unacceptable values are used to configure a parameter of a crypto class (e.g., SHA1 for rule *R-01*). (b) We check if a parameter is configured with constant values by verifying if the same values are found in two different executions of an application (e.g., same key for rule *R-05*). (c) We check if the values of a parameter of a crypto class has enough randomness (e.g., the keys for rule *R-06*). (d) We check if some values of a parameter are reused multiple times during the execution of an application (e.g., the pairs (key, IV) for *R-09*).

```
R-03: SymmEncryption.mode ≠ 'ECB' or
      SymmEncryption.#blocks = 1
R-04: SymmEncryption.mode ≠ 'CBC'
```

For rules *R-03* and *R-04*, we check that the operation modes ECB / CBC are not used. We accept the use of ECB for 1 data block.

```
R-11: KeyDerivation.salt ≥ 64 bits
R-13: KeyDerivation.iter ≥ 1000
```

For key derivation we check that the lengths of the salts in the log are always ≥ 64 bits and the number of iterations is ≥ 1000.

```
R-14: KeyDerivation.pass ∉ BadPass
R-15: score(KeyDerivation.pass) ≥ 3
```

For key derivation, we check if the password is broken (i.e., it belongs to BadPass[2]) or weak. To check if a password is weak we use zxcvbn [47] and consider it bad if it has a score < 3.

```
R-18: RandomGenerator.alg = 'Secure'
```

We check that the algorithm to generate random numbers is Secure, i.e., it should generate truly-random numbers. For example in Java, java.secure.SecureRandom must be used instead of java.util.Random, whose randomness is limited.

```
R-19: AsymmEncryption.alg ≠ 'RSA' or
      AsymmEncryption.key ≥ 2048 bits
R-20: AsymmEncryption.alg ≠ 'RSA' or
      AsymmEncryption.pad ≠ 'NOPADDING'
R-21: AsymmEncryption.alg ≠ 'RSA' or
      AsymmEncryption.pad ≠ 'PKCS1-v1.5'
```

These rules do not admit encryption keys that are < 2048 bits for RSA and require some padding algorithm different from NOPADDING and PKCS1-v1.5 for encryption/decryption [51].

```
R-22: SSL/TLS/Cert.urlprot ≠ 'HTTP'
```

We check that HTTP is never used as a connection protocol.

```
R-24: SSL/TLS/Cert.allhost = 'False'
R-25: SSL/TLS/Cert.allcert = 'False'
R-26: SSL/TLS/Cert.sethost not assigned
```

[2]We used a set of passwords from: https://github.com/cry/nbp.

For rules *R-24* and *R-25*, we check that apps do not naively verify host names and certificates (e.g., they do not verify the host name at all or they trust all certificates). For rule *R-26*, we check that the default host name verifier is not replaced to avoid host name verification, e.g., in Java by creating sockets[3].

B. Constant Values

The checking procedure of Fig. 3 (b) verifies if a parameter of a crypto class is constant or not. For instance, for rule *R-05* we need to ensure that applications do not use static encryption keys that are hard-coded in the app. Ideally, the keys should be generated with a proper random generator. To verify the rules in this category, we examine the logs of two executions of the same application and check that the values that are found in one of the execution log is not present in the other and vice versa. For example, for rule *R-05* we check the following:

```
R-05: { SymmEncryption.key }₁ ∩
      { SymmEncryption.key }₂ = ∅
```

where we used $\{\ \}_1$ to indicate the values collected in the first log and $\{\ \}_2$ the values collected in the second log. In a similar way, we check the rules *R-07*, *R-10*, *R-17*, and *R-23* with the values of SymmEncryption.iv, KeyDerivation.salt, RandomGenerator.seed, and KeyStorage.pass.

C. Badly-derived Values

The checking procedure reported in Fig. 3 (c) verifies if a value is truly random or not. For rule *R-06*, for example, we need to guarantee that the application uses encryption keys that have enough randomness. To verify the rules of this type, we collect all the values of the relevant parameter and we make the following three checks sequentially (box *random?* of Fig. 3 (c)):

1. if the value is obtained from RandomGenerator with alg = 'Secure', then we consider it a legit value;

2. if the value is obtained from RandomGenerator with alg ≠ 'Secure', then we consider it a bad value;

3. otherwise we apply the NIST tests for randomness [49] and if at least one test fails we consider it a bad value.

[3]Android SSL: https://developer.android.com/training/articles/security-ssl.

The first two checks try to determine the origin of the value, i.e., if it has been generated by `RandomGenerator` (parameter `out`). If the origin cannot be determined, e.g., the value is generated in some other ways by the application, then we use the NIST tests. For each NIST test we have three possible outcomes: (i) failure, (ii) success, or (iii) skipped because there are not enough bits to apply the specific test. We consider that an app violates a rule if at least one NIST test fails. This policy can be easily changed by the user. We apply this procedure to rules *R-06* and *R-08*. Verifying the randomness of values is a challenging task. While this test does not ensure that the values that pass the check are truly random, it finds obvious sources of non-randomness. Static approaches do not typically check these types of rules.

D. Reused Values

The checking procedure of Fig. 3 (d) checks if a value or a combination of values of a parameter of a crypto class is reused across the executions of an application. For instance, for rule *R-09*, we have to ensure that the same pair (key, IV) is never reused to encrypt different messages. The checking procedure collects all the values from the log and checks if there are duplicates:

```
R-09: containsDuplicates(
    { (SymmEncryption.key,
      SymmEncryption.iv) }) = False
```

We used this checking procedure for the rules *R-09* and *R-12*. Static approaches do not typically check these types of rules.

VI. IMPLEMENTATION OF CRYLOGGER FOR ANDROID

We implemented *CRYLOGGER* to detect crypto misuses in Java and Android apps by instrumenting classes of the Java Cryptography Extension (JCE) and the Java Cryptography Architecture (JCA), which are part of the Java standard library[4]. These classes provide a common interface for crypto algorithms to all Java apps. This interface is then implemented by 'providers', i.e., specific crypto libraries, e.g., SunJCE, BouncyCastle, etc. Thus, they are the perfect place to detect crypto misuses in Android (as well as Java) apps. TABLE II reports the mapping of the classes of Section IV (Crypto Classes in the table) to the Java classes that we instrumented. In some cases, a single crypto class, e.g., `RandomGenerator`, is mapped to multiple Java classes, e.g., `Random` and `SecureRandom`. In the appendices (TABLE III) we report for each class the member methods that we instrumented and the parameters that we collected for each Java class.

A. Automated Testing of Android Apps

We ran *CRYLOGGER* on 1780 Android apps from the official Google Play Store. These are the most popular free apps of 33 different categories (Section IX). In this section, we discuss how we automated the testing for such a large number of apps.

We implemented a Python script to perform the following nine steps. Step (S1) starts an Android emulator, whose Java library has been instrumented with *CRYLOGGER* (or we can use a real device). (S2) downloads the chosen app from the Google

Crypto Classes	Java Classes
MessageDigest	java.security.MessageDigest
SymmEncryption	javax.crypto.Cipher
AsymmEncryption	javax.crypto.Cipher
	java.security.Signature
KeyDerivation	javax.crypto.spec.PBEKeySpec
	javax.crypto.spec.PBEParameterSpec
RandomGenerator	java.util.Random
	java.security.SecureRandom
KeyStorage	java.security.KeyStore
SSL/TLS/Certif.	java.net.URL
	java.net.ssl.SSLContext
	java.net.ssl.SocketFactory
	java.net.ssl.HttpsURLConnection

TABLE II
Mapping from the crypto library of Section IV to the Java standard library.

Play Store market. (S3) configures the user interface (UI) of the emulator to facilitate random testing (more details below). (S4) installs the app on the emulator with the android debug bridge (ADB)[5]. (S5) uses Monkey to send random events to the UI of the app (the number of UI events is configurable and Monkey can be replaced with other tools). We call 'events' the actions that can be performed on the UI of an app, such as scrolling, touching, inserting text, etc. (S6) collects the crypto log. (S7) uninstalls the app and deletes its data with ADB. (S8) checks the crypto rules and reports the rules that have been violated. (S9) tests another app starting from Step (S4), if it is necessary.

Android apps are UI driven [39]. Therefore to verify an app, there are two main alternatives: manual tests, where a user needs to interact with the UI of the app, and automated tests, where the UI events are generated by a tool [37], e.g., Monkey. Since the results of any dynamic tool, including *CRYLOGGER*, are as good as the UI events used to exercise the app, it is critical to define how to test the apps to detect crypto misuses. Since we wanted to fully automate the testing process, we decided to exclude the option of performing manual tests. We decided to use Monkey for the experimental results in Sections VIII and IX. Monkey is the most popular tool for random-based testing and compared to other tools for random-based generation is known to be the most effective [37]. The main advantage of Monkey is that it is fully automated. It is also fully integrated in Android Studio, and thus supported on all the apps of the Google Play Store and on different Android versions. In addition, it is fast because to generate events it does not need to maintain any information (state) of the app. It has, however, two limitations: (1) random events generate unintended behaviors, for instance, turning off Internet or closing the app [39], and (2) poor app coverage since the events are generated randomly, for example, Monkey cannot perform complex operations, such as app registration or login.

(1) *Unintended Behaviors*: To address this problem, we added Step (S3) mentioned above. This step (i) activates the immersive mode[6], where an app is fixed on the screen and there is no easy way to return to the home screen, (ii) removes the quick settings, so that Monkey cannot interact with system configurations, e.g., Wi-Fi, and (iii) disables physical buttons, e.g., power and volume, to focus the attention of Monkey on the app. We observed that

---

[4]Documentation about JCA and JCE can be found here: https://docs.oracle.com/javase/7/docs/technotes/guides/security/crypto/CryptoSpec.html (Java 7).

[5]Android ADB: https://developer.android.com/studio/command-line/adb.
[6]Immersive: https://developer.android.com/training/system-ui/immersive.

these modifications eliminate most of the unintended behaviors.

(2) *Poor App Coverage:* To improve the coverage, we evaluated many tools for test generation, e.g., SmartDroid [28], Droid-Bot [40], and Dynodroid [41]. Their main drawbacks are that the support is limited (they work on specific versions of Android) and they are typically slower than Monkey, as they need to keep some information about the state of the app and update it to explore new behaviors (e.g., a control-flow graph [40]). Due to these limitations, we decided to use Monkey. We noticed that Monkey is actually capable of triggering many of the crypto misuses, even if the UI events are completely random. Most of the functions that we instrumented (TABLE III) are, in fact, used to initialize some basic, critical crypto classes, and therefore they are relatively easy to trigger. We observed that Monkey achieves $\sim 25\%$ of line coverage on average, but it reports as many crypto misuses as CryptoGuard [6], which employs static analysis (Section VIII). This choice carries some limitations, i.e., the possibility of false negatives, because some parts of the apps are hard to explore (e.g., login). It is worth to mention, however, that *CRYLOGGER* can be configured to use any other UI exercisers as well as manually-written sequences of UI events. For example, if developers have sequences of events to stimulate their apps, it can exploit those to obtain higher coverage. In future, we plan to build our own UI event generator tool specialized for crypto.

B. Details about Crypto Rules Checking

We used the checking procedures explained in Section V to check the crypto rules for the Android apps, but we made few adaptations. The functions that we instrumented for rules *R-24* and *R-25* (TABLE III) take as input some classes for which the developer of the application has to implement some methods, e.g., the method `verify()` to verify the host name. To obtain the values of the parameters `allhost` and `allcert` that are used by rules *R-24* and *R-25*, during the logging, we pass some erroneous values, such as `NULL` or empty strings, to determine if those functions were implemented naively. For the rules that require two executions (see Fig. 3 (b)), we obtain the two logs by running the application on two different instances of the emulator. We also make sure that if we see a value that is in both logs, then this is caused by constants hard-coded in the app.

VII. EXPERIMENTAL SETUP AND BENCHMARKS

We evaluated *CRYLOGGER* on two sets of benchmarks. The first set consists of Android apps. We downloaded 2148 free Android apps from the Google Play Store. These cover the most popular free apps of 33 different categories. We discarded 110 of these apps since they do not use any crypto APIs. We discarded 258 of these apps as they do not work on the Android emulator either because they keep crashing or they require libraries that cannot be installed in the emulator environment. The results of running *CRYLOGGER* on the remaining 1780 apps are discussed in Section IX. We used a random subset of these apps to compare *CRYLOGGER* against CryptoGuard [6] as described in Section VIII. The second set of benchmarks is the CryptoAPI-Bench [26], a set of Java applications that include crypto misuses. The CryptoAPI-Bench was originally proposed

to compare static approaches. We extended it and then used it to compare *CRYLOGGER* against CryptoGuard (see Section VIII).

VIII. RESULTS: COMPARISON WITH CRYPTOGUARD

We compared *CRYLOGGER* against CryptoGuard [6], one of the most effective static tools in detecting crypto misuses in Java-based applications. We could not compare *CRYLOGGER* against a dynamic tool because, to the best of our knowledge, *CRYLOGGER* is the only approach to detect misuses dynamically for a large number of rules (Section III). We chose CryptoGuard among many available static tools, e.g., CryptoLint [5], CrySL [15], because it has been recently shown that CryptoGuard is the tool with the lowest false positive and false negative rates among them [26]. It is also the tool that supports the largest number of crypto rules. We compared *CRYLOGGER* and CryptoGuard by using 2 datasets. The first consists of 150 Android apps we randomly chose from the set of 1780 apps (Section VII). For this dataset, we evaluated the execution times and the number of crypto misuses found by the two tools. The second dataset is the CryptoAPI-Bench [26], a set of Java benchmarks that include crypto misuses. For this dataset, we determined the false positive and the false negative rates of the two tools. We also extended the CryptoAPI-Bench with more benchmarks to cover cases relevant to dynamic approaches.

A. Android Apps: Results

We used 150 free Android apps randomly chosen from the dataset of 1780 apps to compare *CRYLOGGER* and Crypto-Guard[7]. We could not use the entire dataset of 1780 apps of Section VII because the false positives for CryptoGuard must be determined manually (see below). For a fair comparison, we excluded the rules that are supported by *CRYLOGGER*, but not by CryptoGuard, and thus we compared the two tools by checking 16 crypto rules. For each rule, we determined the number of apps that are marked as "vulnerable" by each tool and analyzed the false positive and false negative rates. We used 3 configurations for *CRYLOGGER* where we varied the number of UI events that are generated with Monkey: we used 10k, 30k and 50k random events (same random seed) to see how the number of input events impacts the number of misuses that are identified. In the following, we refer to the 3 configurations as *CRYLOGGER*10, *CRYLOGGER*30 and *CRYLOGGER*50, respectively.

The results of the comparison are reported in Fig. 4 and 5. Each graph is an *upset plot* [53], [54] for a specific rule. An upset plot is an alternative to the Venn diagrams to represent sets and their intersections. In our context, the sets that are represented are the sets of apps that are considered vulnerable by each approach (*CRYLOGGER*10, *CRYLOGGER*30, *CRYLOGGER*50 and CryptoGuard). The horizontal bars are used to indicate the total number of apps that are considered vulnerable by each approach. For instance, for rule *R-03*, CryptoGuard found 17 vulnerable apps among the 150 apps that were analyzed, i.e., 17 apps violate *R-03*, *CRYLOGGER*50 and *CRYLOGGER*30 flagged 21 apps as vulnerable, and finally *CRYLOGGER*10 marked 20 apps as vulnerable. The vertical bars are used to represent the

---

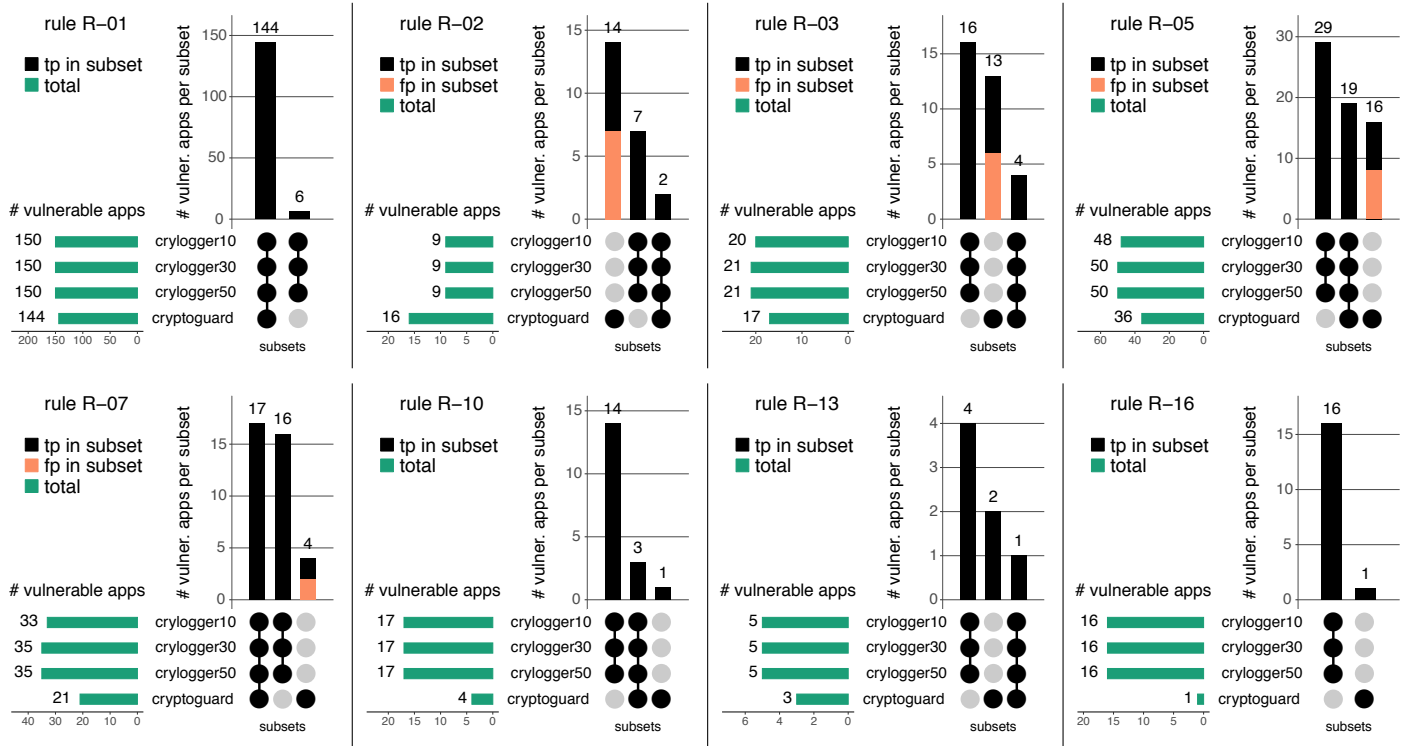[7]https://github.com/franceme/cryptoguard; vers: 03.07.03; commit: ba16c928.

Fig. 4. **(Part 1)** Comparison of *CRYLOGGER* and CryptoGuard [6] on 150 Android apps. Each graph is an *upset plot* [53]. The **horizontal bars** indicate the number of apps flagged as vulnerable by CryptoGuard and *CRYLOGGER* (that is run with 10k, 30k and 50k stimuli). The **vertical bars** indicate the number of apps flagged as vulnerable by a possible intersection of the four approaches (the three largest, non-empty intersections are reported). For example, for *R-02*: 2 apps are considered vulnerable by all approaches, 14 apps are flagged as vulnerable by CryptoGuard, but not by *CRYLOGGER*, and finally 7 apps are considered vulnerable by *CRYLOGGER* only. The vertical bars distinguish the false positives (fp) obtained by reverse engineering and the true positives (tp) for CryptoGuard.

intersections of the sets of apps that are considered vulnerable by each approach. Specifically, each vertical bar indicates the size of the intersection of the sets whose circles below the bar are black. For example, for rule *R-03*: the 3 configurations of *CRYLOGGER* identified 16 crypto misuses that were not found by CryptoGuard; CryptoGuard detected 13 misuses that were not found by the 3 configurations of *CRYLOGGER*, and finally all the approaches agree that 4 apps are vulnerable. The vertical bars for CryptoGuard distinguish the false positives (fp) from the true positives (tp), because CryptoGuard can produce false positives. To make this distinction, we reverse engineered the apps by using APKTool[8] and verified if the API calls flagged as vulnerable by CryptoGuard could actually be called at runtime. We used a very conservative approach to determine the false positives. Starting from the flagged API call, we recursively built the sets of functions that call that API until we obtained a fixed point. If a function that is part of the package of the app is in the set, then we considered the API call a true positive because there is the possibility that it could be called at runtime. If none of the functions in the set is part of the package of the app, then we considered the API call a false positive. If the app was completely obfuscated with ProGuard[9], thereby making it impossible to determine its packages, then we assumed that the vulnerability flagged by CryptoGuard was a true positive. In our case 6 apps were completely obfuscated. This process does

not guarantee that all false positives are identified because some paths in the code of the app could still be not executable (dead code), but it helps to find the obvious sources of false positives.

For most of the rules, excluding some cases (*R-01*, *R-18*, *R-22*, *R-24*, *R-25* and *R-26*), we can observe the following: (1) CryptoGuard detected some crypto misuses that were not found by *CRYLOGGER*; (2) *CRYLOGGER* detected some misuses that were not found by CryptoGuard; (3) the number of misuses detected by *CRYLOGGER* is higher than CryptoGuard, considering that the latter produces many false positives (we discuss some examples of false positives in Section VIII-D). For some rules (*R-01*, *R-18*) we can observe that all the misuses detected by CryptoGuard were also discovered by *CRYLOGGER*. For other rules (*R-22*, *R-24*, *R-25* and *R-26*) we can observe that CryptoGuard found more crypto misuses compared to *CRYLOGGER*, but it produced a significant number of false positives (in some cases the false positive rate is > 50%). These rules are related to SSL/TLS and they require to evaluate the security of the actual implementation of some Java functions, for example, the function `verify` in the case of rule *R-24* or the functions `checkClientTrusted`, `checkServerTrusted` and `getAcceptedIssuers` in the case of rule *R-25*. These tasks are better suited for static analysis because it is necessary to prove that some parameters of the functions are never used or the parameters of the functions do not influence the return value [6]. Overall, these results show that *CRYLOGGER* can complement the results that are obtained through static analysis and it can

---

[8] https://github.com/iBotPeaches/Apktool; vers: 2.4.0; commit: 197d4687.
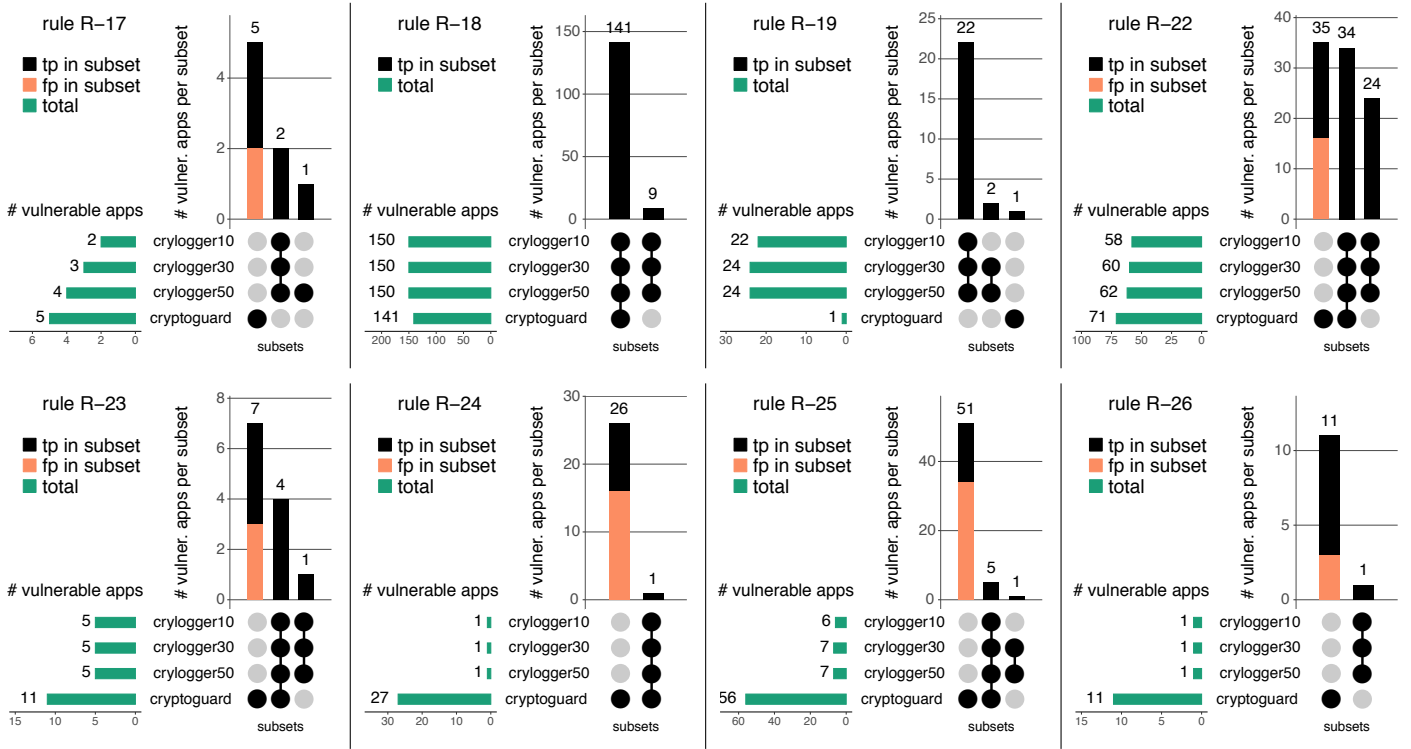[9] ProGuard: https://www.guardsquare.com/en/products/proguard.

Fig. 5. **(Part 2)** Comparison of *CRYLOGGER* and CryptoGuard [6] on 150 Android apps. Each graph is an *upset plot* [53]. The **horizontal bars** indicate the number of apps flagged as vulnerable by CryptoGuard and *CRYLOGGER* (that is run with 10k, 30k and 50k stimuli). The **vertical bars** indicate the number of apps flagged as vulnerable by a possible intersection of the four approaches (the three largest, non-empty intersections are reported). For example, for *R-22*: 35 apps are considered vulnerable by all approaches, 34 apps are flagged as vulnerable by CryptoGuard, but not by *CRYLOGGER*, and finally 24 apps are considered vulnerable by *CRYLOGGER* only. The vertical bars distinguish the false positives (fp) obtained by reverse engineering and the true positives (tp) for CryptoGuard.

be helpful in detecting misuses in Android apps. By combining *CRYLOGGER* with powerful static tools such as CryptoGuard, it is possible to detect crypto misuses effectively. We can also observe that it is sufficient to configure *CRYLOGGER* to use 30k random UI events to trigger most of the crypto misuses. We performed the same experiments on the rules that are not supported by CryptoGuard (see Fig. 9 in the appendices).

B. Android Apps: Execution Time

We measured the average execution time required by the 3 configurations of *CRYLOGGER* and by CryptoGuard to analyze the 150 apps used for the comparison. We obtained that *CRYLOGGER*10 requires on average 146.4 seconds per app, *CRYLOGGER*30 takes 287.4 seconds, and *CRYLOGGER*50 takes 751.7 seconds to perform dynamic analysis. CryptoGuard requires 287.6 seconds. Other static tools are usually much slower. For example, the authors of CryptoLint [5] reported that 22.2% of the apps they analyzed did not terminate in 30 minutes and 6.5% ran out of memory. This shows that the execution time of *CRYLOGGER* is comparable to the time required by CryptoGuard, confirming that both approaches are scalable.

C. Android Apps: Coverage

We measured the line coverage, the method coverage and the class coverage of the apps analyzed with the three configurations of *CRYLOGGER*. We used ACVTool [55] to obtain this information. To calculate the coverage, we considered only the files that are included in the main packages of the apps,

while excluding the files that belong to the third-party libraries because they can contain code not callable from the apps. The average line coverage for *CRYLOGGER*10, *CRYLOGGER*30, and *CRYLOGGER*50 are 22.8%, 25.3%, and 25.4%, respectively. The average method coverage are 25.4%, 27.9%, and 27.9%, respectively. The average class coverage are 32.8%, 35.4%, and 35.7%, respectively. The coverage is relatively low and there are many lines of code that Monkey could not explore ($\sim 75\%$). These results are not surprising because Monkey generates completely random UI events [39]. However, this shows that even if the coverage is low, *CRYLOGGER* can detect misuses as the crypto APIs are easily triggerable with random events.

D. Android Apps: False Positives

Fig. 4 and 5 show that CryptoGuard can produce many false positives, especially for rules *R-22* (false positives: 22.5%), *R-24* (59.3%), *R-25* (57.1%) and *R-26* (27.2%). In Fig. 11 we report two concrete examples of false positives that we found. The first example is for rule *R-22*. We found that many apps were flagged as vulnerable by CryptoGuard because they include the Java class `HttpTesting`. While violating rule *R-22* due to the use of HTTP instead of HTTPS, this class is meant to be used for testing and it is not instantiated at runtime by any of the apps we analyzed. Similarly, for rule *R-24*, many apps were flagged because they contain the Java class `AdjustFactory`[10]. The function reported in the second example of Fig. 11 is used

---

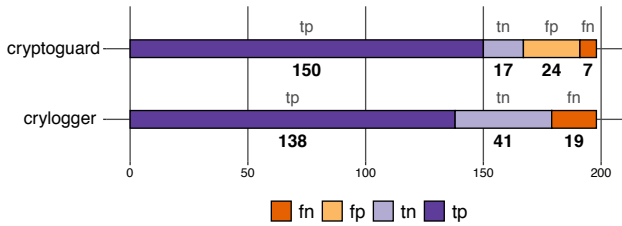[10]The code is available at https://github.com/adjust/android_sdk.

Fig. 6. Comparison of *CRYLOGGER* and CryptoGuard [6] on the CryptoAPI-Bench*. We report the number of false positives (fp), false negatives (fn), true positives (tp) and true negatives (tn). "True positive": there is a crypto misuse that is caught. "True negative": there is not a crypto misuse and it is not caught.

only for testing, as its name suggests, and it is never called at runtime by any of the apps that we analyzed. This function was flagged as vulnerable by CryptoGuard.

E. CryptoAPI-Bench: Results

We compared *CRYLOGGER* against CryptoGuard by using the CryptoAPI-Bench [26][11], a set of Java benchmarks that include crypto misuses. The CryptoAPI-Bench has been proposed to compare CryptoGuard and other static approaches. Therefore, (1) the code is not directly executable, (2) it lacks test cases that are useful for dynamic approaches, and (3) it misses test cases for the rules that are not supported by CryptoGuard. We extended the CryptoAPI-Bench such that (1) the code can be analyzed by static approaches as well as executed by dynamic approaches, (2) we added new test cases that are challenging for dynamic approaches, and (3) we included new test cases for the rules supported by *CRYLOGGER*, but not by CryptoGuard. In this section, we discuss the result of the comparison on the modified CryptoAPI-Bench that we call CryptoAPI-Bench*. For fairness, we consider the rules that are supported by both *CRYLOGGER* and CryptoGuard. For fairness, we also report the results on the original CryptoAPI-Bench in Fig. 12 (in the appendices).

CryptoAPI-Bench contains six types of tests: (1) *basic*: the crypto misuse is in the function `main`; (2) *miscellaneous*: similar to basic, but the parameters for the API calls are saved in data structures or go through data type conversions; (3) *interprocedural*: the misuse is in a function that is called by `main` with 2 or 3 levels of indirection; (4) *path sensitive*: the crypto misuse is in a branch that is always evaluated to `true` at runtime; (5) *field sensitive*: the misuse is in a member function and the relevant parameters are saved in the field of a class; (6) *multiple classes*: the relevant parameters of a misuse are passed from a class to another class to reach the API call. We report an example of each test in Fig. 10 (in the appendices). Some of these tests are challenging for a static tool, but they are all the same from a dynamic tool perspective. Therefore, we decided to add the following type of test: (7) *argument sensitive*: the misuse is triggered only if a specific value is passed as input to `main`.

Fig. 6 shows the results of the comparison of *CRYLOGGER* and CryptoGuard. The bars show the number of true positives (tp), true negatives (tn), false positives (fp) and false negatives (fn). In CryptoAPI-Bench* there are 198 tests in total, 157 true positive tests, i.e., tests in which there is a crypto misuse, and

[11]https://github.com/CryptoGuardOSS/cryptoapi-bench, commit: ace0945.

41 true negative tests, i.e., tests in which there are no misuses. *CRYLOGGER* cannot produce any false positives, but it produces 19 false negatives, all for the tests that are argument sensitive. CryptoGuard produces both false positives and false negatives. The false positives are caused by tests that are path sensitive, and interprocedural tests. The false negatives are caused by the refinements that are applied by CryptoGuard [6], interprocedural tests, and tests that are path sensitive. These results confirm that static tools can be complemented with *CRYLOGGER* to expose more misuses as well as reduce the number of false positives.

IX. RESULTS: VULNERABILITIES IN ANDROID

We run *CRYLOGGER* on the 1780 apps downloaded from the Google Play Store (Section VII). We stimulated the apps with 30k random events as this was a good compromise between running time and number of vulnerabilities found in a subset of these apps (Section VIII). The experiments took roughly 10 days to run on an emulator running Android 9.0.0_r36, to which we allocated 6 cores (Intel Xeon E5-2650) and 16 GB of RAM.

Fig. 7 reports the results of the analysis. The graph reports the total number of apps that violate the 26 crypto rules checked by *CRYLOGGER*. A very high number of apps use broken hash algorithms (*R-01*, 99.1%) and unsafe random generator (*R-18*, 99.7%). These results are more alarming than the ones that were obtained statically in [6], 85.3% and 84.0%, respectively. *CRYLOGGER*, similarly to static tools, cannot determine exactly how hash functions or random numbers are used in the apps by using rules *R-01* and *R-18* only. While for *R-01* it is challenging to determine how hash functions are actually used, for *R-18* we can check if non-truly random numbers are used as values for keys and initialization vectors with *R-06* and *R-08*. These rules are not supported by static tools and they give more precise information about the use of non-truly random numbers. We decided to keep rule *R-18* to compare *CRYLOGGER* against other static tools, but we suggest using rules *R-06* and rule *R-08* for a more precise analysis. Other more subtle uses of hash functions can produce false positives, e.g., when broken hash functions are used with non-sensitive data or when the property of collision resistant is not required. For other rules, e.g., *R-03*, *R-13*, and *R-22*, we obtained results more similar to [6]. A surprising number of apps reuse the same (key, IV) pairs (*R-09*, 31.3%), which was never reported before. Many apps also use badly-generated keys (*R-06*, 36.1%), badly-generated IVs (*R-08*, 6.6%), and reuse salts for different purposes (*R-12*, 6.6%), which are rules that were not checked by other tools before. For rule *R-01* we found that 99.0% of the apps that violate *R-01* use SHA1 and 99.7% use MD5 as message digest algorithm. For *R-02*, we found that 81.0% of the apps that use broken symmetric algorithms use DES, while 16.7% still use Blowfish. We found that 82.8% of the apps that violate *R-13* use ≤ 3 iterations for key derivation, which is much lower compared to the suggested value (1000). For *R-14* and *R-15* we found that 27.1% of the apps use "changeit" as password, while 8.5% use "dontcare". For RSA, we saw that 97.7% use 1024 bits as key size (2048 is the suggested value). These results confirm what was obtained
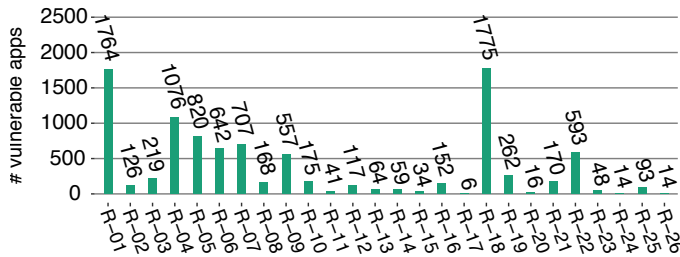
Fig. 7. Number of vulnerable Android apps for each crypto rule. We analyzed 1780 Android apps with *CRYLOGGER* configured to generate 30k random events with Monkey. We downloaded the apps from the official Google Play Store. The dataset of apps was collected between September and October 2019.

in previous works by using static analysis [5], [6] and show that *CRYLOGGER* can analyze a large number of apps automatically.

A. Disclosure of Vulnerabilities

We contacted 306 developers of Android apps and libraries to disclose the vulnerabilities reported in Fig. 7. We respected the disclosure policies of the companies we contacted. Starting from the apps that violate 18 rules (the highest number of violations in our dataset), we contacted all the apps with ≥ 9 rule violations. All the apps are popular: they have from hundreds of thousands of downloads to more than 100 millions. Unfortunately, only 18 developers answered our first email of request and only 8 of them followed back with us multiple times providing useful feedback on our findings. We also contacted 6 developers of popular Android libraries and received answers from 2 of them. The characteristics of the 8 apps and 2 libraries for which we received feedback are reported in the first table from the left of Fig. 8. We preferred to anonymize the apps and libraries because (i) we do not want to associate the feedback we received to the company of the app or its employers, and (ii) we consider some of the attacks possible although developers considered them out-of-scope because they require privilege escalation.

Apps *A-01*, *A-04*, and *A-07* violate rule *R-01*. Their developers told us that MD5 or SHA1 are used for hashing non-sensitive values. App *A-01* violates also rules *R-02* (DES) and *R-03*: the developers justified the use of broken algorithms saying that they do not pose concrete risks to their users. *A-01*, *A-05*, and *A-07* violate rules related to poor encryption parameters, such as constant keys (*R-05*, *R-06*), IVs (*R-07*, *R-08*) and salts (*R-10*). The developers adopted poor encryption practices to encrypt data that are stored locally on the smartphone. They consider these issues outside of their threat model since privilege escalation attacks are required to exploit them. *A-03* uses repeating (key, IV) pairs (*R-09*): the developers agreed that it is a real issue and they plan to fix it. They reused the same pairs because they experienced app crashing when using fresh pairs. *A-02*, *A-05*, *A-06*, and *A-08* use constant passwords (*R-16*, *R-23*) to encrypt data. The developers do not plan to fix these problems because a privilege escalation attack is necessary to access the data. The developers of *A-01*, *A-04* and *A-05* told us that using a short RSA key (*R-19*) does not pose concrete risks. *L-09* is a popular library for advertisements. The library uses the same (key, IV) pairs to store data locally. The same (key, IV) pairs are reused across different apps, i.e., all the apps using this library end up

using the same sequence of (key, IV) pairs. About 30% of the apps in our dataset share the same sequence of pairs which are used to encrypt data in the private folder of each app. The library developers confirmed this issue, but they classified it as out-of-scope. Note that this experiments cannot be replicated by static tools and it is an example of how *CRYLOGGER* can perform inter-app analysis. *L-10* is a common library for advertisements. The library employs weak encryption practices to store data locally. We talked with the library developers. They were aware of the issue and said that the data are not security critical.

This analysis reveals that the threat model of *CRYLOGGER* and all the other static tools is not aligned with the developers' threat model. Developers claim that sensitive data can be encrypted poorly if they are stored only locally because privilege escalation is required to access them. Unfortunately, side-channel attacks can also access the data [56]. While we recommend to always adopt safe crypto practices, one way to to avoid such types of warnings in *CRYLOGGER* is to log when data are stored on the local storage (e.g., in classes such as `File` or `KeyStore`) and discard the corresponding violations. Developers are also more interested to rules that, if violated, pose concrete security threats as also reported in [6]. For example, while setting a minimum size for keys (*R-19*) is important, the effects of its violation are hard to assess. Since the feedback we received from developers is limited to a few apps, we decided to analyze some apps manually to determine if the vulnerabilities of Fig. 7 are exploitable.

B. Analysis of Vulnerabilities

We reverse engineered 28 apps with APKTool and JADX[12]. We chose half of the apps among the most popular apps of our dataset (Section VII) with the highest number of violations. We chose the remaining half randomly. The apps characteristics are shown in Fig. 8. We performed the following steps for reverse engineering: (i) we used APKTool and JADX to obtain the Java code from the binary (apk) of the app, (ii) we analyzed the app with *CRYLOGGER*, which we extended to log the stack trace for each rule violation, and (iii) we manually analyzed the code starting from the flagged API call to understand its purpose in the app. We spent on average 6 hours per app for code analysis.

A significant number of these apps (14/28) are vulnerable to attacks, even though some may be considered out-of-scope by developers. Most of the rules (22/26) are effective in detecting at least one vulnerable app. App *A-13* violates many rules related to encryption. This app uses encryption to manage subscriptions to premium features and users data. The subscription and the users data are stored locally on the app and attacker can read the data as well as fake subscriptions. Similarly, apps *A-18*, *A-20*, *A-24*, *A-25*, *A-33*, and *A-34* store critical users data (emails, answers to security questions, etc.) by using weak encryption algorithms. *A-22*, *A-29*, and *A-30* store SSL/TLS certificates with weak password-based encryption. *A-14* uses a constant seed (*R-17*) to randomly generate keys used for encryption of users data, so the keys can be easily obtained. Apps *A-31*, *A-32*, and

---

[12]https://github.com/skylot/jadx; vers: 1.1.0, commit: cc29da8.

ID	Type (#Downloads)	Analyzed Violations
A-01	File Manager (100M+)	R-02, R-03, R-05, R-07, R-08, R-09, R-10, R-12, R-19
A-02	Data Transfer (10M+)	R-16, R-23
A-03	Video Streaming (10M+)	R-09, R-20, R-22
A-04	Newspaper App (5M+)	R-01, R-19, R-20, R-23
A-05	Social & News (5M+)	R-05, R-06, R-07, R-08, R-10, R-16, R-19
A-06	Language Learning (1M+)	R-16
A-07	Music Streaming (1M+)	R-01, R-05, R-06, R-09
A-08	Video Streaming (1M+)	R-16, R-23
L-01	Advertisement (N.A.)	R-09
L-02	Advertisement (N.A.)	R-07, R-08, R-10

ID	Type (#Downloads)	Analyzed Violations
A-09	Messaging (100M+)	R-01
A-10	Entertainment (100M+)	R-18, R-22
A-11	Movie Reviews (100M+)	R-18, R-19, R-21
A-12	Book Reading (50M+)	R-02, R-03, R-05, R-06
A-13	Passw. Manager (50M+)	R-02, R-03, R-04, R-05, R-06, R-07, R-08
A-14	Passw. Manager (50M+)	R-17
A-15	Screen Utils (10M+)	R-01
A-16	File Manager (10M+)	R-01
A-17	Video Streaming (10M+)	R-04
A-18	Video Streaming (10M+)	R-04, R-07, R-08, R-21, R-23
A-19	Video Streaming (10M+)	R-09, R-20, R-22
A-20	Live Events Info (10M+)	R-11, R-16
A-21	Video Streaming (10M+)	R-11, R-13
A-22	Video Streaming (10M+)	R-14, R-15, R-16
A-23	Newspaper App (5M+)	R-01, R-19, R-20, R-21

ID	Type (#Downloads)	Analyzed Violations
A-24	Mail Manager (5M+)	R-04, R-05, R-06, R-10, R-12, R-13, R-16
A-25	Video Streaming (5M+)	R-19, R-21, R-24, R-25, R-26
A-26	Stocks Manager (5M+)	R-22
A-27	Authentication (5M+)	R-23
A-28	Video Streaming (1M+)	R-10, R-16
A-29	Blog Reading (1M+)	R-14, R-15, R-16
A-30	Book Reading (1M+)	R-14, R-15, R-16
A-31	Healthcare Info (1M+)	R-24, R-25, R-26
A-32	Music Streaming (1M+)	R-24, R-25, R-26
A-33	Newspaper App (500K+)	R-03, R-05, R-06, R-10, R-13, R-16, R-24, R-25, R-26
A-34	Entertainment (100K+)	R-10, R-11, R-13, R-16
A-35	Passw. Manager (100k+)	R-13
A-36	Video Streaming (100K+)	R-22

Fig. 8. The first table from the left reports the characteristics of the Android apps for which we received feedback from their developers. The other tables report the characteristics of the apps that we reverse engineered. The rules reported in the last column of each table are those that were analyzed by the developers or by us.

A-33 are vulnerable to man-in-the-middle attacks because they violate *R-24*, *R-25*, and *R-26*. These apps download copyrighted videos/music as well as ads, which can be intercepted by attackers. The other violations can be considered false positives. Some are caused by 'imprecise' rules. For example, on 3 apps each, rules *R-01* and *R-18* flag secure uses of hash algorithms and random number generators for non-sensitive data. Similarly, *R-04* flags 3 apps that use CBC encryption for scenarios different from client/server. Other violations come from (i) employing weak encryption schemes to obfuscate non-sensitive data and (ii) legacy practices such as using `PCKS#1` as padding scheme in SSL/TLS instead of more secure alternatives such as `OAEP`.

This analysis confirms that the threat model of *CRYLOGGER* and all the other static tools does not completely align with the developers' threat model and some rules produce false positives.

X. DISCUSSIONS AND LIMITATIONS

In this section, we discuss the advantages of dynamic approaches over static approaches and our current limitations.

Why a Dynamic Approach? To date, most of the approaches to detect crypto misuses are based on static analysis, which provides many benefits. Static analysis can analyze the code without executing it, and this is especially important for Android apps since UI test generators are not required. Static analysis can scale up to a large number of applications and, thanks to recent improvements [6], it can analyze massive code bases. Static analysis has, however, some limitations. It can produce false positives, i.e., alarms can be raised on correct calls to crypto APIs due to imprecise slicing algorithms. These alarms add up to those raised on parts of the applications that are not security critical (see Section IX). This makes it hard to analyze a large number of applications. Some static approaches [6] also incur in many false negatives. Some misuses escape detection because the exploration is pruned prematurely to improve scalability. In addition, static analysis misses some crypto misuses in the code that is loaded dynamically. This prevents analyses on critical code [20]. Also, static analysis can be inherently done on a single application only. It is not possible to perform inter-application analysis, as the one we did with *CRYLOGGER* on an Android library (Section IX). On the other hand, dynamic analysis is not a perfect antidote. Dynamic analysis is as good as the test generator that is used to run the applications. We discuss the main limitations of dynamic analysis in the next paragraphs.

False Positives. Although dynamic analysis, theoretically, should avoid false positives, these are possible when detecting crypto misuses (Section IX). It is hard to distinguish critical parts of the application, which should obey to the rules, from less critical parts where the data are not sensitive. In addition, the threat model adopted by app developers can differ from the one adopted in the research community. This requires complex manual analyses. One possible solution is to log additional information in other classes (e.g., `File`) to determine if rule violations can be discarded. This would greatly reduce the false positives, but it is hard to implement with general solutions.

False Negatives. Crypto misuses escape detection if they are not exercised during the execution. In Section VIII, we showed that for many Android apps, *CRYLOGGER* confirmed the results reported by CryptoGuard and found misuses missed by CryptoGuard. In other contexts, it might be harder to trigger the crypto APIs depending on the specific application. One possible solution is to complement *CRYLOGGER* with a static tool in order to expose the misuses that cannot be triggered at runtime.

XI. CONCLUDING REMARKS

We presented *CRYLOGGER*, the first tool that detects crypto misuses dynamically, while supporting a large number of rules. We released *CRYLOGGER* open-source to allow the community to use a dynamic tool alongside static analysis. We hope that application developers will adopt it to check their applications as well as the third-party libraries that they use.

REFERENCES

[1] "[GitHub] lucapiccolboni/crylogger: CRYLOGGER (Version v1.0), Zenodo." [Online]. Available: https://doi.org/10.5281/zenodo.3911285

[2] R. L. Rivest, "Handbook of Theoretical Computer Science," 1990.

[3] J. C. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche, "HACL*: A Verified Modern Cryptographic Library," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2017.

[4] B. B. Brumley, M. Barbosa, D. Page, and F. Vercauteren, "Practical Realisation and Elimination of an ECC-Related Software Bug Attack," in *Cryptographer's Track at the RSA Conference*, 2012.

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

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

[7] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, Y. Markov, A. P. Bianco, and C. Baisse, "The First Collision for Fully SHA-1," in *Proc. of the International Cryptology Conference (CRYPTO)*, 2017.

[8] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security," in *Proc. of the IEEE Symposium on Security and Privacy (SP)*, 2017.

[9] S. Nadi, S. Krger, M. Mezini, and E. Bodden, "Jumping Through Hoops: Why do Java Developers Struggle with Cryptography APIs?" in *Proc. of the International Conference on Software Engineering (ICSE)*, 2016.

[10] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *Proc. of the IEEE Symposium on Security and Privacy (SP)*, 2017.

[11] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source Attribution of Cryptographic API Misuse in Android Applications," in *Proc. of the Asia Conference on Computer & Communications Security (ASIA CCS)*, 2018.

[12] S. Vaudenay, "Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ..." in *Proc. of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptologyi (EUROCRYPT)*, 2002.

[13] E. B. Barker and A. L. Roginsky, "Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths," in *NIST Special Publication 800-131A*, 2018.

[14] "Password-Based Cryptography Specification, IETF (RFC 8018)," https://tools.ietf.org/html/rfc8018.

[15] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *Proc. of the ACM European Conference on Object-Oriented Programming (ECOOP)*, 2019.

[16] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2012.

[17] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler, and R. Kamath, "CogniCrypt: Supporting Developers in Using Cryptography," in *Proc. of the ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2017.

[18] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie, "Modelling Analysis and Auto Detection of Cryptographic Misuse in Android Applications," in *Proc. of the International on Dependable, Automatic and Secure Computing (DASC)*, 2013.

[19] M. Weiser, "Program Slicing," in *Proc. of the International Conference on Software Engineering (ICSE)*, 1981.

[20] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[21] A. Braga, R. Dahab, N. Antunes, N. Laranjeiro, and M. Vieira, "Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software," *IEEE Transactions on Reliability*, 2019.

[22] 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 *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[23] F. Gagnon, M. F. M. Fortier, S. Desloges, J. Ouellet, and C. Boileau, "AndroSSL: A Platform to Test Android Applications Connection Security," in *Proc. of the International Symposium on Foundations and Practice of Security (FPS)*, 2015.

[24] J. Li, Z. Lin, J. Caballero, Y. Zhang, and D. Gu, "K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2018.

[25] Y. Li, Y. Zhang, J. Li, and D. Gu, "iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications," in *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[26] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses," in *Proc. of the IEEE Secure Developement (SecDev)*, 2019.

[27] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" in *Proc. of the International Conference on Software Engineering (ICSE)*, 2013.

[28] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications," in *Proc. of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[29] F. Fischer, H. Xiao, C. Kao, Y. Stachelscheid, B. Johnson, D. Razar, P. Fawkesley, N. Buckley, K. Böttinger, P. Muntean, and J. Grossklags, "Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography," in *Proc. of the USENIX Security Symposium*, 2019.

[30] Y. Wang, P. Leon, K. Scott, X. Chen, A. Acquisti, and L. Cranor, "Privacy Nudges for Social Media: An Exploratory Facebook Study," in *Proc. of the International Conference on World Wide Web (WWW)*, 2013.

[31] M. Green and M. Smith, "Developers are Not the Enemy!: The Need for Usable Security APIs," *IEEE Securityi & Privacy*, 2016.

[32] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A Stitch in Time: Supporting Android Developers in Writing Secure Code," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2017.

[33] S. Ma, D. Lo, T. Li, and R. H. Deng, "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications," in *Proc. of the Asia Conference on Computer & Communications Security (ASIA CCS)*, 2016.

[34] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples," in *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, 2017.

[35] L. Singleton, R. Zhao, M. Song, and H. Siy, "FireBugs: Finding and Repairing Bugs with Security Patterns," in *Proc. of the International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2019.

[36] S. Krüger, K. Ali, and E. Bodden, "CogniCryptGEN: Generating Code for the Secure Usage of Crypto APIs," in *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2020.

[37] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?" in *Proc. of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[38] H. Zheng, D. Li, B. Liang, X. Zeng, W. Zheng, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Towards Getting There in an Industrial Case," in *Proc. of the International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017.

[39] S. Y. Yerima, M. K. Alzaylaee, and S. Sezer, "Machine Learning-based Dynamic Analysis of Android Apps with Improved Code Coverage," in *EURASIP Journal on Information Security*, 2019.

[40] Y. Li, Z. Yang, Y. Guo, and X. Chen, "DroidBot: A Lightweight UI-Guided Test Input Generator for Android," in *Proc. of the ACM/IEEE International Conference on Software Engineering Companion (ICSE-C)*, 2017.

[41] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An Input Generation System for Android Apps," in *Proc. of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2013.

[42] H. Krawczyk, "Cryptographic Extraction and Key Derivation: The HKDF Scheme," in *Proc. of the International Cryptology Conference (CRYPTO)*, 2010.

[43] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 2014.

[44] "Cryptographic Message Syntax, IETF (RFC 5652)," https://tools.ietf.org/html/rfc5652.

[45] T. Jager, S. A. Kakvi, and A. May, "On the Security of the PKCS#1 V1.5 Signature Scheme," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2018.

[46] P. Favre-Bulle, "Security Best Practices: Symmetric Encryption with AES in Java and Android," in *ProAndroidDev (online)*, 2018.

[47] D. L. Wheeler, "zxcvbn: Low-Budget Password Strength Estimation," in *Proc. of the USENIX Security Symposium*, 2016.

[48] P. A. Grassi, M. E. Garcia, and J. L. Fenton, "Digital Identity Guidelines," in *NIST Special Publication 800-63-3*, 2017.

[49] L. E. Bassham, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," in *NIST Special Publication 800-22*, 2010.

[50] D. Boneh, A. Joux, and P. Q. Nguyen, "Why Textbook ElGamal and RSA Encryption Are Insecure," in *Proc. of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptologyi (ASIACRYPT)*, 2000.

[51] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *Proc. of the International Cryptology Conference on Advances in Cryptology (CRYPTO)*, 1998.

[52] S. Goldwasser and S. Micali, "Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information," in *Proc. of the ACM Symposium on Theory of Computing (STOC)*, 1982.

[53] A. Lex, N. Gehlenborg, H. Strobelt, R. Vuillemot, and H. Pfister, "UpSet: Visualization of Intersecting Sets," *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2014.

[54] J. R. Conway, A. Lex, and N. Gehlenborg, "UpSetR: an R Package for the Visualization of Intersecting Sets and their Properties," *Bioinformatics*, 2017.

[55] A. Pilgun, O. Gadyatskaya, S. Dashevskyi, Y. Zhauniarovich, and A. Kushniarou, "An Effective Android Code Coverage Tool," in *Proc. of the ACM Conference on Computer & Communications Security (CCS)*, 2018.

[56] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management," in *Proc. of the USENIX Security Symposium*, 2017.

Package	Class	Function	Logged Data
java.security	MessageDigest	*byte[]* digest (*void*) *int* digest (*byte[]*, *int*, *int*)	`alg`

javax.crypto	Cipher	*void* init (*int*, *Key*, *SecureRandom*)
		void init (*int*, *Key*, *AlgorithmParameters*, *SecureRandom*)
		void init (*int*, *Key*, *AlgorithmParameterSpec*, *SecureRandom*)
		void init (*int*, *Certificate*, *SecureRandom*)
			`alg, mode,`
			`pad, key,`
			`iv`

	Cipher	*byte[]* doFinal (*void*) *int* doFinal (*byte[]*, *int*) *byte[]* doFinal (*byte[]*) *byte[]* doFinal (*byte[]*, *int*, *int*) *int* doFinal (*byte[]*, *int*, *int*, *byte[]*) *int* doFinal (*byte[]*, *int*, *int*, *byte[]*, *int*) *int* doFinal (*ByteBuffer*, *ByteBuffer*)	`out`
java.security	Signature	*void* initVerify (*PublicKey*) *void* initVerify (*Certificate*) *void* initSign (*PrivateKey*) *void* initSign (*PrivateKey, SecureRandom*)	`alg, key`
javax.crypto.spec	PBEKeySpec	PBEKeySpec (*char[]*) PBEKeySpec (*char[]*, *byte[]*, *int*) PBEKeySpec (*char[]*, *byte[]*, *int*, *int*)	`pass, salt,` `iter`
javax.crypto.spec	PBEParameterSpec	PBEParameterSpec (*byte[]*, *int*) PBEParameterSpec (*byte[]*, *int*, *AlgorithmParameterSpec*)	`salt, iter`
java.security	SecureRandom	SecureRandom (*void*) SecureRandom (*byte[]*) *void* setSeed (*byte[]*)	`seed, out`
	SecureRandom	*void* nextBytes (*byte[]*) *void* setSeed (*byte[]*)	
java.util	Random	Random (*void*)	*constructor*
	Random	*int* next (*int*) *void* nextBytes (*byte[]*)	`out`
java.security	KeyStore	*Key* getKey (*String*, *char[]*) *void* load (*InputStream*, *char[]*) *void* load (*LoadStoreParameter*) *void* store (*OutputStream*, *char[]*) *void* store (*LoadStoreParameter*)	`pass`
java.net	URL	URL (*String*, *String*, *int*, *String*) URL (*URL*, *String*, *URLStreamHandler*)	`urlprotl`
javax.net.ssl	HttpsURLConnection	*void* setHostnameVerifier (*HostnameVerifier*) *void* setDefaultHostnameVerifier (*HostnameVerifier*)	`allhost` `sethost`
javax.net.ssl	SSLContext	*void* init (*KeyManger[]*, *TrustManager[]*, *SecureRandom*)	`allcert`
javax.net.ssl	SocketFactory	*SocketFactory* getDefault (*void*)	`sethost`

TABLE III
Java functions that have been instrumented and the parameters that are logged as defined in Fig. 2.
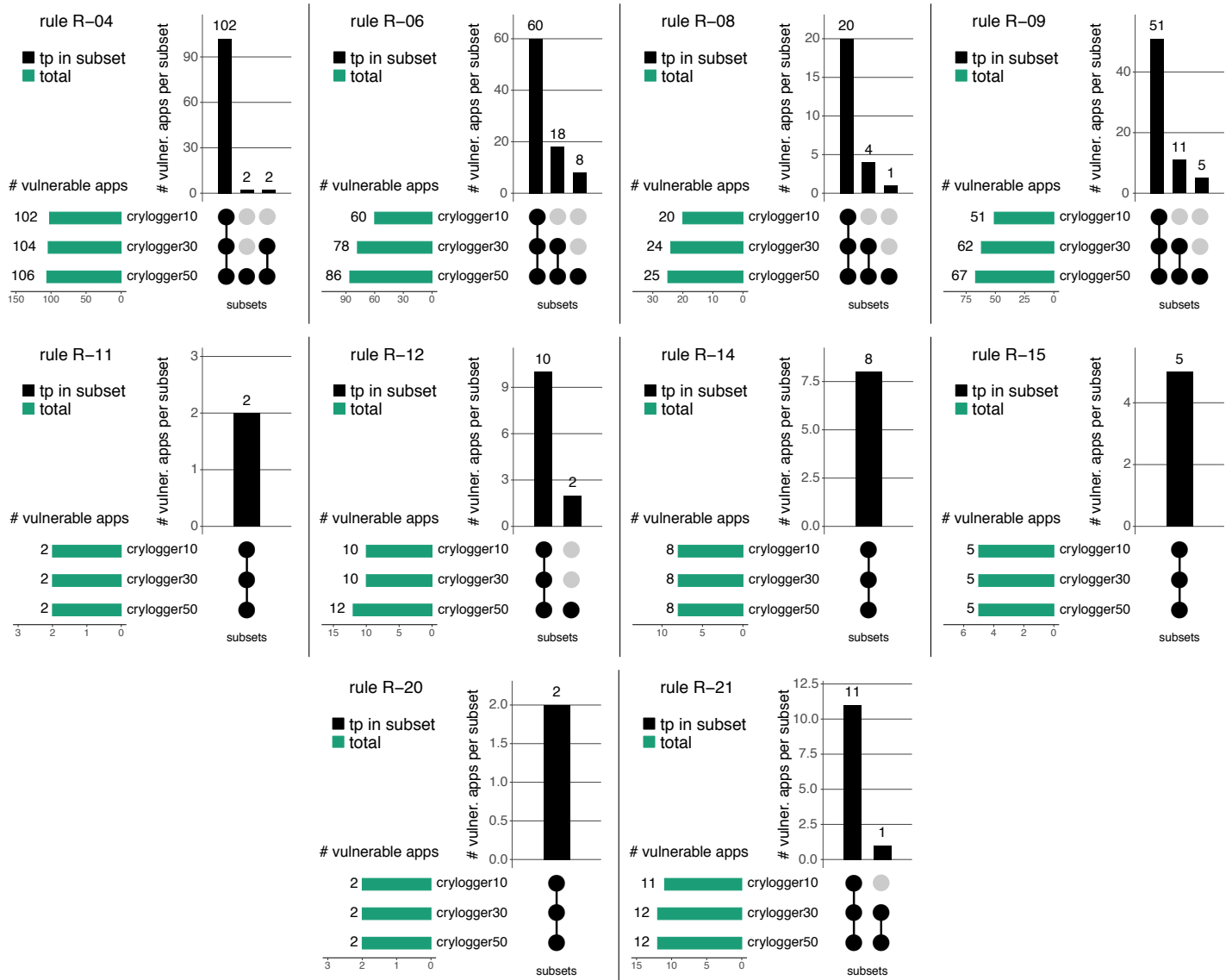
Fig. 9. Comparison of *CRYLOGGER* with 10k, 30k and 50k random stimuli on 150 Android apps. Each graph is an *upset plot* [53], [54]. The **horizontal bars** indicate the number of apps flagged as vulnerable by *CRYLOGGER* with 10k, 30k and 50k stimuli; the **vertical bars** indicate the number of apps flagged as vulnerable by a possible intersection of the approaches (the 3 largest, non-empty intersections are reported). For example, for *R-08*: 20 apps are considered vulnerable by all the approaches, 4 apps are flagged as vulnerable by using 30k and 50k stimuli only, and 1 app is considered vulnerable by using 50k stimuli only.

Listing 1. Basic

```
1 public class Test_X {
2   public static void main(String[] args) {
3     String algorithm = "AES/ECB/PKCS5PADDING";
4     Cipher c = Cipher.getInstance(algorithm);
5   }
6 }
```

Listing 2. Miscellaneous

```
1 public class Test_X {
2   public static void main(String[] args) {
3     String alg = "AES/ECB/PKCS5PADDING";
4     // Use of a simple data structure
5     DataStructure data = new DataStructure(alg);
6     Cipher c = Cipher.getInstance(data.get());
7   }
8 }
```

```
1 public class Test_X {
2   public static void main(String[] args) {
3     String alg = "AES/ECB/PKCS5PADDING";
4     // Conversion to another type
5     Othertype type = ConvertOtherType(alg);
6     Cipher c = Cipher.getInstance(data.get());
7   }
8 }
```

Listing 3. Interprocedural

```
1  public class Test_X {
2    public static void main(String[] args) {
3      String algorithm = "AES/ECB/PKCS5PADDING";
4      method1(algorithm);
5    }
6    public static void method1(String algorithm) {
7      method2(algorithm);
8    }
9    public static void method2(String algorithm) {
10     Cipher c = Cipher.getInstance(algorithm);
11   }
12 }
```

Listing 4. Path Sensitive

```
1 public class Test_X {
2   public static void main(String[] args) {
3     int choice = 2;
4     String algorithm = "AES/ECB/PKCS5PADDING";
5     if (choice > 1)
6       algorithm = "AES/CBC/PKCS5PADDING";
7     Cipher c = Cipher.getInstance(algorithm);
8   }
9 }
```

Listing 5. Field Sensitive

```
1  public class Test_X {
2    String algorithm;
3    public Test_X(String alg) {
4      algorithm = alg;
5    }
6    public method(String alg) {
7      alg = algorithm;
8      Cipher c = Cipher.getInstance(alg);
9    }
10   public static void main(String[] args) {
11     Test_X x = new Test_X("AES/ECB/PKCS5PADDING");
12     x.method("AES/CBC/PKCS5PADDING");
13   }
14 }
```

Listing 6. Multiple Classes

```
1  public class Test_X {
2    public static void main(String[] args) {
3      method1("AES/ECB/PKCS5PADDING");
4    }
5    public static void method1(String algorithm) {
6      Test_Y y = new Test_Y();
7      y.method(algorithm);
8    }
9  }
10 public class Test_X {
11   public void method2(String algorithm) {
12     Cipher c = Cipher.getInstance(algorithm);
13   }
14 }
```

Listing 7. Argument Sensitive

```
1 public class Test_X {
2   public static void main(String[] args) {
3     if (condition(args)) {
4       algorithm = "AES/CBC/PKCS5PADDING";
5       Cipher c = Cipher.getInstance(algorithm);
6     }
7   }
8 }
```

Fig. 10. The types of benchmarks that are present in the CryptoAPI-Bench [26]. We highlighted our modifications to make the benchmarks executable (Section VIII). The first 6 types of benchmarks (*basic, miscellaneous, interprocedural, path sensitive, field sensitive, multiple classes*) were originally proposed in [26]. We added *argument-sensitive* tests so that the CryptoAPI-Bench can be used to evaluate dynamic approaches.

```
1 package com.google.api.client.testing.http;
2 class HttpTesting {
3   static String SIMPLE_URL = "http://google.com"
4   public HttpTesting() {
5     GenericUrl url = new GenericUrl(SIMPLE_URL);
6   } ...
```

```
1 package com.adjust.sdk;
2 class AdjustFactory {
3   public static void useTestConnectionOptions() {
4     con.setHostnameVerifier(new HostnameVerifier() {
5       public boolean verify(String h, SSLSession s)
6       { return true; } ...
```

Fig. 11. Examples of false positives for rules *R-22* and *R-24* for CryptoGuard [6].

(a) Original CryptoAPI-Bench [26]

Rule ID	CryptoGuard [6] TP	TN	FP	FN	CRYLOGGER TP	TN	FN
R-01	24	1	4	0	24	5	0
R-02	30	1	5	0	30	6	0
R-03	6	1	1	0	6	2	0
R-05	5	2	1	2	7	3	0
R-07	8	1	1	0	8	2	0
R-10	7	1	1	0	7	2	0
R-13	5	1	1	2	7	2	0
R-16	7	2	1	1	8	3	0
R-17	13	1	2	1	14	3	0
R-18	1	1	0	0	1	1	0
R-19	4	0	1	1	5	1	0
R-22	6	2	1	0	6	3	0
R-23	7	2	1	0	7	3	0
R-24	1	1	0	0	1	1	0
R-25	3	0	0	0	3	0	0
R-26	4	0	0	0	4	0	0
Total	131	17	20	7	138	37	0

(b) Modified CryptoAPI-Bench

Rule ID	CryptoGuard [6] TP	TN	FP	FN	CRYLOGGER TP	TN	FN
R-01	28	1	4	0	24	5	4
R-02	35	1	5	0	30	6	5
R-03	7	1	5	0	6	6	1
R-05	6	2	1	2	7	3	1
R-07	9	1	1	0	8	2	1
R-10	8	1	1	0	7	2	1
R-13	6	1	1	2	7	2	1
R-16	8	2	1	1	8	3	1
R-17	14	1	2	1	14	3	1
R-18	1	1	0	0	1	1	0
R-19	5	0	1	1	5	1	1
R-22	7	2	1	0	6	3	1
R-23	8	2	1	0	7	3	1
R-24	1	1	0	0	1	1	0
R-25	3	0	0	0	3	0	0
R-26	4	0	0	0	4	0	0
Total	150	17	24	7	138	41	19

(c) New Tests

Rule ID	CRYLOGGER TP	TN	FN
R-04	4	2	1
R-06	6	2	1
R-08	6	2	1
R-09	6	2	1
R-11	7	2	1
R-12	1	1	1
R-14	7	2	1
R-15	7	2	1
R-20	5	1	1
R-21	5	1	1
Total	54	17	10

Fig. 12. Results for the CryptoAPI-Bench [26]. (a) Comparison of CryptoGuard [6] and *CRYLOGGER* on the original CryptoAPI-Bench. In this case, we made the benchmarks executable with a dynamic tool by adding a `main` to all benchmarks. (b) Comparison of CryptoGuard and *CRYLOGGER* on our modified version of the CryptoAPI-Bench. We added tests cases to (i) highlight the problem of false positives (Section IX) and (ii) show the limitations of dynamic approaches in activating paths that are rarely executed. (c) Benchmarks that we added for the rules supported only by *CRYLOGGER* on the modified CryptoAPI-Bench.