CryptoApi-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses

Sharmin Afrose, Sazzadur Rahaman, Danfeng (Daphne) Yao
Department of Computer Science
Virginia Tech
Blacksburg, Virginia
{sharminafrose, sazzad14, danfeng}@vt.edu

Abstract—**Several studies showed that misuses of cryptographic APIs are common in real-world code (e.g., Apache projects and Android apps). There exist several open-sourced and commercial security tools that automatically screen Java programs to detect misuses. In order to compare their accuracy and security guarantees, we develop a comprehensive benchmark named CryptoAPI-Bench. CryptoAPI-Bench consists of 171 unit test cases that cover basic cases, as well as complex cases, including interprocedural, field sensitive, multiple class test cases, and path sensitive data flow of misuse cases. The benchmark also includes correct cases for testing false positive rates. We evaluate CryptoAPI-Bench on four tools, namely, SpotBugs, CryptoGuard, CrySL, and Coverity and present their performance and comparative analysis. Our benchmark is useful for advancing state-of-the-art solutions in the space of misuse detection.**

I. Introduction

Theoretical security guarantees of cryptography are critically dependent on its threat model [32]. A deviation from the threat model of a cryptographic primitive is often refereed as *misuse* [26,47]. Various studies have shown that a vast majority of Java and Android applications misuse cryptographic libraries and APIs, causing devastating security and privacy implications. The most pervasive cryptographic misuses include exposed secrets (e.g., secret keys and passwords), predictable random numbers, use of insecure crypto primitives, vulnerable certificate verification [26,28,30,40,47,48].

Several studies showed that the prominent causes for cryptographic misuses are the deficiency in understanding of security API usage [15,40], complex API designs [15,43], the lack of cybersecurity training [40], insecure code generation tools [46] and insecure/misleading suggestions in Stack Overflow [16,40]. The reality is that most developers, with tight project deadlines and short product turnaround time, spend little effort on improving their knowledge or hardening their code for long-term benefits [19]. Recognizing these practical barriers, automatic cryptographic code generation [34] and misuse detection tools [47] play a significant role in assisting developers with writing and maintaining secure code.

The security community has produced several impressive static (e.g., CryptoLint [26], CrySL [35], FixDroid [45], MalloDroid [28], CryptoGuard [47]) and dynamic code screening tools (e.g., SMV-Hunter [50], and AndroSSL [29]) to detect API misuses in Java. The static analysis does not require a program to execute, rather it is performed on a version of the code (e.g., source code, intermediate representations or binary). Many abstract security rules are reducible to concrete program properties that are enforceable via generic static analysis techniques [18,47]. Consequently, static analysis tools have the potential to cover a wide range of security rules. In contrast, dynamic analysis tools require one to execute a program and spend a significant effort to trigger and detect specific misuse symptoms at runtime. Hence, dynamic analysis tools may be limited in their coverage. A code screening tool needs to be scalable with wide coverage. Thus, static analysis-based tools are usually more favorable than their dynamic counterparts.

However, a major weakness of static analysis tools is their tendency to produce false alerts. False alerts substantially diminished the value of a tool. To reduce the number of false positives, most of the static analysis tools offer a trade-off between completeness and scalability [39]. We define *completeness* as the ability to detect all the misuse instances and *scalability* as the ability to induce low computational overhead to analyze large code-bases. Designing tools that would produce fewer false positives and false negatives with smaller computational overhead help the real-world deployment.

To advance and monitor the scientific progress of domains to produce effective tools, a mechanism for comparative studies is required. Unfortunately, for automatic detection of cryptographic API misuses, no suitable mechanism or benchmark exists. Such a benchmark needs to have the following requirements.

- A benchmark needs to cover a wide range of misuse instances.
- A benchmark needs to cover interesting program properties (e.g., flow-, context-, field-, path-sensitivity).
- A benchmark's test cases should be written in easily compilable source codes so that both source code and binary code analysis tools can be easily evaluated.

None of the existing benchmarks follows these criteria (e.g., DroidBench [17], Ghera [41]). For example, DroidBench [17] only contains binaries. Ghera [41] has sources of provided Android apps. However, both DroidBench and Ghera barely cover cryptographic API misuses.

In this paper, we present CryptoApi-Bench, a comprehensive benchmark with 171 cases for comparing the quality of cryptographic vulnerability detection tools. CryptoApi-Bench covers 16 types of cryptographic misuses, including hardcoded secrets, improper SSL/TLS certificate validations, improper hostname verifications, insecure symmetric and asymmetric cryptographic primitives, insecure hash functions and insecure pseudo-random number generators. In CryptoApi-Bench, there are 40 basic test cases and 131 advanced test cases. Advanced test cases include interprocedural, field sensitive, multiple class, path-sensitive, and combined complex cases.

We run CryptoApi-Bench on four static analysis tools (i.e., SpotBugs, CryptoGuard, CrySL, and Coverity) and perform comparative analysis of these tools. These tools are *i)* capable of detecting cryptographic misuse vulnerabilities and *ii)* open-sourced and/or provide free evaluation license. CrySL [35] and CryptoGuard [47] are open-sourced research prototypes that are actively being maintained to improve their accuracy and coverage. SpotBugs [13] is also an actively maintained open source project, which is the successor of FindBugs. Coverity [4] is one of the most popular static analysis platform for decades.

Our main technical contributions are summarized as follows.

- We provide a benchmark named CryptoApi-Bench, which consists of 171 test cases covering 16 types of Cryptographic and SSL/TLS API misuse vulnerabilities. CryptoApi-Bench utilized various interesting program properties (e.g., field-, context-, and path-sensitivity) to produce a diverse set of test cases. Specifically, CryptoApi-Bench has 40 inter-procedural, 19 field-sensitive, 20 combined (a combination of inter-procedural and field-sensitivity) and 20 path-sensitive cases. Our benchmark is open-sourced and can be found at: https://github.com/CryptoAPI-Bench/CryptoAPI-Bench.
- We evaluate four static analysis tools that are capable of detecting cryptographic misuse vulnerabilities. Our experimental evaluation revealed some interesting insights. Most of the specialized tools to detect cryptographic misuses (e.g., CryptoGuard, CrySL) cover more rules and higher recall than general-purpose tool (e.g., SpotBugs, Coverity). However, Coverity raises fewer false positives than all of the others. Currently, none of these tools supports path-sensitive analysis.

The remainder of this paper is organized as follows. Section II describes the threat models. Section III outlines the design of CryptoApi-Bench. Section IV reviews existing cryptographic vulnerability detection tools. Section V presents the evaluation and performance analysis of the tools on CryptoApi-Bench. Section VII describes the related works. Discussion is given in Section VI. Finally, Section VIII concludes this paper.

II. Threat Model

In this section, we discuss the threat models of Java cryptographic misuses. We consider 16 Java cryptographic API misuse categories as cryptographic threat models and provide secure use cases of each misuse categories. For each threat model, we describe the corresponding cryptographic API, the reason for vulnerability and possible solution.

1) Cryptographic keys: For encryption, it is expected to use a secure and unpredictable key to convert plaintext to ciphertext using `javax.crypto.spec.SecretKeySpec` API that takes a byte array as input. If the Byte array is constant or hardcoded inside the code, the adversary can easily read the cryptographic key and may obtain sensitive information. Therefore, an unpredictable byte array should be used as a parameter in SecretKeySpec to generate a secure key.

2) Passwords in Password-based Encryption (PBE): Password-based Encryption (PBE) is a popular technique of generating a strong secret key using `javax.crypto.spec.PBEKeySpec` API. It takes three parameters (i.e., password, salt, and iteration count) while doing the Password-based Encryption to produce a random and unpredictable key. However, if hardcoded or constant password is used in the code, then malicious attackers may obtain the password and predict the key [26]. Therefore, an unpredictable password should be used as a parameter in PBEKeySpec.

3) Passwords in KeyStore: Cryptographic keys and certificates are sometimes stored using `java.security.KeyStore` API. The KeyStore employs a password to get access into the stored keys and certificates. However, if a hardcoded or constant password is used for KeyStore in the code, it poses a security threat of revealing keys and certificates stored in the KeyStore. Therefore, unpredictable random password should be used in KeyStore.

4) Hostname Verifier: HostnameVerifier in `javax.net.ssl.HostnameVerifier` API verifies the hostname by checking the hostname's authentication and identification. In some cases, verify() method of HostnameVerifier class is set to return true by default so that the verification method can quickly get past of an exception. However, this arrangement causes a security threat, where URL spoofing [14] attacks can be possible. URL spoofing makes it simpler for numerous cyber-attacks (e.g., identity theft, phishing). In Fig. 1, Line 2 returns true without verifying the hostname which is a major source of vulnerability.

```
1    public boolean verify(String hostname, SSLSession sslSession) {
2        return true;
3    }
```

Fig. 1: Skipping hostname verification in the `verify` method of `javax.net.ssl.HostnameVerifier` is insecure.

5) Certificate Validation: Empty methods are oftentimes implemented in `javax.net.ssl.X509TrustManager` interface to connect quickly and easily with clients and remote servers without any certificate validation. In that case, the TrustManager accepts and trusts every entity including the entity that is not signed by a trusted certificate authority. It enables Man-in-the-middle (MitM) attacks [6,28].

6) SSL Sockets: `javax.net.ssl.SSLSocket` connects a specific host to a specific port. However, before the connection, the hostname of the server should be verified and authenticated using `javax.net.ssl.HostnameVerifier` API. However, incorrect implementation omits the hostname verification when the socket is created [10,30].

7) Hypertext Transfer Protocol: HyperText Transfer Protocol (HTTP) sends a request to a server to retrieve a webpage. However, the HTTP allows hackers to intercept and read sensitive information [20]. Therefore, it is recommended to use HyperText Transfer Protocol Secure (HTTPS) that utilize a secured socket layer to encrypt sensitive information. In Fig. 2, a code snippet of secure and insecure URL usage using `java.net.URL` API is presented.

```
1 🐞 URL urlInsecure = "http://time.com/";
2    URL urlSecure = "https://www.google.com";
```

Fig. 2: Use of HTTP URL in `java.net.URL` API is inherently insecure.

8) Pseudorandom Number Generator (PRNG): The generation of a pseudorandom number using `java.util.Random` is vulnerable as the generated random number is not completely random, because it uses a definite mathematical algorithm (Knuth's subtractive random number generator algorithm [33]) that is proven to be insecure. To solve the problem, `java.security.SecureRandom` provides non-deterministic and unpredictable random numbers. In Fig. 3, Line 1 and Line 2 are insecure and secured initialization of random function, respectively.

```
1 🐞 Random r = new Random();
2    SecureRandom sr = new SecureRandom();
3    int insecureSeed = r.nextInt();
4    int secureSeed = sr.nextInt();
5    byte [] bytes = {(byte) 100};
6 🐞 sr.setSeed(bytes);
7    int insecureSeed2 = sr.nextInt();
```

Fig. 3: Generating seeds using `java.util.Random` is insecure. Random secure seeds can be generated using `java.security.SecureRandom` API.

9) Seeds in Pseudorandom Number Generator (PRNG): While using `java.security.SecureRandom`, if a constant or static seed is provided in SecureRandom, then it is possible to have the same outcome on every run. Therefore, due to the predictability concern, developers should use a non-deterministic random seed. In Fig. 3, Line 6 sets constant or hardcoded seed values into the `SecureRandom` which will produce predictable output.

10) Salts in Password-based encryption (PBE): In `javax.crypto.spec.PBEParameterSpec` API, it takes the salt and iteration count to set parameters for Password-based encryption. Using constant or static salts increases the possibility of a dictionary attack. The salt should be a random number that produces a random and unpredictable key. In Fig. 4, Line 2 takes a static/constant salt that is insecure to be used in PBEParameterSpec.

```
1    PBEParameterSpec pbeParamSpec = null;
2 🐞 byte[] salt = {(byte) 0xa2};
3 🐞 int count = 20;
4    pbeParamSpec = new PBEParameterSpec(salt, count);
```

Fig. 4: `javax.crypto.spec.PBEParameterSpec` API usage is insecure if iteration count is less than 1000 and salt is constant/predictable.

11) Mode of Operation: Among the modes of operation, Electronic Codebook (ECB) allows random access to each block. However, the ECB mode of operation is insecure to use in `javax.crypto.Cipher` as ECB-encrypted ciphertext can leak information about the plaintext. Instead of ECB, Cipher Block Chaining (CBC) or Galois/Counter Mode (GCM) is secure to use. Table I provides a list of insecure and secure modes of operation. In Fig. 5, Line 1 is an insecure ECB implementation. Instead of ECB, secure CBC mode of operation should be used.

12) Initialization Vector (IV): To enhance the security of cryptography, initialization vector (IV) is used during encryption and decryption with a secret key. Using static/constant initialization vectors introduce vulnerabilities. Therefore, it is suggested to use an unpredictable random initialization vector in `crypto.spec.IvParameterSpec API`.

TABLE I: Secure and insecure use of mode of operation (Rule 11), symmetric cipher (Rule 14), and cryptographic hash function (Rule 16)

Threat Models	Secure	Insecure
Mode of Operation	CBC, GCM	ECB
Symmetric Cipher	AES	DES, Blowfish, RC4, RC2, IDEA
Hash function	SHA-256	SHA1, MD5, MD4, MD2

13) Iteration Count in Password-based Encryption (PBE): In `javax.crypto.spec.PBEParameterSpec` API, it takes salt and iteration count to set parameters for Password-based Encryption (PBE). In PKCS #5 [42], it is suggested that the number of iteration should be more than 1000 to provide a reasonable security level. Therefore, it is required to use more than 1000 iteration to generate a secure password. In Fig. 4, Line 3 takes an iteration count of 20 that is insecure to be used in PBEParameterSpec.

TABLE II: CryptoApi-Bench: Summary of unit test cases. There are total 171 unit test cases with 40 basic cases and 131 advanced cases (interprocedural, field sensitive, combined case, path sensitive, miscellaneous and multiple class test cases). Total test cases per group and per threat model rules are summarized here. Details information are presented in Section III.

No.	**Threat Model Rules**	**Basic Cases**	**Two-Interproc.**	**Three-Interproc.**	**Field Sensitive**	**Combined Case**	**Path Sensitive**	**Misc.**	**Multiple Class**	**Total Cases per Rules**
1	Cryptographic Key	2	1	1	1	1	1	2	1	**10**
2	Password in PBE	3	1	1	1	1	1	2	1	**11**
3	Password in KeyStore	2	1	1	1	1	1	2	1	**10**
4	Hostname Verifier	2	0	0	0	0	0	0	0	**2**
5	Certificate Validation	3	0	0	0	0	0	0	0	**3**
6	SSL Socket	1	0	0	0	0	0	0	0	**1**
7	HTTP Protocol	2	1	1	1	1	1	0	1	**8**
8	PRNG	2	0	0	0	0	0	0	0	**2**
9	Seed in PRNG	3	2	2	2	2	2	2	2	**17**
10	Salt in PBE	2	1	1	1	1	1	1	1	**9**
11	Mode of Operation	2	1	1	1	1	1	0	1	**8**
12	Initialization Vector	2	1	1	1	1	1	2	1	**10**
13	Iteration in PBE	2	1	1	1	1	1	1	1	**9**
14	Symmetric Ciphers	6	5	5	5	5	5	0	5	**36**
15	Asymmetric Ciphers	1	1	1	0	1	1	0	1	**6**
16	Cryptographic Hash	5	4	4	4	4	4	0	4	**29**
Total Cases per Group		**40**	**20**	**20**	**19**	**20**	**20**	**12**	**20**	**171**

14) Symmetric Ciphers: In symmetric cryptography, the same key is used for encryption and decryption. Some symmetric ciphers, e.g., DES, Blowfish, RC4, RC2, IDEA are considered broken, as brute-force attack is possible for 64-bit ciphers. To overcome the attack, developers need to use AES which can support a block length of 128 bits and key lengths of 128, 192, and 256 bits [1]. Table I provides list of insecure and secure symmetric ciphers. In Fig. 5, Line 1 is an insecure implementation of a symmetric cipher.

```
1 🐞 Cipher cpr = Cipher.getInstance("DES/ECB/PKCS5Padding");
2    cpr.init(Cipher.ENCRYPT_MODE, key)
```

Fig. 5: Use case of `javax.crypto.Cipher` API is insecure if DES symmetric cipher and ECB mode of operation are used.

15) Asymmetric Ciphers: In asymmetric cryptography, two keys, i.e., public key and private key are used for encryption and decryption. However, some asymmetric ciphers, e.g., RSA are considered broken as brute-force attack is possible for 1024-bit ciphers. For this reason, developers need to use RSA with key size 2048 bits or higher.

16) Cryptographic Hash Functions: A cryptographic hash function takes an arbitrary message as input and produces fixed-length alphanumeric string as a character called hash value or message digest which is commonly employed in verifying message integrity, digital signature, and authentication. A cryptographic hash function is contemplated as broken if a collision can be observed, i.e., the same hash value is generated for two different inputs. Table I provides list of insecure and secure hash functions. The list of broken hash functions includes SHA1, MD4, MD5, and MD2. These hash functions produce collisions that cause cryptographic vulnerabilities. Therefore, the developers need to use a strong hash function, e.g., SHA-256. In Fig. 6, a code snippet of the broken hash function test case is shown.

```
🐞 MessageDigest md = MessageDigest.getInstance("MD5");
   md.update(message.getBytes());
   System.out.println(md.digest());
```

Fig. 6: Use case of java.security.MessageDigest API is insecure if MD5 is used. Hash function SHA-256 is secure.

III. Design of CryptoApi-Bench

In this section, we present the design of CryptoApi-Bench. We describe various test case groups based on the threat models described in Section II. These test groups contain sources of vulnerabilities that transmit through other procedures, classes, field variables or conditional statements.

We manually generate the unit test cases guided by 16 types of misuses in the threat model. We divide the test cases into two parts, i.e., basic cases and advanced cases. The advanced cases consist of several groups, i.e., interprocedural cases, field sensitive cases, combined cases, path sensitive cases, miscellaneous test cases, and multiple class test cases. These test cases incorporate the majority of possible variation in the perspective of program analysis to detect cryptographic vulnerability.

A. Basic Cases

Basic unit test cases of benchmark cover only the test cases where vulnerabilities rise within the same method. These benchmarks are quite straightforward cryptographic vulnerable test cases for the analysis tools to detect. For example, a vulnerable cipher is defined and used within the same method (Fig. 5). In CryptoApi-Bench, we include 40 basic cases as depicted in third column of Table II. The basic test cases cover all 16 threat models specified in Section II. Among these test cases, 27 test cases contain cryptographic vulnerability (true positive) and 13 test cases do not contain any cryptographic vulnerability (true negative).

B. Advanced Cases

The advanced cases of benchmark include complex cases in order to test a tool's capability of detecting the cryptographic vulnerability. In advanced cases, we consider three categories of vulnerable data flow analysis, including interprocedural, path sensitivity and field sensitivity. In addition, we consider a combined case consisting of inter-procedural and field sensitivity. We also designed some miscellaneous test cases and multiple class vulnerability test cases. The motivation of including advanced is to improve the precision of detection of the cryptographic vulnerability detection tools. In CryptoApi-Bench, we included 131 advanced cases. The details of advanced cases are depicted in the third to the tenth columns of Table II.

1) Interprocedural Cases: In interprocedural cases, the source of vulnerability originates from a different procedure or method and passes through as an argument. In our cases, we generate two-interprocedural and three-interprocedural cases. In two-interprocedural cases, the source of vulnerability is passed as an argument to another procedure's parameter and then use the vulnerable parameter in the Crypto APIs. In three-interprocedural cases, the source of vulnerability passes as an argument to another procedure and then passed again to another procedure. The goal of the interprocedural test cases is to check the detection tool's interprocedural data flow handling capability. An example code snippet of a two-interprocedural test case is presented in Fig. 7, where it is necessary to detect whether a secure or an insecure algorithm is passed in Line 1 and utilized in Line 3. CryptoApi-Bench contains 40 interprocedural test cases that are represented in the fourth and the fifth columns of Table II.

```
1 public void method2 (String cryptoAlgo);
2 { ...
3     Cipher cipher = Cipher.getInstance(cryptoAlgo)
4     ...
5 }
```

Fig. 7: Example code snippet of an interprocedural test case

2) Field Sensitive Cases: In field sensitive benchmark cases, the source of cryptographic vulnerabilities can be detected by the analysis tools if the tools are capable of performing field sensitive data flow analysis. In Fig. 8, the initialization of `Crypto` class sets the class variable `algo` with a secure or insecure crypto algorithm, which is used in Line 9. CryptoApi-Bench contains 19 field sensitive test cases that are represented in the sixth column of Table II.

TABLE III: Total number of insecure cryptographic API use cases (true positives) and secure cryptographic API use cases (true negatives) in CryptoApi-Bench's 171 test cases.

Threat Model	**Test Cases**	
	True Positive	**True Negative**
Cryptographic Key	7	3
Password in PBE	8	3
Password in KeyStore	7	3
Hostname Verifier	1	1
Certificate Validation	3	0
SSL Socket	1	0
HTTP Protocol	6	2
PRNG	1	1
Seed in PRNG	14	3
Salt in PBE	7	2
Mode of Operation	6	2
Initialization Vector	8	2
Iteration count in PBE	7	2
Symmetric Ciphers	30	6
Asymmetric Ciphers	5	1
Cryptographic Hash	24	5
Total Cases	**135**	**36**

```
 1 class Crypto {
 2     String algo
 3     public Crypto (String defAlgo) {
 4         algo = defAlgo;
 5     }
 6     public void encrypt(String passedAlgo, ... ) {
 7         passedAlgo = algo;
 8         ...
 9         Cipher cipher = Cipher.getInstance(passedAlgo);
10         ...
11     }
12 }
```

Fig. 8: Example code snippet of a field sensitive test case

3) Combined Cases: The combined cases are a bit more complex. In these cases, both interprocedural and field sensitivity are introduced, i.e., both Fig. 7 and Fig. 8 are incorporated to generate complicated test cases. CryptoApi-Bench has 20 combined test cases that are represented in the seventh column of Table II.

4) Path Sensitive Cases: In path sensitive test cases, conditional branch instructions are included in the code of test cases to evaluate proper detection of a precise source of a vulnerability. In Fig. 9, an example code snippet of a path sensitivity case is depicted. Depending on the choice parameter, the Cipher is getting the instance from a secure or an insecure cryptographic algorithm. The eighth column of Table II summarizes 20 path sensitive test cases of CryptoApi-Bench. Among the 36 true negatives shown in Table III, 20 test cases are path sensitive cases.

```
public void method1 (int choice) {
    ...
    Cipher cipher = Cipher.getInstance (cryptoInsecureAlgo1) ;
    if (choice > 1) {
        cipher = Cipher.getInstance (cryptoSecureAlgo2) ;
    }
    cipher.init (Cipher.ENCRYPT_MODE, key) ;
    ...
}
```

Fig. 9: Example code snippet of a path sensitive test case

5) Miscellaneous Cases: Miscellaneous test cases evaluate the tool's abilities to recognize irrelevant constraints and other interfaces, e.g., Map. In Fig. 10, the Map interface of Line 3-6 provides a secure key or insecure key depending on the choice variable. The Map indices (e.g., "a", "b") represent only index values, not security relevant. Similarly, in Line 8, the "UTF-8" represents byte encoding, not any constant or hard-coded value. CryptoAPI-Bench contains 12 miscellaneous test cases that are represented in the ninth column of Table II. Among the 36 true negatives shown in Table III, 3 test cases are miscellaneous test cases.

```
 1  public void method1 (String choice) {
 2     ...
 3     Map<String,String> hm = new HashMap<String, String>();
 4     hm.put("a", secureKeyString);
 5     hm.put("b", insecureKeyString);
 6     String keyString = hm.get(choice);
 7
 8     byte [] bytes = secureKeyString.getBytes("UTF-8");
 9     IvParameterSpec ivSpec = new IvParameterSpec(bytes);
10     ...
11  }
```

Fig. 10: Example code snippet of a false positive test case

6) Multiple Class Cases: In multiple class test cases, the source of vulnerabilities originates from different class methods and passes to another class. An example code snippet of a multiple class case is shown in Fig. 11. It is necessary to detect whether a secure or an insecure algorithm is passed in Line 4 in MultipleClass1 and utilized in Line 9 in MultipleClass2. CryptoAPI-Bench has 20 multiple class test cases that are represented in the tenth columns of Table II.

```
 1  public class MultipleClass1 {
 2     public void method1 (String passedAlgo) {
 3         MultipleClass2 mc = new MultipleClass2 ();
 4         mc.method2 (passedAlgo);
 5     }
 6  }
 7  public class MultipleClass2 {
 8     public void method2 (String cryptoAlgo) {
 9         Cipher cipher = Cipher.getInstance (cryptoAlgo);
10     }
11  }
```

Fig. 11: Example code snippet of a multiple class test case

In Table IV, the list of Java Crypto API libraries covered by CryptoAPI-Bench is shown. We consider mainly 15 such API libraries. In addition, threat models and the total number of test cases covered by each library are also denoted in the Table IV. Our CryptoAPI-Bench benchmark consisting of 171 unit test cases is made available in this GitHub repository: https://github.com/CryptoAPI-Bench/CryptoAPI-Bench.

IV. Existing Cryptographic Vulnerability Detection Tools

TABLE IV: List of Java Crypto API libraries used in CryptoAPI-Bench, the corresponding threat models and test cases per Java Crypto API library. For example, the first entry of the table states that threat model 1 is covered by Java Crypto API "javax.crypto.spec.SecretKeySpec" and there are a total of 10 test cases in CryptoAPI-Bench.

Java Crypto API Libraries	**Threat Models**	**Total Test Cases**
javax.crypto.spec.SecretKeySpec	1	10
javax.crypto.spec.PBEKeySpec	2	11
java.security.KeyStore	3	10
javax.net.ssl.HostnameVerifier	4	2
javax.net.ssl.X509TrustManager	5	3
java.security.cert.X509Certificate	5	3
javax.net.ssl.SSLSocket	6	1
java.net.URL	7	8
java.security.SecureRandom	8, 9	19
javax.crypto.spec.PBEParameterSpec	10, 13	18
javax.crypto.spec.IvParameterSpec	12	10
javax.crypto.KeyGenerator	14	36
javax.crypto.Cipher	11, 14, 15	50
java.security.KeyPairGenerator	15	6
java.security.MessageDigest	16	29

In this section, we summarize the vulnerability detection tools that we choose to run on CryptoAPI-Bench. We consider three criteria while choosing the analysis tools. (1) Open-sourced tools: The open-sourced vulnerability detection tools, i.e., CrySL [35], CryptoGuard [47], SpotBugs [13] are convenient to use as we are able to analyze their codes and understand the reason of their lack of performance. (2) Static analysis tools: We choose static analysis tools that can examine and detect vulnerability without executing the code. SpotBugs, CryptoGuard, CrySL and Coverity [4] are static analysis tools. (3) Free cryptographic vulnerability detection services: We consider Coverity as a provider of free cryptographic vulnerability detection service. Coverity is not open-sourced. However, Coverity provides online service to detect vulnerability.

We also consider GrammaTech [9], QARK [11] and FixDroid [45]. However, GrammaTech is a commercial tool. We were unable to access its trial version. The online SWAMP [25] contains GrammaTech tool to use that only supports vulnerability detection for C and C++. Therefore, we excluded GrammaTech from our list of tools. QARK is a tool that is mainly designed to capture security vulnerabilities in Android applications. FixDroid is built as a research prototype that is embedded as a plugin in Android Studio to conduct a usability study. Our investigation shows that the detection capability of FixDroid and QARK is limited. Though QARK has been maintained and updated, FixDroid has not been updated since 2017.

Therefore, we mainly focus on four tools, i.e., SpotBugs, CryptoGuard, CrySL and Coverity to evaluate on CryptoAPI-Bench.

1) SpotBugs: SpotBugs is a static analysis tool also for capturing deficiencies in Java code. This analysis tool is a successor of another popular static analysis tool named FindBugs, which is no longer maintained [7]. The tool is

TABLE V: Generated alert keywords for each threat model rule in each cryptographic vulnerability detection tool (SpotBugs, CryptoGuard, CrySL and Coverity). For example, for threat model 16 (i.e., Cryptographic Hash), the generated alert keywords in tools are WEAK_MESSAGE_DIGEST, broken hash scheme, ConstraintError, RISKY_CRYPTO, respectively.

Threat Model	SpotBugs	CryptoGuard	CrySL	Coverity
1	HARD_CODE_PASSWORD	Constant keys	RequiredPredicateError	HARDCODED_CREDENTIALS
2	HARD_CODE_PASSWORD	Constant keys	NeverTypeOfError	HARDCODED_CREDENTIALS
3	HARD_CODE_PASSWORD	Predictable password	NeverTypeOfError	HARDCODED_CREDENTIALS
4	WEAK_HOSTNAME_VERIFIER	Manually verify hostname	—	BAD_CERT_VERIFICATION
5	WEAK_TRUST_MANAGER	Untrusted TrustManager	–	BAD_CERT_VERIFICATION
6	–	Does not manually verify socket	ConstraintError	RESOURCE_LEAK
7	—	HTTP protocol	—	—
8	PREDICTABLE_RANDOM	Untrusted PRNG	–	—
9	—	Predictable Seed	TypeStateError	PREDICTABLE_RANDOM_SEED
10	—	Constant Salt	RequiredPredicateError	—
11	CIPHER_INTEGRITY	Broken crypto scheme	ConstraintError	RISKY_CRYPTO
12	STATIC_IV	Constant IV	RequiredPredicateError	—
13	—	<1000 iteration	ConstraintError	—
14	CIPHER_INTEGRITY	Broken crypto scheme	ConstraintError	RISKY_CRYPTO
15	—	Export grade public key	ConstraintError	—
16	WEAK_MESSAGE_DIGEST	Broken hash scheme	ConstraintError	RISKY_CRYPTO

built based on a plugin structure. The tools detect defects by utilizing visitor pattern in class files or bytecodes of Java, state machine, flags. We use the SpotBugs tool (version 3.1.0) available online in SWAMP [25].

2) CryptoGuard: CryptoGuard [47] is a static analysis tool that is operated based on program slicing with novel language-based refinement algorithms. It significantly reduces the false positive rate which is a typical problem for static analysis. Furthermore, CryptoGuard covers 16 cryptographic rules and achieves high precision. The authors showed screening a large number of Apache projects and Android apps to present their high precision rate and low false positive rate. We run the experiment on CryptoGuard (version 03.04.00) available on GitHub [5].

3) CrySL: CrySL [35] is a definition language that is implemented as a compiler, which interprets particular specification to demand driven static analysis. For Java Cryptography Architecture (JCA), CrySL outlined rulesets that report vulnerabilities in code. CrySL is open-sourced and we run the experiment on CrySL (version 2.0) available on GitHub [3].

4) Coverity: Coverity is a commercial tool that analyzes the vulnerabilities of codes. Unlike other tools, it takes the source code and performs its analysis. The Coverity analysis tool is available to use online [4]. We perform the latest analysis using Coverity on 29th March 2019.

V. CryptoApi-Bench Evaluation and Analysis

In this section, we present and analyze our evaluation results on four cryptographic misuse detection tools, i.e., SpotBugs, CryptoGuard, CrySL and Coverity. We show the experimental setup, evaluation criteria and analysis results using CryptoApi-Bench.

A. Experimental Setup

We evaluate mainly four cryptographic analysis tools, i.e., SpotBugs [13], CryptoGuard [47], CrySL [35], Coverity [4] on CryptoApi-Bench. For SpotBugs, we perform the analysis by uploading the CryptoApi-Bench's JAR file to the online SpotBugs tool (version 3.1.0) from Software Assurance Marketplace (SWAMP) [25]. For CryptoGuard, it is open-sourced and available in GitHub repository: https://github.com/CryptoGuardOSS/cryptoguard. We perform the analysis of CryptoGuard (version 03.04.00)[1] on the CryptoApi-Bench's JAR file. CrySL is another open-sourced tool that is available in GitHub repository: https://github.com/CROSSINGTUD/CryptoAnalysis. For CrySL, the analysis also run on CryptoApi-Bench's JAR file with CrySL (version 2.0)[2]. We follow the instructions from GitHub to set up the environment of CryptoGuard and CrySL in our machine to perform the analysis. Coverity is an online commercial tool [4] that takes GitHub link and compressed code files of CryptoApi-Bench in order to start analysis execution. The latest analysis run on Coverity was performed on 29th March 2019.

We initially also consider two other tools, i.e., QARK [11] and FixDroid [45]. QARK is open-sourced available in GitHub repository: https://github.com/linkedin/qark. We set up the environment of QARK[3] and analyze the source codes. For FixDroid, we installed the FixDroid plugins [8] (version 1.2.1) in Android Studio.

B. Evaluation Criteria

We evaluate the vulnerability detection tools by running these tools on CryptoApi-Bench. After performing the analysis, we capture true positive, false positive and false negative rates from the corresponding tool's result log. As our purpose is to detect cryptographic vulnerability detection, we consider only cryptographic misuse alerts and discard others. In Table V, we present the alert keywords that detection tools use while showing a specific cryptographic misuse. For example, the last entry, i.e., threat model 16 (misuse of the cryptographic hash function), is depicted in SpotBugs

[1] Commit id c046892.

[2] Commit id 5f531d1.

[3] Commit id ba1b265.

TABLE VI: CryptoApi-Bench comparison of SpotBugs, CryptoGuard, CrySL and Coverity on all 16 rules with CryptoApi-Bench's 171 test cases. There are 36 secure API use cases (13 in basic and 23 in advanced), which a tool should not raise any alerts on. GTP stands for ground truth positive, which is the number of insecure API use cases in the benchmark. Findings of the table are reported in Section V-C.

No.	Threat Models	GTP	SpotBugs		CryptoGuard		CrySL		Coverity	
			TP	FP	TP	FP	TP	FP	TP	FP
1	Cryptographic Key	7	0	1	5	1	0	9	5	1
2	Password in PBE	8	2	0	6	1	0	9	7	1
3	Password in KeyStore	7	1	1	7	1	0	10	5	1
4	Hostname Verifier	1	1	0	1	0	–	–	1	0
5	Certificate Validation	3	3	0	3	0	–	–	3	0
6	SSL Socket	1	–	–	1	0	1	0	1	0
7	HTTP Protocol	6	–	–	6	1	–	–	–	–
8	PRNG	1	1	0	1	0	–	–	–	–
9	Seed in PRNG	14	–	–	12	2	0	1	1	2
10	Salt in PBE	7	–	–	6	1	6	1	–	–
11	Mode of Operation	6	1	2	6	1	5	1	1	1
12	Initialization Vector	8	7	2	7	1	8	0	–	–
13	Iteration Count in PBE	7	–	–	5	1	5	2	–	–
14	Symmetric Cipher	30	5	10	30	5	25	5	4	4
15	Asymmetric Ciphers	5	–	–	4	1	5	1	–	–
16	Cryptographic Hash	24	4	8	24	4	20	4	4	4
	Total	**135**	**25**	**24**	**124**	**20**	**75**	**43**	**32**	**14**

as "WEAK_MESSAGE_DIGEST", CryptoGuard as "Broken hash scheme", CrySL as "ConstraintError", Coverity as "RISKY_CRYPTO". Table V can assist developers to understand which keyword they should search in the result log to find a specific type of vulnerability. In the following, we provide a brief description of our process of identification of true positive, false positive and false negative alerts.

1) True positive (TP): If a tool generates alert due to the correct reason while screening any specific vulnerable unit test case in CryptoApi-Bench, then the event is considered as a true positive.

2) False positive (FP): The false positive alert can be captured from two different scenarios. If an alert raised by a tool is unexpected (i.e., does not exist in a specific unit test case), then the alert is a false positive. In addition, if a tool gives an inaccurate reason for an expected alert, then it is also considered as a false positive.

3) False negative (FN): A vulnerable test case may not be detected by the evaluation tools. This missed detection is characterized as a false negative.

After analyzing the results by determining the true positive (TP), false positive (FP) and false negative (FN) values, we compute the false positive rate (FPR), false negative rate (FNR), recall and precision to determine the performance of the tools.

C. Analysis of Results

In this section, we describe CryptoApi-Bench evaluation findings on each detection tool based on the result log and performance analysis. Table VI presents the number of true positive and false positive vulnerability threat detection captured by the tools for 16 cryptographic threat models. There are only 6 common cryptographic threat models detected by all tools. To ensure fairness in comparison, we consider only these 6 common cryptographic rules while finding the comparative analysis results of tools based on the basic and advanced benchmark in Table VII and Table VIII, respectively. The analysis results are presented in terms of false positive rate (FPR), false negative rate (FNR), recall and precision.

Analysis Overview: Table VI shows that among the 16 specified high impact cryptographic threat models in Section II, the cryptographic vulnerability detection tools are able to detect a subset of rules.

- SpotBugs, CryptoGuard, CrySL, Coverity covers 10, 16, 12, 10 cryptographic thread models, respectively.
- In total, the benchmark contains 136 vulnerable test cases and among these true positive cases, SpotBugs, CryptoGuard, CrySL, Coverity detects 25, 124, 75, 32 cases, respectively.
- In addition, SpotBugs, CryptoGuard, CrySL, Coverity also generate 24, 20, 43, 14 false alarms, respectively that are included as false positive cases.

We also initially consider QARK and FixDroid. QARK can detect 18 true positive vulnerabilities in total covering both basic and advanced cases covering 4 threat rules, i.e., symmetric cipher, certificate validation, mode of operation and seed in PRNG. QARK generates 4 false alarms in path sensitive test cases of CryptoApi-Bench. FixDroid can detect 4 true positive vulnerabilities in basic cases, i.e., insecure use of symmetric cipher (DES only), mode of operation, iteration count of PBE, and hostname verifier. FixDroid is not designed to capture threats in advanced cases. Therefore, it produces 69 false alarms for 7 rules, i.e., symmetric ciphers, hostname verifier, mode of operation, asymmetric cipher, cryptographic key, iteration count, and salt in PBE. However, originally QARK is designed to detect vulnerabilities in Android applications and FixDroid is designed to use as a plugin. Therefore, due to the limited detection capability of cryptographic misuses, we exclude FixDroid and QARK from our comparison.

1) Analysis on Basic Benchmark: Table VII shows the performance analysis result of four detection tools on six common cryptographic threat models based on the basic benchmark. We capture the following findings based on Table VII.

- SpotBugs does not produce any false positive errors. It detects all cases except one. SpotBugs is not designed to capture threats in the basic case of the vulnerable cryptographic key threat model.
- CryptoGuard also does not produce any false positive errors. It misses a detection in the vulnerable password in PBE cases due to the orthogonal method invocation.
- CrySL produces 6 false positive errors due to maintaining strict rules in Crypto APIs of the cryptographic key, password in PBE, password in KeyStore. These Crypto APIs always raise an alert if type java.lang.String is used as their passing argument whether the string is securely generated or not.
- Coverity does not generate any false positive errors. It can successfully detect every vulnerability except one. Coverity is not designed to capture IDEA as a vulnerable cryptographic algorithm. However, it can capture other vulnerable cryptographic algorithms.

In summary, for all basic cases, SpotBugs, CryptoGuard and Coverity generates a precision of 100%. For CrySL, it produces some false positives and hence generates precision of 62.50%.

2) Analysis on Advanced Benchmark: Table VIII shows the performance analysis result of four detection tools on six common cryptographic threat models based on the advanced benchmark. We capture the following findings based on Table VIII.

- In the prospect of path sensitivity, it is obvious that none of the cryptographic vulnerability detection tools is path-sensitive in their static analysis. SpotBugs, CryptoGuard, CrySL, Coverity generate 10, 13, 13, 12 false positive alerts, respectively. The possible reason for the false positive alert is that for the concerned variable, a container is defined to store all values of the concerned variable. There is no ordered list which shows the latest assignment. Therefore, alerts will be raised if the container contains any vulnerable value that is intended to be used in the Crypto API. A significant reason for having a high false positive rate of 57.89%, 44.83%, 66.67%, 42.86%, respectively is because of the tools being path insensitive.
- SpotBugs is not designed to capture vulnerability threat in advanced cases. Therefore, it produces a huge 100% false negative rate.
- SpotBugs produces 12 false positives for combined cases. In combined cases, SpotBugs failed to detect the source of vulnerability using both interprocedural and field sensitive analysis. For example, in Symmetric Cipher cases, instead of showing correct "CIPHER_INTEGRITY" alert, it produces incorrect "HARD_CODE_PASSWORD" alert.
- CryptoGuard misses 3 vulnerabilities due to the orthogonal method invocation strategy [47]. These cases contain specific built-in methods (e.g., String.valueOf(), etc) that CryptoGuard chooses not to process for efficiency purposes.
- CrySL produces incorrect "RequiredPredicateError", "NeverTypeOfError" alerts for the cryptographic key, password in PBE, password in KeyStore threat model group test cases that contribute to generate a high false positive rate of 66.67%. The reason is that the cryptographic APIs used in these cases follow strict rules in CrySL and search for the type java.lang.String as their passing argument. Therefore, if we use secure unpredictable key String or password String as an argument for crypto APIs, it generates incorrect alerts.
- CrySL produced false positives in combined cases in Version 1.0. Comparison result conducted on CrySL version 1.0 with a small number of cases of CryptoApi-Bench can be found in [47]. They fixed most of the issues of rising false positive alert in Version 2.0. However, in Version 2.0, CrySL does not show any true positives alerts for combined cases.
- Coverity is not designed to detect vulnerable ciphers and cryptographic hash functions in advanced cases. That is the reason for having high false negative values and generating high FNR in Coverity.
- Coverity shows correct results for cryptographic key, password in PBE, password in KeyStore threat model group test as "HARDCODED_CREDENTIALS" for interprocedural, combined cases, multiple class cases. However, in miscellaneous cases and field sensitive cases, Coverity produces 0 and 1 true positive alerts, respectively. Coverity is a closed sourced detection tool. Therefore, we are unable to confirm the reason for the incorrect detection cases.

In summary, for all of the advanced cases, SpotBugs is not designed to identify the advanced vulnerability threats correctly. Therefore, the precision rate is 0%. CryptoGuard detects fairly well (missed only 3 cases) among all detection tools with precision 83.33%. For CrySL produce precision of 63.01%. Coverity generates a precision of 52.00%.

VI. Discussion

Tool insights. Most of these tools (except CryptoGuard) do not cover all the vulnerabilities in our threat model (Table VI). However, their methodologies can be extended to cover most of these vulnerabilities. For example, the technique that Coverity uses to detect constant cryptographic keys can be transferred to detect predictable IVs or fewer iteration counts.

The main differences among different tools are within their approach to trade-off among false positives, false negatives and scalability. Our experimental evaluation reveals that all of these tools produce a number of false positives and false

TABLE VII: CryptoApi-Bench comparison of SpotBugs, CryptoGuard, CrySL and Coverity on six common threat model rules with CryptoApi-Bench's common 20 basic cases. TP, FP, FN, FPR, FNR stand for true positive, false positive and false negative, false positive rate, false negative rate, respectively. Findings of the table are reported in Section V-C1.

Rules	Basic Cases	Vulnerable	SpotBugs			CryptoGuard			CrySL			Coverity		
			TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Cryptographic Key	1.1.1	Y	0	0	1	1	0	0	0	1	1	1	0	0
	1.1.2	N	0	0	0	0	0	0	0	1	0	0	0	0
Password in PBE	2.1.1	Y	1	0	0	1	0	0	0	1	1	1	0	0
	2.1.2	Y	1	0	0	0	0	1	0	0	1	1	0	0
	2.1.3	N	0	0	0	0	0	0	0	1	0	0	0	0
Password in KeyStore	3.1.1	Y	1	0	0	1	0	0	0	1	1	1	0	0
	3.1.2	N	0	0	0	0	0	0	0	1	0	0	0	0
Mode of Operation	11.1.1	Y	1	0	0	1	0	0	1	0	0	1	0	0
	11.1.2	N	0	0	0	0	0	0	0	0	0	0	0	0
Symmetric Ciphers	14.1.1	Y	1	0	0	1	0	0	1	0	0	1	0	0
	14.1.2	Y	1	0	0	1	0	0	1	0	0	1	0	0
	14.1.3	Y	1	0	0	1	0	0	1	0	0	1	0	0
	14.1.4	Y	1	0	0	1	0	0	1	0	0	1	0	0
	14.1.5	Y	1	0	0	1	0	0	1	0	0	0	0	1
	14.1.6	N	0	0	0	0	0	0	0	0	0	0	0	0
Cryptographic Hash	16.1.1	Y	1	0	0	1	0	0	1	0	0	1	0	0
	16.1.2	Y	1	0	0	1	0	0	1	0	0	1	0	0
	16.1.3	Y	1	0	0	1	0	0	1	0	0	1	0	0
	16.1.4	Y	1	0	0	1	0	0	1	0	0	1	0	0
	16.1.5	N	0	0	0	0	0	0	0	0	0	0	0	0
Result		**FPR (%)**	**0**			**0**			**50**			**0**		
		FNR (%)	**7.14**			**7.14**			**28.57**			**7.14**		
		Recall (%)	**92.86**			**92.86**			**71.43**			**92.86**		
		Precision (%)	**100.00**			**100.00**			**62.50**			**100.00**		

TABLE VIII: CryptoApi-Bench comparison of SpotBugs, CryptoGuard, CrySL and Coverity on six common threat model rules with CryptoApi-Bench's common 84 advanced cases. TP, FP, FN, FPR, FNR stand for true positive, false positive and false negative, false positive rate, false negative rate, respectively. Findings of the table are reported in Section V-C2.

Advanced Test Cases	True Positive Count	True Negative Count	SpotBugs			CryptoGuard			CrySL			Coverity		
			TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
Two-Interprocedural	13	0	0	0	13	12	0	1	10	3	3	3	0	10
Three-Interprocedural	13	0	0	0	13	12	0	1	10	3	3	3	0	10
Field Sensitive	13	0	0	0	13	13	0	0	10	2	3	1	0	12
Combined Case	13	0	0	12	13	12	0	1	0	2	13	3	0	10
Path Sensitive	0	13	0	10	0	0	13	0	0	13	0	0	12	0
Miscellaneous Cases	3	3	0	0	3	3	0	0	0	6	3	0	0	3
Multiple Class methods	13	0	0	0	13	13	0	0	10	3	3	3	0	10
Results		**FPR (%)**	**57.89**			**44.83**			**66.67**			**42.86**		
		FNR (%)	**100**			**4.41**			**41.18**			**80.88**		
		Recall (%)	**0**			**95.59**			**58.82**			**19.12**		
		Precision (%)	**0.00**			**83.33**			**55.56**			**52.00**		

negatives. For CryptoGuard, the main source of the false negatives is the clipping of orthogonal method invocations. The approach of orthogonal method invocation is introduced to improve precision and scalability. The main focus of CrySL is to provide a language to specify a class of cryptographic misuse vulnerabilities that can be detected using a generic detection engine. A prime reason behind the false positives is the strictness of the rule definitions that is inherited from the language itself. For example, CrySL raises an alert if a cryptographic key is not generated using a key generator. However, one can legitimately reuse a previously generated key, which CrySL would mistakenly detect as a vulnerability. An impressive aspect of CrySL is that it is constantly being maintained and updated to improve its accuracy. The methodology of SpotBugs is inherently limited to detect advanced cases as they use patterns to detect most of the vulnerabilities.

None of these tools are path-sensitive, i.e., all raising false alerts in path sensitive cases. A possible reason for this failure is that the existing path-sensitive analysis techniques are usually costly, i.e., high runtime complexity.

Our limitation. CryptoApi-Bench cannot be used to evaluate the scalability. All of our test cases are lightweight by design. Our primary focus is to produce easily readable test cases which demand minimal code to express complex program properties.

Currently, our benchmark does not have any cases that involve Java reflection APIs. The primary reason is that the use of Java reflection during cryptographic coding is highly unlikely. Consequently, none of the existing open-sourced tools is designed to detect such cases. However, we plan to include new cases that leverage Java reflection APIs to induce cryptographic misuse vulnerabilities.

VII. Related Work

Vulnerability detection benchmarks. AndroZoo++ [37] is a collection of over eight million Android apps [2] that drives a lot of security, software engineering and malware analysis research. However, vulnerabilities in these apps are not documented, hence not suitable for vulnerability detection benchmarking purpose.

DroidBench [17], a benchmark containing vulnerable android apps, fills the gap by providing specific vulnerability locations within the benchmark. Till date, DroidBench is one of the most popular benchmark to evaluate the performance of vulnerability detection tools in Android literature. In total, DroidBench has 119 APKs from 13 categories [4]. Categories include vulnerabilities that use field and object sensitivity, inter-app communication, inter-component communication, android life-cycle, reflection, etc. However, DroidBench *i)* does not cover cryptographic misuse vulnerabilities and *ii)* does not have source code. To the best of our knowledge, Ghera [41] is the only Android app benchmark that contains app source code. Like DroidBench, most of the vulnerabilities in Ghera are specific to Android apps and barely contains any cryptographic misuse vulnerabilities. To be specific, CryptoApi-Bench and Ghera have only 2 types of vulnerabilities in common.

OWASP Benchmark [24] is fundamentally designed to capture eleven cybersecurity vulnerabilities. However, among the detected vulnerabilities, it builds to address only three Java cryptographic vulnerabilities, i.e., weak encryption algorithm, weak hash algorithm, and a weak random number.

SonarSource [12] released a set of vulnerability samples that can be useful to check for coverage of vulnerability categories. However, these straightforward samples cannot be used to determine the scientific rigor of a tool.

Other benchmarks. The DaCapo benchmarks [22] are designed to evaluate the performance of various components of Java virtual machine (JVM), Garbage collection (GC), Just-in-time (JIT) compiler itself. BugBench [38] is a benchmark to find C/C++ bugs that contains 17 real-world applications. BugBench mostly covers various memory, concurrency and semantic bugs. To detect bugs in the multi-threaded Java programs, a benchmark and framework has been proposed [27,31]. For dynamic software updating system, a standardized benchmark system proposed to check the system's practicality, flexibility, and usability [49]. ManyBugs and IntroClass benchmarks designed to evaluate various C/C++ code repair techniques [36]. Most of the defects in ManyBugs and IntroClass do not impact security, e.g., in the ManyBugs benchmark, more than half of the instances impact correctness, not necessary security.

VIII. Conclusion and Future Work

We believe that for scientific, in-depth and reproducible comparisons benchmark is an important component. In this paper, we present the first benchmark, named CryptoApi-Bench to evaluate the detection accuracy and security guarantees of various cryptographic misuse detection tools. Our benchmark is open-sourced and is available at: https://github.com/CryptoAPI-Bench/CryptoAPI-Bench. We evaluated four static analysis tools that are capable of detecting cryptographic misuses. Our evaluation revealed some interesting insights, i.e., *i)* tools that are specialized to detect cryptographic misuses (e.g., CryptoGuard, CrySL) cover more rules and higher recall than general purpose tools (e.g., SpotBugs, Coverity), *ii)* none of the existing tools is path-sensitive.

We are actively working on expanding CryptoApi-Bench by adding new rules, test cases and covering new cryptographic APIs. In future, we plan to achieve following goals.

- To test the scalability of the tools, our ongoing work is to extend the existing CryptoApi-Bench by integrating Apache benchmark that can be extrapolated to applications on real-world code.
- To motivate the research of cryptographic misuse detection tools for other platforms, we plan to extend CryptoApi-Bench to cover other popular languages, e.g., Python.
- Other non-cryptographic API misuses (e.g., Android APIs to access sensitive information (location, IMEI, passwords, etc.) [23,44], fingerprint protection [21], cloud service APIs for information storage [51]) are also proven to cause catastrophic security consequences. We also plan to include the misuses of these critical non-cryptographic APIs.

IX. Acknowledgement

This work has been supported by the Office of Naval Research under Grant ONR-N00014-17-1-2498.

References

[1] AES Encryption. https://aesencryption.net/. Online; Last accessed: April 3, 2019.

[2] AndroZoo. https://androzoo.uni.lu/. Online; Last accessed: April 3, 2019.

[3] Cognicrypt_SAST: CrySLtoStatic Analysis Compiler. https://github.com/CROSSINGTUD/CryptoAnalysis. Online; Last accessed: April 3, 2019.

[4] Coverity Static Application Security Testing (SAST). https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html. Online; Last accessed: April 1, 2019.

[5] CryptoGuard. https://github.com/CryptoGuardOSS/cryptoguard. Online; Last accessed: April 3, 2019.

[6] Find Security Bugs. https://find-sec-bugs.github.io/. Online; Last accessed: April 3, 2019.

[7] FindBugs is now SpotBugs. https://github.com/findbugsproject/findbugs. Online; Last accessed: April 1, 2019.

[8] FixDroid. https://plugins.jetbrains.com/plugin/9497-fixdroid. Online; Last accessed: April 1, 2019.

[9] GRAMMATECH. https://www.grammatech.com/. Online; Last accessed: April 1, 2019.

[4] Commit id 0fe281b.

[10] Hostname Verification to SSL Socket. https://www.thecodingforums.com/threads/adding-hostname-verification-to-sslsocket.958287/. Online; Last accessed: April 3, 2019.

[11] Quick Android Review Kit (QARK). https://github.com/linkedin/qark. Online; Last accessed: April 1, 2019.

[12] SonarSource, making Code Analyzers. https://rules.sonarsource.com/. Online; Last accessed: April 3, 2019.

[13] SpotBugs: Find Bugs in Java Programs. https://spotbugs.github.io/. Online; Last accessed: April 3, 2019.

[14] URL Spoofing. http://www.securitysupervisor.com/security-q-a/network-security/262-what-is-url-spoofing. Online; Last accessed: April 3, 2019.

[15] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy, SP'17, San Jose, CA, USA, May 22-26*, pages 154–171, 2017.

[16] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *IEEE Symposium on Security and Privacy, SP'16, San Jose, CA, USA, May 23-25*, pages 289–305, 2016.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14*, pages 259–269, 2014.

[18] K. Ashcraft and D. R. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy, (SP'02), Berkeley, California, USA, May 12-15*, pages 143–159, 2002.

[19] H. Assal and S. Chiasson. Security in the Software Development Lifecycle. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS'18, Baltimore, MD, USA, August 12-14*, pages 281–296, 2018.

[20] C. A. Barton, G. A. Clarke, and S. Crowe. Transferring Data via a Secure Network Connection, 2006. US Patent 7,093,121.

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

[22] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'06, Portland, Oregon, USA, October 22-26*, pages 169–190, 2006.

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

[24] E. Burato, P. Ferrara, and F. Spoto. Security Analysis of the OWASP Benchmark with Julia. *The Italian Conference on CyberSecurity (ITASEC)*, 17, 2017.

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

[26] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *ACM Conference on Computer and Communications Security, CCS'13*, pages 73–84, 2013.

[27] Y. Eytani, K. Havelund, S. D. Stoller, and S. Ur. Towards a Framework and a Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, 19(3):267–279, 2007.

[28] S. Fahl, M. Harbach, T. Muders, M. Smith, L. Baumgärtner, and B. Freisleben. Why Eve and Mallory Love Android: An Analysis of Android SSL (in) Security. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 50–61, 2012.

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

[30] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18*, pages 38–49, 2012.

[31] K. Havelund, S. D. Stoller, and S. Ur. Benchmark and Framework for Encouraging Research on Multi-Threaded Testing Tools. In *Proceedings International Parallel and Distributed Processing Symposium, IPDPS'03*, page 286. IEEE, 2003.

[32] C. Herley and P. C. van Oorschot. SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit. In *IEEE Symposium on Security and Privacy, SP'17, San Jose, CA, USA, May 22-26*, pages 99–120, 2017.

[33] D. E. Knuth. *Art of Computer Programming, volume 2: Seminumerical Algorithms*. Addison-Wesley Professional, 2014.

[34] S. Krüger et al. CogniCrypt: Supporting Developers in using Cryptography. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'17*, pages 931–936, 2017.

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

[36] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[37] L. Li, J. Gao, M. Hurier, P. Kong, T. F. Bissyandé, A. Bartel, J. Klein, and Y. L. Traon. AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. *arXiv preprint arXiv:1709.05281*, 2017.

[38] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, volume 5, 2005.

[39] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna. DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers. In *26th USENIX Security Symposium, USENIX Security'17, Vancouver, BC, Canada, August 16-18, 2017*, pages 1007–1024, 2017.

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

[41] J. Mitra and V. Ranganath. Ghera: A Repository of Android App Vulnerability Benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering, PROMISE'17, Toronto, Canada, November 8, 2017*, pages 43–52, 2017.

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

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

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

[45] D. C. Nguyen et al. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *ACM Conference on Computer and Communications Security, CCS'17*, pages 1065–1077, 2017.

[46] M. Oltrogge, E. Derr, C. Stransky, Y. Acar, S. Fahl, C. Rossow, G. Pellegrino, S. Bugiel, and M. Backes. The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators. In *IEEE Symposium on Security and Privacy, SP'18, San Francisco, California, USA, 21-23 May*, pages 634–647, 2018.

[47] 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 *ACM Conference on Computer and communications security, CCS'19*, London, UK, Nov. 2019.

[48] S. Rahaman and D. Yao. Program Analysis of Cryptographic Implementations for Security. In *IEEE Cybersecurity Development, SecDev'17, Cambridge, MA, USA, September 24-26*, pages 61–68, 2017.

[49] E. K. Smith, M. Hicks, and J. S. Foster. Towards Standardized Benchmarks for Dynamic Software Updating Systems. In *4th International Workshop on Hot Topics in Software Upgrades (HotSWUp'12)*, pages 11–15. IEEE, 2012.

[50] 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 *The Network and Distributed System Security Symposium, NDSS'14*, 2014.

[51] C. Zuo, Z. Lin, and Y. Zhang. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud from Mobile Apps. In *IEEE Symposium on Security and Privacy, (SP'19), London, UK*, 2019.