Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks

Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, Danfeng (Daphne) Yao

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. To compare their accuracy and security guarantees, we develop two comprehensive benchmarks named CryptoAPI-Bench and ApacheCryptoAPI-Bench. CryptoAPI-Bench consists of 181 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. The ApacheCryptoAPI-Bench consists of 121 cryptographic cases from 10 Apache projects. We evaluate four tools, namely, SpotBugs, CryptoGuard, CrySL, and another tool (anonymous) using both benchmarks. We present their performance and comparative analysis. The ApacheCryptoAPI-Bench also examines the scalability of the tools. Our benchmarks are useful for advancing state-of-the-art solutions in the space of misuse detection.

Index Terms—Cryptographic API misuses, benchmark, Java.

1 Introduction

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 [1]–[6].

Several studies showed that the prominent causes for cryptographic misuses are the deficiency in understanding of security API usage [4], [7], complex API designs [7], [8], the lack of cybersecurity training [4], insecure code generation tools [9] and insecure/misleading suggestions in Stack Overflow [4], [10]. 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 [11]. Recognizing these practical barriers, automatic cryptographic code generation [12], and misuse detection tools [5] play a significant role in assisting developers with writing and maintaining secure code.

The security community has produced several impressive static (e.g., CryptoLint [3], CrySL [13], FixDroid [14], MalloDroid [1], CryptoGuard [5]) and dynamic code screening tools (e.g., Crylogger [15], SMV-Hunter [16], and AndroSSL [17]) 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 [5], [18]. 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 diminish 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 [19]. 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 several requirements: *i)* It should cover a wide range of misuse instances. *ii)* It should cover interesting program properties (e.g., flow-, context-, field-, path-sensitivity, etc.) [20], [21]. These are different detection capabilities required for capturing certain vulnerabilities. *iii)* 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 [22], Ghera [23]). For example, DroidBench [22] only contains binaries. Ghera [23] has sources

- *S. Afrose, Y. Xiao and D. Yao are with the Department of Computer Science, Virginia Tech, Blacksburg, VA, 24060. E-mail: sharminafrose@vt.edu, yax99@vt.edu, danfeng@vt.edu*
- *S. Rahaman is with the Department of Computer Science, University of Arizona, Tucson, AZ, 85721. E-mail: sazz@cs.arizona.edu*
- *B. P. Miller is with the Computer Sciences Department, University of Wisconsin-Madison, Madison, WI, 53706. E-mail: bart@cs.wisc.edu*

of provided Android apps. However, both DroidBench and Ghera barely cover cryptographic API misuses.

In this paper, we present two benchmarks for cryptographic API misuses. The first one is CryptoAPI-Bench, a comprehensive benchmark for comparing the quality of cryptographic vulnerability detection tools. It consists of 181 unit test cases covering 18 types of cryptographic misuses. Several test cases include interesting program properties [20], [21]. Flow-sensitive correctly computes and analyzes the order of statements in a program. Path-sensitivity analysis computes different dataflow analysis information dependent on conditional branch statements. Field-sensitive analysis distinguishes two fields containing the same object in a class. A context-sensitive analysis is any interprocedural analysis that analyzes the target of a function call.

The second one is ApacheCryptoAPI-Bench which is built upon 10 real-world Apache projects. It contains early versions of activemq-artemis, deltaspike, directory-server, manifoldcf, meecrowave, spark, tika, tomee, wicket projects. We identify 121 crypto cases in them, including 82 basic cases and 39 advanced cases.

We run CryptoAPI-Bench and ApacheCryptoAPI-Bench on four static analysis tools (i.e., SpotBugs [24], CryptoGuard, CrySL, and Tool A (anonymous) and perform a 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 and CryptoGuard are open-sourced research prototypes that are actively being maintained to improve their accuracy and coverage. SpotBugs is also an actively maintained open-source project, which is the successor of FindBugs. Tool A is one of the most popular static analysis platforms for decades.

Our main technical contributions are summarized as follows.

- We provide a benchmark named CryptoAPI-Bench, which consists of 181 test cases covering 18 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. Our benchmark is open-sourced and can be found on GitHub [25].
- We provide another benchmark named ApacheCryptoAPI-Bench for checking the scalability property of the cryptographic vulnerability detection tools. We document 121 test cases covering 12 types of Cryptographic and SSL/TLS API misuse vulnerabilities from 10 real-world Apache projects. The detailed information regarding ApacheCryptoAPI-Bench can be found on GitHub [26].
- We evaluate four static analysis tools that are capable of detecting cryptographic misuse vulnerabilities. Our experimental evaluation revealed some interesting insights. For complex cases, specialized tools (e.g., CryptoGuard, CrySL) detect more cryptographic misuses and cover more rules than general-purpose tools (e.g., SpotBugs, Tool A). Currently, none of these tools supports path-sensitive analysis.

A preliminary version of the work appeared in the Proceedings of the 2019 ACM Conference on Computer and Communications Security (CCS) [5] and the 2019 IEEE Secure Development Conference (SecDev) [27]. We expanded the conference version by adding a new benchmark ApacheCryptoAPI-Bench (Section 4, Table 2) that contains complex real-world Java programs and we test four static tools' performance in real-world code (Section 6.4, Table 7, Table 8). For CryptoAPI-Bench, We also add two new misuse categories (Section 2.4, Section 2.18), 11 new test cases (Table 1), and update tools' performance evaluation (Table 4, Table 5, Table 6).

The remainder of this paper is organized as follows. Section 2 describes cryptographic API misuse categories. Section 3 and Section 4 outlines the design of CryptoAPI-Bench and ApacheCryptoAPI-Bench. Section 5 reviews existing cryptographic vulnerability detection tools. Section 6 presents the evaluation and performance analysis of the tools on the benchmarks. Discussion is given in Section 7. Section 8 describes the related works. Finally, Section 9 concludes this paper.

2 Crypto API misuse categories

In this section, we discuss 18 Java cryptographic API misuse categories. We got the insights of these misuse categories from previous literature [5], [13], [14], NIST documents [28]–[30], and other blogs [31]. We describe reasons for vulnerability and possible secure solutions for these misuse categories.

2.1 Cryptographic Keys: For encryption, it is expected to use an unpredictable key 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.

2.2 Passwords in Password-based Encryption: 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). However, if a hardcoded or constant password is used in the code, then malicious attackers may obtain the password and predict the key [3]. Therefore, an unpredictable password should be used as a parameter in PBEKeySpec.

2.3 Passwords in KeyStore: Cryptographic keys and certificates are sometimes stored using `java.security.KeyStore` API. The KeyStore employs a password to get access to 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, an unpredictable random password should be used in KeyStore.

2.4 Credentials in String: Credentials (passwords, secret keys, etc) should not be stored in the String variable. In Java, String is a final and immutable class stored in the heap. More specifically, it exists in the memory until garbage

collection. Therefore, sensitive information should not be stored in String[32], [33]. Compared with String, it is highly recommended to use mutable data structures (e.g., byte or char array) for sensitive information and clear it immediately after use. This reduces the window of opportunity for an adversary. [34].

2.5 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 an exception. However, this arrangement causes a security threat, where URL spoofing [35] attacks can be possible. URL spoofing makes it simpler for numerous cyber-attacks (e.g., identity theft, phishing).

2.6 Certificate Validation: Empty methods are often 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 may cause Man-in-the-middle (MitM) attacks [1], [36].

2.7 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 [2], [37].

2.8 Hypertext Transfer Protocol: HyperText Transfer Protocol (HTTP) sends a request to a server to retrieve a web page. However, HTTP allows hackers to intercept and read sensitive information [38]. Therefore, it is recommended to use HyperText Transfer Protocol Secure (HTTPS) that utilizes a secured socket layer to encrypt sensitive information.

2.9 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 [39]) that is proven to be insecure. To solve the problem, `java.security.SecureRandom` provides non-deterministic and unpredictable random numbers.

2.10 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, developers should use a non-deterministic random seed.

2.11 Salts in Password-based encryption (PBE): `javax.crypto.spec.PBEParameterSpec` API takes salt as one of the 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.

2.12 Mode of Operation: The Electronic Codebook (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 more secure to use.

2.13 Initialization Vector (IV): The initialization vector (IV) is used during encryption and decryption with several modes of operation. Static/constant initialization vector introduces vulnerabilities for CBC mode of operation. Therefore, it is suggested to use an unpredictable random initialization vector in `crypto.spec.IvParameterSpec API`. Note that, for several modes of operation (e.g., CTR, CBC-MAC), unpredictable random IV is not required.

2.14 Iteration Count in Password-based Encryption (PBE): In `javax.crypto.spec.PBEParameterSpec` API, it takes iteration count as one of the parameters for Password-based Encryption (PBE). In PKCS #5 [40], it is suggested that the number of iteration should be more than 1000 to provide a reasonable security level.

2.15 Symmetric Ciphers: Symmetric ciphers use the same key for encryption and decryption. There are a couple of vulnerable symmetric cipher algorithms, e.g., DES, Blowfish, RC4, RC2, IDEA. For example, DES is a broken block cipher because it uses an outdated block size (64 bits) that allow brute-force attack. RC4 is a flawed stream cipher that produces a biased keystream while a pseudo-random keystream is required for security, thus leading to several attacks (e.g., bit-flipping attack). To overcome the attacks, developers need to use a secure alternative AES which can support a block length of 128 bits and key lengths of 128, 192, and 256 bits [41].

2.16 Asymmetric Ciphers: In asymmetric cryptography, two keys, i.e., a public key and a private key are used for encryption and decryption. RSA is considered insecure for 1024-bit ciphers [29]. For this reason, developers are recommended to use RSA with a key size of 2048 bits or higher.

2.17 Cryptographic Hash Functions: A cryptographic hash function generates a fixed-length alphanumeric hash value or message digest which is commonly used 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. The list of broken hash functions includes SHA1, MD4, MD5, and MD2. Therefore, developers need to use a strong hash function, e.g., SHA-256.

2.18 Cryptographic MAC: A MAC algorithm HmacMD5 and HmacSHA1 are considered insecure as these are susceptible to collision attacks [42]. Therefore, the developers need to use a strong MAC algorithm, e.g., HmacSHA256.

3 Design of CryptoAPI-Bench

In this section, we present the design of the CryptoAPI-Bench. We manually generate 181 unit test cases guided by 18 types of misuses presented in Section 2. We divide all test cases into two types, i.e., basic cases and advanced cases. These test cases incorporate the majority of possible variations in the perspective of program analysis to detect cryptographic vulnerability.

TABLE 1
CryptoAPI-Bench: Summary of unit test cases. There are 181 unit test cases with 45 basic cases and 136 advanced cases (interprocedural, field sensitive, combined case, path sensitive, miscellaneous, and multiple class test cases). Total test cases per type and misuse categories are summarized here. Details information are presented in Section 3.

No.	Misuse Categories	Groups	Basic Cases	Two Interproc.	Three Interproc.	Field Sensitive	Combined Case	Path Sensitive	Misc.	Multiple Class	Total Cases per Categories
1	Cryptographic Key	Predictable Secrets	2	1	1	1	1	1	1	1	**9**
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	Credentials in String		2	1	1	1	1	0	1	1	**8**
5	Hostname Verifier	Vulnerability in SSL/TLS	2	0	0	0	0	0	0	0	**2**
6	Certificate Validation		3	0	0	0	0	0	0	0	**3**
7	SSL Socket		1	0	0	0	0	0	0	0	**1**
8	HTTP Protocol		2	1	1	1	1	1	0	1	**8**
9	PRNG	Predictable PRNGs	2	0	0	0	0	0	0	0	**2**
10	Seed in PRNG		3	2	2	2	2	2	2	2	**17**
11	Salt in PBE	Vulnerable Parameters	2	1	1	1	1	1	1	1	**9**
12	Mode of Operation		2	1	1	1	1	1	0	1	**8**
13	Initialization Vector		2	1	1	1	1	1	2	1	**10**
14	Iteration in PBE		2	1	1	1	1	1	1	1	**9**
15	Symmetric Ciphers	Vulnerable Algorithms	6	5	5	5	5	5	0	5	**36**
16	Asymmetric Ciphers		1	1	1	0	1	1	0	1	**6**
17	Cryptographic Hash		5	4	4	4	4	4	0	4	**29**
18	Cryptographic MAC		3	0	0	0	0	0	0	0	**3**
Total Cases per Type			**45**	**21**	**21**	**20**	**21**	**20**	**12**	**21**	**181**

3.1 Basic Cases

Basic test cases are simple ones where the probable source of vulnerability for Crypto API exists within the same method. For example, Listing 1 shows that Cipher API takes `cryptoAlgo` as an argument. Note that, `cryptoAlgo` contains an insecure cipher algorithm that is defined within the same method `method1`. In CryptoAPI-Bench, we create 45 basic test cases covering all 18 misuse categories. Among these test cases, 30 test cases contain cryptographic vulnerability (i.e., true positive), and 15 test cases do not contain any cryptographic vulnerability (i.e., true negative). These test cases identify a tool's capability to detect a specific misuse category.

```
public void method1 ()
{  ...
   cryptoAlgo = "DES/ECB/PKCS5Padding"
   Cipher cipher = Cipher.getInstance(cryptoAlgo)
   ...
}
```

Listing 1. Example code snippet of a basic test case

3.2 Advanced Cases

The advanced cases are more complex compared to basic cases where the probable source of vulnerability of a Crypto API appears from other methods, classes, field variables, or conditional statements . In CryptoAPI-Bench, we include 136 advanced cases. The distribution of advanced cases is presented from the fourth to tenth columns of TABLE 1.

3.2.1 Interprocedural Cases

In interprocedural cases, the probable source of vulnerability in a Crypto API comes from other methods (i.e., procedures). We create two types of interprocedural cases: two-interprocedural (i.e., involving two methods) and three-interprocedural (i.e., involving three methods). In a two-interprocedural test case, the probable source of vulnerability comes from another method as a parameter. Listing 2 shows the code snippet of a two-interprocedural test case. In `method2`, Cipher API takes `cryptoAlgo` as an argument, and `cryptoAlgo` is not defined in `method2`, rather, it comes from another method `method1`. The assigned value of `cryptoAlgo` in `method1` shows that the test case is insecure. In three-interprocedural test cases, the probable source of vulnerability comes from two consecutive methods (i.e., source defined in one method, passes to another method, and then passes again to be used in Cipher API). CryptoAPI-Bench contains a total of 42 interprocedural test cases. Among them, 21 are two-interprocedural test cases, and 21 are three-interprocedural test cases. The purpose of having the interprocedural test cases is to check the detection tool's interprocedural data flow handling capability.

```
public void method1 ()
{  ...
   cryptoAlgo = "DES/ECB/PKCS5Padding"
   method2(cryptoAlgo)
   ...
}
public void method2 (String cryptoAlgo)
{  ...
   Cipher cipher = Cipher.getInstance(cryptoAlgo)
   ...
}
```

Listing 2. Example code snippet of a two-interprocedural test case

3.2.2 Field Sensitive Cases

In field-sensitive cases, the probable source of cryptographic vulnerabilities can be detected by the analysis tools if the tools are capable of performing field-sensitive data flow analysis. Field-sensitive refers to an analysis that is able to differentiate multiple fields or variables with the same object [21]. In Listing 3, `algo` is an instance or field variable in the `Crypto` class. The constructor `Crypto()` stores `algo` with defAlgo object. A class member function `encrypt()` use this `algo` value in Cipher API. Both `algo` and `defAlgo` contain the same object, i.e., a secure or insecure cipher algorithm. This is a field-sensitive case as the tools need to trace the field variable `algo` as the probable source of vulnerability. CryptoAPI-Bench contains 20 field-sensitive test cases.

```
class Crypto {
    String algo
    public Crypto (String defAlgo) {
        algo = defAlgo;
    }
```

```
    public void encrypt(... ) {
        ...
        Cipher cipher = Cipher.getInstance(algo);
        ...
    }
}
```

Listing 3. Example code snippet of a field sensitive test case

3.2.3 Combined Cases

The combined cases are a bit more complex where both interprocedural and field sensitivity properties are combined, i.e., both Listing 2 and Listing 3 are incorporated to generate complicated test cases. CryptoAPI-Bench has 21 combined test cases.

3.2.4 Path-Sensitive Cases

In path-sensitive test cases, conditional branch instructions are included in the test cases containing the definition of the probable source of a vulnerability. In Listing 4, an example code snippet of a path sensitivity case is given. Depending on the `choice` variable, the `Cipher` is getting the instance from a secure or an insecure cryptographic algorithm. There are 20 path-sensitive test cases in CryptoAPI-Bench.

```
public void method1 (int choice) {
    ...
    Cipher ch = Cipher.getInstance ("DES/ECB/...") ;
    if (choice > 1) {
        ch = Cipher.getInstance ("AES/CBC/...") ;
    }
    ch.init (Cipher.ENCRYPT_MODE, key) ;
    ...
}
```

Listing 4. Example code snippet of a path sensitive test case

3.2.5 Miscellaneous Cases

Miscellaneous test cases evaluate the tool's abilities to recognize irrelevant constraints and other interfaces, e.g., Map. In Listing 5, 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 values. Similarly, in Line 8, the "UTF-8" represents byte encoding, not any constant or hard-coded value. CryptoAPI-Bench contains 12 miscellaneous test cases.

```
public void method1 (String choice) {
    ...
    Map<String,String> hm = new HashMap<String,
    String>();
    hm.put("a", secureKeyString);
    hm.put("b", insecureKeyString);
    String keyString = hm.get(choice);

    byte [] b = secureKeyString.getBytes("UTF-8");
    IvParameterSpec ivSpec = new IvParameterSpec(b);
    ...
}
```

Listing 5. Example code snippet of a miscellaneous test case

3.2.6 Multiple Class Cases

In multiple class test cases, the probable source of vulnerabilities comes from another Java class. An example code snippet of multiple class case is presented in Listing 6. It is necessary to detect whether a secure or an insecure algorithm is passed in Line 4 in `MultipleClass1` and used in Line 9 in `MultipleClass2`. CryptoAPI-Bench has 21 multiple class test cases.

```
public class MultipleClass1 {
    public void method1 (String passedAlgo) {
        MultipleClass2 mc = new MultipleClass2 ();
        mc.method2 (passedAlgo);
    }
}
public class MultipleClass2 {
    public void method2 (String cryptoAlgo) {
        Cipher c = Cipher.getInstance (cryptoAlgo);
    }
}
```

Listing 6. Example code snippet of a multiple class test case

4 Design of ApacheCryptoAPI-Bench

We include the early version of real-world large 10 Apache projects to check the scalability property of different tools. The second and third columns of TABLE 2 show the number of Java files and lines of Java Code in Apache projects. The spark project is the largest among 10 considered projects containing 2,005 Java files with 311,856 lines of code. The meecrowave project contains the lowest number of Java files (40 Java files) and deltaspike contains the lowest number of lines of code (i.e., 5,116 LoC).

We enlist 121 test cases in ApacheCryptoAPI-Bench. Among them, 82 test cases are basic cases, i.e., the vulnerability rise within the same method. There are 39 advanced test cases where probable source vulnerability comes from other methods (interprocedural cases), other classes (multiple class cases), class variables (field sensitive cases), etc. We detect 64 cryptographic misuses, i.e., true positive alerts. Regarding true negatives, we consider only the cases where a tool shows the case as a false alert. With this consideration, we show 57 true negative cases.

We look into the Apache projects in the Benchmark and made detailed documentation. The documentation consists of cryptographic vulnerabilities the project contains, an explanation of the error, the location (file name, method name, line number) of the vulnerabilities. The documentation and corresponding ApacheCryptoAPI-Bench benchmark are available in the GitHub repository [26].

5 Existing Cryptographic Vulnerability Detection Tools

In this section, we summarize the vulnerability detection tools that we choose to run on CryptoAPI-Bench and ApacheCryptoAPI-Bench. We consider three criteria while choosing the analysis tools. (1) Open-sourced tools: The open-sourced vulnerability detection tools, i.e., CrySL [13], CryptoGuard [5], SpotBugs [24] are convenient to use as we are able to analyze their codes and understand the reason for 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 Tool A are static analysis tools. (3) Free cryptographic vulnerability detection services: We consider Tool A as a provider of free cryptographic vulnerability detection service. Tool A is not open-sourced. However, Tool A provides online services to detect vulnerability.

We also consider GrammaTech [43], QARK [44] and FixDroid [14]. However, GrammaTech is a commercial tool. We were unable to access its trial version. The online SWAMP [45] contains GrammaTech tool to use that only

TABLE 2
ApacheCryptoAPI-Bench: Summary of unit test cases. Contents (number of Java file and lines of code) of the considered Apache projects are summarized here. There are total 121 unit test cases with 82 basic cases and 39 advanced cases. Details information are presented in Section 4.

Apache Project	Number of Java Files	Lines of Code	Test Cases				
			Total Case	Basic Case	Advanced Cases	True Positive	True Negative
deltaspike	87	5116	8	5	3	2	6
directory-server	468	20780	36	15	21	19	17
incubator-taverna-workbench	45	9919	8	5	3	8	0
manifoldcf	126	16998	7	4	3	3	4
meecrowave	40	5646	3	3	0	3	0
spark	2005	311856	26	25	1	12	14
tika	225	16558	2	1	1	0	2
tomee	1029	118661	9	7	2	7	2
wicket	204	13442	9	7	2	7	2
artemis-commons	126	8915	15	12	3	7	8
		Total	**121**	**82**	**39**	**64**	**57**

TABLE 3
Generated alert keywords for each misuse category from cryptographic vulnerability detection tools (SpotBugs, CryptoGuard, CrySL, and Tool A). For example, for misuse category 16 (i.e., Cryptographic Hash), the generated alert keywords in tools are WEAK_MESSAGE_DIGEST, broken hash scheme, ConstraintError, RISKY_CRYPTO, respectively.

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

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 Tool A to evaluate on CryptoAPI-Bench.

5.1 SpotBugs

SpotBugs is a static analysis tool also for capturing deficiencies in Java code. The tool is built based on a plugin structure. The tools detect defects by utilizing visitor patterns in class files or bytecodes of Java, state machine, flags. We use the SpotBugs tool (version 3.1.12) available online in SWAMP [45]. However, currently, SWAMP is in the transition to a new host service [46].

5.2 CryptoGuard

CryptoGuard [5] 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 (commitID: 97b220) available on GitHub [47].

5.3 CrySL

CrySL [13] is a domain-specific language for cryptographic libraries. The static analysis $CogniCrypt_{SAST}$ takes the rules provided in the specification language CrySL as input, and performs a static analysis based on the specification of the rules. CrySL is open-sourced and we run the experiment on CrySL (commit ID: 004cd2) available on GitHub [48].

5.4 Tool A

We choose to anonymize Tool A's name. Tool A has an educational license that generally does not allow publishing comparison with other tools.

6 Evaluation and Analysis

In this section, we evaluate the results for four cryptographic misuse detection tools, i.e., SpotBugs, CryptoGuard, CrySL and Tool A. We show the experimental setup, evaluation criteria, and analysis results using both benchmarks.

6.1 Experimental Setup

We evaluate mainly four cryptographic analysis tools, i.e., SpotBugs [24], CryptoGuard [5], CrySL [13], Tool A on both Benchmarks. We follow the instructions from GitHub to set up the environment of CryptoGuard and CrySL in our machine to perform the analysis. We upload JAR files from CryptoAPI-Bench and Apache projects into SpotBugs tool

TABLE 4
CryptoAPI-Bench comparison of SpotBugs, CryptoGuard, CrySL and Tool A on all 18 rules with CryptoAPI-Bench's 181 test cases. There are 37 secure API use cases (15 in basic and 22 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 6.3.

No.	Misuse Categories	GTP	SpotBugs		CryptoGuard		CrySL		Tool A	
			TP	FP	TP	FP	TP	FP	TP	FP
1	Cryptographic Key	7	0	3	5	1	0	8	5	1
2	Password in PBE	8	2	0	7	1	0	10	7	1
3	Password in KeyStore	7	1	1	7	1	0	10	5	1
4	Credentials in String	7	–	–	–	–	0	8	–	–
5	Hostname Verifier	1	–	–	1	0	–	–	1	0
6	Certificate Validation	3	3	0	3	0	–	–	3	0
7	SSL Socket	1	–	–	1	0	–	–	1	0
8	HTTP Protocol	6	–	–	6	1	–	–	–	–
9	PRNG	1	1	0	1	0	–	–	–	–
10	Seed in PRNG	14	–	–	11	2	0	15	1	2
11	Salt in PBE	7	–	–	6	1	6	1	–	–
12	Mode of Operation	6	1	3	6	1	5	1	1	1
13	Initialization Vector	8	3	6	7	1	7	1	–	–
14	Iteration Count in PBE	7	–	–	5	1	5	3	–	–
15	Symmetric Cipher	30	5	11	30	5	25	5	4	4
16	Asymmetric Ciphers	5	–	–	4	1	5	1	–	–
17	Cryptographic Hash	24	4	8	24	4	20	4	4	4
18	MAC Algorithm	2	–	–	–	–	2	0	–	–
Total		**144**	**20**	**32**	**124**	**20**	**75**	**67**	**32**	**14**

available in SWAMP. Tool A is an online tool that takes GitHub link and compressed code files in order to start analysis.

6.2 Evaluation Criteria

We evaluate the vulnerability detection tools by running these tools on our benchmarks. After performing the analysis, we capture true positives, false positives, and false negatives 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 3, we present the alert keywords that detection tools use while showing a specific cryptographic misuse. This 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.

6.2.1 True positive (TP)

If a tool generates an 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.

6.2.2 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 a false positive.

6.2.3 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 recall and precision to determine the performance of the tools.

6.3 CryptoAPI-Bench: 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 4 presents the number of true positive and false positive vulnerability threat detection captured by the tools for 18 cryptographic misuse categories. There are only 6 common cryptographic misuse categories detected by all tools. To ensure fairness in comparison, we consider only these 6 common cryptographic misuses while finding the comparative analysis results of tools based on the basic and advanced benchmark in TABLE 5 and TABLE 6, respectively. The analysis results are presented in terms of false positive rate (FPR), false negative rate (FNR), recall, and precision.

Analysis Overview: TABLE 4 shows that among the 18 specified high impact cryptographic misuse categories in Section 2, the cryptographic vulnerability detection tools are able to detect a subset of rules.

- SpotBugs, CryptoGuard, CrySL, Tool A covers 9, 16, 14, 10 cryptographic misuse categories, respectively.
- In total, the benchmark contains 144 vulnerable test cases and among these true positive cases, SpotBugs, CryptoGuard, CrySL, Tool A detects 20, 124, 75, 32 cases, respectively.
- In addition, SpotBugs, CryptoGuard, CrySL, Tool A also generate 32, 20, 67, 14 false alarms, respectively that are included as false positive cases.

6.3.1 Analysis on Basic Benchmark

TABLE 5 shows the performance analysis result of four detection tools on six common cryptographic misuse categories based on the basic benchmark. We capture the following findings based on TABLE 5.

- SpotBugs shows 3 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 misuse.

TABLE 5
CryptoAPI-Bench comparison of SpotBugs, CryptoGuard, CrySL and Tool A on six common misuse categories with CryptoAPI-Bench's common 21 basic cases. TP, FP, FN stand for true positive, false positive, false negative, respectively. Findings of the table are reported in Section 6.3.1.

Basic Test Cases	True Positive Count	True Negative Count	SpotBugs			CryptoGuard			CrySL			Tool A		
			TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
IntraProcedural	14	6	13	3	1	14	0	0	10	7	4	13	0	1
Result		**Recall (%)**	**92.86**			**100.00**			**71.43**			**92.86**		
		Precision (%)	**81.25**			**100.00**			**58.82**			**100.00**		

TABLE 6
CryptoAPI-Bench comparison of SpotBugs, CryptoGuard, CrySL and Tool A on six common misuse categories with CryptoAPI-Bench's common 84 advanced cases. TP, FP, FN stand for true positive, false positive, false negative, respectively. Findings of the table are reported in Section 6.3.2.

Advanced Test Cases	True Positive Count	True Negative Count	SpotBugs			CryptoGuard			CrySL			Tool A		
			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	2	0	0	3	3	0	0	0	5	3	0	0	3
Multiple Class methods	13	0	0	0	13	13	0	0	10	3	3	3	0	10
Results		**Recall (%)**	**0.00**			**95.59**			**58.82**			**19.12**		
		Precision (%)	**0.00**			**83.33**			**56.34**			**52.00**		

- CrySL produces 7 false positive errors due to their rules associated with Crypto APIs for the cryptographic key, password in PBE, and password in KeyStore.
- Tool A does not generate any false positive errors. It can successfully detect every vulnerability except one. Tool A is not designed to capture IDEA as a vulnerable cryptographic algorithm.
- For insecure uses of pseudo-random number generators, SpotBugs and CryptoGuard flag all uses of java.util.Random.

In summary, for all basic cases, CryptoGuard and Tool A generate a precision of 100%. For SpotBugs and CrySL, it produces some false positives and hence generates precision of 81.25%, 58.82% respectively.

6.3.2 Analysis on Advanced Benchmark

TABLE 6 shows the performance analysis result of four detection tools on six common cryptographic misuse categories based on the advanced benchmark. We capture the following findings based on TABLE 6.

- In the prospect of path sensitivity, it is obvious that none of the cryptographic vulnerability detection tools is path-sensitive in their static analysis. The tools generate 10, 13, 13, 12 false positive alerts for path sensitive cases, 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 that 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 because of the tools being path insensitive.
- SpotBugs is not designed to capture vulnerability threats in advanced cases. Therefore, it shows 0% precision and recall.
- 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 the correct "CIPHER_INTEGRITY" alert, it produces an incorrect "HARD_CODE_PASSWORD" alert.
- CryptoGuard performs better than other tools in terms of both precision and recall. The reasons behind this include 1) Cryptoguard performs dataflow analysis based on forward slicing and backward slicing that efficiently handles the advanced cases, 2) CryptoGuard follows several refinement insights that systematically remove irrelevant constants, hence reducing false positives. However, as being a static analysis tool, CryptoGuard cannot handle path-sensitive cases. In addition, CryptoGuard missed 3 vulnerabilities due to clipping orthogonal method invocation (i.e., limiting the depth to visit callee method).
- CrySL produces incorrect "RequiredPredicateError" alerts for the cryptographic key, password in PBE, and password in KeyStore test cases, causing false positives. In some of these cases, CrySL detection rules are unable to recognize secure byte arrays and would still incorrectly generate alerts.
- Tool A 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 Tool A. Tool A is a close-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 a precision of 83.33%. CrySL produces precision of 56.34%. Tool A generates a precision of 52.00%.

6.4 ApacheCryptoAPI-Bench: Analysis of Results

TABLE 7 presents the number of true positive and false positive vulnerability threats detected by the tools. CrySL fails to analyze spark and artemis-commons project. Tool A fails to analyze artemis-commons project. SpotBugs and

TABLE 7
ApacheCryptoAPI-Bench comparison of SpotBugs, CryptoGuard, CrySL and Tool A on 10 Apache projects. GTP stands for ground truth positive, which is the number of insecure API use cases in the Apache codes.

Apache Project	GTP	SpotBugs			CryptoGuard			CrySL			Tool A		
		TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
deltaspike	2	2	0	0	2	0	0	2	3	0	2	0	0
directory-server	19	11	0	8	5	0	14	18	6	1	5	0	14
incubator-traverna-workbench	8	2	0	6	4	0	4	7	0	1	3	0	5
manifoldcf	3	0	3	3	0	3	3	3	2	0	2	1	1
meecrowave	3	3	0	0	2	0	1	2	0	1	2	0	1
spark	12	9	12	3	12	14	0	–	–	–	4	0	8
tika	0	0	0	0	0	0	0	0	2	0	0	0	0
tomee	7	3	1	4	4	2	3	6	0	1	3	1	4
wicket	3	0	2	3	3	2	0	2	2	1	0	0	3
artemis-commons	7	5	8	2	5	8	2	–	–	–	–	–	–
Total	**64**	**35**	**26**	**29**	**37**	**29**	**27**	**40**	**15**	**5**	**21**	**2**	**36**

CryptoGuard successfully analyze all 10 projects. Overall, we capture the following findings.

- Tool A has a low false positive value. However, SpotBugs and CryptoGuard have high false positive values of 26, 29, respectively. The main reason is that CryptoGuard and SpotBugs consider all usages of Java.util.Random as vulnerable whereas majority of the random is used in a non-security context. We have discussed the reason for generating high false positives for CrySL in Section 6.3.2.
- SpotBugs, CryptoGuard, CrySL, and Tool A can accurately detect 35, 37, 40, and 21 alerts respectively from 64 alerts. The main reason for missed alarms is that no tool can detect all 18 types of vulnerabilities as shown in TABLE 3. For example, SpotBugs and CryptoGuard cannot capture vulnerable crypto algorithm usage in SecretKeySpec API. Among the successfully compiled programs (i.e., from 8 Apache projects), CrySL captures 40 out of 45.
- After analyzing ten Apache projects, we find that there are 82 basic cases, whereas, the number of advance cases is only 39. Therefore, in the real world codes, the number of basic cases is much higher than advanced cases. Vulnerability detection tools should consider expanding their coverage to detect more categories of vulnerabilities.
- From TABLE 7, we observe that CrySL fails to analyze two Apache projects: spark and artemis-commons. CrySL throws StackOverFlowError (i.e., memory error) during analyzing objects for spark. The probable reason is the larger number of files and lines of code Spark contains for analysis. For artemis-commons, CrySL throws NullPointerErrorException during analysis due to the reference variable not pointing to any object. Tool A fails to analyze only the artemis-commons project. Tool A is closed source, therefore, we are unable to confirm the reason for this failure. TABLE 8 shows the runtime on Apache projects for only CryptoGuard and CrySL. For Tool A and SpotBugs, we use the web version that takes all scan requests for users and reports results after complete scanning. Therefore, we cannot calculate their original runtime for comparison. Among the 8 successful analyzed projects, we observe average runtime for CrySL is 14.64 seconds and CryptoGuard is 11.46 seconds. For the largest Apache project Spark (LoC: 311,856), CryptoGuard successfully analyzes in 88.68 seconds and CrySL produces incomplete analysis report after running for 46.84 seconds. Overall, SpotBugs and CryptoGuard successfully analyze all 10 Apache projects. Therefore, SpotBugs, CryptoGuard are scalable for large projects.

6.5 Verifiability

Our benchmarks are open-sourced and are available on GitHub [25], [26]. It contains the Java cryptographic API test cases. The detailed documentation and explanation are provided there.

TABLE 8
Runtime for analyzing Apache projects. Star (*) symbol indicates that the analysis was unsuccessful.

Apache Projects	LoC	Runtime (sec)	
		CryptoGuard	CrySL
deltaspike	5.1K	4.31	6.95
directory-server	20.8K	8.96	23.03
incubator-taverna-workbench	9.9K	12.69	7.94
manifoldcf	17K	7.07	8.20
meecrowave	5.6K	4.67	7.24
spark	311.9K	88.68	46.84*
tika	16.6K	7.46	8.15
tomee	118.7K	40.52	34.81
wicket	13.4K	5.99	20.83
artemis-commons	8.9K	5.63	19.82*

7 Discussion

Tool insights. No tool can cover all categories of vulnerabilities (TABLE 4). However, their methodologies can be extended to cover most of these vulnerabilities. For example, the technique that Tool A uses to detect constant cryptographic keys can be transferred to detect static IVs or fewer iteration counts.

The main differences among different tools are within their approach to trade-offs among false positives, false negatives. Our experimental evaluation reveals that all of these tools produce a number of false positives and false negatives. CryptoGuard performs on-demand inter-procedural dataflow analysis. Its backward data flow analysis starts from the slicing criteria and explores upward (↑) and orthogonally (→) on-demand. Orthogonal method invocation chains always return to the call sites. By leveraging this insight, CryptoGuard offers a performance vs scalability tradeoff by limiting the depth of the orthogonal invocations (which is "clipping of orthogonal method invocations"). In the current implementation, the depth is set to be 1.

That means CryptoGuard will skip deeper orthogonal callee methods, which may result in false negatives. However, the advantage of the orthogonal method invocation technique is that it helps to improve precision.

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. For the version that we tested, CrySL would raise 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 detecting 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.

CryptoAPI-Bench cannot be used to evaluate scalability property. All of our test cases are lightweight by design. The primary focus is to produce easily readable test cases that demand minimal code to express complex program properties. On the other hand, all of the projects in ApacheCryptoAPI-Bench are complex programs including a lot of files and lines of code. The primary focus is to test the vulnerability detection tool's scalability property and extrapolation to applications on real-world code.

Case Studies. TABLE 4 shows that many misuse cases are still uncovered by tools (e.g., CryptoGuard, SpotBugs, Tool A cannot handle MAC misuses) that should be addressed to expand coverage. Among the covered rules, there are also some deficiencies. For example, CryptoGuard and SpotBugs can capture RC4 as a vulnerable cipher but not ARCFOUR cipher algorithm as the static code does not specify ARCFOUR as a vulnerable cipher. Static or predictable initialization vector defined in another method, class, file, or as field variable (i.e., advanced cases) cannot be captured using SpotBugs and Tool A. In another advanced case, a java file in manifoldcf contains `SecretKeySpec key = new SecretKeySpec(secretKey.getEncoded(), "AES")`. This secretKey parameter is initialized as a field variable with a static string value of "NowIsTheTime" and passed through three procedures. This complex case cannot be captured by SpotBugs, CryptoGuard, or Tool A.

Our limitation. Currently, our benchmark does not contain cryptographic cases, e.g., digital signature, CBC-MAC misuses in MAC, other modes of operations (e.g., CTR). We plan to include test cases based on these cryptographic vulnerabilities in our CryptoAPI-Bench benchmark. Furthermore, 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.

8 Related Work

Vulnerability detection benchmarks. AndroZoo++ [49] is a collection of over eight million Android apps [50] 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 purposes.

DroidBench [22], 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 benchmarks to evaluate the performance of vulnerability detection tools in Android literature. In total, DroidBench has 119 APKs from 13 categories (Commit id 0fe281b). 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 [23] 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 contain any cryptographic misuse vulnerabilities. To be specific, CryptoAPI-Bench and Ghera have only 2 types of vulnerabilities in common.

OWASP Benchmark [51] 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 [31] released a set of vulnerability samples that can be useful to check for coverage of vulnerability categories. A verification tool for five common audit controls is proposed for ensuring continuous compliance [52]. MASC framework [53] is designed to evaluate static analysis tools using mutation testing. However, it considers limited complex cases.

Other benchmarks. The DaCapo benchmarks [54] are designed to evaluate the performance of various components of Java virtual machine (JVM), Garbage collection (GC), Just-in-time (JIT) compiler itself. BugBench [55] 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 have been proposed [56], [57]. Coding practice and recommendations are provided for 28 enterprise applications that use Spring security framework [58]. ManyBugs and IntroClass benchmarks are designed to evaluate various C/C++ code repair techniques [59]. 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.

9 Conclusion and Future Work

We believe that for scientific, in-depth, and reproducible comparisons benchmark is an important component. In this paper, we present CryptoAPI-Bench and ApacheCryptoAPI-Bench to evaluate the detection accuracy, scalability, and security guarantees of various cryptographic misuse detection tools. Our benchmarks are open-sourced

and are available on GitHub. 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, Tool A), *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 the future, we plan to achieve the following goals.

- 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.) [60], [61], fingerprint protection [62], cloud service APIs for information storage [63]) are also proven to cause catastrophic security consequences. We also plan to include the misuses of these critical non-cryptographic APIs.

ACKNOWLEDGMENTS

This work has been supported by the National Science Foundation under Grant No. CNS-1929701 and the Virginia Commonwealth Cyber Initiative (CCI).

REFERENCES

[1] S. Fahl, M. Harbach, T. Muders *et al.*, "Why Eve and Mallory Love Android: An Analysis of Android SSL (in) Security," in *the ACM Conference on Computer and Communications Security, CCS'12*, 2012, pp. 50–61.

[2] M. Georgiev, S. Iyengar, S. Jana *et al.*, "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*, 2012, pp. 38–49.

[3] M. Egele, D. Brumley, Y. Fratantonio *et al.*, "An Empirical Study of Cryptographic Misuse in Android Applications," in *ACM Conference on Computer and Communications Security, CCS'13*, 2013, pp. 73–84.

[4] N. Meng, S. Nagy, D. Yao *et al.*, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *International Conference on Software Engineering, ICSE'18*, May 2018.

[5] S. Rahaman, Y. Xiao, S. Afrose *et al.*, "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects," in *ACM Conference on Computer and communications security, CCS'19*, Nov. 2019, pp. 2455–2472.

[6] S. Rahaman and D. Yao, "Program Analysis of Cryptographic Implementations for Security," in *IEEE Cybersecurity Development, SecDev'17, Cambridge, MA, USA*, 2017, pp. 61–68. [Online]. Available: https://doi.org/10.1109/SecDev.2017.23

[7] Y. Acar, M. Backes, S. Fahl *et al.*, "Comparing the Usability of Cryptographic APIs," in *IEEE Symposium on Security and Privacy, SP'17, San Jose, CA, USA, May 22-26*, 2017, pp. 154–171.

[8] S. Nadi, S. Krüger, M. Mezini *et al.*, "Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?" in *International Conference on Software Engineering, ICSE'16*, 2016, pp. 935–946.

[9] M. Oltrogge, E. Derr, C. Stransky *et al.*, "The Rise of the Citizen Developer: Assessing the Security Impact of Online App Generators," in *IEEE Symposium on Security and Privacy, SP'18*, 2018, pp. 634–647.

[10] Y. Acar, M. Backes, S. Fahl *et al.*, "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*, 2016, pp. 289–305.

[11] H. Assal and S. Chiasson, "Security in the Software Development Lifecycle," in *Fourteenth Symposium on Usable Privacy and Security, SOUPS'18*, 2018, pp. 281–296.

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

[13] S. Krüger, J. Späth, K. Ali *et al.*, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *European Conference on Object-Oriented Programming, ECOOP'18*, 2018, pp. 10:1–10:27.

[14] 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*, 2017, pp. 1065–1077.

[15] L. Piccolboni, G. Di Guglielmo, L. P. Carloni *et al.*, "Cryllogger: Detecting Crypto Misuses Dynamically," *arXiv preprint arXiv:2007.01061*, 2020.

[16] D. Sounthiraraj, J. Sahs, G. Greenwood *et al.*, "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.

[17] F. Gagnon, M. Ferland, M. Fortier *et al.*, "AndroSSL: A Platform to Test Android Applications Connection Security," in *International Symposium on Foundations and Practice of Security, FPS'15*, 2015, pp. 294–302.

[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)*, 2002, pp. 143–159.

[19] A. Machiry, C. Spensky, J. Corina *et al.*, "DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers," in *26th USENIX Security Symposium, USENIX Security'17, Vancouver, BC, Canada*, 2017, pp. 1007–1024.

[20] Wikipedia contributors, "Data-flow Analysis — Wikipedia, The Free Encyclopedia," last accessed: Sep 9, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Data-flow_analysis

[21] J. Späth, K. Ali, and E. Bodden, "Context-, Flow-, and Field-sensitive Data-flow Analysis Using Synchronized Pushdown Systems," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: https://doi.org/10.1145/3290361

[22] S. Arzt, S. Rasthofer, C. Fritz *et al.*, "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*, 2014, pp. 259–269.

[23] 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*, 2017, pp. 43–52.

[24] "SpotBugs: Find Bugs in Java Programs," https://spotbugs.github.io/, online; Last accessed: Dec 3, 2020.

[25] S. Afrose, "CryptoAPI-Bench," https://github.com/CryptoAPI-Bench/CryptoAPI-Bench, 2019.

[26] ——, "ApacheCryptoAPI-Bench," https://github.com/CryptoAPI-Bench/ApacheCryptoAPI-Bench, 2020.

[27] S. Afrose, S. Rahaman, and D. Yao, "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 49–61.

[28] E. Barker and A. Roginsky, "Transitioning the Use of Cryptographic Algorithms and Key Lengths," in *Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD*, 2019.

[29] E. Barker, L. Chen, A. Roginsky *et al.*, "Recommendation for Pairwise Key Establishment Using Integer Factorization Cryptography," in *Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD*, 2019.

[30] "NIST Computer Security Resource Center," https://csrc.nist.gov/projects, online; Last accessed: Sep 3, 2021.

[31] "SonarSource Static Code Analysis," https://rules.sonarsource.com/, online; Last accessed: Dec 3, 2020.

[32] "Oracle," https://docs.oracle.com/javase/8/docs/api/java/security/Permission.html, online; Last accessed: Sep 3, 2021.

[33] "Secure Code Review: 8 Security Code Review Best Practices," https://snyk.io/blog/secure-code-review/, online; Last accessed: Sep 3, 2021.

[34] "Secure Coding Guidelines for Java SE," https://www.oracle.com/java/technologies/javase/seccodeguide.html, online; Last accessed: Sep 3, 2021.

[35] "URL Spoofing," http://www.securitysupervisor.com/security-q-a/network-security/262-what-is-url-spoofing.html, online; Last accessed: Dec 3, 2020.

[36] "Find Security Bugs," https://find-sec-bugs.github.io/, online; Last accessed: Dec 3, 2020.

[37] "Hostname Verification to SSL Socket," https://www.the-codingforums.com/threads/adding-hostname-verification-to-sslsocket.958287/, online; Last accessed: Dec 3, 2020.

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

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

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

[41] "AES Encryption," https://aesencryption.net/, online; Last accessed: Dec 3, 2020.

[42] M. Bellare, "New Proofs for NMAC and HMAC: Security Without Collision Resistance," *Journal of Cryptology*, vol. 28, no. 4, pp. 844–878, 2015.

[43] "GRAMMATECH," https://www.grammatech.com/, online; Last accessed: Dec 3, 2020.

[44] "Quick Android Review Kit (QARK)," https://github.com/linkedin/qark, online; Last accessed: Dec 3, 2020.

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

[46] "Transition of SWAMP software," https://continuousassurance.org/blog/, online; Last accessed: Dec 3, 2020.

[47] "CryptoGuard," https://github.com/CryptoGuardOSS/cryptoguard, online; Last accessed: Dec 3, 2020.

[48] "Cognicrypt_SAST: CrySLtoStatic Analysis Compiler," https://github.com/CROSSINGTUD/CryptoAnalysis, online; Last accessed: Dec 3, 2020.

[49] L. Li, J. Gao, M. Hurier *et al.*, "AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community," *arXiv preprint arXiv:1709.05281*, 2017.

[50] "AndroZoo," https://androzoo.uni.lu/, online; Last accessed: Dec 3, 2020.

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

[52] M. Kellogg, M. Schäf, S. Tasirans *et al.*, "Continuous Compliance," in *35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2020, pp. 511–523.

[53] A. S. Ami, N. Cooper, K. Kafle *et al.*, "Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques," *arXiv preprint arXiv:2107.07065*, 2021.

[54] S. M. Blackburn *et al.*, "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*, 2006, pp. 169–190.

[55] S. Lu, Z. Li, F. Qin *et al.*, "BugBench: Benchmarks for Evaluating Bug Detection Tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, vol. 5, 2005.

[56] 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*. IEEE, 2003, p. 286.

[57] Y. Eytani, K. Havelund, S. D. Stoller *et al.*, "Towards a Framework and a Benchmark for Testing Tools for Multi-Threaded Programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 267–279, 2007.

[58] M. Islam, S. Rahaman, N. Meng *et al.*, "Coding Practices and Recommendations of Spring Security for Enterprise Applications," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 49–57.

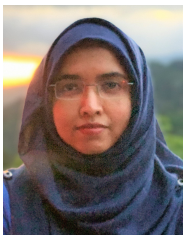
[59] C. Le Goues, N. Holtschulte, E. K. Smith *et al.*, "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[60] A. Bosu, F. Liu, D. Yao *et al.*, "Collusive Data Leak and More: Large-Scale Threat Analysis of Inter-app Communications," in *ACM ASIA Conference on Computer and Communications Security, AsiaCCS'17*, 2017, pp. 71–85.

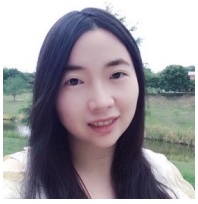
[61] Y. Nan, Z. Yang, X. Wang *et al.*, "Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps," in *25th Annual Network and Distributed System Security Symposium, NDSS'18*, 2018.

[62] A. Bianchi, Y. Fratantonio, A. Machiry *et al.*, "Broken Fingers: On the Usage of the Fingerprint API in Android," in *25th Annual Network and Distributed System Security Symposium, NDSS'18*, 2018.

[63] 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.

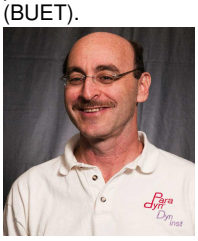


Sharmin Afrose is a Ph.D. student in the department of computer science at Virginia Tech. She is working under the supervision of Professor Danfeng (Daphne) Yao. Her research interests include software security for Java Cryptographic API and AI bias in healthcare. She received a BS in Computer Science and Engineering from the Bangladesh University of Engineering and Technology (BUET).

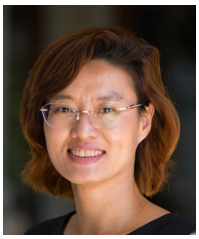


Ya Xiao is a Ph.D. student of computer science department at Virginia Tech. She is working under the supervision of Professor Danfeng (Daphne) Yao. Her research interests include neural network based software security solutions, program analysis, and neural cryptanalysis. She has been awarded the Bit Shares Fellowship and the Dennis G. Kafura Fellowship at Virginia Tech. She received M.S. degree and B.S. degree from Beijing University of Posts and Telecommunications (BUPT).

Sazzadur Rahaman is an assistant professor in the Department of Computer Science at the University of Arizona. He works towards making security research more democratized and affordable. He is broadly interested in computer security and privacy problems, specifically in building robust systems and methodologies by using program analysis, formal verification, applied cryptography, and machine learning-based techniques. Sazzadur completed his Ph.D. from Virginia Tech. He received his B.Sc. in computer science at Bangladesh University of Engineering and Technology (BUET).



Barton P. Miller is a Vilas Distinguished Achievement Professor, and Amar & Belinder Sohi Professor in Computer Sciences at the University of Wisconsin, Madison. He directs the software assurance and training program for Trusted CI, the NSF Cybersecurity Center of Excellence and co-directs the MIST software vulnerability assessment project in collaboration with his colleagues at the Autonomous University of Barcelona, where they are addressing issues related to maritime cybersecurity. He also directs the Paradyn Tools project, which is investigating program scalability and binary code analysis and instrumentation technologies for use in profiling, debugging, forensics, and cyber-security. He received his B.A. degree from the University of California, San Diego in 1977, and M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley in 1980 and 1984. Professor Miller is a Fellow of the ACM.



Danfeng (Daphne) Yao is a Professor of Computer Science at Virginia Tech. She is an Elizabeth and James E. Turner Jr. '56 Faculty Fellow and CACI Faculty Fellow. Her research interests include building cyber defenses, as well as machine learning for digital health, with a shared focus on accuracy and deployment. She creates new models, algorithms, techniques, and deployment-quality tools for securing large-scale software and systems. Her tool CryptoGuard helps large software companies and Apache projects harden their cryptographic code. She systematized program anomaly detection in the book Anomaly Detection as a Service. Her patents on anomaly detection are extremely influential in the industry, cited by patents from major cybersecurity firms and technology companies, including FireEye, Symantec, Qualcomm, Cisco, IBM, SAP, Boeing, and Palo Alto Networks. In 2018, she was named an ACM Distinguished Scientist. Previously, she received the NSF CAREER Award and ARO Young Investigator Award. Dr. Yao is the ACM SIGSAC Vice Chair and has been a member of the ACM SIGSAC executive committee since 2017. She received her Ph.D. degree from Brown University (Computer Science), M.S. degrees from Princeton University (Chemistry) and Indiana University (Computer Science), Bloomington, B.S. degree from Peking University in China (Chemistry).