This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?

Ying Zhang, Mahir Kabir, Ya Xiao, Danfeng (Daphne) Yao, Na Meng

Abstract—The Java platform provides various cryptographic APIs to facilitate secure coding. However, correctly using these APIs is challenging for developers who lack cybersecurity training. Prior work shows that many developers misused APIs and consequently introduced vulnerabilities into their software. To eliminate such vulnerabilities, people created tools to detect and/or fix cryptographic API misuses. However, it is still unknown (1) how current tools are designed to detect cryptographic API misuses, (2) how effectively the tools work to locate API misuses, and (3) how developers perceive the usefulness of tools' outputs. For this paper, we conducted an empirical study to investigate the research questions mentioned above. Specifically, we first conducted a literature survey on existing tools and compared their approach design from different angles. Then we applied six of the tools to three popularly used benchmarks to measure tools' effectiveness of API-misuse detection. Next, we applied the tools to 200 Apache projects and sent 57 vulnerability reports to developers for their feedback. Our study revealed interesting phenomena. For instance, none of the six tools was found universally better than the others; however, CogniCrypt, CogniGuard, and Xanitizer outperformed SonarQube. More developers rejected tools' reports than those who accepted reports (30 vs. 9) due to their concerns on tools' capabilities, the correctness of suggested fixes, and the exploitability of reported issues. This study reveals a significant gap between the state-of-the-art tools and developers' expectations; it sheds light on future research in vulnerability detection.

Index Terms—Detection of cryptographic API misuses, developers' feedback, empirical

1 Introduction

JCA (Java Cryptography Architecture [1]) and **JSSE (Java Secure Socket Extension [2])** are two cryptographic frameworks provided by the standard Java platform. The application programming interfaces (APIs) offered by these frameworks intend to ease developers' secure programming like generating keys and establishing secure communications.

However, the APIs are actually not easy to use for two reasons. First, some APIs are overly complicated but poorly documented [3]–[5]. Second, developers lack necessary cybersecurity training; they are unaware of the security implications of coding options (e.g., the parameter values, calling sequences, or overriding logic of APIs) [5]–[8]. Consequently, many developers misused cryptographic APIs, built security functionalities insecurely, and introduced vulnerabilities or weaknesses to software. Specifically, Fischer et al. found that the cryptographic API misuses posted on StackOverflow [9] were copied and pasted into 196,403 Android applications available on Google Play [10]. Rahaman et al. revealed similar API misuses in 39 high-quality Apache projects [11]. Fahl et al. [12] and Georgiev et al. [13] separately showed that hackers could exploit such API-related vulnerabilities to steal data (e.g., user credentials).

Tools were recently built to scan Java applications, to detect *the specialized category of security vulnerabilities—misuses of cryptographic APIs* [10], [11], [14]–[17]. However, it is unclear what are the strengths and weaknesses of these tools, how well they help developers improve existing secure coding practices, and how we can design better approaches. Therefore, for this paper, we conducted a novel empirical study to explore (1) how current tools are designed to detect cryptographic API misuses, (2) how effectively the tools work to locate API misuses, and (3) how developers perceive the usefulness of tools' outputs. Specifically, there are four steps in our study method. First, we searched in the ACM digital library for state-of-the-art tools, which analyze Java-based applications for any API misuses relevant to JCA and JSSE. Second, we compared the approach design of tools in terms of API usage pattern representations, pattern-matching logic, input/output infos, and public availability.

Third, among all explored tools, we identified six publicly available, executable, and comparable tools: CogniCrypt [18], CryptoGuard [11], CryptoTutor [19], FindSecBugs [20], SonarQube [21], and Xanitizer [22]. To empirically compare tools' effectiveness, we applied the six tools to three public program benchmarks: CryptoBench [23], MUBench [24], and OWASP Benchmark [25]. Based on the ground truth of cryptographic API misuses and manual validation, we evaluated tools' precision, recall, and F-score rates. Fourth, to assess the relevance of tool outputs, we also applied the 6 tools to another dataset of 200 Apache projects, and filed 57 pull requests (PRs) to seek for developers' feedback. After sending our descriptions to the owners of 35 projects, we received 47 responses and analyzed the information to learn about developers' opinions. Our research explores the following research questions (RQs):

RQ1: *How are current tools designed to detect cryptographic API misuses?* We found that current tools represent misuse patterns as either hardcoded rules in tool implementation, Java code snippets, or templates described with domain-specific languages (DSL). These tools match Java programs

• *Y. Zhang, M. Kabir, Y. Xiao, D. Yao, and N. Meng are with the Department of Computer Science, Virginia Tech, Blacksburg, VA 24060. E-mail: {yingzhang, mdmahirasefk, yax99, danfeng, nm8247}@vt.edu*

against known misuse patterns via static program analysis, clone detection, or machine learning to reveal API misuses. Among different design options, most tools adopt hard-coded rules and inter-procedural static analysis probably because such a design can effectively locate misuses.

RQ2: *How effectively do current tools work to locate cryptographic API misuses?* The six experimented tools focus on slightly different pattern sets and achieved distinct trade-offs between precision and recall. Specifically, CryptoGuard outperformed other tools on CryptoBench, getting 85% F-score; Xanitizer acquired the highest F-score when being applied to OWASP Benchmark and MUBench (i.e., 100% and 72%). There is no tool universally better than the others.

RQ3: *How do developers perceive the usefulness of tools' outputs?* According to the 47 responses we received, most developers (i.e., 30) rejected the reported vulnerabilities, fewer developers (i.e., 17) wanted to address the reported issues, and even fewer developers (i.e., 9) replaced API misuses by following tool-generated guidance. The tools' reports usually did not change developers' coding practices. We identified three factors that prevent developers from addressing reported issues. First, the fixing suggestions are vague and incomplete. Second, developers need evidence of security exploits enabled by those vulnerabilities. Third, some detected misuses were in test code or security-irrelevant program contexts, and developers believed those issues to cause no security consequence.

To sum up, in this paper, we made the following research contributions:

- We analyzed the approach design of existing detectors for cryptographic API misuses, and empirically compared six of those detectors. No prior work did such a comprehensive and systematic evaluation.
- We conducted a novel study with developers, via describing for them the security vulnerabilities reported by current tools. We got surprising feedback.
- By manually inspecting developers' feedback and related program context, we characterized the gap between existing tools and developers' expectations.

In the following sections, we will first introduce the misuse patterns of cryptographic APIs (Section 2). Then we will present our literature survey, which describes the existing API-misuse detectors to answer RQ1 (Section 3). Next, we will describe our empirical evaluation of six state-of-the-art tools to explore RQ2 (Section 4). Finally, we will explain our study with developers to investigate RQ3 (Section 5). At https://figshare.com/s/30bd909fb08804d14255, we open-sourced our experiment results.

2 Misuse Patterns of Cryptographic APIs

The Java platform provides two important frameworks to enable security implementation: JCA and JSSE. JCA provides APIs to implement concepts of cryptography such as digital signatures, message digests, certificates and their validation, encryption, key generation and management, and secure random number generation [1]. JSSE enables secure internet communications; it includes APIs for creation of secure channels, data encryption, server authentication, message integrity, and optional client authentication [2].

Among all APIs defined in JCA and JSSE, there are 13 Java types (i.e., classes or interfaces) frequently mentioned in the API-misuse patterns summarized by prior research [10]–[12], [15], [17], [26], [27]. As shown in Table 1, 10 Java types are from JCA and the other 3 types are from JSSE. Each of these Java types has one or more method APIs that are prone to misuse, each API may be misused in one or multiple ways, and each API misuse pattern is considered a code vulnerability. To succinctly represent all API misuse patterns in Table 1, we list all Java type APIs (the container classes/interfaces of methods) instead of the misused method APIs, and summarize the misuse patterns as well as related correct usage. In Table 1, column **Insecure** describes API misuse patterns, while **Secure** summarizes the correct usage patterns with security guarantees.

The **Common Weakness Enumeration (CWE)** [28] is a category system for software weaknesses and vulnerabilities. To facilitate understanding of API misuse patterns, we map the patterns to six CWE categories to explain their security implications:

1. CWE-327: Use of a Broken or Risky Cryptographic Algorithm. When the methods APIs of three Java types (i.e., `Cipher.getInstance(...)`, `MessageDigest.getInstance(...)` and `SecretKeyFactory.getInstance(...)`) get invoked with improper parameters (e.g., `"DES"` and `"MD5"`), security experts consider those invocations to be vulnerable. This is because the parameter values indicate usage of broken or risky cryptographic algorithms, and the usage may result in the exposure of sensitive information [29]. For instance, `"DES"` on line 10 in Listing 1 implies using the symmetric-key algorithm DES (Data Encryption Standard), which is proven insecure [30]. Thus, the method invocation on line 10 is considered an instance of API misuse.

Listing 1: A code snippet that misuses three APIs

```
1  private static byte[] desKey = "12345678".getBytes();
2  private static byte[] iv = "12345678".getBytes();
3  public static void insecureEncrypt(String in) {
4    try {
5  // Declare an IV parameter with constant (CWE-330).
6      IvParameterSpec ivSpec = new IvParameterSpec (iv);
7  // Create a secret key with a hardcoded constant (CWE-798).
8      SecretKey key = new SecretKeySpec(desKey, "DES");
9  // Declare a DES cipher although DES is provenly insecure (CWE-327).
10     Cipher c=Cipher.getInstance("DES/CBC/PKCS5Padding");
11     c.init(Cipher.ENCRYPT_MODE, key, ivSpec);
12     ... } ... }
```

Listing 2: Insecure method overriding for `TrustAllManager`

```
1  private static TrustManager createTrustAllManager() {
2    return new X509TrustManager() {
3  // Override checkClientTrusted (...) to have empty body (CWE-295).
4      @Override
5      public void checkClientTrusted(...) throws
           CertificateException {}
6  // Override checkServerTrusted (...) to have empty body (CWE-295).
7      @Override
8      public void checkServerTrusted(...) throws
           CertificateException {}
9    ... };}
```

2. CWE-295: Improper Certificate Validation. When the method APIs `HostnameVerifier.verify(...)`, `TrustManager.checkClientTrusted(...)`, and `TrustManager.checkServerTrusted(...)` get overridden with (almost) empty code implementation, security experts consider those overridden methods to be vulnerable. This

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

TABLE 1: The insecure and secure usage patterns of method APIs related to 13 Java classes/interfaces

Java Type API	Framework	Insecure	Secure	CWE Category
`Cipher`	JCA	The passed name of a requested transformation is RC2, RC4, RC5, DES, DESede, 3DES, AES/ECB, RSA with NoPadding or Blowfish.	The parameter value is AES/GCM, AES/CCM, AES/CFB, AEC/CBC, or {RSA, RSA/ECB, RSA/None} with OAEP padding.	CWE-327: Use of a Broken or Risky Cryptographic Algorithm
`HostnameVerifier`	JSSE	Allow all hostnames.	Disallow the hostnames that do not pass validation.	CWE-295: Improper Certificate Validation
`IvParameterSpec`	JCA	Create an initialization vector (IV) with a constant.	Create an IV with an unpredictable random value.	CWE-330: Use of Insufficiently Random Values
`KeyPairGenerator`	JCA	Create an RSA key pair whose key size $<$ 2048 bits, or create an ECC key pair whose key size $<$ 224 bits.	RSA key size $>=$ 2048 bits, or ECC key size $>=$ 224 bits.	CWE-326: Inadequate Encryption Strength
`KeyStore`	JCA	When loading a keystore from a given input stream, the provided password is a hardcoded constant non-null value.	The password is retrieved from some external source (e.g., database or file)	CWE-798: Use of Hard-coded Credentials
`MessageDigest`	JCA	The used hash algorithm is MD2, MD5, SHA-1, or SHA-224.	The parameter value is SHA-256, SHA-512 or SHA-3.	CWE-327
`PBEKeySpec`	JCA	Create a PBEKey based on a constant salt.	Set the salt to an unpredictable random value.	CWE-330
`PBEParameterSpec`	JCA	Create a parameter set for password-based encryption (PBE) by setting salt size $<$ 64 bits, iteration count $<$1000, or a constant salt.	Set salt size $>=$ 64 bits, iteration count $>=$1000. Set the salt to an unpredictable random value.	CWE-326, CWE-330
`SecretKeyFactory`	JCA	Create a secret key with the algorithm DES, DESede, ARCFOUR, or PBEWithMD5AndDES.	Create a secret key with AES or PBEWithHmacSHA256AndAES_128.	CWE-327
`SecretKeySpec`	JCA	Create a secret key with a hardcoded constant.	Create a secret key with a raw key (i.e., credential) retrieved from some external source or an unpredictable random value dynamically generated.	CWE-798
`SecureRandom`	JCA	Use `Random` to generate random values, set `SecureRandom` to use a constant seed, or invoke `SecureRandom.setSeed()` before calling any `nextXXX()` method (e.g., `nextInt()`).	Replace `Random` with `SecureRandom`. If `setSeed(...)` is called, use a parameter that is generated by `nextBytes()`.	CWE-330
`SSLContext`	JSSE	Use the protocol SSL, SSLv2.0, SSLv3.0, TLSv1.0, or TLSv1.1.	Use the protocol TLSv1.2 or TLSv1.3.	CWE-757: Selection of Less-Secure Algorithm During Negotiation
`TrustManager`	JSSE	Trust all clients and servers.	Check clients or servers.	CWE-295

is because with naïve or empty code implementation, a software program does not validate, or incorrectly validates, hostnames and/or certificates; it may allow an attacker to spoof a trusted entity by interfering in the communication path between the host and client [31]. For instance, Listing 2 shows the empty bodies for `checkXXX(...)` methods of `TrustManager`. Such naïve method overriding actually voids the intended protection mechanism offered by JSSE.

3. CWE-330: Use of Insufficiently Random Values. When some methods of four Java types (i.e., `IvParameterSpec`, `PBEKeySpec`, `PBEParameterSpec`, and `SecureRandom`) get called with constants or predictable random values, the method calls are considered insecure. This is because when software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information [32]. Line 6 of Listing 1 presents an exemplar API misuse, which creates an `IvParameterSpec` object with a constant array derived from the string literal `"12345678"`.

4. CWE-326: Inadequate Encryption Strength. When certain methods of two Java types (i.e., `KeyPairGenerator` and `PBEParameterSpec`) are invoked, if the parameter values are constants within specific value ranges, the invocations are considered vulnerable. This is because when generating keys or creating parameters used for password-based encryption (PBE), if a program specifies relatively low numbers for key lengths, salt sizes, or iteration counts, the leveraged encryption scheme is theoretically sound but not strong enough for the level of protection required. The weak encryption scheme can be subjected to brute force attacks that have a reasonable chance of succeeding using current attack methods and resources [33]. Listing 3 shows an exemplar misuse of `KeyPairGenerator` APIs, which initializes a generator to create RSA key pairs with the 1024-bit key size; however, the key size should be no smaller than 2048.

Listing 3: An exemplar misuse of the `KeyPairGenerator` APIs

```
1 KeyPairGenerator gen=KeyPairGenerator.getInstance("RSA");
2 // The RSA key size should be at least 2048 (CWE-326).
3 gen.initialize(1024);
4 KeyPair kp = gen.generateKeyPair();
```

5. CWE-798: Use of Hardcoded Credentials. When a program calls `KeyStore.load(...)` or `SecretKeySpec(...)` with a hardcoded constant as the credential parameter, the method invocation is treated vulnerable. The reason is that hard-coded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the software administrator [34]. Line 8 in Listing 1 shows an exemplar API misuse of this category. In the example, `SecretKeySpec(...)` is invoked with `desKey`, which credential comes from a hardcoded constant `"12345678"`.

6. CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade'). When a program calls `SSLContext.getInstance(...)` with any of the following parameters: `"SSL"`, `"SSLv2"`, `"SSLv3"`, `"TLSv1.0"`, and `"TLSv1.1"`, the invocation is treated insecure. Secure Socket Layer (SSL) and TLS (Transport Layer Security) are standard protocols for keeping an internet connection secure and safeguarding the transmitted data [35]. As a successor of SSL, TLS is more secure. Security exerts recommend to enforce TLS 1.2 as the minimum protocol version and to disallow older versions like TLS 1.0. Failure to do so could open the door to downgrade attacks: a malicious actor who is able to intercept the connection could modify the requested protocol version and downgrade it to a less secure version [36], [37].

Summary: We defined Table 1 to include only the API misuse patterns *frequently* mentioned by literature for three reasons. First, different literatures sometimes define conflicting patterns, so focusing on the most common ones can avoid arguable cases. Second, these patterns enable us to empirically compare the effectiveness of different tools on the same benchmarks. Third, these patterns are representative, so our study based on them can reflect developers' general perception of security-API misuses.

3 A Literature Survey of Current Tools

Tools were built to automatically detect Java cryptographic API misuses. These tools typically start with certain representations of misuse patterns, adopt different techniques to scan programs for pattern matches, and generate reports when matches are found.

Table 2 shows an overview of existing detectors. This list is complete to the best of our knowledge. We searched for literatures published in 2020, with keywords "secure API misuse Java" in the ACM digital library. We then read the retrieved papers together with their references to identify all relevant tools. For each tool, this table summarizes the pattern representation, pattern-matching strategy, and output; it also characterizes two properties: availability and the input format. **Availability** describes whether a tool is publicly available or just has its methodology described in literature. **Input format** reflects whether a tool can analyze Java source code, JAR files, or APK files. Among the 20 tools studied, there are 15 research prototypes from academia, and 5 tools from industry or open source communities.

3.1 Research Prototypes from Academia

Most tools that fall into this category scan the APK files of Android apps. Among the 15 tools, 10 tools hardcode patterns as built-in rules probably due to the simplicity of such representations; another 5 tools represent patterns with code snippets or templates written in a domain-specific language. 11 tools conduct inter-procedural static analysis for pattern matching, perhaps due to the relatively higher precision of such analysis than intra-procedural analysis and other techniques. Five tools suggest customized repairs.

MalloDroid [12] scans the decompiled code of Android apps to detect potential vulnerabilities related to SSL. It uses intra-procedural static analysis to i) extract networking API calls and valid HTTP(S) URLs, ii) check the validity of SSL certificates for all extracted HTTPS hosts, and iii) identify apps that validate certificates inadequately (CWE-295).

CryptoLint [15] is similar to MalloDroid, because it also extends Androguard [49]—a tool to decompile APK files into Dalvik bytecode and to statically analyze the bytecode. However, CryptoLint hardcodes six rules:

1) Do not use ECB mode for encryption (CWE-327).
2) Do not use a non-random IV for CBC encryption (CWE-330).
3) Do not use constant encryption keys (CWE-798).
4) Do not use constant salts for PBE (CWE-330).
5) Do not use fewer than 1,000 iterations for PBE (CWE-326).
6) Do not use static seeds to seed `SecureRandom(...)` (CWE-330).

For each located potentially vulnerable API call (e.g., `Cipher.getInstance(v)`), CryptoLint conducts inter-procedural backward slicing to decide whether the used parameter value is insecure (e.g., `v="AES/ECB"`). Although its design has been followed by later tools, CryptoLint is not publicly available.

BinSight [38] reimplements CryptoLint. However, its design seems better as it maps Java classifiers to their container software (i.e., an Android app or a third-party library) in a semi-automated way. When Android apps are obfuscated and identifiers are renamed, it is challenging to correctly map detected vulnerabilities to the original code or software libraries. To overcome this challenge, BinSight implements several heuristics to automate identifier mapping.

CDRep [39] automates both the detection and repair of security API misuses for Android apps. CDRep reimplements the design of CryptoLint for vulnerability detection. To repair vulnerabilities, CDRep leverages seven manually created patch templates, with each template usable to fix one misuse pattern.

CryptoTutor [19] helps students locate and repair cryptographic API misuses in Java code. Similar to CDRep, CryptoTutor applies program slicing and inter-procedural data flow analysis to locate misuses. Its built-in rules are also related to the vulnerabilities of CWE-327, CWE-330, CWE-798, and CWE-326. However, CryptoTutor focuses on a larger pattern set; in addition to the misuses examined by CryptoLint, CryptoTutor also checks API misuses that

1) use weak hash functions (e.g., MD5),
2) use weak encryption algorithms (e.g., DES),
3) use weak random number generators (e.g., `Random(...)`, and
4) use short-length keys or salts for encryption.

CryptoTutor repairs vulnerabilities by editing abstract syntax trees (ASTs) for code transformation. Once API misuses are located in the code submitted by a student, CryptoTutor provides coding feedback to help the student understand why the program is incorrect.

Crypto Misuse Analyzer (CMA) [40] scans Dalvik bytecode of Android apps and checks for API misuses related to CWE-327, CWE-295, CWE-330, CWE-326, and CWE-798. CMA first uses inter-procedural static analysis to identify all execution paths that may invoke certain cryptographic APIs. Based on the analysis result, CMA instruments code to perform dynamic analysis, log execution profiles, and record how cryptographic APIs are invoked at runtime. Finally, CMA matches execution profiles with predefined API-misuse models, to decide whether any API is misused.

CryptoChecker [41] also detects cryptographic API misuses based on built-in rules. Before hardcoding rules into CryptoChecker, Paletov et al. first built a rule inference tool called DiffCode. There are three steps in DiffCode. First, DiffCode mines code changes from GitHub repositories based on their usage of particular crypto APIs (e.g., `SecertKeySpec`). Second, to filter out irrelevant changes from the mined corpus, DiffCode represents invoked APIs and related parameter values with directed acyclic graphs (DAGs). By comparing DAGs, DiffCode extracts API usage changes and then clusters similar changes to infer API misuse patterns.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

TABLE 2: Overview of existing detectors for security API misuses

Name	Availability	Input Format			Pattern Representation		Pattern-Matching Strategy			Output	
		Java	JAR	APK	Built-in Rules	Other	Intra-	Inter-	Other	Misuse	Repair
MalloDroid [12]	✓			✓	✓		✓			✓	
CryptoLint [15]				✓	✓			✓		✓	
BinSight [38]				✓	✓			✓		✓	
CDRep [39]				✓	✓			✓		✓	✓
CryptoTutor [19]	✓	✓			✓			✓		✓	✓
CMA [40]				✓	✓			✓	✓	✓	
CryptoChecker [41]		✓			✓		-	-	-	✓	
Amandroid [42]	✓			✓	✓			✓		✓	
CogniCrypt [18]	✓	✓	✓	✓		✓		✓		✓	
Hotfixer [43]		✓	✓	✓		✓		✓		✓	✓
Fischer et al.'s tool [10]		✓		✓		✓	✓		✓	✓	
Xu et al.'s tool [44]				✓		✓		✓	✓	✓	
CryptoGuard [11]	✓		✓	✓	✓			✓		✓	
VuRLE [45]		✓				✓	✓			✓	✓
Vulvet [46]				✓	✓			✓		✓	✓
AndroBugs [47]	✓			✓	✓		✓			✓	
FindSecBugs [20]	✓		✓		✓			✓		✓	
MobSF [48]	✓	✓		✓	✓	✓			✓	✓	
SonarQube [21]	✓	✓			✓			✓		✓	
Xanitizer [22]	✓	✓	✓		✓			✓		✓	

"-" means the information is not publicly available.

The mined rules are related to CWE-327, CWE-330, CWE-326, and CWE-798. However, CryptoChecker's pattern set is much smaller than that of CryptoTutor. It is unclear what technique CryptoChecker adopts to match patterns.

Amandroid [42], [50] is a general-purpose static analysis framework, to decide points-to information for all objects in a flow- and context- sensitive way across Android app components. This technique seems more accurate than prior work. As the researchers noted, the event-driven nature and inter-component communication (ICC) of Android apps make traditional analysis insufficient and imprecise, and require additional processing to connect the control flow graphs of some seemingly irrelevant functions. Wei et al. [42] demonstrated that Amandroid can be easily extended to find API misuses that adopt ECB mode for encryption (CWE-327).

CogniCrypt [18] supports developers to properly use APIs in two ways. First, for some common tasks (e.g., data encryption), CogniCrypt generates code from high-level task descriptions in English. Second, CogniCrypt takes in rules defined in a domain-specific language (DSL)—CrySL [27]—to detect API misuses. Each CrySL rule has five mandatory sections:

(1) OBJECTS declares Java objects;
(2) EVENTS lists all security APIs involved;
(3) ORDER uses a regular expression to define correct API call sequences;
(4) CONSTRAINTS defines constraints on objects; and
(5) ENSURES defines predicates on the relationship among objects.

Because CogniCrypt translates CrySL rules into context-sensitive, flow-sensitive, and demand-driven static analysis, users can extend the tool capability by defining new rules. For detected API misuses, CogniCrypt offers fixing guidance (e.g., replacing an insecure parameter value with a secure one). Its pattern set is related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757.

Hotfixer [43] adopts CogniCrypt to detect API misuses, and applies fixes at runtime without stopping the program execution. To dynamically update software, Hotfixer transforms handcrafted software patches into hotfixes that are usable by Java agents, and checks the program execution status before applying any patch. For instance, if a method is running and a redefinition of that method is suggested as a security patch, then the method's old implementation will continue running until the execution finishes. Hotfixer ensures that only future runs of that method will execute the new implementation.

Fischer et al.'s tool [10] detects misuses in two ways: machine learning and clone detection. Specifically, their first approach uses *tf-idf* to generate features from source code. It trains a support vector machine (SVM) with an annotated dataset of code snippets that use APIs securely or insecurely. The trained model predicts whether a code snippet misuses any security API. Their second approach converts both known vulnerable Java code and Android apps to the same representation, i.e., the internal representation (IR) of WALA [51]—a widely used program analysis framework. The approach scans Android apps for clones (i.e., similar code) of the known vulnerable code, by finding isomorphic subgraphs in IR-based program dependency graphs (PDG). Both tools focus on misuse patterns related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757.

Xu et al.'s tool [44] is similar to that of Fischer et al., as it also detects misuses via machine learning. The approach first analyzes the Dalvik bytecode of APK files to extract all possible API invocation sequences from Android apps, and uses CogniCrypt to label secure and insecure call sequences. With the labeled dataset, the tool trains (1) a Hidden Markov Model (HMM) to predict how likely a given API sequence is secure, and (2) an n-gram model to further locate the misused API(s) in a problematic call sequence. As this tool adopts CogniCrypt to label training data, the pattern set it learns overlaps with, but can be no larger than that of CogniCrypt.

CryptoGuard [11], [52] extends Soot [53], [54]—a widely used program analysis framework—to statically analyze Java bytecode. It focuses on vulnerabilities of CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. To achieve high precision rates when detecting vulnerabilities, CryptoGuard conducts both backward and forward slicing in a context- and field- sensitive way. However, because

eight of CryptoGuard's rules are about the usage of constant values, naïvely applying existing slicing techniques can falsely report constants that are covered by program slices but totally irrelevant to security. Therefore, CryptoGuard defines refinement algorithms to remove false alarms based on the domain knowledge of cryptography.

VuRLE [45] detects and fixes vulnerabilities. There are two phases in VuRLE: learning and repair. In Phase I, given vulnerable programs and corresponding repaired code, VuRLE first extracts edits by comparing the ASTs of each ⟨vulnerable, repaired⟩ code pair; it represents each edit as a sequence of AST edit operations (e.g., node insertion). Next, VuRLE clusters similar edits based on the longest common subsequences (LCSs) between edit operations; for each cluster, VuRLE generalizes a ⟨template, edit⟩ pair. Here, the template abstractly represents a vulnerable code pattern, while the edit pattern represents the repair. In Phase II, given a vulnerable program, VuRLE scans code for matches of any inferred template. For each template match, VuRLE customizes the corresponding edit pattern, and applies the customized changes to repair vulnerabilities. Its pattern set is about CWE-327, CWE-295, CWE-798, and some categories outside our research scope (e.g., resource leakage).

Vulvet [46] extends Soot to statically analyze the Dalvik bytecode of Android apps; it detects and fixes cryptographic API misuses as well as other types of vulnerabilities (e.g., ICC-related). The patterns Vulvet focuses on are related to CWE-327, CWE-295, CWE-330, and CWE-798. Vulvet automatically resolves vulnerabilities by instrumenting patches to the Jimple code of Android apps, where Jimple is an internal representation of Soot.

3.2 Tools from Industry or Open Source Communities

We found five tools that are from either industry or open source communities. None of them has any paper published to describe the tool design or implementation. Thus, our descriptions below are based on tool websites, manual code inspection of open-source tools, and our first-hand user experience with tools. Most of these tools hardcode misuse patterns as built-in rules and conduct inter-procedural analysis; none of the tools suggests customized repairs.

AndroBugs [47] is an open-source framework to scan Android apps for vulnerabilities. Among all the vulnerability categories AndroBugs considers, two categories are within our research scope: CWE-295 and CWE-798. AndroBugs implements a naïve string-match method, to detect API misuses that match certain regular expressions (regex).

FindSecBugs [20] is the SpotBugs [55] plugin for security audits of Java web applications. Here, SpotBugs is an open-source tool that statically analyzes Java bytecode for software bugs. FindSecBugs performs inter-procedural static program analysis to find API misuses related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. The misuse patterns in FindSecBugs are hardcoded as built-in rules; however, the software architecture provides extensible interfaces for developers to easily add or remove rules. For each detected vulnerability, FindSecBugs can provide general guidance on repairs by showing code examples; it does not suggest concrete repairs applicable to any specific program context.

MobSF [48] is an open-source, automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, manual analysis, and security assessment framework. It performs both static and dynamic analysis to detect a variety of vulnerabilities. Among the categories listed in Table 1, MobSF locates API misuses related to CWE-327, CWE-295, CWE-330, and CWE-798. To identify improper certificate validation (CWE-295), MobSF hardcodes built-in rules and performs dynamic program analysis. To reveal other API misuses, MobSF holds an independent YAML file to represent misuse patterns with regular expressions (regex), and conducts regex-based string match.

SonarQube [21] is an open-source tool that conducts inter-procedural static analysis to detect bugs, code smells, and security vulnerabilities. SonarQube hardcodes built-in rules for vulnerabilities of CWE-327, CWE-295, CWE-330, CWE-326, and CWE-757. For each detected vulnerability, SonarQube presents general guidance on repairs by showing (1) secure code examples and (2) relevant CWE entries.

Xanitizer [22] is a closed-source commercial tool for the security audit of Java web applications. We were able to use Xanitizer by requesting for the user license. According to our experience with the tool, Xanitizer conducts inter-procedural static analysis to reveal API misuses of CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. It scans not only Java code and JAR files, but also configuration files and templates for rendering the HTML output. For each detected vulnerability, Xanitizer offers a high-level repair suggestion (e.g., "specify crypto-provider") together with relevant CWE entries.

Finding 1: *For RQ1, existing tools are different in terms of their availability, input formats, pattern representations, pattern-matching strategies, and outputs. Most tools represent patterns as built-in rules, conduct inter-procedural analysis, and report detected API misuses as outputs.*

4 Empirical Comparison of Tools

To empirically compare the tools listed in Table 2, we first tried to download all tools and deploy them to our desktop, and then applied the successfully deployed ones to existing datasets. The configuration of our desktop includes (1) OS: Linux Mint 20, (2) CPU: i7-8700, (3) memory size: 32 GB, and (4) JVM heap size: 30 GB. In this section, we will first introduce the experimented tools (Section 4.1) and evaluation datasets (Section 4.2). Then we will explain our evaluation metrics (Section 4.3) and results (Section 4.4).

4.1 Tools Used in Experiments

Within the tools listed in Table 2, nine tools are unavailable and do not support any free trial. Although BinSight is open-source, we could not compile or run it, neither did the authors respond to our email requests. Thus, we still consider BinSight unavailable. Among the remaining 10 tools, MalloDroid, Amandroid, AndroBugs, and MobSF are only applicable to Android apps. Because our evaluation datasets include only six Android apps (see Section 4.2), which are insufficient to evaluate any tool, we decided not to experiment with these four tools. Finally, we have six tools usable in the empirical comparison of tools' detection

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

TABLE 3: The API misuse patterns covered by each tool

Java Type API	CogniCrypt	CryptoGuard	CryptoTutor	FindSecBugs	SonarQube	Xanitizer
`Cipher`	✓	✓	✓	✓	✓	✓
`HostnameVerifier`		✓		✓	✓	✓
`IvParameterSepc`	✓	✓	✓	✓	✓	✓
`KeyPairGenerator`	✓	✓		✓	✓	✓
`KeyStore`	✓	✓		✓		✓
`MessageDigest`	✓	✓	✓	✓	✓	✓
`PBEKeySpec`	✓	✓		✓	✓	✓
`PBEParameterSpec`	✓	✓	✓		✓	✓
`SecrectKeyFactory`	✓		✓			✓
`SecrectKeySpec`	✓	✓	✓	✓		✓
`SecureRandom`	✓	✓	✓	✓	✓	✓
`SSLContext`	✓	✓		✓	✓	✓
`TrustManager`	✓	✓	✓	✓	✓	✓

"✓" means a Java class/interface has at least one method-API misuse pattern covered by a tool

TABLE 4: The tool versions adopted

Tool	Version or Commit Id on Github
CogniCrypt	2.7.1
CryptoGuard	Release_04.05.03_2020-11-25-02-42
CryptoTutor	v202107
FindSecBugs	1.10.1
SonarQube	8.5.1.38104
Xanitizer	5.1.3

capability: CogniCrypt, CryptoGuard, CryptoTutor, FindSecBugs, SonarQube, and Xanitizer. The first three tools are from academia, while the last three come from industry. Table 4 shows the tool versions we adopted. As shown in Table 3, among the six tools, Xanitizer covers the most misuse patterns while CryptoTutor covers the fewest.

4.2 Benchmark Datasets

To evaluate the effectiveness of tools, we searched extensively online for third-party benchmarks that label programs based on their correct or incorrect usage of security APIs. We found three datasets:

(1) CryptoBench [23], [56] includes 171 handcrafted programs that use the APIs of JCA and JSSE. In particular, 136 of the programs have cryptographic API misuses, while the other 35 programs use APIs correctly. To precisely comprehend API usage, a tool needs to do intra-procedural analysis for 40 programs, and perform inter-procedural analysis for the other 131 programs. Among the 136 vulnerable programs, only 129 programs contain the API misuses we focus on (see Table 1).

(2) MUBench [24], [57] is a benchmark of API-misuse detectors. Among the different versions of MUBench, we downloaded a recent version created in 2019 [58], which contains instances of cryptographic API misuses collected from 62 Java programs. These programs include 6 Android apps and 56 non-Android applications. We managed to compile 37 out of the 56 Java applications into JAR files, which correspond to 149 labeled instances of API misuses. Therefore, we used these 149 instances as ground truth in our evaluation.

(3) OWASP Benchmark [25], [59] is a Java test suite designed to evaluate the effectiveness of automated vulnerability detection. It gathers the vulnerabilities recently reported on CWE [28], and has been recommended as an evaluation dataset for Application Security Testing tools. We downloaded the latest version (v1.2) of this benchmark, which includes 2,740 Java programs. Because not all programs involve security APIs, we focused on the data of three categories: weak cryptography, weak hashing, and weak randomness. In this way, our experiment includes 975 programs from the original dataset, containing 477 programs with labeled misuses of security APIs and 498 programs with correct uses.

TABLE 5: The security APIs covered by each benchmark

Java Type API	CryptoBench	MUBench	OWASP Benchmark
`Cipher`	✓	✓	✓
`HostnameVerifier`	✓		
`IvParameterSepc`	✓	✓	
`KeyPairGenerator`	✓		
`KeyStore`	✓		
`MessageDigest`	✓	✓	✓
`PBEKeySpec`	✓		
`PBEParameterSpec`	✓	✓	
`SecrectKeyFactory`		✓	
`SecrectKeySpec`	✓	✓	
`SecureRandom`	✓		✓
`SSLContext`			
`TrustManager`	✓		

"✓" means that a Java class/interface has at least one method API called by a program benchmark.

Actually not every benchmark covers all the API misuses summarized by prior work. To ensure rigorous evaluation of tools, we manually inspected the security APIs labeled in each benchmark. We present the mapping between benchmarks and security APIs in Table 5. As shown in the table, CryptoBench covers the usage of most APIs (i.e., 11 Java types). The data of OWASP Benchmarks is only relevant to three Java classes: `Cipher`, `MessageDigest`, and `SecureRandom`. We chose to use existing benchmark datasets instead of creating new ones for two reasons. First, these benchmarks are public and were manually crafted by different groups of people, which makes our empirical comparison representative, and easy to reproduce by other people. Second, some of the benchmarks (e.g., OWASP Benchmark and MUBench) are widely accepted and have great industrial impacts, which enables our empirical results to better characterize the state-of-the-art tools and inspire future research.

4.3 Evaluation Metrics

We used four metrics to measure tool effectiveness: precision, recall, F-score, and runtime overhead.

Precision (P) measures among all reported misuses, how many of them are actual misuses (i.e., true positives):

$$P = \frac{\text{\# of true misuses detected}}{\text{Total \# of detected misuses}}. \quad (1)$$

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

TABLE 6: The precision, recall, and F-score of tools measured based on CryptoBench (%)

Java Type API	CogniCrypt			CryptoGuard			CryptoTutor			FindSecBugs			SonarQube			Xanitizer		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
`Cipher (36)`	85	78	81	85	97	91	67	22	33	62	36	46	43	17	24	83	83	83
`HostnameVerifier (1)`	-	0	-	-	0	-	-	0	-	100	100	100	50	100	67	100	100	100
`IvParameterSepc (8)`	71	63	67	88	88	88	0	0	-	80	100	89	-	0	-	89	100	94
`KeyPairGenerator (5)`	83	100	91	80	80	80	-	0	-	-	0	-	-	0	-	83	100	91
`KeyStore (7)`	75	86	80	88	100	93	-	0	-	100	29	44	-	0	-	100	29	44
`MessageDigest (24)`	94	67	78	86	100	92	67	26	36	67	33	44	80	83	82	83	83	83
`PBEKeySpec (8)`	63	63	63	86	75	80	-	0	-	100	25	40	100	50	67	73	100	84
`PBEParameterSpec (14)`	77	71	74	85	79	81	67	14	24	-	0	-	-	0	-	-	0	-
`SecrectKeySpec (8)`	78	88	82	100	38	55	50	13	20	67	25	36	-	0	-	50	13	20
`SecureRandom (15)`	100	7	13	86	80	83	50	13	21	100	7	13	100	60	75	-	0	-
`TrustManager (3)`	-	0	-	-	0	-	-	0	-	100	100	100	100	100	100	100	100	100
Overall (129)	**81**	**64**	**72**	**86**	**84**	**85**	**61**	**15**	**24**	**73**	**31**	**43**	**75**	**33**	**46**	**83**	**60**	**70**

"-" means that a tool does not report any misuse for certain Java type APIs, so the related precision and F score values cannot be calculated.

Given a reported set of misuses S_1, suppose that the labeled set of misuses (i.e., ground truth) is S_2. Theoretically, we can automatically evaluate precision based on the intersection between two sets, i.e., $P = |S_1 \cap S_2|/|S_1|$. Such automatic evaluation requires the ground truth (i.e., S_2) to be complete. Namely, the labeled set of misuses in each benchmark should cover all actual misuses existing in codebases.

Because CryptoBench and OWASP are manually crafted datasets with injected API misuses, their ground truth sets are complete. However, MUBench consists of software from the real world, and the labeled set was crafted based on manual inspection or tool results. The ground truth of MUBench is incomplete and thus unusable for automatic evaluation. To correctly compute precision of tools on MUBench, we manually inspected all reported misuses and decided whether they were true positives based on our security knowledge. Namely, given a reported API misuse, if our manual inspection of the program context confirms the misuse, we consider the report to be a true positive; otherwise, it is a false positive.

Recall (R) measures among all known API misuses in benchmarks, how many of them are detected by a tool:

$$R = \frac{\text{\# of true misuses detected}}{\text{Total \# of known true misuses}}. \quad (2)$$

Given a reported set of misuses S_1, suppose that the labeled set of misuses is S_2. We evaluated recall using $|S_1 \cap S_2|/|S_2|$.

F-score (F) is the harmonic mean of P and R, to reflect a trade-off between the two metrics:

$$F = \frac{2 \times P \times R}{P + R}. \quad (3)$$

F varies within [0, 1]. The higher F scores are desirable, because they demonstrate better trade-offs between precision and recall. Suppose that we have 100 known API misuses in a codebase; a tool reports 120 misuse instances, with 80 of them being true misuses. Then $P = 80/120 = 67\%$, $R = 80/100 = 80\%$, $F = 2 \times 80\% \times 67\%/(80\% + 67\%) = 73\%$.

Runtime Overhead measures the time cost of each tool. The lower overhead, the better.

4.4 Results Based on Benchmarks

Table 7 shows the measured runtime overheads. Within the six tools, Xanitizer spent the most time when being applied to CryptoBench and OWASP (i.e., 95 and 490,729 seconds); CryptoGuard got the highest time cost when being applied to MUBench (i.e., 1,918 seconds). Two reasons can explain the observed time differences among tools. First, these tools adopt distinct static analysis techniques (e.g., inter-procedural vs. intra-procedural) to match patterns, and some techniques are more time-consuming than others. Second, tools focus on different vulnerability patterns, although some patterns are irrelevant to cryptographic APIs. Since we were unable to revise tools to disable all patterns irrelevant to our investigation, the measured costs are higher than the actual time costs incurred by automatic detection of cryptographic APIs.

TABLE 7: Time cost comparison between tools (seconds)

Tool	CryptoBench	MUBench	OWASP
CogniCrypt	7	133	10,773
CryptoGuard	11	1,918	9,045
CryptoTutor	22	530	-*
FindSecBugs	4	59	20,352
SonarQube	28	387	2,188
Xanitizer	95	724	490,729

* CryptoTutor does does not run successfully with the OWASP benchmark.

Finding 2 (for RQ2): *The measured time costs imply that given hundreds of programs to scan, the experimented tools usually respond within six hours (18,000 seconds).*

In terms of detection capability, CryptoGuard achieved the highest F score (85%) among all tools when being applied to CryptoBench. Meanwhile, Xanitizer acquired the highest F scores (i.e., 72% and 100%) when being applied to MUBench and OWASP datasets. In the following subsections, we will further discuss the precision, recall, and F-score of tools on each dataset.

4.4.1 CryptoBench

Table 6 shows the evaluation results of different tools on CryptoBench. Each row in the table corresponds to API misuses related to one Java type (e.g., `Cipher`); each number mentioned in the first column (e.g., 36) counts the labeled cryptographic API misuse instances for a Java type. Among the 11 Java types covered by CryptoBench, CryptoTutor and SonarQube separately reported API misuses for 6 Java types; the other tools reported API misuses related to 9 Java types. There are only two Java types whose API misuses are commonly detected by all tools: `Cipher` and `MessageDigest`; CryptoGuard consistently outperformed the others when handling these common cases. Our observation indicates that it is promising to improve vulnerability detection by combining the results of different tools, although we have not seen any hybrid approach built in this way.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

TABLE 8: The precision, recall, and F-score of tools measured based on MUBench (%)

Java Type API	CogniCrypt			CryptoGuard			FindSecBugs			SonarQube			Xanitizer		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
`Cipher (50)`	92	66	77	100	68	81	100	66	80	100	76	86	86	78	82
`IvParameterSpec (13)`	88	62	72	100	31	47	69	46	55	-	0	-	65	46	54
`MessageDigest (31)`	89	58	70	100	81	89	77	68	72	86	55	67	85	77	81
`PBEParameterSpec (7)`	79	100	88	100	100	100	-	0	-	100	29	44	100	57	73
`SecretKeyFactory (9)`	84	100	91	-	0	-	-	0	-	-	0	-	82	100	90
`SecretKeySpec (39)`	48	62	54	100	8	14	100	3	5	-	0	-	100	28	44
Overall (149)	**77**	**66**	**71**	**100**	**49**	**66**	**84**	**41**	**55**	**93**	**38**	**54**	**85**	**62**	**72**

TABLE 9: The precision, recall, and F-score of tools measured based on OWASP Benchmark (%)

Java Type API	CogniCrypt			CryptoGuard			FindSecBugs			SonarQube			Xanitizer		
	P	R	F	P	R	F	P	R	F	P	R	F	P	R	F
`Cipher (130)`	83	100	91	83	100	91	100	100	100	83	100	91	100	100	100
`MessageDigest (129)`	100	69	82	100	69	82	100	69	82	-	0	-	100	100	100
`SecureRandom (218)`	-	0	-	100	100	100	100	100	100	-	0	-	100	100	100
Overall (477)	**89**	**46**	**61**	**94**	**92**	**93**	**100**	**92**	**96**	**83**	**27**	**41**	**100**	**100**	**100**

```
1. public class BrokenCryptoABICase12 {
2.   public static void method2(String c) throws
  NoSuchPaddingException, NoSuchAlgorithmException,
  InvalidKeyException {
3.     String cryptoAlgo = c;
4.     method1(cryptoAlgo);
5.   }
6.   public static void method1(String crypto) throws
  NoSuchPaddingException, NoSuchAlgorithmException,
  InvalidKeyException {
7.     KeyGenerator keyGen=KeyGenerator.getInstance(crypto);
8.     SecretKey key = keyGen.generateKey();
9.     Cipher cipher = Cipher.getInstance(crypto);
10.    cipher.init(Cipher.ENCRYPT_MODE, key);
11.  }
12.  public static void main (String [] args) throws
  NoSuchPaddingException, NoSuchAlgorithmException,
  InvalidKeyException {
13.    String crypto = "Blowfish";
14.    method2(crypto);
15.  }
16.}
```

Fig. 1: A program in which SonarQube and FindSecBugs could not find the API misuse

The overall F-score comparison among tools is CryptoGuard>CogniCrypt>Xanitizer>SonarQube>FindSecBugs>CryptoTutor. CryptoTutor obtained much lower measurements than other tools for two reasons. First, it has implementation issues, which prevented the tool from correctly identifying API misuses in many cases. Second, CryptoTutor scans code for fewer patterns than other tools, and could not reveal the API misuses beyond its pattern set. SonarQube and FindSecBugs worked worse than CogniCrypt, CryptoGuard, and Xanitizer for two reasons. First, SonarQube and FindSecBugs have smaller pattern sets. Second, both tools apply intra-procedural instead of inter-procedural analysis to locate some API misuses (e.g., using constant IV values), although the inter-procedural program analysis is more desirable. As CryptoBench has many programs that require sophisticated inter-procedural analysis, neither SonarQube nor FindSecBugs handled well those programs.

Fig. 1 shows a program whose API misuse was not detected by either SonarQube or FindSecBugs. On line 9, `Cipher.getInstance(...)` is invoked with parameter `crypto`, whose actual value "`Blowfish`" implies the adoption of a provenly insecure algorithm. To facilitate understanding, in Fig. 1, we underlined all statements involved in the backward slice of that API invocation (i.e., lines 2–4, 6, 9, 13–14); we also marked the data-dependencies between statements with arrowed blue curves. As implied by the data dependencies, a tool has to conduct inter-procedural backward slicing in order to reveal the API misuse. Nevertheless, SonarQube and FindSecBugs failed to do that.

4.4.2 MUBench

We applied five tools (except CryptoTutor) to MUBench, because CryptoTutor is quite unique. When running tools other than CryptoTutor, we managed to launch tools via commands, apply each tool to every program benchmark, and automatically process the output text (or files) to compute tools' P/R/F values. However, CryptoTutor is an interactive coding assistance tool that is delivered as an Eclipse plugin. To launch the tool, we have to first launch a new instance of Eclipse, import all programs into the IDE for each benchmark, manually right-click all programs to launch CryptoTutor, read outputs in GUI panes, and manually inspect subject programs as needed to compute P/R/F values. Such frequent manual interference can be extremely time-consuming, when the subject programs are large and CryptoTutor does not output the code locations of identified API misuses. As we could not afford the manual effort, we did not apply CryptoTutor to MUBench.

Our results are shown in Table 8. We observed similar phenomena in this table and Table 6. Among the six Java types covered by MUBench, SonarQube revealed API misuses for the fewest Java types (i.e., three). Only two types have API misuses commonly detected by all tools: `Cipher` and `MessageDigest`. SonarQube achieved the highest F score (i.e., 86%) for misuses of `Cipher`'s method APIs, and CryptoGuard achieved the highest F score (i.e., 89%) for `MessageDigest`-related misuses. No tool consistently outperformed others. The overall F-score comparison is Xanitizer>CogniCrypt>CryptoGuard>FindSecBugs>SonarQube. FindSecBugs and SonarQube obtained much lower F scores than CogniCrypt, CryptoGuard, and Xanitizer.

4.4.3 OWASP

As shown in Table 9, OWASP Benchmark covers a lot fewer Java types than the two benchmarks. Among the three Java types covered, SonarQube only detected API misuses for `Cipher`. Xanitizer worked perfectly to report misuses without any incorrect result. The overall F-score comparison among tools is

```
In benchmark.properties,
1.   cryptoAlg1=DES/ECB/PKCS5Padding
2.   cryptoAlg2=AES/CCM/NoPadding
3.   …
In Properties.java,
4.   …
5.   public String getProperty(String var1, String var2) {
6.     String var3 = this.getProperty(var1);
7.     return var3 == null ? var2 : var3;
8.   }
9.   …
In BenchmarkTest00358.java,
10.  …
11.  java.util.Properties benchmarkprops = new
     java.util.Properties();
12.  benchmarkprops.load(this.getClass().getClassLoader().get
     ResourceAsStream("benchmark.properties"));
13.  String algorithm=benchmarkprops.getProperty("cryptoAlg2",
     "AES/ECB/PKCS5Padding");
14.  javax.crypto.Cipher c
     = javax.crypto.Cipher.getInstance(algorithm);
15.  …
```

Legend: Data-dependency

Fig. 2: A program that is falsely reported to be vulnerable by CryptoGuard and CogniCrypt

Xanitizer>FindSecBugs>CryptoGuard>CogniCrypt>SonarQube. This comparison seems contradictory with what we observed in Tables 6 and 8. Namely, compared with CryptoGuard and CogniCrypt, FindSecBugs worked better on this dataset but worse on the other datasets.

To understand the contradiction, we further inspected the codebases of FindSecBugs and OWASP benchmark. We realized that in the benchmark, there are multiple programs having the same code underlined in Fig. 2. The common code calls `Cipher.getInstance(...)` with parameter `algorithm`, whose actual string value "`AES/CCM/NoPadding`" implies the adoption of a secure algorithm. However, CryptoGuard and CogniCrypt are not rigorous enough to determine that `algorithm` does not hold the value of "`AES/ECB/PKCS5Padding`". Consequently, they falsely inferred that "`AES/ECB/PKCS5Padding`" is passed to `Cipher.getInstance(...)`, and reported API misuses. In comparison, FindSecBugs nicely handled these programs and did not report any false positive.

Finding 3 (for RQ2): *No tool consistently worked best. However, CogniCrypt, CryptoGuard, and Xanitizer always outperformed SonarQube, probably due to their sophisticated inter-procedural analysis and larger pattern sets.*

Among the experimented tools, we observed the highest F-score that they can achieve on CryptoBench is 85% (by CryptoGuard); the highest measurement on MUBench is 72% (by Xanitizer). These numbers imply that there is still improvement space for new approaches to detect misuses with higher F-scores. Additionally, we noticed that each adopted benchmark only covers at most 11 of the 13 Java types frequently involved in cryptographic API misuses. It means that to better assess the effectiveness of different tools, we also need new benchmarks that cover various API misuses related to all Java types.

5 User Study

To understand how existing tools help with developers' secure coding practices, we performed a user study. We reported cryptographic API misuses found in open-source projects to owner developers, to seek for their feedback.

TABLE 10: Summary of developers' responses to 57 PRs

Java Type API	PRs filed	Developers' Feedback		
		Positive	Negative	No Response
`Cipher`	7	5	2	0
`HostnameVerifier`	2	1	1	0
`IvParameterSepc`	1	1	0	0
`KeyPairGenerator`	3	1	1	1
`KeyStore`	7	0	6	1
`MessageDigest`	7	1	6	0
`PBEParameterSpec`	5	4	0	1
`SecretKeyFactory`	3	0	1	2
`SecretKeySpec`	4	0	2	2
`SecureRandom`	7	0	5	2
`SSLContext`	4	1	2	1
`TrustManager`	7	3	4	0
Total	**57**	**17**	**30**	**10**

Specifically, we ranked all Apache projects on GitHub in the descending order of their popularity (i.e., star counts). We then scanned the source code of top-ranked projects to find 200 projects that use any of the 13 Java types listed in Table 1. We chose to explore Apache projects because (1) they are usually well maintained, and (2) the project developers are experienced and often respond to pull requests (PRs).

Next, we applied all 5 experimented tools to the 200 Apache project to reveal cryptographic API misuses. Because the vulnerability reports by different tools contain true API misuses, together with false ones and other security issues out of scope, we needed to manually refine those reports before contacting developers for their feedback. To reduce our manual effort, in each tool's outputs, we sampled 15 projects. As we used 5 tools, in total we sampled 75 projects based on the reported vulnerabilities.

According to our manual analysis, the tools reported 416 true positives among the sampled projects. As it is infeasible for developers to respond to all instances, we further sampled 57 instances by taking a couple of steps. In Step 1, we classified all instances based on the Java types they are associated with. In Step 2, we randomly chose seven unique instances for each Java type to file PRs, in order to get developers' feedback on different kinds of misuses. When there are insufficient instances reported for any Java type (e.g., `IvParameterSpec`), we included all instances. In each PR, we specified (a) the code location of an instance, (b) the security implication, (c) one or two CWE entries showing the potential exploits, and (d) tool-generated guidance on fixes. When developers asked us to file issue reports, we also created issues to describe the above-mentioned information.

Based on our interactions with developers so far, we have classified developers' opinions into three categories: positive feedback, negative feedback, and no response. Surprisingly, developers rejected 53% of PRs (i.e., 30/57), agreed with us for 30% of PRs (i.e., 17/57), and did not respond for 18% of PRs (i.e., 10/57). PRs related to `Cipher` got the highest positive rate (i.e., 5/7). However, PRs related to another four Java types received zero positive response, including `KeyStore`, `SecretKeyFactory`, `SecretKeySpec`, and `SecureRandom`. Such comparison implies that developers considered certain vulnerability reports to be more important than others.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3150302

(a) An incomplete fixing suggestion by existing tools

```
1.  public static String encrypt(String text, SecretKey key) {
2.    …
5.    cipher.init(Cipher.ENCRYPT_MODE, key);
6.    … }
```

(b) The complete fixing suggestion that developers need

```
2.  public static String encrypt(String text, SecretKey key) {
3.    …
8.    … }
```

Fig. 3: The incomplete and complete fixing suggestions for the misuse `Cipher.getInstance("AES/ECB/PKCS5PADDING")`

Finding 4 (for RQ3): *Developers would like to address the security issues mentioned in 17 PRs, but rejected 30 PRs. This phenomenon implies that developers are usually negative towards the reported security-API misuses.*

Among the 17 PRs with positive feedback, developers finally accepted 9 PRs. They mentioned two challenges posed by the other eight PRs:

Challenge 1. Incomplete fixing suggestion. In four PRs, tool-generated guidance does not offer all needed information for repairs. For instance, a project invokes `Cipher.getInstance("AES/ECB/PKCS5PADDING")`, which misuse was located by tools and developers were recommended to replace the parameter with `"AES/CBC/PKCS5PADDING"`. However, naïvely applying this edit can cause a runtime error, because the substitute parameter requires for an additional initialization vector (IV) parameter when the cipher is used for encryption/decryption. With more details, Fig. 3 contrasts an incomplete fixing suggestion implied by existing tools and the complete suggestion that developers need. As reflected by Fig. 3 (b), when replacing the insecure parameter value with a secure one, developers must remember to also update a related API call `cipher.init(...)` (lines 6–7) and create an IV parameter for that updated call (line 1 and lines 9–15).

As current tools do not guarantee the comprehensiveness or completeness of their coding suggestions, developers hesitate to modify code based on the partial information.

Challenge 2. Complex repair procedures. Some developers concurred with the revealed vulnerabilities in six PRs, but could not cope with the complexity of secure solutions. For instance, when implementing `TrustManager`, developers understood that they should not blindly trust all clients and servers, neither should they have empty implementation for the interface methods `checkClientTrusted(...)` and `checkServerTrusted(...)`. However, to remove the vulnerability, they have to not only revise code in order to check the certificates of both clients and servers, but also download certificate files to local machines and properly configure a set of local files. This process is challenging and time-consuming, and developers have almost zero tool assistance for non-code artifact configuration.

Finding 5 (for RQ3): *For 8 PRs, developers were willing to address the reported vulnerabilities but could not do that. They need tools to provide more detailed suggestions on repairing edits and non-code artifact configuration.*

Developers rejected 30 PRs for follwing reasons:

Reason 1. No exploit demo. For four PRs, developers do not trust the described API misuses or related security implications; they required actual security attacks to demonstrate the security exploit. Particularly in one PR, we pinpointed the insufficient key length of an RSA key pair, provided guidance on fixes, and included CWE-327 [60] as a reference. However, developers still need more convincing reasons before accepting the PR. They replied, "*we were unable to identify any security impact. As such, this has been marked as Not Applicable. If you still believe this to be valid, please submit a new report which includes detailed information demonstrating and exploiting the security impact for this issue*".

Reason 2. False positives without actual security impact. Among the 26 PRs, developers believed that the reported vulnerabilities in 10 PRs exist only in outdated code (i.e., archived files or repositories), or in test suites that will not be included into the released software products. As the reported vulnerabilities will not influence their software products, developers do not want to fix the reported misuses. For example, some `TrustManager` implementation was intentionally designed to trust all incoming connections in test code; developers mentioned that they understood the listed concerns in PRs. However, they have two reasons not to fix the reported issues: (1) all these test files will not be shipped with the projects; (2) the trust-all mechanism was implemented on purpose.

For the remaining 16 PRs, developers considered the reported vulnerabilities to be totally irrelevant, as the API misuses are not located in security-sensitive software implementation. 11 of these PRs are about the usage of `MessageDigest` and `Random`. Although tools consider "`MD5`" and "`SHA-1`" as insecure parameters to use when calling `MessageDigest.getInstance(...)`, developers totally disagreed on that. They defended that in their circumstances, these APIs were called not for cryptographic hashing or signature requests; instead, the APIs were used only to generate checksums for data integrity checks. Thus, they do not believe the usage of MD5 or SHA-1 to be security-sensitive.

Listing 4: A scenario where `MD5` is irrelevant to security [61]

```
1 public MD5Function(List<Expression> children) throws
      SQLException {
2   super(children);
3   try {
4     messageDigest = MessageDigest.getInstance("MD5");
5   } catch (NoSuchAlgorithmException e) {
6     throw new RuntimeException(e);
7   }
8 }
```

Listing 4 shows a scenario where MD5 is used to generate message digests [61]. Concerning the code, developers explained, "*From a cursory check through Phoenix code, I see two uses of MD5: 1. An 'MD5' SQL function that allows users to generate MD5 fingerprints of columns. 2. To compare columns of primary and index tables against each other in the index scrutiny tool. Neither of these needs a cryptographic hash*".

Similarly, for `Random`, although tools consider the API to be a cryptographically weak random value generator and consistently suggest `SecureRandom` as a secure substitute, developers disagreed. For instance, the code in Listing 5 uses `Random` to randomly pick an endpoint as the target for gossip (lines 18–22). The developers responded "*The gossiper does not use any random calls for cryptography*". This response indicates that developers only used randomly generated values for purposes irrelevant to cryptography, so they did not believe that their API usage is vulnerable.

Listing 5: A case where `Random` is irrelevant to security [62]

```
1  public class Gossiper implements
       IFailureDetectionEventListener, GossiperMBean
2  {
3    private final Random random = new Random();
4    ...
5    /**
6     * Returns true if the chosen target was also a seed.
          False otherwise
7     *
8     * @param message
9     * @param epSet   a set of endpoint from which a random
          endpoint is chosen.
10    * @return true if the chosen endpoint is also a seed.
11    */
12   private boolean sendGossip(Message<GossipDigestSyn>
         message, Set<InetAddressAndPort> epSet) {
13     List<InetAddressAndPort> liveEndpoints =
           ImmutableList.copyOf(epSet);
14     int size = liveEndpoints.size();
15     if (size < 1)
16       return false;
17     /* Generate a random number from 0 -> size */
18     int index = (size == 1) ? 0 : random.nextInt(size);
19     InetAddressAndPort to = liveEndpoints.get(index);
20     ...
21     boolean isSeed = seeds.contains(to);
22     GossiperDiagnostics.sendGossipDigestSyn(this, to);
23     return isSeed;
24   }
25 }
```

Finding 6 (for RQ3): *Developers rejected 30 PRs because they disagreed upon the criteria tools adopted to recognize vulnerabilities. They need tools to demonstrate security exploits of vulnerabilities, and to skip issues located in test cases, archived code, and security-irrelevant context.*

6 Threats to Validity

Threats to External Validity: Our empirical findings may be limited to the misuse patterns we focused on, the tools we experimented with, the datasets used, and the developers who responded to our PRs. To mitigate this limitation, we intentionally included the misuse patterns frequently mentioned in literature, ran as many tools as possible, randomly sampled the most popular 200 Apache projects, and proactively discussed with developers on filed PRs. In the future, we will include more patterns, expand our datasets, and file more PRs.

Threats to Construct Validity: CryptoBench and OWASP Benchmark solely have crafted code with injected API misuse instances; they may not represent actual API misuses in real-world software. MUBench contains cryptographic API misuses in open-source programs; however, the ground truth of labeled misuses seems incomplete, and they may not represent API misuses in closed-source software. Therefore, our tool evaluation results may not reflect these tools' actual effectiveness in the real world. In the future, we will construct more comprehensive benchmarks using more real-world programs.

Threats to Internal Validity: In the user study, we manually checked tools' outputs, removed false alarms, and only sampled true misuses to file PRs. It is possible that our manual analysis is subject to human bias. To mitigate the problem, we had two authors inspect each sampled instance. In this way, we ensured that every filed PR contains a true misuse based on the pattern set defined in literature; when developers considered any PRs to be false positives, it indicates the discrepancy between developers' belief and research literature.

7 Recommendations on Future Tools

Our work compares existing tools and reveals the gap between tools' capabilities and developers' expectations. Our findings lead us to give the following recommendations.

Improve the F-score and relevance of misuse detection. Tool developers can improve over the state-of-the-art detectors in three ways: (1) to increase the accuracy of inter-procedural program analysis, (2) to skip vulnerabilities in test or outdated code, and (3) to perform context-aware analysis that examines API usage only inside the implementation of security functionalities. Especially for (3), new tools may need to characterize program context by locating cryptographic API usage, tracking the propagation of any value produced by those API calls, and deciding whether those value propagations are related to any security concept (e.g., encryption).

Provide detailed and customized fixing suggestions. To persuade developers into removing detected API misuses, it is important to provide actionable suggestions on how to correctly use those APIs in developers' circumstances. Developers found existing repair guidance to be insufficient. Therefore, to better help developers, we still need tools to suggest both code solutions and related non-code configurations. Each suggested code solution should be complete: it not only replaces problematic API calls with correct ones, but also adjusts related code for correct program syntax and semantics. Each non-code configuration should be clear enough for developers to follow in order to properly manipulate their local file system.

Automate project-specific exploit generation. To help developers better diagnose the vulnerabilities in their programs, future tools can generate hacking code and provide the recipe for successful attacks. In this way, developers can gain first-hand experience of security attacks, view their projects' security issues from a different perspective, deepen their understanding of secure coding, and further improve their cryptographic API usage in the future.

8 Related Work

Several studies are relevant to our work [63]–[67]. For instance, Gao et al. applied CogniCrypt to different versions of Android apps; they observed that app developers are generally unaware of cryptographic API misuses and hence usually do not fix such issues [65]. Gao et al. also performed another empirical study on the evolution of Android app vulnerabilities [67]. The researchers found that (1) most vulnerabilities could survive at least three app updates; (2)

part of third-party libraries were the major contributors of most vulnerabilities; (3) all kinds of vulnerabilities were reintroduced by developers, while encryption-related ones were reintroduced most often; (4) some vulnerabilities may foreshadow malware. Our study is different from both studies, as it compares tools that automatically detect cryptographic API misuses, and investigates developers' feedback on tool outputs.

Amann et al. [64] did a systematic evaluation of static API-misuse detectors. They qualitatively compared 12 existing detectors. They also applied four of the studied tools to MUBench, to evaluate detection capabilities and analyze the root causes for low precision and recall. However, none of the studied tools focus on cryptographic API misuses.

Oyetoyan et al. [63] studied static application security testing (SAST) tools and explored developers' opinions on those tools. The researchers applied five open-source tools (i.e., SonarQube, FindSecBugs, Lapse+ [68], JLint [69], and FindBugs [70]) and a commercial tool to two program benchmarks: OWASP Benchmark and NIST Test Suite [71]. They also interviewed six developers to understand the desired features in SAST tools. They reported similar findings to ours, including (1) one tool is not enough to cover all weakness categories and (2) the capability of current tools is generally low. Our research is different in two aspects. First, we focused on cryptographic API misuses; Oyetoyan et al. focused on 13 weakness categories (e.g., code quality), many of which are irrelevant to security. Second, our user study with developers is more rigorous and representative. We interacted with more developers (47 vs. 6) on concrete API misuses and general repair suggestions.

Tupsamudre et al. [66] surveyed four SAST tools (FindSecBugs, SonarQube, CryptoGuard, and CogniCrypt) to explore (1) how tools detect password storage vulnerabilities, and (2) whether the tool-generated fixes comply with the guidelines by OWASP or NIST. The researchers found that none of the tools covered all vulnerabilities related to password storage, and tools' suggestions are either imprecise or inconsistent with the latest guidelines. They also did a study with eight developers, asking each developer to replace insecure SHA-1 based password storage implementation with the PBKDF2 solution suggested by tools. The results show that, in the absence of examples, developers chose insecure values for PBKDF2 parameters (salt, iteration count, key length). Thus, although the usage of PBKDF2 matches tools' suggestions, the resulting password storage code may be insecure in practice.

Our research corroborates the findings mentioned in Tupsamudre et al.'s work but is different in two ways. First, we studied more tools, conducted more experiments, and examined more API-misuse patterns; thus, our work has a wider and deeper scope. Second, there are more participants in our user study (47 vs. 8); they provided feedback on not only tool outputs but also future directions. Thus, we revealed more challenges and research opportunities.

9 Conclusion

With the existence of tools that detect cryptographic API misuses in Java programs, some people believed that the research problem is well solved. Our work intended to assess current tools in different aspects and to reveal the gaps between existing work and developers' needs. Namely, we explored the question: *Are existing tools good enough to help developers eliminate cryptographic API misuses?*

Our quantitative and qualitative analysis of existing tools revealed several interesting findings. First, there is no tool consistently outperforming other tools. Currently, the most advanced tools detect API misuses using interprocedural program analysis. However, developers still need better detectors, which conduct more accurate interprocedure analysis and perform context-aware analysis to report API misuses in security-focused implementation. Second, although some tools provide general guidance on misuse repairs, they are insufficient to help developers correctly remove misuses. More detailed and customized repairing suggestions are still desperately needed. Third, although some tools explain reported API misuses by citing vulnerabilities described on CWE, such citations are sometimes unconvincing to developers. It will be better if future tools can automatically synthesize program-specific attacks and detail the procedure of security exploits.

Our study shows that the problem of cryptographic API misuse detection is far from being well solved. In the future, we will build tools to suggest better repairs and to synthesize exploits.

References

[1] "Java cryptography architecture," https://docs.oracle.com/javase/9/security/java-cryptography-architecture-jca-reference-guide.htm.

[2] "Java Secure Socket Extension (JSSE) Reference Guide," https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm, 2020.

[3] M. Green and M. Smith, "Developers are not the enemy!: The need for usable security apis," *IEEE Security & Privacy*, vol. 14, no. 5, pp. 40–46, 2016.

[4] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: Why do Java developers struggle with cryptography APIs?" in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE. New York, NY, USA: ACM, 2016, pp. 935–946. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884790

[5] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 372–383.

[6] "Developers lack skills needed for secure DevOps, survey shows," https://www.computerweekly.com/news/450424614/Developers-lack-skills-needed-for-secure-DevOps-survey-shows, 2017.

[7] " Too few cybersecurity professionals is a gigantic problem for 2019," https://techcrunch.com/2019/01/27/too-few-cybersecurity-professionals-is-a-gigantic-problem-for-2019/, 2019.

[8] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags, "How reliable is the crowdsourced knowledge of security implementation?" in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, May 2019, pp. 536–547.

[9] "StackOverflow," https://stackoverflow.com, 2020.

[10] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl, "Stack overflow considered harmful? the impact of copy&paste on android application security," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 121–136.

[11] 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 *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 2455–2472.

[12] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why eve and mallory love android: An analysis of android ssl (in) security," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 50–61.

[13] 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 *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 38–49.

[14] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 447–456. [Online]. Available: https://doi.org/10.1145/1858996.1859089

[15] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 73–84.

[16] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018. [Online]. Available: http://arxiv.org/abs/1801.01681

[17] F. Fischer, H. Xiao, C.-Y. Kao, Y. Stachelscheid, B. Johnson, D. Razar, P. Fawkesley, N. Buckley, K. Böttinger, P. Muntean, and J. Grossklags, "Stack overflow considered helpful! deep learning security nudges towards stronger cryptography," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 339–356. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/fischer

[18] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, "Cognicrypt: supporting developers in using cryptography," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 931–936.

[19] L. Singleton, R. Zhao, M. Song, and H. Siy, "Cryptotutor: Teaching secure coding practices through misuse pattern detection," in *Proceedings of the 21st Annual Conference on Information Technology Education*, 2020, pp. 403–408.

[20] "Find security bugs," 2021. [Online]. Available: https://find-sec-bugs.github.io/

[21] "SonarQube," https://github.com/SonarSource/sonarqube, 2021.

[22] "Xanitizer by rigs it - because security matters," 2021. [Online]. Available: https://www.rigs-it.com/xanitizer/

[23] "Cryptoguardoss/cryptoapi-bench," https://github.com/CryptoGuardOSS/cryptoapi-bench, 2020.

[24] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A Benchmark for API-Misuse Detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR 2016, 2016. [Online]. Available: http://dx.doi.org/10.1145/2901739.2903506

[25] "OWASP Benchmark," https://owasp.org/www-project-benchmark/, 2021.

[26] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 519–534.

[27] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "Crysl: An extensible approach to validating the correct usage of cryptographic apis," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[28] "CWE - Common Weakness Enumeration," https://cwe.mitre.org/index.html, 2021.

[29] "CWE-327: Use of a Broken or Risky Cryptographic Algorithm," https://cwe.mitre.org/data/definitions/327.html, 2021.

[30] G. Singh and Supriya, "A study of encryption algorithms (rsa, des, 3des and aes) for information security," *International Journal of Computer Applications*, vol. 67, pp. 33–38, 2013.

[31] "CWE-295: Improper Certificate Validation," https://cwe.mitre.org/data/definitions/295.html, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/295.html

[32] "CWE-330: Use of Insufficiently Random Values," https://cwe.mitre.org/data/definitions/330.html, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/330.html

[33] "CWE-326: Inadequate Encryption Strength," https://cwe.mitre.org/data/definitions/326.html, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/326.html

[34] "CWE-798: Use of Hard-coded Credentials," https://cwe.mitre.org/data/definitions/798.html, 2021. [Online]. Available: https://cwe.mitre.org/data/definitions/798.html

[35] R. Oppliger, *SSL and TLS: Theory and Practice*. USA: Artech House, Inc., 2009.

[36] "CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade')," https://cwe.mitre.org/data/definitions/757.html, 2021.

[37] "Weak SSL/TLS protocols should not be used," https://rules.sonarsource.com/java/tag/cwe/RSPEC-4423, 2021.

[38] I. Muslukhov, Y. Boshmaf, and K. Beznosov, "Source attribution of cryptographic api misuse in android applications," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 133–146.

[39] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 711–722.

[40] S. Shao, G. Dong, T. Guo, T. Yang, and C. Shi, "Modelling analysis and auto-detection of cryptographic misuse in android applications," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. IEEE, 2014, pp. 75–80.

[41] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 450–464, 2018.

[42] F. Wei, S. Roy, and X. Ou, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," in *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, 2014, pp. 1329–1341.

[43] K. L. Newbury, "Automated Hotfixes for Misuses of Crypto APIs," Master's thesis, University of Alberta, 2020.

[44] Z. Xu, X. Hu, Y. Tao, and S. Qin, "Analyzing cryptographic api usages for android applications using hmm and n-gram," in *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE, 2020, pp. 153–160.

[45] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," in *European Symposium on Research in Computer Security*. Springer, 2017, pp. 229–246.

[46] J. Gajrani, M. Tripathi, V. Laxmi, G. Somani, A. Zemmari, and M. S. Gaur, "Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation," *Digital Threats: Research and Practice*, vol. 1, no. 2, pp. 1–25, 2020.

[47] "AndroBugs," https://www.androbugs.com, 2021.

[48] "MobSF/Mobile-Security-Framework-MobSF," https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2021.

[49] "AndroGuard," https://androguard.readthedocs.io/en/latest/\#, 2021.

[50] "Argus saf," http://pag.arguslab.org/argus-saf, 2021.

[51] "Wala ir," https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR), 2020.

[52] "Cryptoguardoss/cryptoguard," https://github.com/CryptoGuardOSS/cryptoguard, 2020.

[53] "Soot," https://github.com/soot-oss/soot, 2020.

[54] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, p. 13.

[55] Spotbugs, "spotbugs/spotbugs." [Online]. Available: https://github.com/spotbugs/spotbugs

[56] 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.

[57] "Mubench," https://github.com/stg-tud/MUBench, 2021.

[58] "Add 201 parametric crypto (JCA) misuses," https://github.com/stg-tud/MUBench/pull/427, 2019.

[59] "OWASP/Benchmark," https://github.com/OWASP/Benchmark, 2021.

[60] "Cwe-327 : Avoid weak encryption providing not sufficient key size (jee) — cast appmarq," https://www.appmarq.com/public/tqi,1039028, CWE-327-Avoid-weak-encryption-providing-not-sufficient-key-size-JEE, 2021, (Accessed on 03/22/2021).

[61] "MD5Function.java," https://github.com/apache/phoenix/blob/7987a74e6cea1103a028e128f98e2fb3c2252b82/phoenix-core/src/main/java/org/apache/phoenix/expression/function/MD5Function.java, 2017.
[62] "Gossiper.java," https://github.com/apache/cassandra/blob/79e693e16e2152097c5b27d2d7aaa1763e34f594/src/java/org/apache/cassandra/gms/Gossiper.java, 2020.
[63] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. Soares Cruzes, "Myths and facts about static application security testing tools: An action research at telenor digital," in *Agile Processes in Software Engineering and Extreme Programming*, J. Garbajosa, X. Wang, and A. Aguiar, Eds. Cham: Springer International Publishing, 2018, pp. 86–103.
[64] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2019.
[65] J. Gao, P. Kong, L. Li, T. F. Bissyande, and J. Klein, "Negative results on mining crypto-api usage rules in android apps," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 388–398.
[66] H. Tupsamudre, M. Sahu, K. Vidhani, and S. Lodha, "Fixing the fixes: Assessing the solutions of sast tools for securing password storage," in *Financial Cryptography and Data Security*, M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, Eds. Cham: Springer International Publishing, 2020, pp. 192–206.
[67] J. Gao, L. Li, P. Kong, T. F. Bissyande, and J. Klein, "Understanding the evolution of android app vulnerabilities," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 212–230, 2021.
[68] P. M. Pérez, J. Filipiak, and J. M. Sierra, "Lapse+ static analysis security software: Vulnerabilities detection in java ee applications," in *Future Information Technology*, J. J. Park, L. T. Yang, and C. Lee, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 148–156.
[69] "Jlint," http://artho.com/jlint/, 2021.
[70] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004. [Online]. Available: https://doi.org/10.1145/1052883.1052895
[71] "Nist samate: static analysis tool exposition (sate) iv," https://samate.nist.gov/SATE.html, 2012.