Industrial Experience of Finding Cryptographic Vulnerabilities in Large-scale Codebases

Ya Xiao[1], Yang Zhao[2], Nicholas Allen[2], Nathan Keynes[2], Danfeng (Daphne) Yao[1], Cristina Cifuentes[2]
[1]Computer Science, Virginia Tech, Blacksburg, VA
[2]Oracle Labs, Brisbane, Australia
{yax99, danfeng}@vt.edu, {yang.yz.zhao, nicholas.allen, cristina.cifuentes}@oracle.com

***Abstract*—Enterprise environments need to screen large-scale (millions of lines of code) codebases for vulnerability detection, resulting in high requirements for precision and scalability of a static analysis tool. At Oracle, Parfait [1] is one such bug checker, providing precision and scalability of results, including inter-procedural analyses. CryptoGuard [2] is a precise static analyzer for detecting cryptographic vulnerabilities in Java™[1] code built on Soot. In this paper, we describe how to integrate CryptoGuard into Parfait, with changing intermediate representation and relying on a demand-driven IFDS framework in Parfait, resulting in a precise and scalable tool for cryptographic vulnerabilities detection. We evaluate our tool on several large real-world applications and a comprehensive Java cryptographic vulnerability benchmark, CryptoAPI-Bench [3]. Initial results show that the new cryptographic vulnerability detection in Parfait can detect real-world cryptographic vulnerabilities in large-scale codebases with few false positives and low runtime.**

***Index Terms*—Cryptography, misuse, API, static analysis**

I. Introduction

Precise and scalable bug checking tools are important for companies to guarantee the security of large scale projects. Parfait [1] is a *scalable* static code analysis tool designed for large-scale codebases to find security and quality defects written in C/C++, Java, PL/SQL, and SQL languages. In particular, Parfait focuses on defects from the lists of CWE Top 25 [4] and OWASP Top 10 [5]. CryptoGuard [2] is a *precise* static analyzer detecting Java cryptographic API misuses. It contains an inter-procedural flow-, context- and field-sensitive data-flow analysis for screening cryptography code and significantly reduces the false alarms by a set of refined slicing algorithms. Our work aims to achieve the *precise* and *scalable* detection for Java cryptographic API misuses based on the scalable framework of Parfait and the refinement insights from CryptoGuard.

Many Java cryptographic APIs have been considered error-prone [6]–[12]. APIs in Java Cryptography Architecture (JCA) and Java Cryptography Extension (JCE) libraries are too complicated for developers without cryptography expertise to configure securely [13], [14]. The confusing official documents and the misleading insecure code examples in popular coding forums (e.g. StackOverflow) make the practices even worse [9], [15]. The results of misusing these APIs are serious [7], [16], [17], causing various vulnerabilities from exposing the secret keys to using vulnerable ciphers. A survey shows that the vulnerabilities in the "cryptography issues" category of the Common Vulnerabilities and Exposures (CVE) database have been dominated (83%) by the Cryptography API misuses [18].

The detection of cryptographic API misuses can be mapped to a set of program analysis problems [19]. Most of these vulnerabilities involve using constants or predictable sources to generate secrets and random numbers. Starting from prescribed sensitive arguments (e.g. cipher algorithms, keys) of cryptographic APIs, we conduct backward data-flow analysis to capture their constant sources. However, there are some major challenges. First, the number of false positives from static analyzers could be extremely high [20], [21]. Many constants (e.g. resource identifiers) used in the generation of the cryptographic materials are irrelevant to their security properties, which aggravates the false positive issue [3]. Second, the scalability of screening large-scale projects is always a challenge for static analyzers [22].

Based on the precise detection in CryptoGuard [2], we aim to implement a precise and scalable cryptographic vulnerability detection in Parfait. However, due to the different implementations, we cannot directly incorporate CryptoGuard in Parfait. First, Parfait is supported by LLVM while CryptoGuard is based on Soot [23]. CryptoGuard defines five refinements according to Jimple IR of Soot. Moreover, Parfait has a layered framework to optimize the analysis ensemble. We need to make the analyses for cryptographic vulnerability detection compatible with the Parfait framework. Finally, the data-flow analysis in Parfait follows the inter-procedural, finite, distributive subset (IFDS) framework [24]. The IFDS framework is a precise inter-procedural data-flow solution in polynomial time by transforming the analysis into a graph-reachability problem. It outperforms the ordinary flow-set based data-flow analysis in CryptoGuard in terms of time complexity. We need to achieve cryptographic vulnerability detection in the context of a set of IFDS algorithms.

Our contributions can be summarized as follows:

- We incorporated the cryptographic vulnerability detection into Oracle tool Parfait. Specifically, we implemented the backward inter-procedural, flow-, context-, field-sensitivity analysis under Parfait/LLVM support. Our analysis includes a set of refined IFDS-based analysis algorithms under Parfait's layered framework. These re-

The work was performed while the first author was at Oracle Labs as an intern.

[1]Java is a registered trademark of Oracle and/or its affiliates.

fined algorithms have the same effects as CryptoGuard [2] to reduce false positives and achieve high precision.

- We applied the cryptographic vulnerability detection on large-scale applications and found many potential vulnerabilities with relatively high precision (93.44%). The runtime for codebases that include 2.4K to 1321K lines of code ranges from 2 seconds to 36 minutes. Most of the projects can be finished within 10 minutes. We further demonstrated some noteworthy examples to better understand the practices.
- We evaluated our Parfait cryptographic vulnerability detection on a comprehensive CryptoAPI-Bench. It demonstrates that our detection achieves a high precision (86.62%) and recall (98.40%) overall. The precision excluding the path-sensitivity test cases even reaches 100%. We further analyzed the design choices and their impact on precision and recall by several test cases.

II. Background

Our detection tool aims Java cryptographic API misuses. We introduce the covered vulnerabilities as well as the background about CryptoGuard and Parfait in this section.

A. Java Cryptographic API misuses

We summarize the Java cryptographic API misuses covered by our detection from the developers' perspective, showing the involved error-prone APIs and their common vulnerable usages in Table I. The involved Java classes include:

`SecureRandom` *Class.* Any nonce used in cryptography operations should be generated from `SecureRandom` instead of `Random`. Furthermore, setting a static or predictable seed via the constructors or `setSeed` methods[2] is vulnerable.

`MessageDigest` *Class.* Passing a broken hash algorithm (e.g. MD5) to the `getInstance` API is vulnerable.

`Cipher` *Class.* The `getInstance` API is error-prone of using broken ciphers or insecure mode. The specific vulnerable usages include 1) passing a weak cipher algorithm (e.g. `"DES"`); 2) specifying `"ECB"` mode for a block cipher (e.g. `"AES/ECB/NoPadding"`); 3) a block cipher without explicitly specifying a mode (e.g. `"AES"`) because the vulnerable ECB mode is used by default.

`KeyStore` and *Key Specification Classes.* The APIs in `KeyStore` and the various key specification classes (e.g. `SecretKeySpec`, `PBEKeySpec`) accept secrets (e.g. password, key materials) through their arguments. Any method call accepting a hard-coded or predictable secret is vulnerable.

Algorithm Parameters Classes. `IvParameterSpec` and `PBEParameterSpec` classes manage the initial vector (IV), salt, and PBE iteration count. IVs and salts that are static or predictable cause vulnerabilities. Besides, the iteration count is required to be not fewer than 1000.

`javax.net.ssl` *Classes.* The methods of Java classes `TrustManager`, `HostnameVerifier`, and `SSLSocketFactory` in `javax.net.ssl` package provide the SSL/TLS services. Vulnerabilities usually happen when developers override the default methods or skip necessary steps to bypass proper verifications.

B. CryptoGuard

CryptoGuard [2] applies the backward program slicing to discover constant sources and configurations causing Java cryptographic API misuses. It develops a set of refined slicing algorithms to achieve high precision.

False Positive Reduction. CryptoGuard invents five refinement insights to remove the language-specific irrelevant elements that cause false positives. During analysis, the state indicators (e.g. `getBytes("UTF-8")`), resource identifiers (e.g. keys of a map), bookkeeping indices (e.g. size parameters of an array), contextually incompatible constants, and constants in infeasible paths are removed by refinements conditioned on their Jimple representations.

Runtime Improvement. The most costly parts of the inter-procedural analysis are usually the iterative orthogonal explorations. CryptoGuard improves the runtime by limiting the orthogonal explorations to depth 1. Deeper orthogonal method calls are handled by the refinement insights.

C. Data-flow Analysis in CryptoGuard and Parfait.

Parfait implements many functionalities supporting program analysis. An important feature of Parfiat that has not appeared in CryptoGuard [2] is the IFDS analysis framework[3].

Data-flow Analysis in CryptoGuard. CryptoGuard achieves data-flow analysis based on Soot's `FlowAnalysis` library. `FlowAnalysis` implements the intra-procedural data-flow analysis that keeps a flow set and updates it along the data-flow traces as shown in Fig. 1(b). CryptoGuard iteratively runs its intra-procedural analysis for callee and caller methods on the call graph. This design might cause re-exploring callee methods multiple times. To reduce complexity, its implementation sets the default depth of the clipping callee method exploration to 1.

IFDS in Parfait. Parfait implements the data-flow analysis as well as the IFDS framework. As shown in Fig. 1(c), the IFDS framework handles the analysis by building edges among the data facts (i.e. variables) and summarizing the edges between two program points on the super control-flow graph. It can avoid unnecessary re-analysis as much as possible.

Parfait Framework. To improve scalability, Parfait offers a layered framework to optimize the ensemble of static program analyses. According to the time cost, the analyses are scheduled from the quickest to the slowest. In this way, more bugs can be found with a lower time overhead. Specifically, in the cryptographic vulnerability detection, we break down and dynamically schedule the analyses into different layers according to the depth of callers. We introduce more details in Section III-B.

[2]This API has two different method signatures (setSeed(long seed) and setSeed(byte[] seed)), we skip them for simplicity.

[3]The project Hero [25] implements the IFDS framework on top of Soot, however, the CryptoGuard only uses the FlowAnalysis library in Soot, which does not provide IFDS.

TABLE I

ERROR-PRONE JAVA CRYPTOGRAPHIC APIS COVERED BY PARFAIT'S CRYPTOGRAPHIC API MISUSES DETECTION AND THE INVOLVED VULNERABILITIES IN CWE. THE SEVERITY INFORMATION IS FROM CRYPTOGUARD [2].

Class	Method Names	Vulnerable Usage	Severity	CWE
Random	constructor	used in cryptography operations	M	338: Use of Cryptographically Weak PRNG
SecureRandom	constructor setSeed	pass static or predictable seed	M	337: Predictable Seed in PRNG
MessageDigest	getInstance	pass weak algorithm	H	328: Reversible One-Way Hash
Cipher	getInstance	pass weak algorithm pass ECB mode for block ciphers	L	327: Use of a Broken or Risky Cryptographic Algorithm
KeyStore	load store setKeyEntry getKey	pass hard-coded password	H	259: Use of Hard-coded Password
SecretKeySpec	constructor	pass hard-coded key materials	H	321: Use of Hard-coded Cryptographic Key
PBEKeySpec	constructor	pass hard-coded password pass static or predictable salt pass iteration <1000	H M L	259: Use of Hard-coded Password 760: Use of a One-Way Hash with a Predictable Salt 916: Use of Password Hash With Insufficient Computational Effort
PBEParameterSpec	constructor	pass static or predictable salt pass iteration <1000	M M	760: Use of a One-Way Hash with a Predictable Salt 916: Use of Password Hash With Insufficient Computational Effort
IvParameterSpec	constructor	pass static or predictable IV	M	329: Not Using a Random IV with CBC Mode
TrustManager	checkClientTrusted checkServerTrusted getAcceptedIssuers	override to skip validation override to skip validation override to return null	H	303: Incorrect Implementation of Authentication Algorithm
HostnameVerifier	verify	override to always return True	H	303: Incorrect Implementation of Authentication Algorithm
SSLSocketFactory	createSocket	miss hostname verification	H	304: Missing Critical Step in Authentication

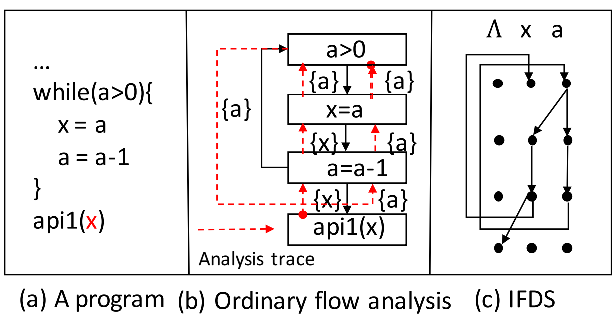

Fig. 1. The comparison of the ordinary iterative analysis and IFDS analysis. (a) is a program demo. (b) shows the ordinary flow-set based analysis by collecting and updating a flow set. (c) shows the IFDS based analysis process which builds edges and then summarizes edges.

III. DETECTION METHODS AND IMPLEMENTATION

Our detection covers all the misuses shown in Table I. Two scalability enablers of it are the layered framework of Parfait and the summarization mechanism in IFDS to handle callee methods.

A. Detection Methods

The detecting logic is similar to CryptoGuard which maps the cryptographic API misuses to the data-flow analysis problems. In terms of the specific detection methods, there are three groups.

Group 1: Inter-procedural Backward Data-flow Analysis. This group includes the API misuses determined by constant sources. Specifically, these are APIs in Table I of Java Class `SecureRandom`, `MessageDigest`, `Cipher`, `KeyStore`, `SecretKeySpec`, `PBEKeySpec`, `PBEParameterSpec`, and `IvParameterSpec`. We require an inter-procedural backward data-flow analysis to capture the constant sources of the API arguments. We apply different detection rules to the collected constant sources according to the vulnerability types. The detection rules include whether it is a constant, whether it is a number less than 1000, or whether it matches to some weak algorithms.

Group 2: Intra-procedural Pattern Matching. The vulnerabilities related to `TrustManager`, `HostnameVerifier`, and `SSLSocketFactory` in Table I belong to this group. These vulnerabilities often happen within one method that is responsible for authentication operations. We find them by the intra-procedural pattern matching. Specifically, for `HostnameVerifier`, we detect whether the return value of the method `verify` is always "True" regardless of the verification. For `TrustManager`, we detect three vulnerable patterns in the `checkClientTrusted` and `checkServerTrusted` methods including 1) missing verification behavior; 2) catching the verification exception without throwing it; 3) missing verification under a certain path. For `SSLSocketFactory`, we perform the intra-procedural pattern matching to check whether the `HostNameVerifier.verify` method is called after the `SSLSocketFactory` instance creation.

Group 3: Sanitizer VS. Verifier. In cryptography operations, `Random` is not strong enough [26]. However, it is unreasonable to report every `Random` used in a program as a vulnerability. Therefore, we regard `Random` as a verifier and `SecureRandom` as a sanitizer for the traced arguments in group 1. Accordingly, we only report `Random` in these cryptographic usages.

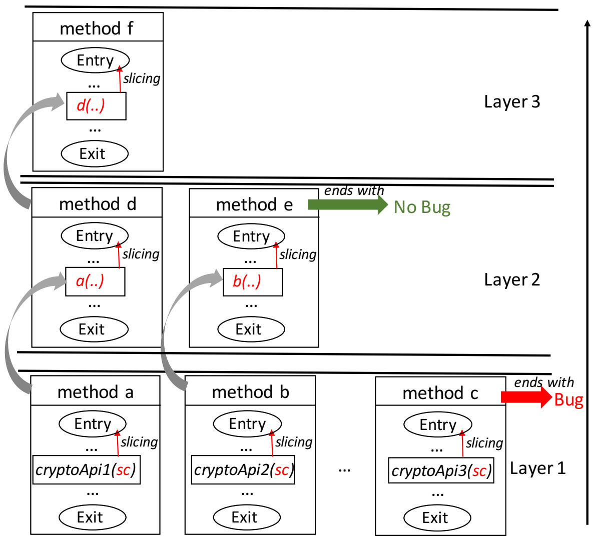

Fig. 2. The inter-procedural analysis under parfait's layered framework

B. *Cryptographic Vulnerability Detection Implementation*

Supported by Parfait, we implement the inter-procedural flow-, context-, and field-sensitive backward data-flow analysis for cryptographic vulnerabilities detection.

Layered Scheduler for Caller Methods. Parfait optimizes the analysis ensemble to improve scalability. Figure 2 demonstrates the backward analyses that are broken down and assigned to different layers. The analyses are scheduled layer by layer. At each layer, the backward analysis ends up at the entry point of the current method with three situations. First, a real bug is verified. Second, the potential bug is sanitized as no bug. Third, further analyses are required in its caller methods. Further analyses will be scheduled at the next layer. In this way, the analysis requiring less time can be performed first. It also avoids the duplicated parts of two potential vulnerabilities detection traces. This layered framework effectively improves the efficiency of finding bugs.

Flow Functions in IFDS. There are several flow functions used to define the analysis. In our cryptographic vulnerability detection, they are:

- `flow`: This function specifies the data-flow edges through ordinary non-call instructions. Specifically, it applies to the LLVM instructions `ReturnInst`, `LoadInst`, `StoreInst`, and `BitCastInst`.
- `phiFlow`: This function specifies the data-flow edges through the LLVM `phi` instruction.
- `returnVal`: The function specifies the data-flow edges between the `ReturnInst` of the callee method and its callsite. The summary edges of the callee method are queried at this point to handle the callee method.
- `passArgs`: The function specifies the data-flow edges between the arguments of the callee method and the parameters passed in its callsite.
- `callFlow`: The function handles the data-flow edges regardless of the callee method. Most of the refinements happen here to handle the callee method whose implementation is unavailable.

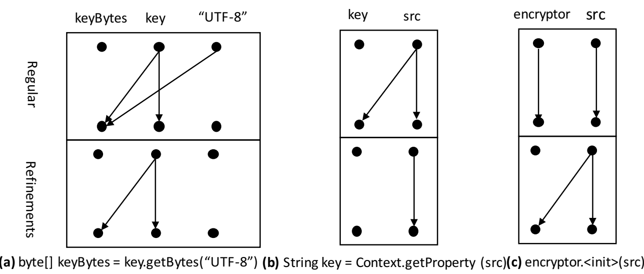

Fig. 3. Refinements represented in IFDS. The callFlow propagating edges for three situations. The above one is the default propagating edges of callFlow. The bottom one is the refined propagating edges.

The major differences of these flow functions between the analysis for cryptographic vulnerabilities and taint analysis are the data-flow edges from constants. The cryptographic vulnerability detection covers the edges flowing out from constants and refines them according to five refinement insights, which does not happen in the taint analysis. Furthermore, cryptography vulnerability detection redefines the default data-flow edges in `callFlow`. More details are in Section III-C.

Summarization for Callee Methods. Another design improving the scalability is the summarization mechanism for the callee methods. After a method is explored, the summary edges for it are stored for future usage. Parfait exhaustively summarizes all methods in advance and queries the summary edges of the callee methods on demand. All the methods are summarized in a bottom-up manner according to the call graph, beginning from leaf methods to their callers. This design guarantees every method is only explored once. Hence, the re-exploration for callee methods is eliminated to avoid complexity explosion.

C. *Refinement Insights Implementation*

By applying the backward IFDS analysis, all the constant sources that reach the sensitive arguments are captured. However, it is quite difficult to distinguish between truly insecure sources and pseudo-influences, which leads to an extremely high false-positive rate. CryptoGuard [2] summarizes five types of pseudo-influences including state indicators, resource identifiers, bookkeeping indices, contextually incompatible constants, and constants in infeasible paths based on observation. The refinement insights are used to eliminate the pseudo-influences. We apply these refinement insights on the context of IFDS algorithms and LLVM IR instructions.

CryptoGuard described the refinement insights with Jimple IR. To remove state indicators, it eliminates tracing for callee arguments on condition that they appear in Jimple assign statements marked with `virtualinvoke`. To remove resource identifiers, CryptoGuard eliminates tracing for the arguments appearing in Jimple assign statements marked with `interfaceinvoke` or `staticinvoke`. In the form of IFDS, they are translated into reachable edges applied to different types of LLVM instructions. Specifically, we change

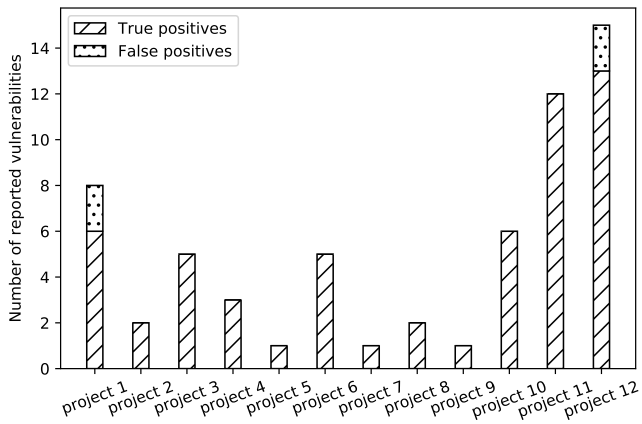

Fig. 4. Number of vulnerabilities of 12 real-world projects. Projects 2-10 achieves zero false positive. 10 of them are Oracle internal codebases. Projects 9 and 10 are open-source projects.

the default `callFlow` function for three kinds of call instructions. They are 1) assignment callsites for instance methods 2) assignment callsites for static methods 3) non-assignment callsites for class constructors. The examples are shown in Fig. 3.

IV. Real-world Findings and Accuracy Analysis

We tested the cryptographic vulnerability detection in Parfait on several large real-world codebases and a comprehensive cryptographic vulnerability benchmark (CryptoAPI-Bench [3]) to evaluate the performance.

A. *Real-world Findings*

We ran Parfait on 12 large codebases. 10 of them are Oracle internal products, 2 are open-source projects. There are 61 reported vulnerabilities and 57 of them are manually verified as true positives. The precision is 93.44%. We have reported the detected vulnerabilities to corresponding developers. In terms of the open-source projects, we further find that the vulnerabilities are either in their non-production (development) mode or fixed in their latest versions. We show several real-world detected cases below.

```
public class DesEncrypter{
    private byte[] salt = { (byte) 0xC9, (byte) 0xDB
    , (byte) 0xA3, (byte) 0x52, (byte) 0x56, (byte)
    0x35, (byte) 0xE8, (byte) 0xB0};
    private int iterationCount = 20;
    public DesEncrypter(final String passPhrase){
        initDesEncrypter(passPhrase);}
    private void initDesEncrypter(final String
    passPhrase){
    ...
    AlgorithmParameterSpec paramSpec = new
    PBEParameterSpec(salt,iterationCount);}}
```

Listing 1. A real-world vulnerability about using constant salt and insufficient iteration count (We modified the code to make the codebase unidentifiable.)

Listing 1 shows vulnerabilities of using constant salt and insufficient iteration count as PBE parameters. This case represents the most common vulnerable pattern of the sensitive cryptographic materials (e.g. password, salt, IV, etc) to be hard-coded in the initialization.

```
1 public String padding_salts(String salts){
2     StringBuffer sb = new StringBuffer();
3     for(int i=salts.getBytes().length; i<16; i++){
4         sb.append((byte)i&0xfe)}
5     String padded_salts = salts+sb.toString();
6     return padded_salts;}
```

Listing 2. A real-world vulnerability about insufficient entropy salts

Listing 2 is a noteworthy real-world example. It introduces a vulnerability of using salts with insufficient entropy. When a random salt is iteratively assigned by the same variable, its value space is reduced significantly and hence makes the exhaustive attack feasible. Our analysis reports a constant number 16 at Line 3 involved in the construction of the salts. However, to accurately capture the insufficient entropy issue, symbolic execution is required.

```
1 public SecureRandom getObject() throws Exception{
2     SecureRandom rnd = SecureRandom.getInstance(
      algorithm);
3     if(seed != null){
4         byte[] seedBytes = FileCopyUtils.
      copyToByteArray(seed.InputStream());
5         rnd.setSeed(seedBytes); //manual seeding
6     }else{
7         rnd.nextBytes(new byte[1]) //self-seeding
8     }}
```

Listing 3. An example from CVE-2019-3795

Listing 3 shows a detected vulnerability in open-source project Spring Security. It is published in CVE database [27]. This vulnerability appears in Spring Security versions 4.2.x before 4.2.12, 5.0.x before 5.0.12, and 5.1.x before 5.1.5. Although not involving a hard-coded seed, the `SecureRandom` instance relies on an unreliable `InputStream` at Line 4 as the `seed`. Inspired by this real-world vulnerability, we apply a more strict rule for `SecureRandom.setSeed` to avoid unreliable seeding. Only self-seeding and manual seeding by the method `SecureRandom.generateSeed()` are considered as secure. A self-seeding (secure) will be automatically enforced if the API `nextBytes` is called immediately after the `SecureRandom` instantiation [28].

```
public void checkClientTrusted(X509Certificate[]
    certs, String authType) throws
    CertificateException{
    throw new UnsupportedOperationException("
    checkclientTrusted is unsupported in "+ this.
    getClass().getName());}
```

Listing 4. A real-world false positive case about TrustManager

Listing 4 shows a reported case for bypassing certificate verification. Although it is a false positive case, what happened in the code is not recommended and required to be fixed. This case completely disables the certificate verification by simply throwing the `UnsupportedOperationException` for all certificates. We reported it because it matches a vulnerable pattern, that is, missing verification (e.g. the default `checkclientTrust`). This situation is caused by the development mode and will be fixed in the production mode.

TABLE II
EVALUATION RESULTS ON 158 TEST CASES OF CRYPTOAPI-BENCH. THERE ARE BASIC CASES (INTRA-PROCEDURAL), DIFFERENT INTER-PROCEDURAL CASES THAT REQUIRES ACROSS METHODS, ACROSS CLASSES, FIELD SENSITIVITY, PATH-SENSITIVITY, AND HEURISTICS TO HANDLE.

Type	Total Cases	Insecure Cases	Secure Cases	Reported Cases	False Positives	False Negatives	Precision	Recall
Basic Cases	27	24	3	24	0	0	100%	100%
Multiple methods	57	56	1	54	0	2	100%	96.43%
Multiple Classes	23	18	5	18	0	0	100%	100%
Field Sensitivity	19	18	1	18	0	0	100%	100%
Path Sensitivity	19	0	19	19	19	0	0 %	0 %
Heuristics	13	9	4	9	0	0	100%	100%
Total	**158**	**125**	**33**	**142**	**19**	**2**	**86.62%**	**98.40%**

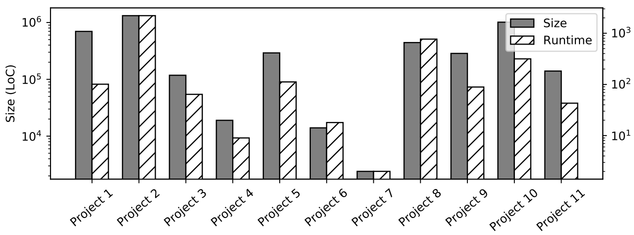

Fig. 5. Runtime of different projects

Runtime Analysis. The scalability is always one of the most important concerns. We list the runtime of several projects in Figure 5. The detection is run on the machine with Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz, 128G memory, and Oracle Linux Server release 6.9 operating system. The analysis can be finished within 10 minutes for most of these projects even including millions of lines of code (Project 10). Due to the access limitation of these internal projects, we are not able to run CryptoGuard on these projects to compare. However, our cryptographic vulnerability detection has more designs to achieve better scalability compared with CryptoGuard.

B. Accuracy Analysis on Crypto-Bench

We tested Parfait on 158 test cases for 13 cryptographic vulnerability types of CryptoAPI-Bench. CryptoAPI-Bench includes various kinds of test units from basic ones to more advanced cases. The basic test cases only require intra-procedural analysis to handle. The advanced cases are inter-procedural ones that require the analyses across multiple methods, multiple classes, achieving field sensitivity, and path sensitivity.

The breakdown numbers are shown in Table II. The overall precision and recall are 86.62% and 98.40%, respectively. All the false positive cases come from path sensitivity cases, which verifies that our tool has achieved high precision for the cases excluding path-sensitive ones. We analyze several examples to further reveal the details of Parfait cryptographic vulnerability detection and discuss possible improvements.

Application Perspective VS. Library Perspective. Parfait differs from CryptoGuard in the vulnerability definitions in some situations. An example is given in Listing 7 in the Appendix. When the potentially vulnerable method is not called therefore the concerned field variable is left undetermined, Parfait considers it as a non-vulnerable case. However, CryptoGuard applies a forward slicing for this field variable to find out the possible assignments in the initialization. If a constant is assigned in the initialization, CryptoGuard still considers it as a vulnerability. If the detected issues are in applications, Parfait's design is better to avoid overestimating the vulnerabilities. If they are in libraries, CryptoGuard's design is better to discover the potential buggy method although they may not be called yet.

Potential Improvement. There are two potential improvements to fix the false-negative cases. First, a false negative could be caused by missing the summarization for `clinit` method. An example is shown in Listing 8 in the Appendix. This deficiency is derived from the fact that `clinit` has not appeared in the Parfait's call graph. A fix for this issue could be updating the call graph construction to cover the `clinit` of every class. Second, a false-negative case shown in Listing 6 is caused by incompatible types between the captured source (i.e. `String`) and the sensitive argument (i.e. `int`). This corner case can be improved by checking the type compatibility through the type casting in Java language.

Limitations. Our cryptographic vulnerability detection still has limitations to handle path-sensitive cases and pointer issues. We show a path-sensitive false-positive case in Listing 5 in the Appendix. Furthermore, another potential cause for false positives could be the pointer issues. Due to the limitation of static analysis, there may be over-approximation in our call graph construction, which leads to potential false positives. However, path-sensitivity and pointer precision are too costly to achieve. Our analysis aims to massive-sized projects, therefore we have to balance them to come up with better overall performance.

V. CONCLUSION

We implemented a precise and scalable cryptographic vulnerability detection based on the scalable bug checker Parfait and precise cryptographic vulnerability detection tool CryptoGuard. Leveraging the refinement insights from CryptoGuard, our detection reproduced the high precision results (few or no false positives) achieved by CryptoGuard. Experiments show 93.44% precision for 12 real-world large-scale projects and 100% precision for CryptoAPI-Bench excluding the path-sensitivity cases. Benefited from the IFDS and layered framework of Parfait, the cryptographic vulnerability detection also achieves good runtime performance for large-scale codebases. The runtime for 11 large-scale codebases ranges from 2 seconds to 36 minutes. Most of the codebases can be screened within 10 minutes.

References

[1] C. Cifuentes and B. Scholz, "Parfait: Designing a Scalable Bug Checker," in *Proceedings of the 2008 workshop on Static analysis*, 2008, pp. 4–11.

[2] 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.

[3] 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.

[4] *2019 CWE Top 25 Most Dangerous Software Errors*, September 18, 2019. [Online]. Available: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

[5] *OWASP Top Ten*. [Online]. Available: https://owasp.org/www-project-top-ten/

[6] 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.

[7] 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.

[8] D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A Stitch in Time: Supporting Android Developers in Writing Secure Code," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1065–1077.

[9] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty, "Secure Coding Practices in Java: Challenges and Vulnerabilities," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 372–383.

[10] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017, Abu Dhabi, United Arab Emirates, April 2-6, 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds. ACM, 2017, pp. 71–85. [Online]. Available: https://doi.org/10.1145/3052973.3053004

[11] K. Tian, D. Yao, B. G. Ryder, G. Tan, and G. Peng, "Detection of repackaged android malware with code-heterogeneity features," *IEEE Trans. Dependable Secur. Comput.*, vol. 17, no. 1, pp. 64–77, 2020. [Online]. Available: https://doi.org/10.1109/TDSC.2017.2745575

[12] N. Patnaik, J. Hallett, and A. Rashid, "Usability smells: An analysis of developers' struggle with crypto libraries," in *Fifteenth Symposium on Usable Privacy and Security ({SOUPS} 2019)*, 2019.

[13] 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*, 2016, pp. 935–946.

[14] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the Usability of Cryptographic APIs," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 154–171.

[15] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You Get Where You're Looking for: The Impact of Information Sources on Code Security," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 289–305.

[16] 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*, 2012, pp. 38–49.

[17] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1296–1310.

[18] D. Lazar, H. Chen, X. Wang, and N. Zeldovich, "Why does Cryptographic Software Fail? A Case Study and Open Problems," in *Proceedings of 5th Asia-Pacific Workshop on Systems*, 2014, pp. 1–7.

[19] S. Rahaman and D. Yao, "Program Analysis of Cryptographic Implementations for Security," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 61–68.

[20] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 299–308.

[21] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[22] F. Gauthier, N. Keynes, N. Allen, D. Corney, and P. Krishnan, "Scalable static analysis to detect security vulnerabilities: Challenges and solutions," in *2018 IEEE Cybersecurity Development (SecDev)*. IEEE, 2018, pp. 134–134.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A Java Bytecode Pptimization Framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[24] T. Reps, S. Horwitz, and M. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[25] E. Bodden, "Inter-procedural Data-flow Analysis with IFDS/IDE and Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012, pp. 3–8.

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

[27] "NVD: CVE-2019-3795 Detail," https://nvd.nist.gov/vuln/detail/CVE-2019-3795, 2019, [online; Last Modified: 05/20/2019].

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

Appendix

```java
    String defaultKey = "defaultkey";
    int choice = 2;
    byte[] keyBytes = defaultKey.getBytes();
    //keyBytes-->key material after phiFLow
    if(choice>1){
        //nothing-->key material
        SecureRandom random = new SecureRandom();
        keyBytes = String.valueOf(random.ints()).getBytes();
    }
    keyBytes = Arrays.copyOf(keyBytes,16);
    SecretKeySpec keySpec = new SecretKeySpec(keyBytes, "AES");
```

Listing 5. A false positive caused by path sensitivity

```java
public class LessThan1000IterationPBEABICase2 {
    public static final String DEFAULT_COUNT = "20";
    private static char[] COUNT;
    private static char[] count;
    public static void main(){ //Bug condition: "20"<1000?
        LessThan1000IterationPBEABICase2 lt = new LessThan1000IterationPBEABICase2();
        go2();    //"20"-->PBE iteration
        go3();    //this.COUNT-->PBE iteration
        lt.key2(); //this.count-->PBE iteration
    }
    private static void go2(){
        COUNT = DEFAULT_COUNT.toCharArray();
    }
    private static void go3(){
        count = COUNT;
    }
    public void key2(){ //this.count-->PBE iteration
        ...
        pbeParamSpec = new PBEParameterSpec(salt, Integer.parseInt(String.valueOf(count)));
    }
}
```

Listing 6. A false negative case caused due to type matching

```java
1  public class PredictableCryptographicKeyABSCase1 {
2      Crypto crypto;
3      public PredictableCryptographicKeyABSCase1() throws Exception {
4          String passKey = PredictableCryptographicKeyABSCase1.getKey("pass.key");
5          if(passKey == null) {
6              crypto = new Crypto("defaultkey");
7          }
8          crypto = new Crypto(passKey);
9      }
10     //this.crypto.defaultKey-->secret key; no caller for encryptPass, terminate
11     public byte[] encryptPass(String pass, String src) throws Exception {
12         String keyStr = PredictableCryptographicKeyABSCase1.getKey(src);
13         return crypto.method1(pass, keyStr);
14         //keyStr-->secret key; this.crypto.defaultKey-->secret key
15     }
16     public static String getKey(String s) {
17         return System.getProperty(s);
18     }
19 }
20 class Crypto {
21     Cipher cipher;
22     String algoSpec = "AES/CBC/PKCS5Padding";
23     String algo = "AES";
24     String defaultKey;
25     public Crypto(String defkey) throws NoSuchPaddingException, NoSuchAlgorithmException {
26         cipher = Cipher.getInstance(algoSpec);
27         defaultKey = defkey;
28     }
29     //key-->secret key; this.defaultKey-->secret key
30     public byte[] method1(String txt, String key) throws UnsupportedEncodingException, InvalidKeyException, BadPaddingException, IllegalBlockSizeException {
31         if(key.isEmpty()){
32             key = defaultKey;
33         }
34         byte[] keyBytes = key.getBytes("UTF-8");
35         byte [] txtBytes = txt.getBytes();
36         keyBytes = Arrays.copyOf(keyBytes,16);
37         SecretKeySpec keySpec = new SecretKeySpec(keyBytes,algo); //A potential bug
38         cipher.init(Cipher.ENCRYPT_MODE,keySpec);
39         return cipher.doFinal(txtBytes);
40     }
41 }
```

Listing 7. A test cases considered non-vulnerable by Parfait but vulnerable by CryptoGuard. The backward analysis in Parfait terminates at Line 11 and leaves this.crypto.defaultKey as a variable due to no caller of this method.

```java
public class PredictablePBEPasswordABICase2 {
    public static String KEY = "sagar";
    public static char [] DEFAULT_ENCRYPT_KEY = KEY.toCharArray(); //"sagar"-->this.DEFAULT_ENCRYPT_KEY happens in clinit
    private static char[] encryptKey;
    ...
    public static void main(String [] args) { //this.DEFAULT_ENCRYPT_KEY-->PBE password
        ...
    }
}
```

Listing 8. A false negative case caused due to the summarization