2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)

FireBugs: Finding and Repairing Bugs with Security Patterns

Larry Singleton, Rui Zhao, Myoungkyu Song, Harvey Siy
Dept. of Computer Science, University of Nebraska, Omaha, NE 68182, USA
{larrysingleton, ruizhao, myoungkyu, hsiy}@unomaha.edu

***Abstract*—Security is often a critical problem in software systems. The consequences of the failure lead to substantial economic loss or extensive environmental damage. Developing secure software is challenging, and retrofitting existing systems to introduce security is even harder. In this paper, we propose an automated approach for Finding and Repairing Bugs based on security patterns (FireBugs), to repair defects causing security vulnerabilities. To locate and fix security bugs, we apply security patterns that are reusable solutions comprising large amounts of software design experience in many different situations. In the evaluation, we investigated 2,800 Android app repositories to apply our approach to 200 subject projects that use `javax.crypto` APIs. The vision of our automated approach is to reduce software maintenance burdens where the number of outstanding software defects exceeds available resources. Our ultimate vision is to design more security patterns that have a positive impact on software quality by disseminating correlated sets of best security design practices and knowledge.**

I. Introduction

Developing and maintaining secure software is a difficult undertaking. Many mobile applications were written with little consideration for software security issues. As mobile applications become commonly used, the potential consequences of attacks that exploit these vulnerabilities also increase in severity and impact [1]. Therefore, there is a need to repair these systems by detecting and patching the vulnerabilities. An automated approach is needed to cope with the scale of the work involved in repairing a large number of such systems.

Although existing program repair approaches [2], [3] showed promising results, they have inherent limitations; their algorithms rely on random program generation that could produce nonsensical patches, and the techniques often repair minor or rare bugs that are only of academic interest. There is a need for automatic repair approaches which enable applications to safely recover from or avoid *common security bugs or attacks*. In addition, it is desirable for such approaches to help developers understand patches correctly so they can review and confirm that the repairs have addressed defects safely.

In this paper, we introduce an automated approach, **Fi**nding and **Re**pairing **Bugs** based on security patterns (FireBugs). Our approach leverages security patterns [4] to generate a repair patch automatically which is dynamically adapted into detected vulnerabilities at runtime. Developers typically address each new vulnerabilities relying on the penetrate-and-patch approach, i.e., fixing the problem after an attack has occurred. However, it is hard to anticipate the seemingly steady stream of newly discovered security vulnerabilities [5]. To cope with this shortcoming, FireBugs allows developers to systematically specify declarative expressions in a repair template for either old and new vulnerabilities. It then searches for vulnerable program locations based on the template, and automatically generate repair patches for detected security bugs. To provide correctness, safety, and security guarantees, relevant regression tests are selectively executed achieving significant savings in time [6]. To fix a deployed buggy program, a patch is weaved dynamically at runtime by avoiding recompiling or restarting the application [7].

In the evaluation, we investigated 2,800 Android app repositories where 200 apps use `javax.crypto` APIs to implement their security features. Given the subject projects, our experiment measures the repair capability of FireBugs, how accurately FireBugs can detect security bugs and how correctly it generates repair patches to pass all tests and to not fail those encoding bugs.

The main contributions of this research are:

- Security patterns representing required behaviors for security functionalities, while encoding common repair patterns to fix vulnerable anomalies susceptible to security bugs.
- FireBugs, an automated approach that leverages security patterns to detect security vulnerabilities and generates a repair patch that is dynamically weaved in a program at runtime.
- Experimental analysis for the evaluation of FireBugs's repair capability by inspecting 2,800 repositories to apply FireBugs to 200 Android apps that use `javax.crypto` APIs for security implementations.

II. Related Work

A. Encryption-related Security Problems

Of the many types of vulnerabilities commonly encountered, encryption-related problems appear to be prevalent. Lazar et al. [8] analyzed a sample of application bugs from the nationally curated Common Vulnerabilities and Exposures (CVE) database and found that 83% are caused by the misuse of cryptographic libraries. Nadi et al. [9] found that it is hard for developers, who

978-1-7281-3395-9/19/$31.00 ©2019 IEEE
DOI 10.1109/MOBILESoft.2019.00014

Misuse Pattern	[10]	[11]	[12]
ECB Mode	√	√	√
Risky/broken Symmetric Encryption Algorithm			√
RSA algorithm without OAEP			√
Reversible One-way Hash			√
Non-random IV for CBC Encryption	√	√	√
Constant Encryption Keys	√	√	
Static Seeds for SecureRandom		√	
Insufficient Key Length			√
Reusing Same Cryptographic Key			√
Constant Salts for PBE		√	
Fewer than 1000 Iterations for PBE		√	

TABLE I: Encryption misuse patterns detected by existing tools.

lack of domain knowledge in cryptography, to determine the correct way to use Java cryptographic APIs, which are too complex to use.

B. Software Vulnerability Patterns

Several tools have been shown to be effective at detecting cryptographic vulnerabilities by finding common patterns in the code and its execution, see Table I. For example, iCryptoTracer [10] uses static analysis techniques to identify crucial cryptographic APIs and analyzes the runtime logs to trace the use of cryptographic APIs in three patterns in iOS applications. Of 98 applications diagnosed by iCryptoTracer, 64 contain security flaws caused by cryptographic misuse. CryptoLint [11] is a lightweight static analysis tool for identifying cryptographic misuse in Android apps based on seven misuse patterns. In 11,748 Android applications, CryptoLint found 88% had used cryptography inappropriately. Crypto Misuse Analyzer [12] combines static and dynamic analysis techniques to monitor the cryptographic APIs invoked in Android applications and then determine if there is any cryptographic misuse based on 13 misuse patterns. In 45 applications, the tool found 40 applications had misused cryptographic APIs. In our paper, we investigated 2,800 repositories to analyze 200 Android apps as to whether they had misused `javax.crypto` APIs for their security implementations.

C. Automatic Detection and Program Repair

The closest existing work is CDRep [13], a static analysis tool that searches and fixes cryptographic misuse in Android apps with seven rules from Egele et. al's work [11], most of which are related to the use of constant values. CDRep repaired vulnerabilities by manually created templates. Different from CDRep, FireBugs processes the values weakly generated by problematic pseudorandom number generators besides the constant ones. In contrast to CDRep that only focuses on bytecode, our approach allows developers to interactively inspect the code changes for understanding repaired locations and debugging the root cause of vulnerabilities [14], [15]. FireBugs is designed to provide the code inspection support during maintenance activities for code reviews by extracting edits [16] of bug-fixing changes, analyzing the impact and risk of the changes.

Arzt et al. [17] proposed an automatic program generation tool, OpenCCE to guide adequate uses of cryptographic APIs. OpenCCE generates code for securely using cryptographic APIs in a program, and it also validates code that uses cryptography based on user's specifications by answering high-level questions. In contrast, FireBugs focuses on isolating security bugs and generating *safe* repair patches by regression testing.

III. FireBugs: Finding and Repairing Security Bugs

In this project, we create a software tool, FireBugs, such that, given a buggy program, it automatically locates vulnerabilities and generates a repair patch that passes a test suite encoding the required behavior and does not fail those encoding vulnerabilities. We leverage the notion that most common vulnerabilities have recurring patterns and thus the patch can be generated by utilizing a template-based approach. For applying the generated patch to vulnerable locations at runtime, we leverage a dynamic adaptation technique by using Aspect-Oriented Programming [18].

Figure 1 shows the workflow of our approach where (1) we collected 2,800 Android app repositories and identified seven common security patterns for two categories of cryptographic misuse, (2) we are currently working on phases I and II for static analysis for bug detection and code transformation tool for patch generation, and (3) we have established a thorough evaluation plan for our experimental analysis along side engineering a runtime adaptation environment on phase III.

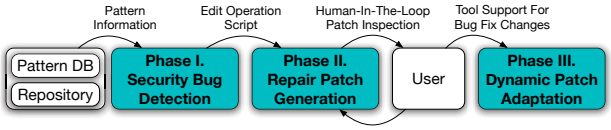

Fig. 1: Overview of FireBugs's workflow.

A. Security Patterns

There are three requirements for secure use of cryptography on data protection.

- The algorithm must be strong and resistant to cryptanalytic attacks.
- The key required by the encryption algorithms, including both symmetric and public-key encryption schemes and some message authentication functions, must be distributed to the encrypter and decrypter in a secure fashion and must be kept secure.
- The key and the nonce, e.g., the salt and the initialization vector (IV), in the use of the cryptography must generated by the true-random number generator (TRNG) or the pseudorandom number generator (PRNG) with a secure seed.

Based on these requirements, FIREBUGS focuses on the fix of two categories of cryptographic misuse, weak cryptographic algorithms and weak parameter values used in them. The first category contains four types of misuses on the selection of weak encryption algorithms, weak hash function, weak random number generator, and weak password-based encryption algorithms as shown in TableII. In this project, we only consider the standard cipher algorithms. Within all standard encryption algorithms provided by the `javax.crypto` library, we categorize the AES CTR mode as weak because all NIST-approved block cipher modes of operations involve feedback [19], and the Blowfish algorithm as weak because it was proved to be vulnerable to birthday attack [20]. Although RC4 has been criticized for its security, in essence, the problem is not with RC4 itself but the way in which the keys are generated for using RC4 [19]. In terms of algorithms for the password-based encryption (PBE), the password-based key derivation function 1 (PBKDF1) algorithm was no longer recommended [21].

The second category contains three types of misuses.

- The selection/generation of parameter values for encryption algorithms, including short-length, constant, predictable, or reused values for the keys, IVs, salts, and seeds. The secure manner should be using at least 128-bit, pseudo-randomly generated or securely derived values for encryption parameters.
- The improper post-processing of the encryption key, including writing or sending key values to files or the network.
- The use of a small number for the iteration count in PBE algorithms. A minimum of 1000 iterations is recommended [21].

In Figure2, we show a cryptographic misuse example which contains both weak encryption algorithms and the weak key value. An encryption key *key* is generated from a secret string *secret* at line 4. An AES-ECB encryption algorithm is initialized at line 6. The *plaintext* is encrypted by AES-ECB with the *key* at lines 8 and 9. A SHA-1 hash function is used to obtain the hash value from the *plaintext* at lines 11–13. Based on our specification of cryptographic misuse shown above, this code example uses (1) the weak ECB mode of the AES encryption algorithm at line 6, (2) the weak SHA-1 hash function at line 11, and (3) a constant value for the key at line 1.

B. Security Bug Detection and Repair Patch Generation

Static Analysis. FIREBUGS utilizes program slicing and data flow analysis techniques to only identify semantically dependent statements for finding vulnerabilities in each program. For example, Figure 3 shows how FIREBUGS tracks dependent statements, where s_n denotes a sequence of statements on control flow graph nodes; a dotted-line denotes a backward analysis direction; a solid line denotes a forward analysis direction; an underlined statement denotes a location identified by an aliases analysis, and; a dotted-circle denotes a correspondence of the propagated variable. Given program dependence graphs generated from program slicing, FIREBUGS

	Weak	Strong
Encryption Algorithm and its Mode	DES AES AES-ECB AES-CTR Blowfish RC2	3-DES AES-CBC AES-CFB AES-OFB AES-CCM AES-GCM RSA Elliptic Curve RC4, RC5
Hash Function	MD2, MD5 SHA-1	HMAC SHA-2 SHA-3
Random Number Generator		TRNG PRNG
Password-Based Encryption	PBKDF1	PBKDF2

TABLE II: Selection of Cryptographic Algorithms

```
1  String secret = "Secret";
2  String plaintext = "PlainMessage";
3  // Generate a key from the secret
4  SecretKeySpec key = new SecretKeySpec(secret.getBytes(),
                                         "AES");
5  // Select a cipher algorithm
6  Cipher cipher =Cipher.getInstance("AES/ECB/PKCS5PADDING");
7  // Encrypt
8  cipher.init(Cipher.ENCRYPT_MODE, key);
9  byte[] ciphertext = cipher.doFinal(plaintext.getBytes());
10 // Hash
11 MessageDigest digest = MessageDigest.getInstance("SHA-1");
12 digest.update(plaintext.getBytes());
13 byte[] hashValue = digest.digest();
```

Fig. 2: A cryptographic misuse example

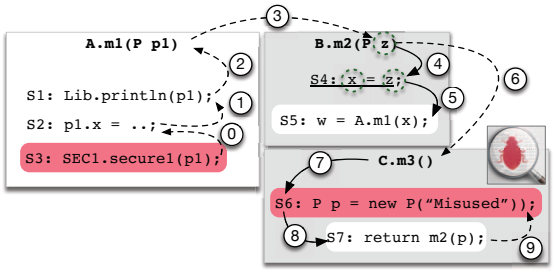

Fig. 3: Applying control and data flow analyses for vulnerability detection.

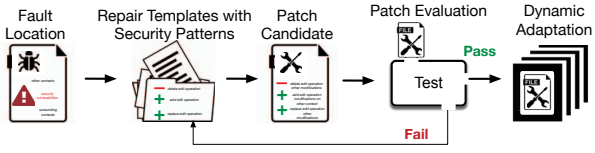

Fig. 4: Generating and testing repair patches for adapting to fix security bugs at runtime.

tracks a variable `p1` backward, starting at `S3` specified by security patterns in Section III-A. It then identifies semantic dependencies between procedures `A.m1`, `B.m2`, and `C.m3` using inter-procedural control and data flow information. FireBugs leverages security patterns to report a security API misuse in an object creation at `S6`, which leads to vulnerabilities in `SEC1.secure1()` at `S3`.

Code Transformation. Figure 4 shows how FireBugs automatically generates and evaluates repair patches to dynamically fix bugs at runtime. Given fault locations, FireBugs takes as input a program with security bugs and a set of repair templates. It parses the source code into Abstract Syntax Trees (AST). While scanning a given program's AST, it analyzes the vulnerable locations and their adjacent locations to determine whether the matched locations can be edited by repair templates. If editable locations are found, FireBugs generates a repair patch that modifies the given program's AST based on the corresponding edit operations in the templates.

FireBugs defines a set of repair edit operations that compute AST differences between a pre- and post-repair patch [16]. For the program modification, it represents these differences in repair templates by encoding the required behaviors and vulnerabilities of each security pattern. The edit operations provide structural constraints for a concrete repair patch.

FireBugs performs change impact analysis [22] to identify a subset of regression test suites whose execution behavior may have been altered by repair changes. It constructs call graphs by tracing the execution and then determines a subset to be executed against the modified version by correlating the changes against the call graphs of the pre-repair program.

To apply a repair patch to a buggy program, FireBugs incorporates an approach of Aspect-Oriented Programming [18]. It encapsulates adaptation concerns for repairs with aspects to be woven into joint points of matched vulnerable locations through dynamic runtime adaptation [7]. FireBugs composes crosscutting repair concerns specified in repair templates by using the dynamic AOP technique.

IV. Experimentation and Preliminary Results

To evaluate the effectiveness of our tool, we use existing open source projects, such as 2,800 Android app repositories where 200 apps use `javax.crypto` APIs to implement their security features. We wrote a batch script which crawls GitHub repositories based on four search queries: (1) Java programming language, (2) Android platform, (3) commits pushed after January 1st 2018, and (4) star rates more than 100. Table III shows the size of each subject project, measuring the number of LOC and the number of classes, methods, and fields. 200 subject projects are partitioned by 11 categories based on a manual review of the project descriptions.

	LOC	#CLASS	#METHOD	#FIELD
C_1	229,389	2,944	22,917	13,731
C_2	48,233	386	3,165	2,250
C_3	10,435	267	1,020	429
C_4	31,340	393	2,991	1,296
C_5	33,626	414	2,907	1,805
C_6	41,541	441	3,866	1,890
C_7	50,550	504	4,217	3,029
C_8	28,831	570	2,727	2,988
C_9	22,426	253	1,480	794
C_{10}	62,658	594	4,317	3,386
C_{11}	29,673	303	2,346	1,424

TABLE III: The 200 subject Android projects partitioned by 11 categories with the average number of Lines of Source Code (LOC) and the average number of classes, methods, and fields. C_n is a category of Android apps, including browser, client, demo, dev tool, entertainment, finance, health, library, security, social, and utility

In these subject Android apps, a number of vulnerabilities have been uncovered over the years. Open source projects maintain a public repository that contains a history of all code modifications so that previous versions can be reconstructed. A defect tracking system, where vulnerabilities are reported and fixed, can be used to find security bugs in real-world applications. This makes it possible to identify when professional software developers have patched such problems.

Our overall strategy is to identify, collect, and analyze a corpus of such project repositories. For each project, we identify a subset of code modifications where vulnerabilities have been fixed. For each vulnerability, we extract two versions of the system, V_1 and V_2, representing the versions of the code before and after the vulnerability was fixed, respectively. We then analyze V_1 and apply our generated patch to it, creating a new version V_g. The behavior of V_g is compared to V_2 to verify that the vulnerability in V_1 has been fixed in an identical manner. The behavior can be compared by running tests which are typically included in the project repository. For fixes where tests are not available, they can be created using existing test generators such as Randoop [23].

V. Conclusion

Security failures often cause crucial issues in software systems. Isolating security faults in software is challenging and inspecting legacy systems to find security bugs is even more difficult. This paper presents FireBugs to find and repair security bugs based on seven common security patterns. In the evaluation, we investigated 2,800 Android app repositories to apply our approach to 200 subject projects that use `javax.crypto` APIs. Our vision is to provide tool-assisted capabilities to automatically reduce software maintenance burdens of the number of outstanding software defects that exceed available resources. Our ultimate goal is to design more security patterns that have a positive impact on software quality by disseminating correlated sets of best security design practices and knowledge.

References

[1] Warwick Ashford. Economic impact of cyber crime is significant and rising. *Computer Weekly*, 2018.

[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1), 2012.

[3] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.

[4] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. Organizing security patterns. *IEEE Software*, 24(4), 2007.

[5] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. Growing a pattern language (for security). In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 139–158. ACM, 2012.

[6] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 211–222. ACM, 2015.

[7] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 141–147. ACM, 2002.

[8] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: A case study and open problems. In *Proceedings of Asia-Pacific Workshop on Systems (APSys)*, pages 7:1–7:7, 2014.

[9] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do Java developers struggle with cryptography APIs? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 935–946, 2016.

[10] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic analysis on misuse of cryptography functions in iOS applications. In *Network and System Security*, pages 349–362, 2014.

[11] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in Android applications. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 73–84, 2013.

[12] S. Shuai, D. Guowei, G. Tao, Y. Tianchang, and S. Chenjie. Modelling analysis and auto-detection of cryptographic misuse in Android applications. In *Proceedings of the IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80, 2014.

[13] Siqi Ma, David Lo, Teng Li, and Robert 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 (ASIA CCS)*, pages 711–722, 2016.

[14] Peter C Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 202–212. ACM, 2013.

[15] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[16] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11), 2007.

[17] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. Towards secure integration of cryptographic software. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 1–13, 2015.

[18] Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, 2004.

[19] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, 6th edition, 2013.

[20] Karthikeyan Bhargavan and Gaëtan Leurent. On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, 2016.

[21] RFC 2898: PKCS #5: Password-based cryptography specification. https://www.ietf.org/rfc/rfc2898.txt.

[22] Xiaoxia Ren, Ophelia C Chesley, and Barbara G Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE Transactions on Software Engineering*, 32(9):718–732, 2006.

[23] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 815–816. ACM, 2007.