Example-Based Vulnerability Detection and Repair in Java Code

Ying Zhang
yingzhang@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Ya Xiao
yax99@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Md Mahir Asef Kabir
mdmahirasefk@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Danfeng (Daphne) Yao
danfeng@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

Na Meng
nm8247@vt.edu
Virginia Tech
Blacksburg, Virginia, USA

ABSTRACT

The Java libraries JCA and JSSE offer cryptographic APIs to facilitate secure coding. When developers misuse some of the APIs, their code becomes vulnerable to cyber-attacks. To eliminate such vulnerabilities, people built tools to detect security-API misuses via pattern matching. However, most tools do not (1) fix misuses or (2) allow users to extend tools' pattern sets. To overcome both limitations, we created Seader—an example-based approach to detect and repair security-API misuses. Given an exemplar ⟨insecure, secure⟩ code pair, Seader compares the snippets to infer any API-misuse template and corresponding fixing edit. Based on the inferred info, given a program, Seader performs inter-procedural static analysis to search for security-API misuses and to propose customized fixes.

For evaluation, we applied Seader to 28 ⟨insecure, secure⟩ code pairs; Seader successfully inferred 21 unique API-misuse templates and related fixes. With these ⟨vulnerability, fix⟩ patterns, we applied Seader to a program benchmark that has 86 known vulnerabilities. Seader detected vulnerabilities with 95% precision, 72% recall, and 82% F-score. We also applied Seader to 100 open-source projects and manually checked 77 suggested repairs; 76 of the repairs were correct. Seader can help developers correctly use security APIs.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **Security and privacy → Software security engineering**.

KEYWORDS

Vulnerability repair, pattern inference, inter-procedural analysis

ACM Reference Format:
Ying Zhang, Ya Xiao, Md Mahir Asef Kabir, Danfeng (Daphne) Yao, and Na Meng. 2022. Example-Based Vulnerability Detection and Repair in Java Code. In *30th International Conference on Program Comprehension (ICPC '22), May 16–17, 2022, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524610.3527749

ICPC '22, May 16–17, 2022, Virtual Event, USA

ACM ISBN 978-1-4503-9298-3/22/05.
https://doi.org/10.1145/3524610.3527749

1 INTRODUCTION

JCA (Java Cryptography Architecture [24]) and JSSE (Java Secure Socket Extension [5]) are two cryptographic frameworks, provided by the standard Java platform. These frameworks offer security APIs to ease developers' secure software development. For instance, some of the APIs support key generation and secure communication. However, these libraries are not easy to use for two reasons. First, some APIs have overly complicated usage that is poorly documented [21, 38]. Second, developers lack the necessary cybersecurity training to correctly implement security features [2, 3, 36]. Prior work shows that developers misused security APIs [18, 40], and thus introduced vulnerabilities into software [16, 20]. For instance, Fischer et al. found that the security-API misuses posted on StackOverflow [6] were copied and pasted into 196,403 Android applications available on Google Play [18]. Fahl et al. [16] and Georgiev et al. [20] showed that such API misuses in software could be exploited by hackers to steal data (e.g., user credentials).

Existing tools are insufficient to help developers eliminate security-API misuses. Table 1 summarizes both capability and extensibility of the mainstream techniques, and compares the tools with our new approach Seader. As shown in the table, existing tools usually represent cryptographic API misuses as built-in rules [9, 11, 15, 29, 40]; users cannot easily extend these tools to detect more API-related vulnerabilities. As more security libraries emerge and evolve, we believe that vulnerability detectors should have good extensibility to keep their pattern sets of API-misuses up-to-date. Although CogniCrypt [26] offers a domain-specific language (DSL), CrySL [27], for users to prescribe the usage templates of cryptographic APIs, users need to spend lots of time learning CrySL and crafting templates. VuRLE [30] infers templates from user-provided code examples. However, its algorithm does not observe the unique characteristics of security API-misuses (e.g., using an integer within certain range); thus, VuRLE cannot always detect or fix misuses effectively.

Additionally, most existing tools merely report misuses, without suggesting any customized fixes. When developers lack the cybersecurity knowledge to understand the reported misuses, they may continue making mistakes when trying to fix those issues independently [43]. Although CDRep [29] and VuRLE [30] can suggest customized fixes, they are separately limited by (1) the inextensible hardcoded pattern set and (2) the intra-procedural analysis adopted for template matching. Please refer to Section 5 for more details.

To overcome the limitations of existing systems, we introduce Seader (short for "security-API misuse detection and repair")—our

Table 1: Comparison of SEADER against the existing detectors for security-API misuses

Tool	API-Misuse Representation			Misuse-Matching Strategy			Output	
	Built-in Rule	Template	Other	Intra-procedural Analysis	Inter-procedural Analysis	Other	Misuse	Repair
CryptoLint [15]	✓				✓		✓	
CDRep [29]	✓				✓		✓	✓
CogniCrypt [26]		✓			✓		✓	
CryptoGuard [40]	✓				✓		✓	
FindSecBugs [9]	✓				✓		✓	
Fischer et al.'s tool [18]			✓	✓		✓	✓	
SonarQube [11]	✓				✓		✓	
VuRLE [30]		✓		✓			✓	✓
SecureSync [39]			✓	✓			✓	
SEADER		✓			✓		✓	✓

new approach for vulnerability detection and repair from a data-driven perspective. As shown in Figure 1, there are two phases in SEADER: pattern inference and pattern application. In Phase I, suppose that a domain expert (e.g., security researcher) provides

- ***I***—insecure code with certain security-API misuse, and
- ***S***—the secure counterpart showing the correct API usage.

SEADER compares the two code snippets and detects program changes that can transform ***I*** to ***S***. Next, based on those changes, SEADER conducts *intra-procedural* analysis to derive a vulnerability-repair pattern. Each pattern has two parts: (i) a vulnerable code template together with matching-related information, and (ii) the abstract fix. SEADER stores all inferred patterns into a JSON file. In Phase II, given a program ***P***, SEADER loads patterns from the JSON file, and conducts *inter-procedural* program analysis to match code with any template. For each code match, SEADER concretizes the corresponding abstract fix, and suggests code replacements to developers.

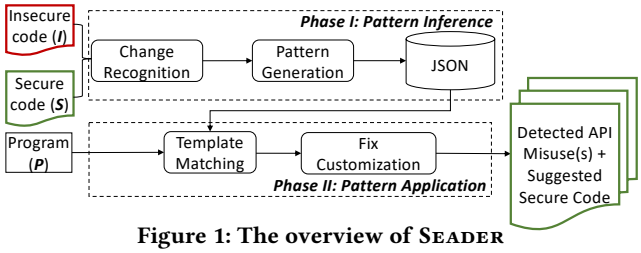

Figure 1: The overview of SEADER

According to the existing API-misuse patterns mentioned in prior work [18, 40], there are three unique kinds of security-API misuses that are hard to express with plain code examples, and are thus difficult to infer for existing program differencing-based approaches (e.g., VuRLE and SecureSync). Such misuses are about API invocations with (i) constants instead of random values, (ii) multiple alternative specialized constants, or (iii) constants in certain value ranges (see Section 3.5 and Table 3 for more details). To facilitate users to describe these patterns via code examples, we defined three novel specialized ways of example specification, and developed SEADER to specially infer patterns from those examples.

For evaluation, we crafted 28 ⟨insecure, secure⟩ code pairs based on the API-misuse patterns summarized by prior research. After SEADER inferred patterns from those pairs, we further applied SEADER to two program datasets to evaluate its effectiveness in vulnerability detection and repair. When applied to the first dataset, SEADER detected vulnerabilities with 95% precision, 72% recall, and 82% F-score. After applying SEADER to the second dataset, we inspected 77 repairs output by SEADER and found 76 of them correct.

To sum up, we made the following research contributions:

- We developed SEADER—a new approach that performs *intra-procedural analysis* to infer vulnerability-repair patterns from ⟨insecure, secure⟩ code examples, does *inter-procedural analysis* to match code with vulnerability templates, and customizes abstract fixes to suggest repairs. No prior work combines intra- with inter-procedural analysis in such a way.
- SEADER supports specialized ways of example specification, which enable users to define examples for API misuses related to arbitrary constant parameters, constant parameters within certain ranges, and alternative constants. No prior work has such speciality to strengthen the expressiveness of example-based pattern specification.
- We conducted a comprehensive evaluation with SEADER. We observed that for vulnerability detection, SEADER achieved a higher F-score than three state-of-the-art tools. For repair suggestion, SEADER achieved 99% (76/77) accuracy.

SEADER's extensibility is realized by its capability of inferring patterns from provided $\langle I, S \rangle$ code examples. As security experts offer examples for new misuse patterns, SEADER can infer those patterns to extend its pattern set. Additionally, SEADER repairs misused APIs by applying the inferred knowledge to given codebases. We open-sourced our program and datasets at https://github.com/NiSE-Virginia-Tech/ying-ICPC-2022.

2 A MOTIVATING EXAMPLE

This section overviews our approach with several code examples. Prior work shows that the security of symmetric encryption schemes depends on the secrecy of shared key [15]. Thus, developers should not generate secret keys from constant values hardcoded in programs [18]. Suppose a security expert **Alex** wants to detect and fix such vulnerabilities using SEADER. Alex needs to craft (1) an insecure code example to show the API misuse, and (2) a secure example for the correct API usage. As shown in Figure 2, the insecure code ***I*** invokes the constructor of `SecretKeySpec` by passing in a constant array. Here, `ByteLiterals.CONSTANT_ARRAY` is the specialized way that SEADER requires users to adopt when they represent any byte-array constant. Meanwhile, the secure code ***S*** invokes the same API with `key`—a generated unpredictable value.

Given the two examples, SEADER generates abstract syntax trees (ASTs) and compares them for any AST edit operation. For Figure 2, SEADER creates an expression update and multiple statement insertions. The update operation replaces `ByteLiterals.CONSTANT_ARRAY` with `key`. Next, based on the updated expression in ***I***, SEADER conducts data-dependency analysis to find any security API that uses the expression, and treats it as a **critical API**. Such critical APIs

Insecure code (*I*)

```
void test() {
  SecretKey sekey= new SecretKeySpec(ByteLiterals.CONSTANT_ARRAY,
        "AES");  }
```

Secure code (*S*)

```
// store the key as a field for reuse purpose
byte[] key = keyInit();

// create a key based on an unpredictable random value
public byte[] keyInit() {
  try {
    KeyGenerator keyGen=KeyGenerator.getInstance("AES");
    keyGen.init(256);
    SecretKey secretKey = keyGen.generateKey();
    byte[] keyBytes= secretKey.getEncoded();
    return keyBytes;
  } catch (Exception e) {
    e.printStackTrace();
    return null;
  }
}
void test() {
  SecretKey sekey= new SecretKeySpec(key, "AES"); }
```

Figure 2: A pair of examples to show the vulnerability and repair relevant to secret key creation

Vulnerable code template (T)

```
SecretKey $v_0$ = new SecretKeySpec(ByteLiterals.CONSTANT_ARRAY, "AES");
```

Matching-related data:

critical API: javax.crypto.spec.SecretKeySpec.SecretKeySpec(byte[], String)
other security APIs: {}

Abstract fix (F)

Replace the matched statement with:

```
SecretKey $v_0$ = new SecretKeySpec($v_1$, "AES");
```

Add these lines before the container method of the matched statement:

```
// store the key as a field for reuse purpose
byte[] $v_1$ = $m_0$();
// create a key based on an unpredictable random value
public byte[] $m_0$() {
  try {
    KeyGenerator $v_4$=KeyGenerator.getInstance("AES");
    $v_4$.init(256);
    SecretKey $v_3$ = $v_4$.generateKey();
    byte[] $v_2$= $v_3$.getEncoded();
    return $v_2$;
  } catch (Exception $v_5$) {
    $v_5$.printStackTrace();
    return null;
  }
}
```

Figure 3: The pattern inferred from the code pair in Figure 2

are important for Seader to later detect similar vulnerabilities in other codebases. Afterwards, Seader generalizes a **vulnerability-repair pattern** from the examples by abstracting away concrete variable/method names and edit-irrelevant code. As shown in Figure 3, the generalized pattern has two parts: the vulnerability template (T) together with matching-related data, and an abstract fix (F). Such pattern generalization ensures the transformation applicable to codebases with distinct program contexts.

With a pattern inferred from the provided code pair, Alex can further apply Seader to an arbitrary program ***P***, to detect and fix any occurrence of the described vulnerability. In particular, given a program whose simplified version is shown in Listing 1, Seader first scans for any invocation of the critical API `SecretKeySpec(...)`. If no such invocation exists, Seader concludes that ***P*** does not have the above-mentioned vulnerability; otherwise, if the API is invoked (see line 8 in Listing 1), Seader then searches for any code matching the template in Figure 3. The template-matching process conducts inter-procedural analysis and checks for two conditions:

Listing 1: A simplified version of *P*

```
1 public class CEncryptor {
2   private char[] passPhrase;
3   private String alg = "AES";
4   public CEncryptor(String passPhrase) {
5     this.passPhrase = passPhrase.toCharArray();
6   }
7   public Result encrypt(byte[] plain) throws Exception {
8     SecretKey secret = new SecretKeySpec(new String(passPhrase).
          getBytes(), alg);
9   ...
10 }
11 public class Main {
12   public static void main(String[] args)
13   CEncryptor aes0 = new CEncryptor("password");
14   aes0.encrypt((byte[])args[0]);
15   ...
16 }
```

Replace the matched statement with:

```
SecretKey secret = new SecretKeySpec($v_1$, "AES");
```

Add these lines before the method encrypt(byte[] plain):

```
1.  // store the key as a field for reuse purpose
2.  byte[] $v_1$ = $m_0$();
3.  // create a key based on an unpredictable random value
4.  public byte[] $m_0$() {
5.    try {
6.      KeyGenerator $v_4$=KeyGenerator.getInstance("AES");
7.      $v_4$.init(256);
8.      SecretKey $v_3$ = $v_4$.generateKey();
9.      byte[] $v_2$= $v_3$.getEncoded();
10.     return $v_2$;
11.   } catch (Exception $v_5$) {
12.     $v_5$.printStackTrace();
13.     return null;
14.   }
15. }
```

Figure 4: A customized fix for *P* suggested by Seader

C1: Is the first parameter derived from a constant?
C2: Does the second parameter exactly match `"AES"`?

If any invocation of `SecretKeySpec(...)` satisfies both conditions, Seader reports the code to be vulnerable. Notice that if we only check line 8 of Listing 1, neither `new String(passPhrase). getBytes()` nor `alg` satisfies any condition. Thanks to the usage of inter-procedural analysis, Seader can perform backward slicing to trace how both parameters are initialized. Because `alg` is a private field of `CEncryptor`, whose value is initialized on line 3 with `"AES"`, Seader decides that C2 is satisfied. Similarly, `passPhrase` is another field whose value is initialized with a parameter of the constructor `CEncryptor(...)` (lines 4-6). When `CEncryptor(...)` is called with parameter `"password"` before the invocation of `SecretKeySpec(...)` (lines 7-14), C1 is satisfied. Therefore, Seader concludes that line 8 matches the template; it matches concrete variable `secret` with the template variable `$v_0$`.

For the found code match, Seader customizes the abstract fix shown in Figure 3 by replacing the abstract variable `$v_0$` with concrete variable `secret`. As shown in Figure 4, the customized fix first initializes a `KeyGenerator` instance with the algorithm "`AES`" and the key size "`256`", to generate an unpredictable AES key (lines 6–8). Next, the AES key is converted to a byte array (line 9), which value can be stored into a Java field so that the value is reusable by both encryption and decryption modules. Additionally, inside the method `encrypt(...)`, the original vulnerable statement is updated to create a secret key using the generated byte array.

3 APPROACH

There are two challenges to overcome in our research:

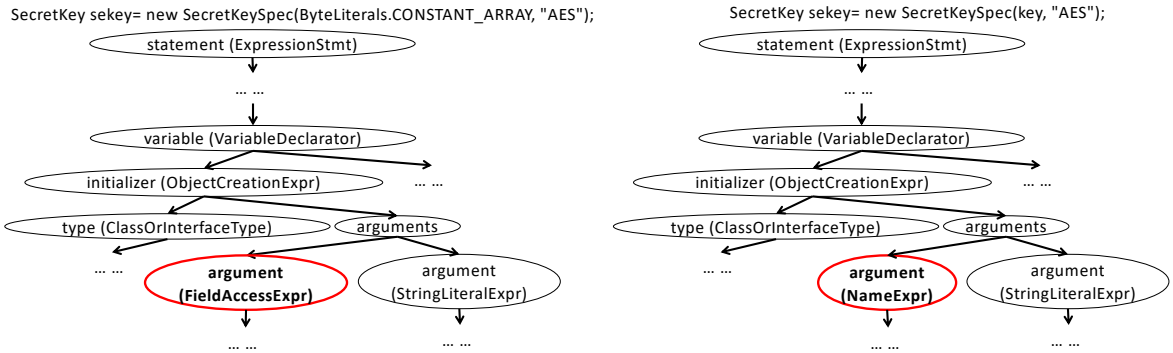

Figure 5: The simplified ASTs of the two statements related to a statement-level update operation

(1) How can we infer generalized vulnerability-repair patterns from concrete ⟨insecure, secure⟩ code examples?
(2) How can we ensure that the inferred patterns are applicable to code that is different from the original examples?

To address these challenges, as shown in Figure 1, we designed two phases in Seader. The first phase takes two steps to infer vulnerability-repair patterns from ⟨insecure, secure⟩ code examples; the second phase contains another two steps to apply inferred patterns to given programs. In this section, we will first describe each of the four steps in detail (Section 3.1-Section 3.4). Next, we will explain the three specialized ways of example specification, which can facilitate users to demonstrate certain API misuses (Section 3.5).

3.1 Change Recognition

Given an $\langle I, S \rangle$ example pair, Seader compares code to locate (1) the root cause of any vulnerability demonstrated by I and (2) the security patch shown in S. Specifically, Seader applies syntactic program differencing to the code pair, to reveal any edit operation(s) that can transform I to S. This step consists of two parts: statement-level change recognition and expression-level change recognition.

3.1.1 Statement-level change recognition. Seader first uses JavaParser [23] to generate ASTs for I and S, and then compares ASTs to create three types of edit operations:

- **delete (Node** a**)**: Delete node a.
- **insert (Node** a**, Node** b**, int** k**)**: Insert node a and position it as the $(k+1)^{th}$ child of node b.
- **update (Node** a**, Node** b**)**: Replace a with b. This operation changes a's content.

Specifically, when comparing any two statements $s_i \in I$ and $s_s \in S$, Seader checks whether the code string of s_i exactly matches that of s_j; if so, Seader considers s_i unchanged while I is transformed to S. Otherwise, if the code strings of s_i and s_j are different, Seader normalizes both statements by replacing concrete variables (e.g., `key`) with abstract ones (e.g., `$v_0`), and replacing constant values (e.g., `"AES"`) with abstract constants (e.g., `$c_0`). We denote the normalized representations as n_i and n_s. Next, Seader computes the Levenshtein edit distance [28] between n_i and n_s, and computes the similarity score [19] with:

$$sim = 1 - \frac{edit\_distance}{max\_length(n_i, n_s)}$$

The similarity score sim is within $[0, 1]$. When $sim = 1$, n_i and n_j are identical. We set a threshold $th = 0.8$ such that if $sim >= th$, n_i and n_j are considered to match. In this way, Seader can identify update operation(s). Compared with string-based match, the normalization-based match is more flexible, because it can match any two statements that have similar syntactic structures but distinct variables or constants. Finally, if a statement $s_i \in I$ does not find a match in S, Seader infers a delete operation; if $s_s \in S$ is unmatched, Seader infers an insert operation.

3.1.2 Expression-level change recognition. For each statement-level update, Seader tries to identify any finer-granularity edit (i.e., expression replacement) to better comprehend changes, and to prepare for later pattern generation (see Section 3.2). When s_i is updated to s_s, Seader conducts top-down matching between ASTs to identify edits. Namely, while traversing both trees in a preorder manner, Seader compares roots and inner nodes based on the AST node types, and compares leaf nodes based on the code content. Such node traversal and comparison continue until Seader finds all unmatched subtrees or leaves.

For the example code shown in Figure 2, with statement-level change recognition, Seader reveals one statement update and multiple statement additions. Figure 5 shows the simplified ASTs of both before- and after- versions for the updated statement. By comparing the ASTs in a top-down manner, Seader finds the first arguments sent to the constructor to differ (e.g., `FieldAccessExpr` vs. `NameExpr`). Thus, Seader creates a finer-granularity operation to replace the statement-level update: **update (`ByteLiterals.CONSTANT_ARRAY, key`)**.

Notice that we decided not to use existing tools, such as GumTree [17] and ChangeDistiller [19], to recognize changes for a variety of reasons. First, GumTree often mismatches nodes against developers' intent [32]. GumTree can generate four types of edit operations: add, delete, update, and move. However, in our research, we need only three edit types: add, delete, and update, so that Seader can infer API-misuse patterns from recognized changes. Second, ChangeDistiller only detects statement-level changes, without identifying expression-level changes. Additionally, it also generates four edit types. To avoid (1) fixing bugs in GumTree and (2) revising current tools to report three instead of four types of edit operations, we created our own program differencing algorithm.

3.2 Pattern Generation

When security experts present an $\langle I, S \rangle$ example pair to demonstrate any API misuse, we expect that they provide the code snippets to show only one vulnerability and its repair. Additionally, based on our experience with security-API misuses, each vulnerability is

Insecure code (*I*)

```
1 void test(int iterations) {
2   byte[] salt = new byte[4];
3   AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt,
        iterations); }
```

Secure code (*S*)

```
1 void test(int iterations) {
2   byte[] salt = new byte[8];
3   AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt,
        iterations); }
```

Figure 6: An $\langle I, S\rangle$ where a critical API `PBEParameterSpec(...)` indirectly depends on a updated constant

usually caused by the misuse of one security API. Therefore, to infer a general vulnerability-repair pattern from a given code pair, we need to overcome two technical challenges:

- How can we identify the security API whose misuse is responsible for the vulnerability (i.e., critical API)?
- How should we capture any relationship between the critical API and its surrounding code?

3.2.1 Task 1: Identifying the critical API. Starting with the edit script E created in Section 3.1, Seader looks for any update operation $update(e, e')$. If there is such an operation, Seader searches for the security API whose invocation is data-dependent on e or e', and considers the API to be *critical*. For the example shown in Figure 5, the critical API is `SecretKeySpec(byte[], String)` because it is invoked with the updated expression as the first argument. Similarly, Figure 6 presents another example where a numeric literal is updated from `4` to `8`. With data-dependency analysis, Seader reveals that the constants are used to define variable `salt`, while `salt` is used as an argument when `PBEParameterSpec(...)` is invoked. Therefore, the method invocation depends on the updated expression, and the security API `PBEParameterSpec(byte[], int)` is considered *critical*.

If there is no update operation in E, Seader searches for any overridden security API that encloses all edit operations, and considers the overridden API to be *critical*. Take the code pair shown in Figure 7 as an example. By comparing I with S, Seader can identify one statement deletion and multiple statement insertions. As there is no update operation and all edit operations are enclosed by an overridden method `verify(String, SSLSession)` (indicated by `@Override`), Seader further locates the interface or super class declaring the method (e.g., `HostnameVerifier`). If the overridden method together with the interface/super class matches any known security API, Seader concludes the overridden method to be *critical*.

Lastly, if no update operation or overridden security API is identified, Seader checks whether there is any deletion of security API call in E; if so, the API is *critical*. To facilitate later template matching (Section 3.3), for each identified critical API, Seader records the method binding information (e.g., `javax.crypto.spec.SecretKeySpec.SecretKeySpec(byte[], String)`).

3.2.2 Task 2: Extracting relationship between the critical API and its surrounding code. When a vulnerable code example has multiple statements (e.g., Figures 6 and 7), we were curious how the critical API invocation is related to other statements. On one extreme, if the invocation is irrelevant to all surrounding statements, we should not include any surrounding code into the generalized pattern. On the other extreme, if the invocation is related to all surrounding code, we should take all code into account when inferring a vulnerability-repair pattern. Thus, this task intends to

Insecure code (*I*)

```
1 public class HostVerifier implements HostnameVerifier {
2   @Override
3   public boolean verify(String hostname, SSLSession sslSession){
4     return true;     }}
```

Secure code (*S*)

```
1 public class HostVerifier implements HostnameVerifier {
2   @Override
3   public boolean verify(String hostname, SSLSession sslSession){
4     // Please change "example.com" as needed
5     if ("example.com".equals(hostname)) {
6       return true;
7     }
8     HostnameVerifier hv = HttpsURLConnection.
          getDefaultHostnameVerifier();
9     return hv.verify(hostname, sslSession);     }}
```

Figure 7: A pair of examples from which Seader infers the critical API to be an overridden method

decide (1) which statements of I to include into the vulnerable code template, (2) what additional security API call(s) to analyze for template matching (see Section 3.3), and (3) which statements of S to include into the abstract fix.

Seader performs intra-procedural data-dependency analysis. If a statement defines a variable whose value is (in)directly used by the critical API invocation, the statement is extracted as *edit-relevant context*. Seader uses such context to characterize the demonstrated vulnerability. For the insecure code I in Figure 6, since the API call (line 3) data-depends on variable `salt`, lines 2-3 are extracted as context. Additionally, when the critical API is an overridden method, its code implementation in I is considered edit-relevant context (see lines 3-4 in Figure 7). Based on the extracted edit-relevant context, Seader abstracts all variables to derive a vulnerable code template T, and records mappings M between abstract and concrete variables. In addition to the critical API, Seader also extracts binding information for any other security API invoked by the contextual code. Compared with edit-relevant context, these APIs provide more succinct hints. In our later template-matching process, these APIs can serve as "*anchors*" for Seader to efficiently decide whether a program slice is worth further comparison with the template.

To locate the fix-relevant code in secure version S, Seader identifies any unchanged code in the edit-relevant context, the inserted statements, and the new version of any updated statement. For the secure code S shown in Figure 6, lines 2-3 are fix-relevant, because line 2 is the new version of an updated statement and line 3 is unchanged contextual code. Similarly, for the secure code S shown in Figure 7, lines 3-9 are fix-relevant, because lines 3 presents the critical API while lines 4-9 are inserted statements. Based on the above-mentioned variable mappings M and fix-related code, Seader further abstracts variables used in the fix-related code to derive an abstract fix F. Seader ensures that the same concrete variables used in I and S are mapped to the same abstract variables.

To sum up, given a $\langle I, S\rangle$ pair, Seader produces a pattern $Pat = \langle T, F\rangle$, which has a vulnerable code template T, an abstract fix F, and metadata to describe T (i.e., bindings of security APIs).

3.3 Template Matching

Given a program P, Seader uses a static analysis framework—WALA [7]—to analyze the program JAR file (i.e., bytecode). As shown by lines 1.2–1.4 in Algorithm 1, to find any code in P that matches the template T, Seader first searches for the critical API

(i.e., invocation or method reimplementation). If the critical API does not exist, SEADER concludes that there is no match for T. Next, if the critical API is invoked at least once, for each invocation, SEADER conducts inter-procedural backward slicing to retrieve all code *Sli* on which the API call is data-dependent (i.e., `getBackwardSlice(x)`). When T invokes one or more security APIs in addition to the critical API, SEADER further examines whether *Sli* contains matches for those extra APIs; if not, the matching trial fails (see lines 1.8–1.9). Next, SEADER checks whether the matched code in *Sli* preserves the data dependencies manifested by T (i.e., `dataDependConsist(T, Sli)`). If those data dependencies also match, SEADER reveals a vulnerability (see lines 1.10–1.11).

Alternatively, if the critical API is reimplemented, for each reimplementation, SEADER compares the code content against T, and reports a vulnerability if they match (see lines 1.13–1.14). At the end of this step, if any vulnerability is detected, SEADER presents the line number where the critical API is invoked or is declared as an overridden method, and shows related matching details. The matching details include both code matches and abstract-concrete variable mappings.

Actually, we designed our algorithm of template matching based on three considerations. First, as developers provide code examples in Java but WALA analyzes JAR files, template matching should leverage the minimum information (i.e., security APIs and variable data dependencies) to overcome any discrepancy between program representations (i.e., source code vs. bytecode). Second, although SEADER infers templates from simple code examples via intra-procedural analysis, we need to match code with templates step-by-step via inter-procedural analysis, so that SEADER can find matches even if the program context is more complicated. Third, many security-API misuses are relevant to parameter usage or method overriding, so our matching algorithm observes such unique characteristics to establish matches.

Algorithm 1: Matching Program P to template T

```
   Input: P, T, D /* program, template, and related metadata      */
   Output: Matched /* a set of code matches from P to T          */
1.1 Candi := ∅, Matched := ∅;
   /* 1. search for matches of the critical API                  */
1.2 foreach code line x ∈ P do
1.3     if x invokes D(critical) || x declares D(critical) then
1.4         Candi := Candi ∪ x;
1.5 foreach x ∈Candi do
1.6     if x invokes D(critical) then
            /* 2(a). For API call, do program slicing and look for
               matches of other security APIs                     */
1.7         Sli = getBackwardSlice(x);
1.8         if (Sli has all matches for D(other)) == false then
1.9             continue;
            /* 3. check whether the data dependencies between
               security APIs in T match those in Sli             */
1.10        if dataDependConsist(T, Sli) then
1.11            Matched:=Matched ∪ {Sli, mappings};
1.12    else
            /* 2(b). For API overriding, check the code           */
1.13        if contentMatch(code(P, x), T) then
1.14            Matched := Matched ∪ {code(P, x), mappings};
```

Table 2: The stubs defined to ease example specification

Class	Members	Semantics
StringLiterals	StringLiterals (String... a)	This constructor creates a `StringLiterals` object with one or more string literals.
	getAString()	This method randomly returns one of the strings originally used to construct the `StringLiterals` object.
ByteLiterals	CONSTANT_ ARRAY	This field serves as a placeholder for a byte-array constant, whose value can be unspecified.
CharLiterals	CONSTANT_ ARRAY	This field serves as a placeholder for a char-array constant, whose value can be unspecified.

3.4 Fix Customization

This step involves two types of customization: variable customization and edit customization. To customize variables, based on the matching details mentioned in Section 3.3, SEADER replaces abstract variables in F with the corresponding concrete ones. We denote this customized version as F_c. For edit customization, SEADER suggests code replacements in two distinct ways depeding on the inferred edit operations mentioned in Section 3.1. Specifically, if there is only one update operation inferred, SEADER simply recommends an alternative expression to replace the original expression. Otherwise, SEADER presents F_c for developers to consider.

Notice that SEADER does not directly modify P to repair any vulnerability for two reasons. First, when template T contains multiple statements, it is possible that the corresponding code match involves statements from multiple method bodies. Automatically editing those statements can be risky and cause unpredictable impacts on program semantics. Second, some fixes require for developers' further customization based on their software environments (e.g., network configurations, file systems, and security infrastructures). As implied by Figure 7, the abstract fix derived from S will contain a comment `"//Please change 'example.com' as needed"`, so will the customized fix by SEADER. This comment instructs developers to replace the standard hostname based on their circustances.

3.5 Specialized Ways of Example Specification

We believe that by crafting $\langle I, S \rangle$ code pairs, security experts can demonstrate the misuse and correct usage of security APIs. However, we also noticed some scenarios where plain Java examples cannot effectively reflect the vulnerability-repair patterns. To solve this problem, we defined three **stub Java classes** (i.e., fake classes) for user adoption and invented **three specialized ways of example definition**. As shown in Table 2, the stub classes offer stub methods or fields to facilitate constant-related example specification. This section explains the scenarios where our special specification methods are needed.

*Scenario 1. An API misuse involves an **arbitrary** constant value instead of any **particular** constant.* Plain examples only show the usage of particular constant values, but cannot generally represent the constant concept. Consider the vulnerability introduced in Section 2. Without using `ByteLiterals.CONSTANT_ARRAY`, a domain expert has to define a plain example to show the API misuse, such as

```
SecretKey sekey = new SecretKeySpec("ABCDE".getBytes(), "AES");
```

SEADER is designed to preserve all string literals from I when generalizing template T, and to look for those values when matching code with T. Consequently, given the above-mentioned example, SEADER will inevitably embed `"ABCDE"` into the inferred template. To help users avoid such unwanted literal values in T, we

Insecure code (*I*)

```
1 StringLiterals literals=new StringLiterals("AES", "RC2", "RC4",
      "RC5", "DES", "blowfish", "DESede", "ARCFOUR");
2 Cipher.getInstance(literals.getAString());
```

Secure code (*S*)

```
1 StringLiterals literals = new StringLiterals("AES/GCM/NoPadding"
      ,"RSA/ECB/OAEPWithSHA-1AndMGF1Padding");
2 Cipher.getInstance(literals.getAString());
```

Figure 8: A code pair where multiple alternative secure and insecure options are specified simultaneously

defined `ByteLiterals.CONSTANT_ARRAY` and `CharLiterals.CONSTANT_ARRAY`. These static fields can be used as placeholders or wildcards for constant arrays, to represent the general constant concept in examples. When Seader detects such fields in examples, it keeps them as they are in T and later matches them with constant values in P.

Scenario 2. An API misuse has multiple alternative insecure (or secure) options. Given a parameter of certain security API, suppose that there are (1) m distinct values to cause API misuse and (2) n alternatives to ensure correct API usage, where $m \geq 1$, $n \geq 1$. To express all possible combinations between the insecure and secure options via plain Java examples, users have to provide $m \times n$ pairs of examples, which practice is inefficient and undesirable. To solve this issue, we defined two stub methods in `StringLiterals`. As shown in Figure 8, one is a constructor of `StringLiterals`, which can take in any number of string literals as arguments (see line 1 in I) and store those values into an internal list structure. The other method is `getAString()`, which randomly picks and returns a value from that list (see line 2 in I). In this way, a domain expert can efficiently enumerate multiple secure/insecure options in just one code pair.

The examples in Figure 8 show that when security API `Cipher.getInstance(...)` is called, the parameter may have one of the insecure values (e.g., `"AES"`). Such vulnerability can be addressed when the value is replaced by one of the three secure options (e.g., `"AES/GCM/NoPadding"`). Given the example in Figure 8, Seader extracts insecure and secure options from `StringLiterals`-related statements, detects vulnerabilities in P if the security API is invoked with any insecure option, and suggests all secure alternatives.

Scenario 3. An API misuse requires for a parameter value in a specific range. Given an integer parameter p of certain API, suppose that there is a threshold value th such that the API invocation is secure only when $p \geq th$. To enumerate all possible vulnerable cases and related repairs via plain examples, theoretically, a user has to provide $(th - Integer.MIN\_VALUE) \times (Integer.MAX\_VALUE - th + 1)$ code pairs, which practice is infeasible. Therefore, we invented a special way of example definition, which requires users to provide only (1) one insecure example by setting p to a concrete value less than th and (2) one secure example by setting $p = th$. As shown in Figure 6, if a security expert wants to describe the pattern that *the array size of the first parameter should be no less than 8*, then s/he can define I by creating an array with a smaller size (i.e., 4) and define S by setting the size to 8. Seader can identify the integer literals used by I and S, and infer the secure value range $size \geq 8$.

4 EVALUATION

This section first describes the evaluation datasets and metrics, and then presents Seader's effectiveness of pattern inference. Next, it explains the tool effectiveness of pattern application, including vulnerability detection and repair. We did all experiments on Linux Mint 20.3 Cinnamon, version 5.2.7; we used Intel Core i7-8700 processor and 32GB memory.

4.1 Datasets

We used one dataset to evaluate pattern inference, and two datasets to evaluate pattern application.

4.1.1 A dataset to evaluate pattern inference. Prior research revealed a number of security-API misuses and related correct usage in Java [1, 14, 16, 18, 25, 31, 33, 36, 40, 42]. To evaluate Seader's effectiveness of pattern inference, we referred to those well-described API misuses and fixes while crafting code examples for Seader. Table 3 lists the 13 security class APIs we focused on, the insecure usage of certain method API(s) frequently mentioned by prior work, and the secure usage. With this domain knowledge, we handcrafted 28 $\langle I, S \rangle$ pairs. Among the pairs, 19 pairs are defined in the specialized ways introduced in Section 3.5, and 9 pairs are defined with plain Java examples. Within the 19 pairs, 8 pairs, 6 pairs, and 5 pairs separately belong to Scenarios 1–3.

4.1.2 Two datasets to evaluate pattern application. The first dataset is a third-party benchmark, consisting of 86 real vulnerabilities from 10 Apache open-source projects [8, 12]. We decided to use this dataset for two reasons. First, it was created by other researchers, so it can be used to objectively assess the effectiveness of different vulnerability detectors. Second, most of the 86 vulnerabilities belong to the 13 security classes shown in Table 3, so they can properly measure Seader's capability of pattern application. The second dataset contains 100 widely used Apache open-source projects. To create this dataset, we first ranked the Apache projects available on GitHub [4] in a descending order of their popularity (i.e., star counts). Next, we located the top 100 projects that satisfy the following constraints: (1) the project uses the security APIs that Seader examines; (2) the project is compilable because Seader analyzes the compiled JAR files. The resulting dataset is used to evaluate Seader's effectiveness of repair suggestion.

4.2 Metrics

As with prior work [40], we leveraged the following three metrics to measure tools' capability of vulnerability detection:

Precision (P) measures among all reported vulnerabilities, how many of them are true vulnerabilities.

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of reports}}$$

When a tool reports a set of vulnerabilities S_1 and the known set of vulnerabilities is S_2, we intersected S_1 with S_2 to automatically compute precision. Namely, $P = |S_1 \cap S_2|/|S_1|$.

Recall (R) measures among all known vulnerabilities, how many of them are detected by a tool.

$$R = \frac{\text{\# of correct reports}}{\text{Total \# of known vulnerabilities}}$$

When a tool reports a set of vulnerabilities S_1, we intersected S_1 with the set of known vulnerabilities S_2 to automatically compute recall, i.e., $R = |S_1 \cap S_2|/|S_2|$.

F-score (F) is the harmonic mean of precision and recall; it can reflect the trade-off between precision and recall.

$$F = \frac{2 \times P \times R}{P + R}$$

Table 3: The API misuses and related fixes summarized by prior work [1, 18, 33, 40]

Id	Security Class API	Insecure	Secure
1	Cipher	The algorithm and/or mode is set as AES, RC2, RC4, RC5, DES, DESede, AES/ECB, Blowfish, ARCFOUR, or RSA/None/NoPadding.	The algorithm and/or mode is set as AES/GCM/NoPadding, RSA/ECB/OAEPWithSHA-1AndMGF1Padding, AES/CFB/PKCS5Padding, or RSA/CBC/PKCS5Padding.
2	HostnameVerifier	Allow all hostnames.	Implement logic to actually verify hostnames.
3	IvParameterSpec	Create an initialization vector (IV) with a constant.	Create an IV with an unpredictable random value.
4	KeyPairGenerator	Create an RSA key pair where key size < 2048 bits or create an ECC key pair where key size < 224 bits.	RSA key size >= 2048 bits, ECC key size >= 224 bits.
5	KeyStore	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).
6	MessageDigest	The algorithm is MD2, MD5, SHA-1, or SHA-224.	The algorithms is SHA-256, SHA-512 or SHA-3.
7	PBEKeySpec	Create a PBEKey based on a constant salt.	Use an unpredictable random salt value to create the key.
8	PBEParameterSpec	Create a parameter for password-based encryption (PBE) by setting salt size < 64 bits or iteration count < 1000, or by using a constant salt.	Salt size >= 64 bits, iteration count >=1000. Use an unpredictable randomly generated salt value.
9	SecretKeyFactory	Create secret keys with algorithm DES, DESede, ARCFOUR, PBEWithMD5AndDES, or PBKDF2WithHmacSHA1.	Create secret keys with AES or PBEWithHmacSHA256AndAES_256.
10	SecretKeySpec	Create a secret key with a constant value, or using the algorithm DES, DESede, Blowfish, HmacSHA1, ARCFOUR, PBEWithMD5AndDES, or PBKDF2WithHmacSHA1.	Create a secret key with an unpredictable randomly generated value, or using the algorithm AES or PBEWithHmacSHA256AndAES_128.
11	SecureRandom	Use Random to generate random values, or set SecureRandom to use a constant seed.	Use SecureRandom instead of Random, and ensure the seed to be a random value.
12	SSLContext	Use the protocol SSL, TLSv1.0, or TLSv1.1.	Use the protocol TLSv1.2 or TLSv1.3
13	TrustManager	Trust all clients or servers	Check clients and/or check servers.

Table 4: The 28 code pairs for pattern inference

	Single statement	Multiple statements
Identical	4	5
Abstract	1	18

4.3 Effectiveness of Pattern Inference

As mentioned in Section 4.1, we crafted 28 code pairs to evaluate Seader's effectiveness of pattern inference. We categorized the 28 pairs based on two criteria:

C1. Do I and S contain single or multiple statements?
C2. Does pattern inference abstract variables?

The two conditions actually reflect the difficulty levels or challenges of these pattern inference tasks. For instance, if I or S has multiple statements, Seader conducts data-dependency analysis to locate the edit-relevant context in I or to reveal the fix-relevant code in S. If I or S uses variables, Seader abstracts all variable names to ensure the general applicability of inferred patterns. As shown in Table 4, there are four simplest pairs; Seader can handle these pairs without conducting any data-dependency analysis or identifier generalization. Meanwhile, there are 18 most complicated cases that require Seader to analyze data dependencies and generalize identifiers. In our evaluation, Seader correctly inferred patterns from all pairs. When some pairs present secure/insecure options (e.g., distinct string literals) for the *same* critical API, Seader merged the inferred patterns. In this way, Seader derived 21 unique patterns.

Finding 1: *Our experiment shows Seader's great capability of pattern inference. Seader shows impressive extensibility by inferring patterns from various examples.*

4.4 Effectiveness of Vulnerability Detection

To assess Seader's capability of vulnerability detection, we used a third-party dataset (see Section 4.1.2). We applied Seader and three state-of-the-art vulnerability detectors (i.e., CogniCrypt [26], CryptoGuard [40], and FindSecBugs [9]) to all subject programs. The tool versions we used include CogniCrypt-2.7.1, commit 94135c5 of CryptoGuard, and findsecbugs-cli-1.10.1. To ensure that CogniCrypt has enough memory during execution, instead of using its default configuration, we set the maximum heap size to 30G (-Xmx30g) and the maximum stack size to 60M (-Xss60m). For other tools, we adopted the default tool configuration in our experiments. Seader spent 262 seconds analyzing all programs. As shown in Table 5, Seader outperformed the other tools by acquiring the highest average recall (72%) and F-score (82%). It obtained the same average precision rate—95%—as CryptoGuard and FindSecBugs, which rate is much higher than that of CogniCrypt (i.e., 58%).

Seader reported API misuses in eight projects, while the other three tools reported issues in nine projects. As the dataset labels no vulnerability in `tika.jar` and all tools found zero vulnerability in that project, we could not measure P, R, or F for these tools. Seader was unable to analyze `tomee.jar`, because WALA does not always work well with the JAR files built by Maven [10]. We believe that once WALA developers overcome the limitation between WALA and Maven JARs in the future, Seader can also analyze `tomee.jar`. Among the four tools under comparison, CryptoGuard obtained a slightly lower F-score than Seader (78% vs. 82%), followed by FindSecBugs and CogniCrypt. Two possible reasons can explain Seader's higher F-score. First, thanks to its great extensibility, Seader has a larger pattern set of API misuses. Second, its inter-procedural analysis can accurately detect API misuses in more complex scenarios.

Analysis of False Positives. We manually inspected the cases where Seader did not report misuses correctly. We found one reason to explain why Seader did not achieve 100% precision: the ground truth is incomplete, as it labels some instead of all invocations of `Random()`. We currently consider the extra calls of `Random()` found by Seader to be false positives, although the actual precision rate is higher.

Analysis of False Negatives. Seader missed some labeled API misuses, because the corresponding misuse patterns are not covered by our 21 inferred patterns. Some of these missing patterns can be added to Seader if we feed the tool with more code examples. One missing pattern cannot get added even if we provide $\langle I, S \rangle$ pairs to Seader. The pattern is to replace `new URL("http://...")` with `new URL("https://...")`. HTTPS is HTTP with encryption. Nowadays

Table 5: Evaluation results on the 86-vulnerability dataset [8]

Apache Project	# of Labeled Vulnerabilities	CogniCrypt			CryptoGuard			FindSecBugs			Seader		
		P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)	P(%)	R(%)	F(%)
deltaspike.jar	2	40	100	57	100	100	100	100	100	100	100	100	100
directory-server.jar (apacheds-kerberos-codec)	19	51	95	67	100	26	42	100	58	73	94	84	89
incubator-taverna-workbench.jar	5	57	80	67	100	80	89	100	40	57	80	80	80
manifoldcf.jar (mcf-core)	3	17	33	22	60	100	75	75	100	86	75	100	86
meecrowave.jar	3	100	100	100	100	67	80	100	67	80	100	100	100
spark.jar	27	100	26	41	100	100	100	100	85	92	100	93	96
tika.jar	0	-	-	-	-	-	-	-	-	-	-	-	-
tomee.jar (openejb-core)	7	60	43	50	83	71	77	60	43	50	-	-	-
wicket.jar	5	40	40	40	100	100	100	100	40	57	100	60	75
artemis-commons.jar	15	100	40	57	100	27	42	100	33	50	100	40	57
Overall	86	58	53	56	95	66	78	95	62	75	95	72	82

"-" means no value is computed, because there is no labeled API misuse in the ground truth dataset or there is no tool-reported misuse.

Table 6: The sampled vulnerabilities and repairs

Security Class API	# of Reports	Vulnerability Detection			Repair Suggestion	
		Basic	Intra-	Inter-	Parameter or API replacement	Code replacement
Cipher	6	5		1	6	
HostnameVerifier	2	2				2
IvParameterSpec	4		1	3		4
KeyPairGenerator	3		1	2	3	
KeyStore	5	1		4		5
MessageDigest	7	5		2	7	
PBEParameterSpec	7	1	4	2	4	3
SecretKeyFactory	4	2		2	4	
SecretKeySpec	11	6	1	4	9	8
SecureRandom	5	5			5	
SSLContext	5	5			5	
TrustManager	12	12				12
Total	71	44	7	20	43	34

all websites are supposed to use HTTPS instead of HTTP for secure communication, so any URL string hardcoded in programs should always start with "`https`" instead of "`http`". In this pattern, the difference between insecure and secure code lies in the string literal, which is not handled by Seader currently. To derive the pattern from given code examples, we need to extend Seader and our specialized ways of example definition, to accurately locate and properly represent any difference within strings.

Finding 2: *On the third-party Apache dataset, Seader outperformed existing tools by achieving the highest precision, recall, and F-score on average. Our experiment indicates Seader's great capability of vulnerability detection.*

4.5 Effectiveness of Repair Suggestion

By applying Seader to the second dataset mentioned in Section 4.1.2, we got hundreds of vulnerabilities reported together with repair suggestions. Due to the time limit, we did not check every vulnerability as well as their repair(s); instead, we manually sampled the vulnerability reports and suggested repairs for 71 misuse instances.

To ensure the representativeness of our manual inspection results, we took four steps to create the sample set. First, we clustered all bug reports based on API-misuse patterns. Second, we randomly picked 10 reports from each cluster for further checking. If any cluster contained nine or fewer reports, we picked all reports. Third, when bug reports referred to duplicated code snippets, we removed duplicates to simplify our manual task, getting 71 sampled vulnerabilities. Fourth, we mapped the 71 samples to the security classes that they correspond to, and created Table 6. Notice that because some security classes (e.g., SecretKeySpec and TrustManager) have multiple method APIs that are prone to misuses (i.e., have multiple API-misuse patterns), the corresponding rows contain more than 10 samples (e.g., 11 for SecurityKeySpec and 12 for TrustManager).

After the first author created the sample set, both the first and fifth authors independently checked bug reports and fixing suggestions. The two authors compared their manual inspection results for cross-checking; they had extensive discussion for any opinion divergence and even involved the second author into discussion, until reaching a consensus.

Among the examined vulnerabilities, Seader revealed 44 misuses without doing any backward slicing (see the **Basic** column); it successfully matched templates with single Java statements. Seader revealed seven misuses via intra-procedural slicing because in each of these scenarios, Seader located and analyzed multiple statements to match the related template. Seader revealed 20 misuses via inter-procedural slicing, because multiple statements from different Java entities (i.e., methods or fields) demonstrate each of the misuses.

70 of the vulnerabilities are true positives; the remaining one was falsely reported. This is because during its analysis, Seader checks whether the second parameter of `KeyStore.load(InputStream stream, char[] password)` is derived from a hardcoded constant; if so, the API call is considered insecure. Such analysis logic can effectively identify any password derived from a hardcoded secret. However, in our experiment, it incorrectly reported a scenario where the password is loaded from a file, whose name is hardcoded as a string literal. In the future, we will overcome this limitation by implementing heuristics (e.g., regular expressions) in Seader, to differentiate between constants serving for distinct purposes.

Additionally, among the 77 suggested repairs, 43 repairs are solely about parameter/API replacement; Seader does not need to generate any code or customize any identifier to propose these fixes. Meanwhile, 34 repairs involve both multi-statement fixes and identifier customization. Notice that the total number of repair suggestions (i.e., 77) is larger than the vulnerability count (i.e., 71). This is because Seader provides multiple suggestions for six vulnerabilities, as each of the code snippets matches two templates simultaneously and Seader suggests a repair for each match. Finally, we found all repairs by Seader to be correct. However, as Seader has a false positive when reporting API misuses, we count the repairs for the other 70 misuses as correct suggestions.

Finding 3: *We manually checked 77 fixes generated by Seader, and found 76 of them to be correct. It indicates that Seader has great capability of repair suggestion.*

5 RELATED WORK

The related work of our research includes automatic detection of security-API misuses, and example-based program transformation.

5.1 Detection of Security-API Misuses

Tools were built to detect security-API misuses [9, 11, 15, 16, 18, 22, 26, 27, 29, 30, 40]. As shown in Table 1, most tools statically analyze programs based on hardcoded or built-in rules. Specifically, CryptoLint hardcoded six API misuse patterns. For each located potentially vulnerable API call (e.g., `Cipher.getInstance(v)`), CryptoLint conducts backward slicing to decide whether the used parameter value is insecure (e.g., `v="AES/ECB"`). CDRep reimplements the design of CryptoLint for misuse detection. It also repairs detected misuses leveraging manually created patch templates. Such tools are not easy to extend, because tool builders or users have to modify tool implementation to expand the rulesets of vulnerabilities.

Fischer et al. [18] built a tool to detect misuses in two ways: machine learning and graph matching. Both methods detect vulnerabilities based on the similarity between given programs and labeled (in)secure code. However, this tool does not rigorously reason about misuse patterns; it cannot pinpoint the exact location of misused API in vulnerable code. CogniCrypt [26] supports rule definition via a domain-specific language CrySL [27]. Each CrySL rule specifies **correct** API usage, and CogniCrypt detects misuses by scanning programs for rule violation. CogniCrypt has three limitations. First, manually prescribing rules with CrySL can be tedious and error-prone for tool users. Second, CrySL cannot express the API misuses related to constant placeholders and constants within certain ranges. Third, CogniCrypt does not customize fixes.

VuRLE [30] is most relevant to our work. VuRLE also detects and repairs vulnerabilities based on the $\langle I, S\rangle$ code examples provided by users. Seader complements VuRLE in three ways. First, Seader infers each pattern from one instead of multiple code pairs, so it works well when users have only one pair. Second, Seader conducts inter-procedural analysis and adopts succinct info (security APIs and data dependencies) to match code with templates, while VuRLE uses intra-procedural analysis and tree matching. Thus, Seader can find more matches. Listing 1 is an exemplar program, where the API misuse is only identifiable when a tool conducts inter-procedural program analysis. VuRLE cannot locate the API misuse but Seader can. Third, Seader supports specialized example definitions, and VuRLE does not. As the source code of VuRLE is not publicly available, we cannot empirically compare it with Seader.

5.2 Example-Based Program Transformation

Based on the insight that developers modify similar code in similar ways, researchers built tools to infer program transformations from exemplar code change examples, and to manipulate code or suggest changes accordingly [13, 34, 35, 37, 39, 41, 44]. For instance, given one or multiple code change examples, LASE [35] and REFAZER [41] infer a program transformation from the examples; they then use the transformation to locate similar code to edit, and apply customized transformations to those locations. Given a set of vulnerable and patched code fragments $K = \{(A_1, A_1'), (A_2, A_2'), \ldots, (A_n, A_n')\}$, SecureSync [39] scans the source code of programs to find fragments, which are similar to vulnerable code A_i but dissimilar to the patched code $A_i'(i \in [1, n])$.

Sharing the same insight, we designed Seader to detect and fix vulnerabilities based on code examples. However, Seader is different from prior work for two reasons. First, Seader infers a program transformation via *intra-procedural* analysis, but conducts pattern matching via *inter-procedural* analysis. All the tools mentioned above are limited to intra-procedural analysis. Our unique design makes Seader more powerful when it searches for pattern matches; it can find matches that go beyond the method boundary and span multiple Java methods. Second, Seader supports three specialized ways of example specification, which can describe transformations that are not expressible via plain code examples. Based on our experience with security-API misuses, these unique specification methods are necessary and helpful.

6 THREATS TO VALIDITY

All inferred patterns and detected vulnerabilities are limited to our experiment datasets and two cryptographic libraries: JCA and JSSE. The observations may not generalize well to other subject programs (e.g., closed-source projects) or other security libraries. We actually also manually checked API misuses in Spring Security—a widely-used third-party security framework, and found more misuse patterns that can be handled by Seader (e.g., the parameter value of BCryptPasswordEncoder's constructor should not be less than 10). Seader can handle the API misuses that involve (1) calling certain method APIs with incorrect parameter values, (2) calling certain method APIs in incorrect sequential orders, and (3) incorrectly overriding certain method APIs. Therefore, Seader is generalizable in terms of (1) the API misuse patterns to handle, and (2) security libraries to cover.

In some repair suggestions provided by Seader, there are placeholders that we need developers to further customize (see "`//Please change 'example.com' as needed`" in Figure 7). Such placeholders should be filled based on developers' software environments, or even require extra configurations outside the codebase (e.g., generating and loading SSL certificates). In the future, we will provide clearer suggestions on hands-on experience and create interactive tools that guide developers to apply complete repairs step-by-step.

7 CONCLUSION

We created Seader—a new approach to take in ⟨insecure, secure⟩ code examples, infer vulnerability-repair patterns from examples, and apply those patterns for vulnerability detection and repair suggestion. Compared with prior work, Seader offers a more powerful means for security experts to extend the pattern set of API-misuse detectors, and concretizes security expertise as customized fixing edits for developers. Our evaluation shows Seader's great capabilities of pattern inference and application; it detects API misuses and suggests fixes with high accuracy. In the future, we will widen Seader's applicability by specifying more code pairs. We will also extend Seader's capability by adding support for more kinds of API misuse patterns (e.g., patterns involving Java annotations).

ACKNOWLEDGMENTS

We thank anonymous reviewers and Dr. Eric Bodden for their valuable comments. This work was supported by NSF-1845446 and NSF-1929701.

REFERENCES

[1] 2016. SLOTH: TLS 1.2 vulnerability (CVE-2015-7575). https://access.redhat.com/articles/2112261.

[2] 2017. Developers lack skills needed for secure DevOps, survey shows. https://www.computerweekly.com/news/450424614/Developers-lack-skills-needed-for-secure-DevOps-survey-shows.

[3] 2019. 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/.

[4] 2020. GitHub. https://github.com.

[5] 2020. Java Secure Socket Extension (JSSE) Reference Guide. https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm.

[6] 2020. StackOverflow. https://stackoverflow.com.

[7] 2020. WALA IR. https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR).

[8] 2021. ApacheCryptoAPI-Bench. https://github.com/CryptoAPI-Bench/ApacheCryptoAPI-Bench/tree/main/apache_codes.

[9] 2021. Find Security Bugs. https://find-sec-bugs.github.io/

[10] 2021. Maven. https://maven.apache.org.

[11] 2021. SonarQube. https://github.com/SonarSource/sonarqube.

[12] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, Danfeng, and Yao. 2021. Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks. arXiv:2112.04037 [cs.CR]

[13] Kijin An, Na Meng, and Eli Tilevich. 2018. Automatic Inference of Java-to-Swift Translation Rules for Porting Mobile Applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 180–190. https://doi.org/10.1145/3197231.3197240

[14] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. 2019. How Reliable is the Crowdsourced Knowledge of Security Implementation?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 536–547. https://doi.org/10.1109/ICSE.2019.00065

[15] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (Berlin, Germany) *(CCS '13)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/2508859.2516693

[16] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. 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* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/2382196.2382205

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. https://doi.org/10.1145/2642937.2642982

[18] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy amp;Paste on Android Application Security. In *2017 IEEE Symposium on Security and Privacy (SP)*. 121–136. https://doi.org/10.1109/SP.2017.31

[19] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. https://doi.org/10.1109/TSE.2007.70731

[20] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. 2012. 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* (Raleigh, North Carolina, USA) *(CCS '12)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2382196.2382204

[21] Matthew Green and Matthew Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy* 14, 5 (2016), 40–46. https://doi.org/10.1109/MSP.2016.111

[22] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V.N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. 2015. Vetting SSL Usage in Applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy*. 519–534. https://doi.org/10.1109/SP.2015.38

[23] Roya Hosseini and Peter Brusilovsky. 2013. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, Vol. 1009. University of Pittsburgh, 60–63.

[24] Java Cryptography Architecture 2021. Java Cryptography Architecture. https://docs.oracle.com/javase/9/security/java-cryptography-architecture-jca-reference-guide.htm.

[25] B. Kaliski. 2000. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational). http://www.ietf.org/rfc/rfc2898.txt

[26] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, and Ram Kamath. 2017. CogniCrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 931–936. https://doi.org/10.1109/ASE.2017.8115707

[27] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2021. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. *IEEE Transactions on Software Engineering* 47, 11, 2382–2400. https://doi.org/10.1109/TSE.2019.2948910

[28] VI Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707.

[29] Siqi Ma, David Lo, Teng Li, and Robert H Deng. 2016. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 711?722. https://doi.org/10.1145/2897845.2897896

[30] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. 229–246. https://doi.org/10.1007/978-3-319-66399-9_13

[31] James Manger. 2001. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS #1 v2.0. In *Advances in Cryptology — CRYPTO 2001*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–238.

[32] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. 2019. Beyond GumTree: A Hybrid Approach to Generate Edit Scripts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 550–554. https://doi.org/10.1109/MSR.2019.00082

[33] Florian Mendel, Tomislav Nad, and Martin Schläffer. 2013. Improving Local Collisions: New Attacks on Reduced SHA-256. In *Advances in Cryptology – EUROCRYPT 2013*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–278.

[34] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2011. Systematic Editing: Generating Program Transformations from an Example. *SIGPLAN Not.* 46, 6, 329–342. https://doi.org/10.1145/1993316.1993537

[35] Na Meng, Miryung Kim, and Kathryn S. McKinley. 2013. Lase: Locating and applying systematic edits by learning from examples. In *2013 35th International Conference on Software Engineering (ICSE)*. 502–511. https://doi.org/10.1109/ICSE.2013.6606596

[36] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. 2018. Secure Coding Practices in Java: Challenges and Vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 372–383. https://doi.org/10.1145/3180155.3180201

[37] Robert C. Miller and Brad A. Myers. 2001. Interactive Simultaneous Editing of Multiple Text Regions. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 161–174.

[38] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE)*. ACM, New York, NY, USA, 935–946. https://doi.org/10.1145/2884781.2884790

[39] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2010. Detection of Recurring Software Vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 447–456. https://doi.org/10.1145/1858996.1859089

[40] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 2019. 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* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2455–2472. https://doi.org/10.1145/3319535.3345659

[41] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 404–415. https://doi.org/10.1109/ICSE.2017.44

[42] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. 2015. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525. https://doi.org/10.17487/RFC7525

[43] Harshal Tupsamudre, Monika Sahu, Kumar Vidhani, and Sachin Lodha. 2020. Fixing the Fixes: Assessing the Solutions of SAST Tools for Securing Password Storage. In *Financial Cryptography and Data Security: FC 2020 International Workshops, AsiaUSEC, CoDeFi, VOTING, and WTSC, Kota Kinabalu, Malaysia, February 14, 2020, Revised Selected Papers* (Kota Kinabalu, Malaysia). Springer-Verlag, Berlin, Heidelberg, 192?206. https://doi.org/10.1007/978-3-030-54455-3_14

[44] Shengzhe Xu, Ziqi Dong, and Na Meng. 2019. Meditor: Inference and Application of API Migration Edits. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 335–346. https://doi.org/10.1109/ICPC.2019.00052