VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples

Siqi Ma[1(✉)], Ferdian Thung[1], David Lo[1], Cong Sun[2], and Robert H. Deng[1]

[1] Singapore Management University, Singapore, Singapore
{siqi.ma.2013,ferdiant.2013,davidlo,robertdeng}@smu.edu.sg
[2] Xidian University, Xi'an, China
suncong@xidian.edu.cn

Abstract. Vulnerability becomes a major threat to the security of many systems. Attackers can steal private information and perform harmful actions by exploiting unpatched vulnerabilities. Vulnerabilities often remain undetected for a long time as they may not affect typical systems' functionalities. Furthermore, it is often difficult for a developer to fix a vulnerability correctly if he/she is not a security expert. To assist developers to deal with multiple types of vulnerabilities, we propose a new tool, called VuRLE, for automatic detection and repair of vulnerabilities. VuRLE (1) learns transformative edits and their contexts (i.e., code characterizing edit locations) from examples of vulnerable codes and their corresponding repaired codes; (2) clusters similar transformative edits; (3) extracts edit patterns and context patterns to create several repair templates for each cluster. VuRLE uses the context patterns to detect vulnerabilities, and customizes the corresponding edit patterns to repair them. We evaluate VuRLE on 279 vulnerabilities from 48 real-world applications. Under 10-fold cross validation, we compare VuRLE with another automatic repair tool, LASE. Our experiment shows that VuRLE successfully detects 183 out of 279 vulnerabilities, and repairs 101 of them, while LASE can only detect 58 vulnerabilities and repair 21 of them.

Keywords: Automated template generation · Vulnerability detection · Automated program repair

1 Introduction

Vulnerability is a severe threat to computer systems. However, it is difficult for a developer to detect and repair a vulnerability if he/she is not a security expert. Several vulnerability detection tools have been proposed to help developers to detect and repair different kinds of vulnerabilities, such as cryptographic misuse [17], cross-site scripting (XSS) [26], component hijacking vulnerability [28], etc.

Prior studies on automatic vulnerability repair typically focus on one type of vulnerability. These studies require custom manually-generated templates or custom heuristics tailored for a particular vulnerability. CDRep [17] is a tool to repair cryptographic vulnerabilities. It detects and repairs the misuses of cryptographic algorithms. By *manually* summarizing repair patterns from correct primitive implementations, repair templates are generated. Similar to CDRep, Wang et al. [26] proposed an approach to repair string vulnerabilities in web applications. They *manually* construct input string patterns and attack patterns. Based on the input-attack patterns, the tool can compute a safe input, and convert a malicious input into a safe input. AppSealer [28] defines *manually* crafted rules for different types of data to repair component hijacking vulnerabilities by using taint analysis. By applying dataflow analysis, it can identify tainted variables, and further repair those variables based on the defined rules.

Manually generating repair templates and defining repair rules are tedious and time consuming activities. As technology and computer systems advance, different vulnerabilities may occur and fixing each of them likely requires different repair patterns. Unfortunately, it is very expensive or even impractical to manually create specific templates or rules for all kinds of vulnerabilities. The above facts highlight the importance of developing techniques that can generate repair templates automatically.

To help developers repair common bugs, Meng et al. [20] proposed LASE that can automatically generate a repair template. LASE automatically learns an edit script from two or more repair examples. However, LASE's inference process has two major limitations. First, it can only generate a general template for a type of bug. However, a bug can be repaired in different ways based on the *context* (i.e., preceding code where a bug appears in). Second, it cannot learn multiple repair templates from a repair example that involves repair multiple bugs.

To address the above limitations, we design and implement a novel tool, called VuRLE (Vulnerability Repair by Learning from Examples), that can help developers automatically detect and repair multiple types of vulnerabilities. VuRLE works as follows:

1. VuRLE analyzes a training set of repair examples and identifies *edit blocks* – each being series of related edits and its context from each example. Each example contains a vulnerable code and its repaired code.
2. VuRLE clusters similar edit blocks in to groups.
3. Next, VuRLE generates several repair templates for each group from pairs of highly similar edits.
4. VuRLE then uses the repair templates to identify vulnerable code.
5. VuRLE eventually selects a suitable repair template and applies the transformative edits in the template to repair a vulnerable code.

VuRLE addresses the first limitation of LASE by generating many repair templates instead of only one. These templates are put into groups and are used collectively to accurately identify vulnerabilities. VuRLE also employs a heuristics that identifies the most appropriate template for a detected vulnerability. It addresses the second limitation by breaking a repair example into several code segments. It then extracts an edit block from each of the code segment. These edit blocks may cover different bugs and can be used to generate different repair templates. This will result in many edit blocks though, and many of which may not be useful in the identification and fixing of vulnerabilities. To deal with this issue, VuRLE employs a heuristics to identify suitable edit blocks that can be generalized into repair templates.

We evaluate VuRLE on 279 vulnerabilities from 48 real-world applications using 10-fold cross validation setting. In this experiment, VuRLE successfully detects 183 (65.59%) out of 279 vulnerabilities, and repairs 101 of them. This is a major improvement when compared to LASE, as it can only detects 58 (20.79%) out of the 279 vulnerabilities, and repairs 21 of them.

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Section 3 elaborates the learning phase of our approach and Sect. 4 presents the repair phase of our approach. Experimental results are presented in Sect. 5. Related work is presented in Sect. 6. Section 7 concludes the paper.

2 Overview of VuRLE

In this section, we introduce how VuRLE repairs vulnerabilities. Figure 1 shows the workflow of VuRLE. VuRLE contains two phases, **Learning Phase** and **Repair Phase**. We provide an overview of working details of each phase below.

Learning Phase. VuRLE generates templates by analyzing edits from repair examples in three steps (Step 1–3).

1. **Edit Block Extraction.** VuRLE first extracts *edit blocks* by performing Abstract Syntax Tree (AST) *diff* [8] of each vulnerable code and its repaired code in a training set of known repair examples.

   The difference between a pair of vulnerable and repaired code may be in several code segments (i.e., contiguous lines of code). For each pair of vulnerable and repaired code segments, VuRLE outputs an edit block which consists of two parts: (1) a sequence of *edit operations*, and (2) its *context*. The first specifies a sequence of AST node insertion, deletion, update, and move operations to transform the vulnerable code segment to the repaired code segment. The latter specifies a common AST subtree corresponding to code appearing before the two code segments.
2. **Edit Group Generation.** VuRLE compares each edit block with the other edit blocks, and produces groups of similar edit blocks.

   VuRLE creates these *edit groups* in several steps. First, it creates a graph where each edit block is a node, and edges are added between two edit blocks iff they share the longest common substring [11] of edit operations with a substantial size. Next, it extracts connected components [12] from these graphs. Finally, it applies a DBSCAN [5]-inspired clustering algorithm, to divide edit blocks in each connected component into edit groups.
3. **Repair Template Generation.** In each edit group, VuRLE generates a *repair template* for each pair of edit blocks that are adjacent to each other in the connected component (generated as part of Step 2).

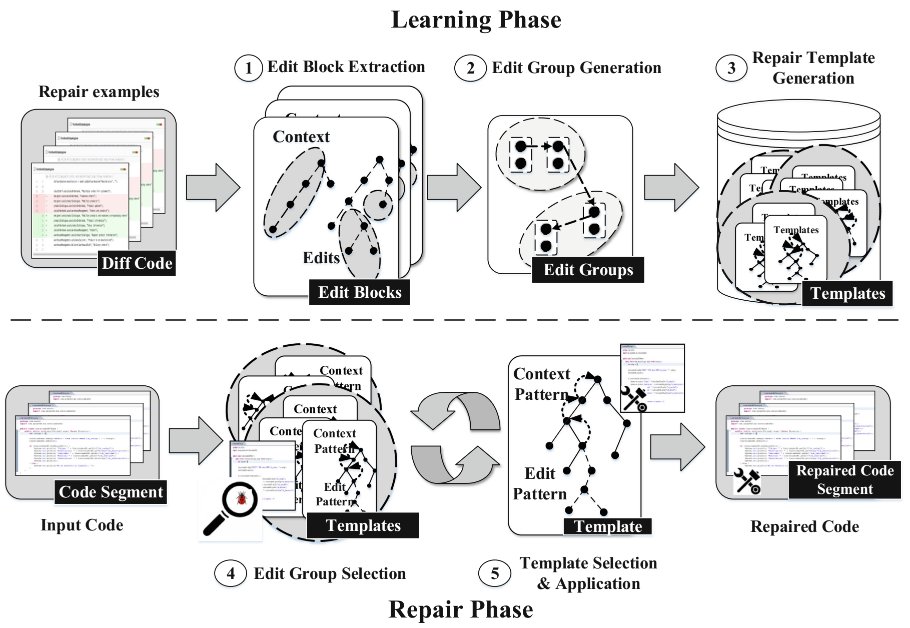


Fig. 1. Workflow of VuRLE: (1) VuRLE generates an edit block by extracting a sequence of edit operations and its context. (2) VuRLE pairs the edit blocks and clusters them into edit groups (3) VuRLE generates repair templates, and each contains an edit pattern and a context pattern. (4) VuRLE selects the best matching edit group to detect for vulnerabilities (5) VuRLE selects and applies the most appropriate repair template within the selected group.

Each repair template has an *edit pattern* and a *context pattern*. An edit pattern specifies a sequence of transformative edits, while a context pattern specifies the location of the code where the transformative edits should be applied. To create the edit pattern, VuRLE identifies the longest common substring of edit operations in the two edit blocks. To create the context pattern, VuRLE compares the code appearing in the context part of the two edit blocks. To generalize the patterns, VuRLE abstracts concrete identifier names and types appearing in the patterns into *placeholders*.

The context pattern is used to identify vulnerable code, while the edit pattern is used to repair identified vulnerabilities in the repair phase.

Repair Phase. VuRLE detects and repairs vulnerabilities by selecting the most appropriate template in two steps (Step 4–5). These two steps are repeated a number of times until no more vulnerable code segments are detected.

4. **Edit Groups Selection.** Given an input code and a set of repair templates, VuRLE compares code segments of the input code with edit groups and identifies an edit group that best matches it.

5. **Template Selection and Application.** The most matched edit group may have multiple templates that match an input code segment. VuRLE enumerates the matched templates one by one, and applies the transformative edits specified in the edit pattern of the template. If the application of the transformative edits results in redundant code, VuRLE proceeds to try the next template. Otherwise, it will flag the code segment as a vulnerability and generates a repaired code segment by applying the transformative edits.

3 Learning Phase: Learning from Repair Examples

In this phase, VuRLE processes a set of vulnerability repair examples to produce groups of similar repair templates. The three steps involved in this phase (Edit Block Extraction, Edit Block Group Extraction, and Repair Template Generation) are presented in more details below.

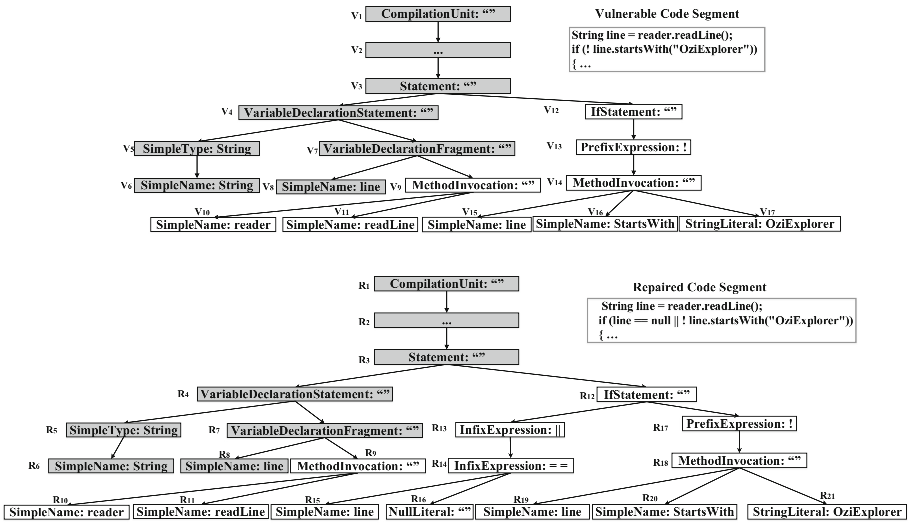
3.1 Edit Block Extraction

For each repair example, VuRLE uses Falleri et al.'s GumTree [7] to compare the AST of a vulnerable code and its repaired code. Each node in an AST corresponding to a source code file can be represented by a 2-tuple: (*Type*, *Value*). The first part of the tuple indicates the type of the node, e.g., VariableDeclarationStatement, SimpleType, SimpleName, etc. The second indicates the concrete value stored in the node, e.g., String, readLine, "OziExplorer", etc.

Using GumTree, VuRLE produces for each repair example a set of edit blocks, each corresponds to a specific code segment in the AST diff between a vulnerable code and its repaired code. Each edit block consists of a sequence of edit operations, and its context. The sequence can include one of the following edit operations:

- **Insert**(**Node** u, **Node** p, **int** k): Insert node u as the k^{th} child of parent node p.
- **Delete**(**Node** u, **Node** p, **int** k): Delete node u, which is the k^{th} child of parent node p.
- **Update**(**Node** u, **Value** v): Update the old value of node u to the new value v.
- **Move** (**Node** u, **Node** p, **int** k): Move node u and make it the k^{th} child of parent p. Note that all children of u are moved as well, therefore this moves a whole subtree.

For each sequence of edit operations, VuRLE also identifies its context. To identify this context, VuRLE uses GumTree to extract an AST subtree that appears in both vulnerable and repaired ASTs and is relevant to nodes affected by the edit operations. This subtree is the largest common subtree where each of its leaf nodes is a node with SimpleName type that specifies a variable that is used in the sequence of edit operations. We make use of the `getParents` method of GumTree to find this subtree.



```
String line = reader.readLine();
if (! line.startsWith("OziExplorer"))
{ ...
```

```
String line = reader.readLine();
if (line == null || ! line.startsWith("OziExplorer"))
{ ...
```

Fig. 2. Vulnerable and repaired code segments and their ASTs

To illustrate the above, consider Fig. 2. It shows the ASTs of a vulnerable code segment and its corresponding repaired code segment. Performing AST diff on these two ASTs produces a sequence of edit operations which results in the deletion of nodes V_{12} to V_{17}, and the insertion of nodes R_{12} to R_{21} into the subtree rooted at V_3. It also produces a context which corresponds to the common AST subtree highlighted in gray.

3.2 Edit Group Generation

VuRLE generates edit groups in two steps: (1) edit graph construction; (2) edit block clustering. We describe these two steps in detail below.

Edit Graph Construction. VuRLE creates a graph, whose nodes are edit blocks extracted in the previous step. The edges in this graph connect similar edit blocks. Two edit blocks are deemed similar iff their edit operations are similar. To check for this similarity, VuRLE extracts the longest common substring (LCS) [11] from their edit operation sequences. The two edit blocks are then considered similar if the length of this LCS is larger than a certain threshold T_{Sim}. Each edge is also weighted by the reciprocal of the corresponding LCS length. This weight represents the distance between the two edit blocks. We denote the distance between two edit blocks e_1 and e_2 as $dist(e_1, e_2)$.

Edit Block Clustering. Given an edit graph, VuRLE first extracts connected components [12] from it. For every connected component, VuRLE clusters edit blocks appearing in it.

To cluster edit blocks in a connected component (CC), VuRLE follows a DBscan-inspired clustering algorithm. It takes in two parameters: ε (maximum cluster radius) and ρ (minimum cluster size). Based on these two parameters, VuRLE returns the following edit groups (EGS):

$$EGS(CC) = \{N_\varepsilon(e_i) \mid e_i \in CC \wedge |N_\varepsilon(e_i)| \geq \rho\} \tag{1}$$

In the above equation, $N_\varepsilon(e_i)$ represents a set of edit blocks in CC whose distance to e_i is at most ε. Formally, it is defined as:

$$N_\varepsilon(e_i) = \{e_j \in CC \mid dist(e_i, e_j) \leq \varepsilon\} \tag{2}$$

The value of ρ is set to be 2 to avoid generating groups consisting of only one edit block. The value of ε is decided by following Kreutzer et al.'s code clustering method [14]. Their heuristic has been shown to work well in their experiments. The detailed steps are as follows:

1. Given an edit graph, VuRLE first computes the distance between each connected edit block. Two edit blocks that are not connected in the edit graph has an infinite distance between them.
2. VuRLE then orders the distances in ascending order. Let $\langle d_1, d_2, \ldots, d_n\rangle$ be the ordered sequence of those distances.
3. VuRLE finally sets the value of ε by finding the largest gap between two consecutive distances $d_{\langle j+1\rangle}$ and $d_{\langle j\rangle}$ in the ordered sequence. Formally, ε is set as $\varepsilon = d_{\langle j^*\rangle}$, where $j^* = argmax_{1\leq j\leq n}(\frac{d\langle j+1\rangle}{d\langle j\rangle})$.

To illustrate the above process, Fig. 3 presents two connected components (CCs), $\{E_1, E_2, E_3, E_5, E_6\}$ and $\{E_0, E_7\}$. VuRLE first orders the distances into [0.12, 0.14, 0.17, 0.25]. It then computes the largest gap between two consecutive

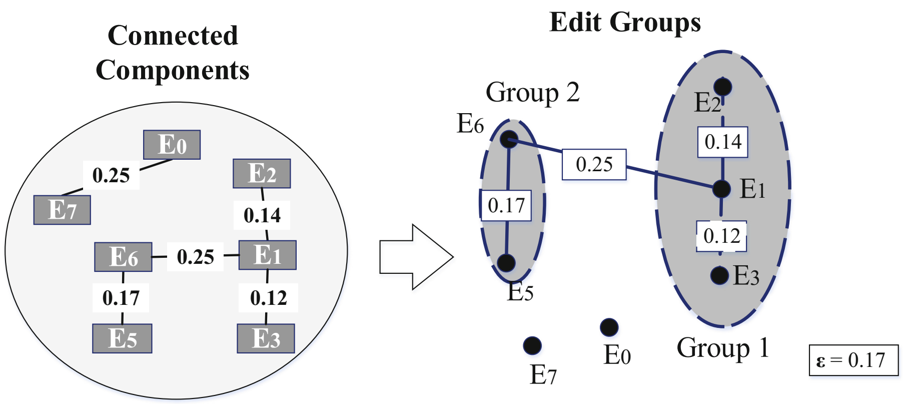


Fig. 3. Edit block clustering: CCs to edit block groups

distances, and identifies a suitable value of ε, which is 0.17. Based on $\varepsilon = 0.17$ and $\rho = 2$, VuRLE creates two groups of edit blocks for the first CC: $\{E_1, E_2, E_3\}$, and $\{E_5, E_6\}$. It generates none for the second CC.

3.3 Templates Generation

For each edit group, VuRLE identifies pairs of edit blocks in it that are adjacent nodes in the edit graph. For each of these *edit pairs*, it creates a repair template. A repair template consists of an edit pattern, which specifies a sequence of transformative edits, and a context pattern, which specifies where the edits should be applied.

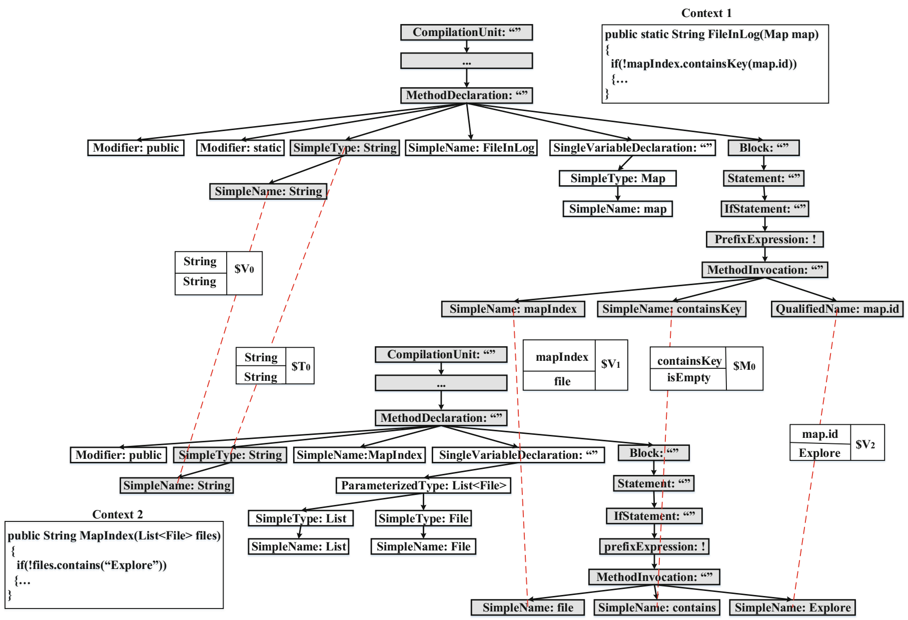
To create an edit pattern from a pair of edit blocks, VuRLE compares the edit operation sequences of the two edit blocks. It then extracts the longest common substring (LCS) from the two sequences. This LCS is the edit pattern.

To create a context pattern from a pair of edit blocks, VuRLE processes the context of each edit block. Each context is a subtree. Given a pair of edit block contexts (which is a pair of AST subtrees, ST_1 and ST_2), VuRLE proceeds in the following steps:

1. VuRLE performs pre-order traversal on ST_1 and ST_2.
2. For each subtree, it extracts an ordered set of paths from the root of the subtree to each of its leaf nodes. The two ordered sets PS_1 and PS_2 represent the context of ST_1 and ST_2 respectively. We refer to each of these paths as a *concrete* context sequence.
3. VuRLE then compares the corresponding elements of PS_1 and PS_2. For each pair of paths, if they share a longest common substring (LCS) of size T_{Sim}, we use this LCS to represent both pairs and delete the paths from PS_1 and PS_2. We refer to this LCS as an *abstract* context sequence.
4. VuRLE uses the remaining concrete sequences and identified abstract sequences as the context pattern.

As a final step, for each template, VuRLE replaces all concrete identifier types and names with placeholders. All occurrences of the same identifier type or name will be replaced by the same placeholder.

Figure 4 illustrates how VuRLE generates a context pattern by comparing two contexts. VuRLE performs pre-order traversal on AST subtrees of context 1 and context 2, generating an ordered set of paths for each context. After comparing the two set, VuRLE finds the matching paths highlighted in gray. For each matching pair of nodes that is of type SimpleName or SimpleType, VuRLE creates placeholders for it. There are five matching pair of nodes fulfilling this criteria, which are indicated by the dashed lines. Thus, VuRLE creates five placeholders named $\$V_0$, $\$V_1$, $\$V_2$, $\$T_0$, and $\$M_0$ from them.



Context 1
```
public static String FileInLog(Map map)
{
  if(!mapIndex.containsKey(map.id))
  {...
}
```

Context 2
```
public String MapIndex(List<File> files)
{
  if(!files.contains(“Explore”))
  {...
}
```

Fig. 4. Context pattern generation

4 Repair Phase: Repairing Vulnerable Applications

In this phase, VuRLE uses repair templates generated in the learning phase to detect whether an input code is vulnerable and simultaneously applies appropriate edits to repair the vulnerability. The two steps involved in this phase (Edit Group Selection and Template Selection) are presented in more details below. They are performed iteratively until VuRLE can no longer detect any vulnerability.

4.1 Edit Group Selection

To detect whether an input code is vulnerable, VuRLE needs to find the edit group with the highest *matching score*. VuRLE compares the input code (IC) with each edit group (EG) and computes the matching score as follows:

$$S_{matching}(IC, EG) = \sum_{T \in templates(EG)} S_{matching}(IC, T) \tag{3}$$

In the above equation, $templates(EG)$ is the set of templates corresponding to edit group EG, and $S_{matching}(IC, T)$ is the matching score between template T and IC. VuRLE computes the matching score between the template T and input code IC as follows:

1. VuRLE first generates an AST of the input code.
2. VuRLE performs pre-order traversal on this AST to produce an ordered set of paths. Each path is a sequence of AST nodes from the root of the AST to one of its leaf node. Let us denote this as IP.
3. VuRLE compares IP with the context of template T. If sequences in T can be matched with sequences in IP, the number of matching nodes is returned as a matching score. Abstract sequences need to be fully matched, while concrete sequences only need to be partially matched. Otherwise, the matching score is 0.

4.2 Template Selection

In the most matched edit group EG, there are likely to be multiple corresponding templates (i.e., $templates(EG)$ has more than one member). In this final step, we need to pick the most suitable template.

To find a suitable template, VuRLE tries to apply templates in $templates(EG)$ one-by-one according to their matching scores. To apply a template, VuRLE first finds a code segment whose context matches with the context of the template. It then replaces all placeholders in the template with concrete variable names and types that appear in the context of the code segment. Next, VuRLE applies each transformative edits specified in the edit operation sequence of the template to the code segment.

If the application of a template results in redundant code, VuRLE proceeds to try the next template. The template selection step ends when one of the templates can be applied without creating redundant code. The code segment where the template is applied to is marked as being vulnerable and the resultant code after the transformative edits in the template is applied is the corresponding repaired code.

5 Evaluation

This section evaluates the performance of VuRLE by answering two questions below:

RQ1 (Vulnerability Detection). How effective is VuRLE in detecting whether a code is vulnerable?

RQ2 (Vulnerability Repair). How effective is VuRLE in repairing the detected vulnerable codes? Why some vulnerable codes cannot be repaired by VuRLE?

The following sections first describe the settings of our experiments, followed by the results of the experiments which answer the above two questions.

5.1 Experiment Setup

Dataset. We collect 48 applications written in Java from GitHub[1] that have more than 400 stars. These applications consist of Android, web, word-processing and multimedia applications. The size of Android applications range from 3–70 MB while the size of other applications are about 200 MB. Among these applications, we identify vulnerabilities that affects them by manually analyzing commits from each application's repository. In total, we find 279 vulnerabilities. These vulnerabilities belong to several vulnerable types listed in Table 1.

Table 1. Types of vulnerabilities in our dataset

Vulnerability type	Description
Unreleased resource	Failing to release a resource [19] before reusing it. It increases a system's susceptibility to Denial of Service (DoS) attack
Cryptographic vulnerability	Inappropriate usage of encryption algorithm [4,17] or usage of plaintext password storage. It increases a system's susceptibility to Chosen-Plaintext Attack (CPA), brute force attack, etc.
Unchecked return value	Ignoring a method's return value. It may cause an unexpected state and program logic, and possibly a privilege escalation bug
Improper error handling	Showing an inappropriate error handling message. It may cause a privacy leakage, which reveals useful information to potential attackers
SSL vulnerability	Unchecked hostnames or certificates [6,10]. It makes a system susceptible to eavesdroppings and Man-In-The-Middle attacks [2]
SQL injection vulnerability	Unchecked input of SQL. It makes a system susceptible to SQL injection attack, which allows attackers to inject or execute SQL command via the input data [16]

Experiment Design. We use 10-fold cross validation to evaluate the performance of VuRLE. First, we split the data into 10 groups (each containing roughly 28 vulnerabilities). Then, one group is defined as a test group, and the other 9 groups as a training group. The test group is the input of the repair phase, while the training group is the input of the learning phase. We repeat the process 10 times by considering different group as test group. We examine the repaired code manually by comparing it with the real repaired code provided by developers. Furthermore, we compare VuRLE with LASE [20], which is state-of-the-art tool for learning repair templates. When running VuRLE, by default we set T_{Sim} to three.

[1] Github: https://github.com/.

To evaluate the vulnerability detection performance of our approach, we use precision and recall as the evaluation metrics, which are defined as follows.

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

where TP is the number of correctly detected vulnerabilities, FP is the number of wrongly detected vulnerabilities, and FN is the number of vulnerabilities that are not detected by our approach.

To evaluate the vulnerability detection performance of our approach, we use success rate as the evaluation metric. Success rate is the proportion of the correctly detected vulnerabilities that can be successfully repaired.

5.2 RQ1: Vulnerability Detection

To answer this RQ, we count the number of vulnerabilities that can be detected by VuRLE and compute the precision and recall on the entire dataset.

Table 2. Detection result: VuRLE vs LASE

	# of Detected vulnerabilities	Precision	Recall
VuRLE	183	64.67%	65.59%
LASE	58	52.73%	20.79%

Table 2 shows the number of detected vulnerabilities, precision, and recall of VuRLE and LASE. VuRLE successfully detects 194 vulnerabilities out of the 279 vulnerabilities, achieving a recall of 65.59%. On the other hand, LASE can only detect 58 vulnerabilities out of the 279 vulnerabilities, achieving a recall of only 20.79%. Thus, VuRLE detects 215.52% more vulnerabilities compared to LASE. In terms of precision, VuRLE improves over LASE by 22.64%. It means that VuRLE proportionally generates less false positives than LASE.

5.3 RQ2: Vulnerability Repair

To answer this RQ, we investigate the number of vulnerabilities that can be repaired successfully. We also investigate how VuRLE can repair some bugs than cannot be repaired by LASE. We also discuss some causes on why VuRLE cannot repair some bugs.

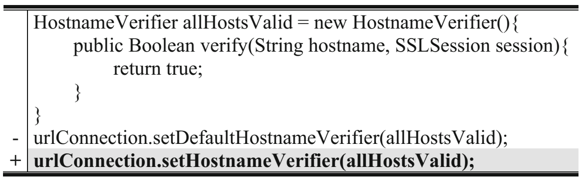
Table 3 presents the success rate of VuRLE and LASE. The success rate of VuRLE is much higher than the success rate of LASE. VuRLE successfully

Table 3. Vulnerability repair: VuRLE and LASE

	# of repaired vulnerabilities	Success rate
VuRLE	101	55.19%
LASE	21	36.21%

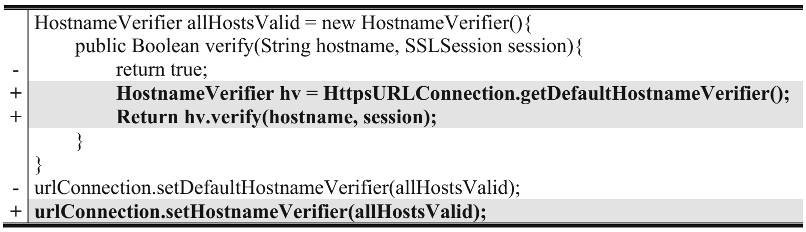
repairs 101 vulnerabilities (55.19%), and LASE can only repairs 21 vulnerabilities, with a success rate of 36.21%. Thus, VuRLE can repair 380.95% more vulnerabilities compared to LASE. In terms of success rate, it improves over LASE by 52.42%.

Figure 5 provides a repair example generated by LASE and VuRLE on the same input code. The piece of code in the example contains a vulnerability that allows any hostname to be valid. LASE generates an overly general repair template, which only include invocation to `setHostnameVerifier`. It generate such template since each repair example invokes the `setHostNameVerifier` method after they define the `setDefaultHostnameVerifier` method, but the definition of the verifier method itself is different. On the other hand, VuRLE generates two repair templates that can repair this vulnerability. One of the template is for modifying the `verify` method, and another is for invoking the `setDefaultHostnameVerifier` method.



```
  HostnameVerifier allHostsValid = new HostnameVerifier(){
      public Boolean verify(String hostname, SSLSession session){
          return true;
      }
  }
- urlConnection.setDefaultHostnameVerifier(allHostsValid);
+ urlConnection.setHostnameVerifier(allHostsValid);
```

(a) Patch Generated by LASE



```
  HostnameVerifier allHostsValid = new HostnameVerifier(){
      public Boolean verify(String hostname, SSLSession session){
-         return true;
+         HostnameVerifier hv = HttpsURLConnection.getDefaultHostnameVerifier();
+         Return hv.verify(hostname, session);
      }
  }
- urlConnection.setDefaultHostnameVerifier(allHostsValid);
+ urlConnection.setHostnameVerifier(allHostsValid);
```

(b) Patch Generated by VuRLE

Fig. 5. A vulnerability repaired by LASE and VuRLE

Among 183 detected vulnerabilities, VuRLE cannot repair some of them. We discuss the main causes as follows:

Unsuccessful Placeholder Resolution. When replacing placeholders with concrete identifier names and types, VuRLE may use a wrong type or name to fill the placeholders. For example, the required concrete type is "double", but the inferred concrete type is "int". Moreover, VuRLE may not be able to concretize some placeholders since they are not found in the matching contexts.

Lack of Repair Examples. In our dataset, some vulnerabilities, such as *Cryptographic Misuses* and *Unchecked Return Value*, have many examples. Thus, a more comprehensive set of repair templates can be generated for these kinds of vulnerabilities. However, some vulnerabilities, such as *SSL Socket Vulnerability*, only have a few examples. Thus, VuRLE is unable to derive a comprehensive set of repair template to repair these kinds of vulnerabilities.

Partial Repair. For some cases, VuRLE can only generate a partial repair. This may be caused either by the inexistence of similar repairs or because VuRLE only extracts a partial repair pattern.

6 Related Work

This section describes related work on vulnerability detection and automatic vulnerability repair.

Vulnerability Detection. A number of works on detecting software vulnerabilities. Taintscope [26] is a checksum-aware fuzzing tool that is able to detect checksum check points in programs and checksum fields in programs' inputs. Moreover, it is able to automatically create valid input passing the checksum check. TaintScope can detect buffer overflow, integer overflow, double free, null pointer dereference and infinite loop. Sotirov [25] propose a static source analysis technique to detect vulnerabilities, such as buffer overflow, format string bugs and integer overflow. They classify the vulnerabilities and extracted common patterns for each type of vulnerability. Mohammadi et al. [21] focus on the XSS (Cross Site Scripting) vulnerability detection. Instead of analyzing source code, they detect XSS vulnerability that is caused by improper encoding of untrusted input data.

Medeiros et al. [18] propose a combination approach to detect vulnerabilities in web applications. They combine taint analysis and static analysis. Taint analysis is for collecting human coding knowledge about vulnerabilities. Furthermore, they generate several classifiers to label the vulnerable data. These classifiers achieve a low false positive rate. Fu et al. [9] propose a static analysis approach to detect SQL injection vulnerability at compile time. Their approach symbolically executes web applications written in ASP.NET framework and detects an SQL injection vulnerability if it can generates an input string that matches with a certain attack pattern. Kals et al. [13] propose SecuBat, a scanner that can detect vulnerabilites in web applications. SecuBat provides a framework that allows user to add another procedure that can detect a particular vulnerability. Similar like SecuBat, Doupé et al. [3] propose a web vulnerability scanner that is aware of web application state. It infers the web application's state machine by observing effects of user actions on the web application's outputs. This state machine is traversed to discover vulnerabilities. Balduzzi et al. [1] propose an automatic technique to discover HTTP parameter pollution vulnerabilities in web applications. It automatically launches an attack by injecting an encoded parameter into one of the known parameters and discovers a vulnerability if the attack is successful.

Similar like the above works, our work also detects vulnerabilities. However, those works are specialized to detect a certain type of vulnerabilities. On the other hand, our work can detect many types of vulnerabilities, under the condition that examples for repairing the corresponding vulnerabilities can be provided.

Vulnerability Repair. To repair vulnerabilities automatically, it is common to generate a repair pattern for a specific vulnerability. FixMeUp [24] is proposed to repair access-control bugs in web applications. It automatically computes an interprocedural access-control template (ACT), which contains a low-level policy specification and program transformation template. FixMeUp uses ACT to identify a faulty access-control logic and performs the repair. CDRep [17] detects cryptographic misuse in Android applications and repairs vulnerabilities automatically by applying manually-made repair templates. It makes use of seven repair templates, each for a particular type of cryptographic misuse. Yu et al. [27] propose an approach to sanitize user's input in web applications. Given a manually defined input pattern and its corresponding attack pattern, their approach checks whether an input has the same pattern with an input pattern and identifying whether the input is safe from the corresponding attack. If it is not, they convert a malicious input into a benign one. Smirnov and Chiueh [23] proposed DIRA, a tool that can transform program source code to defend against buffer overflow attacks. At running time, the transformed program can detect, identify, and repair itself without terminating its execution. Repair is achieved by restoring programs state to the one before the attack occurs, which was achieved through manually pre-defined procedures. Sidiroglou and Keromytis [22] propose an automated technique to repair buffer overflow vulnerabilities using manually-made code transformation heuristics and testing the repair in a sandboxed environment. Lin et al. [15] propose AutoPAG, a tool that automatically repairs out-of-bound vulnerabilities. It automatically generates a program patch by using pre-defined rules for repairing a particular case of out-of-bound vulnerability.

Different than the above works, our work aims to automatically repair different kinds of vulnerabilities by learning from examples. The closest to our work is LASE [20], which generates a repair template from repair examples. Different than ours, it cannot generate many repair templates from repair examples, which may include fixes for different vulnerabilities. It also cannot generate many sequences of edits when given a repair example, each corresponds to a certain code segment in the example.

7 Conclusion and Future Work

In summary, we propose a tool, called VuRLE, to automatically detect and repair vulnerabilities. It does so by learning repair templates from known repair examples and applying the templates to an input code. Given repair examples, VuRLE extracts edit blocks and groups similar edit blocks into an edit group. Several repair templates are then learned from each edit group. To detect and repair vulnerabilities, VuRLE finds the edit group that matches the most with the input code. In this group, it applies repair templates in order of their matching score until it detects no redundant code (in which case a vulnerability is detected and repaired) or until it has applied all repair templates in the edit group (in which case no vulnerability is detected). VuRLE repeats this detection and repair process until no more vulnerabilities are detected.

We have experimented on 48 applications with 279 real-world vulnerabilities and performed 10-fold cross validation to evaluate VuRLE. On average, VuRLE can automatically detect 183 (65.59%) vulnerabilities and repair 101 (55.19%) of them. On the other hand, the state-of-the-art approach named LASE can only detect 58 (20.79%) vulnerabilities and repair 21 (36.21%) of them. Thus, VuRLE can detect and repair 215.52% and 380.95% more vulnerabilities compared to LASE, respectively.

In the future, we plan to evaluate VuRLE using more vulnerabilities and applications written in various programming languages. We also plan to boost the effectiveness of VuRLE further so that it can detect and repair more vulnerabilities. Additionally, we plan to design a new approach to detect and repair vulnerabilities without examples.

References

1. Balduzzi, M., Gimenez, C.T., Balzarotti, D., Kirda, E.: Automated discovery of parameter pollution vulnerabilities in web applications. In: Network and Distributed System Security Symposium (NDSS) (2011)
2. Conti, M., Dragoni, N., Lesyk, V.: A survey of man in the middle attacks. IEEE Commun. Surv. Tutorials **18**(3), 2027–2051 (2016)
3. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: a state-aware black-box web vulnerability scanner. In: USENIX Security Symposium, vol. 14 (2012)
4. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in android applications. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security, pp. 73–84. ACM (2013)
5. Ester, M., Kriegel, H.P., Sander, J., Xu, X., et al.: A density-based algorithm for discovering clusters in large spatial databases with noise. In: Knowledge Discovery and Data Mining (KDD), vol. 96, no. 34 (1996)
6. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: 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, pp. 50–61. ACM (2012)

7. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering, ASE 2014, Vasteras, Sweden, 15–19 September 2014. pp. 313–324 (2014). http://doi.acm.org/10.1145/2642937.2642982
8. Fluri, B., Wuersch, M., PInzger, M., Gall, H.: Change distilling: tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. **33**(11), 725–743 (2007)
9. Fu, X., Lu, X., Peltsverger, B., Chen, S., Qian, K., Tao, L.: A static analysis framework for detecting SQL injection vulnerabilities. In: 31st Annual International Computer Software and Applications Conference, COMPSAC 2007, vol. 1, pp. 87–96. IEEE (2007)
10. Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: 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, pp. 38–49. ACM (2012)
11. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
12. Hopcroft, J., Tarjan, R.: Algorithm 447: efficient algorithms for graph manipulation. Commun. ACM **16**(6), 372–378 (1973)
13. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: Secubat: a web vulnerability scanner. In: Proceedings of the 15th International Conference on World Wide Web, pp. 247–256. ACM (2006)
14. Kreutzer, P., Dotzler, G., Ring, M., Eskofier, B.M., Philippsen, M.: Automatic clustering of code changes. In: Proceedings of the 13th International Conference on Mining Software Repositories, pp. 61–72. ACM (2016)
15. Lin, Z., Jiang, X., Xu, D., Mao, B., Xie, L.: AutoPaG: towards automated software patch generation with source code root cause identification and repair. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, pp. 329–340. ACM (2007)
16. Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in Java applications with static analysis. In: Usenix Security, vol. 2013 (2005)
17. Ma, S., Lo, D., Li, T., Deng, R.H.: CDRep: automatic repair of cryptographic misuses in android applications. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, pp. 711–722. ACM (2016)
18. Medeiros, I., Neves, N., Correia, M.: Detecting and removing web application vulnerabilities with static analysis and data mining. IEEE Trans. Reliab. **65**(1), 54–69 (2016)
19. Meghanathan, N.: Source code analysis to remove security vulnerabilities in Java socket programs: a case study. arXiv preprint arXiv:1302.1338 (2013)
20. Meng, N., Kim, M., McKinley, K.S.: LASE: locating and applying systematic edits by learning from examples. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 502–511. IEEE Press (2013)
21. Mohammadi, M., Chu, B., Lipford, H.R., Murphy-Hill, E.: Automatic web security unit testing: XSS vulnerability detection. In: 2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST), pp. 78–84. IEEE (2016)
22. Sidiroglou, S., Keromytis, A.D.: Countering network worms through automatic patch generation. IEEE Secur. Priv. **3**(6), 41–49 (2005)
23. Smirnov, A., Chiueh, T.C.: DIRA: automatic detection, identification and repair of control-hijacking attacks. In: Network and Distributed System Security Symposium (NDSS) (2005)

24. Son, S., McKinley, K.S., Shmatikov, V.: Fix me up: repairing access-control bugs in web applications. In: Network and Distributed System Security Symposium (NDSS) (2013)
25. Sotirov, A.I.: Automatic vulnerability detection using static source code analysis. In: Ph.D thesis (2005)
26. Wang, T., Wei, T., Gu, G., Zou, W.: TaintScope: a checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: 2010 IEEE Symposium on Security and Privacy (SP), pp. 497–512. IEEE (2010)
27. Yu, F., Shueh, C.Y., Lin, C.H., Chen, Y.F., Wang, B.Y., Bultan, T.: Optimal sanitization synthesis for web application vulnerability repair. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 189–200. ACM (2016)
28. Zhang, M., Yin, H.: AppSealer: automatic generation of vulnerability-specific patches for preventing component hijacking attacks in android applications. In: Network and Distributed System Security Symposium (NDSS) (2014)