Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions

Hyunji Hong
Department of Computer Science and Engineering, Korea University
hyunji_hong@korea.ac.kr

Seunghoon Woo
Department of Computer Science and Engineering, Korea University
seunghoonwoo@korea.ac.kr

Heejo Lee*
Department of Computer Science and Engineering, Korea University
heejo@korea.ac.kr

ABSTRACT

Online Q&A fora such as Stack Overflow assist developers to solve their faced coding problems. Despite the advantages, Stack Overflow has the potential to provide insecure code snippets that, if reused, can compromise the security of the entire software.

We present Dicos, an accurate approach by examining the change history of Stack Overflow posts for discovering insecure code snippets. When a security issue was detected in a post, the insecure code is fixed to be safe through user discussions, leaving a change history. Inspired by this process, Dicos first extracts the change history from the Stack Overflow post, and then analyzes the history whether it contains security patches, by utilizing pre-selected features that can effectively identify security patches. Finally, when such changes are detected, Dicos determines that the code snippet before applying the security patch is insecure.

To evaluate Dicos, we collected 1,958,283 Stack Overflow posts tagged with C, C++, and Android. When we applied Dicos on the collected posts, Dicos discovered 12,458 insecure posts (*i.e.*, 14,719 insecure code snippets) from the collected posts with 91% precision and 93% recall. We further confirmed that the latest versions of 151 out of 2,000 popular C/C++ open-source software contain at least one insecure code snippet taken from Stack Overflow, being discovered by Dicos. Our proposed approach, Dicos, can contribute to preventing further propagation of insecure codes and thus creating a safe code reuse environment.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Q&A forum; Insecure code snippet discovery; Software security.

ACM Reference Format:
Hyunji Hong, Seunghoon Woo, and Heejo Lee. 2021. Dicos: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In *Annual Computer Security Applications Conference (ACSAC '21), December 6–10, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3485832.3488026

*Heejo Lee is the corresponding author

ACSAC '21, December 6–10, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488026

1 INTRODUCTION

Developers commonly leverage online Q&A fora such as Stack Overflow and Quora [1, 7, 35], because copying and pasting code snippets from Stack Overflow are being considered as improving productivity and providing more stable code. However, this can cause problems when developers reuse code snippets without understanding the code implications, such as the propagation of insecure code snippets. In 2018, for example, there was an issue that Docker could not be worked when Razor Synapse was running in the background [3]. According to a Reddit post, this issue was caused by reusing the insecure code snippet from Stack Overflow [20].

The approach to resolving such issues consists of two steps: (1) discovering insecure code snippets from Stack Overflow and (2) detecting software containing the insecure code snippets. The problem is that the process of discovering insecure code snippets is challenging, as opposed to the second step, which can be accomplished with techniques such as code clone detection [10, 11, 21, 30].

Existing approaches are limited in terms of the discovery coverage; most of them were only able to discover insecure code snippets containing selected security-related APIs (*e.g.*, cryptographic APIs) [7, 23, 32, 34]. By contrast, unknown security patch detection approaches [15, 18, 25] that can be used to discover insecure code snippets, yield many false alarms owing to ineffective feature selection, *e.g.*, they consider only the fragmentary characteristics of the security patch [15] (details are introduced in section 7).

To overcome such shortcomings, we present Dicos (**D**iscovering **I**nsecure **CO**de **S**nippets), which is an accurate approach by examining the change history of Stack Overflow posts for discovering insecure code snippets. The key idea of Dicos, unlike previous approaches that have discovered insecure code snippets considering only the latest revision of a post, is the analysis of the change history of a post based on user discussions.

Our approach. Dicos first selects effective features and then discovers insecure code snippets by leveraging user discussions in Stack Overflow.

Selecting effective features that prominent only in insecure codes is challenging. The features used in existing approaches were either too finer or too coarse, and thus not guarantee efficient insecure code snippet discovery (see section 7). To address this challenge, Dicos first collects a number of features that are used in previous approaches, and then verifies their effectiveness using Common Vulnerabilities and Exposures (CVE). In particular, Dicos attempted to identify features that predominantly appear in CVE patches. As a result, Dicos determines that the following three features could be utilized effectively in discovering insecure code snippets: (1) security-sensitive APIs, (2) security-related keywords, and (3) control flow information (see section 3).

Next, to discover insecure code snippets, Dicos leverages *user discussions* in Stack Overflow. When a security issue was detected in the code snippet, the insecure code is fixed to a safe one through user discussions, leaving a change history (see subsection 2.1). Inspired by this, Dicos first extracts the change history from the post, analyzes the change history using selected three features, and then determines whether the post contains insecure code snippets.

Dicos extracts the change history from the answer post; we pay attention to the changes in description, comments, and code snippets, which mainly changed when a security issue was reported and fixed. In particular, extracting change history from code snippets is error-prone because a post can contain multiple code snippets; completely different code snippets can be paired between the two post revisions. To resolve this issue, Dicos cleaves each code snippet: if a code snippet contains a function, Dicos extracts the function and considers it as a new code snippet. Dicos then detects the most similar code snippets between the two revisions of the post, determines them as a code snippet pair, and then extracts the change history from the paired code snippets (see subsection 4.1).

Thereafter, Dicos analyzes the change history whether it contains a security patch by using the three selected features. Dicos determines that a change was a security patch if: (1) control flows or (2) security-sensitive APIs of the code snippet were changed, or (3) security-related keywords were added to the description or comments. Finally, Dicos determines the older revision of a post (*i.e.*, before applying the detected security patch) containing two or more features in its change history as the *insecure post*, and the code snippet included in the insecure post as the *insecure code snippet*.

Evaluation. To evaluate Dicos, we collected 1,958,283 Stack Overflow answer posts tagged with C, C++, and Android from the *SOTorrent* dataset [2], of which 668,520 posts contained the change history. In the experiment, Dicos discovered **12,458 insecure posts (2%)** with 14,719 insecure code snippets. To verify the discovery results, we manually reviewed the following five groups for C/C++ and Android insecure posts, respectively: (1) all posts with three features, (2) the top 200 posts with two features (ranked by #votes), (3) randomly selected 100 posts with two features (to prevent biased validation results), (4) the top 200 posts with only one feature, and (5) the top 100 posts without any features. Consequently, we confirmed that Dicos discovered insecure code snippets with 93% precision and 94% recall for C/C++ posts, with 86% precision and 89% recall for Android posts (details are presented in subsection 5.2).

We then compared the discovery results of Dicos to those of the closely related approaches [7, 23]. First, Dicos discovered 2,454 Android insecure posts in the same dataset with Fischer *et al.* [7], which is **9 times more** than that reported by their approach (*i.e.*, 278 insecure posts). Among their results, 62 were targets of Dicos (*i.e.*, posts containing code change history), and Dicos was able to cover 50 out of 62 insecure posts (81%). Next, Dicos discovered 7,241 C/C++ insecure posts in the same dataset with Verdi *et al.* [23], which is **105 times more** than that discovered by them (*i.e.*, 69 insecure posts). Among their results, 36 were targets of Dicos, and Dicos was able to discover 22 of them (61%) as insecure. The result

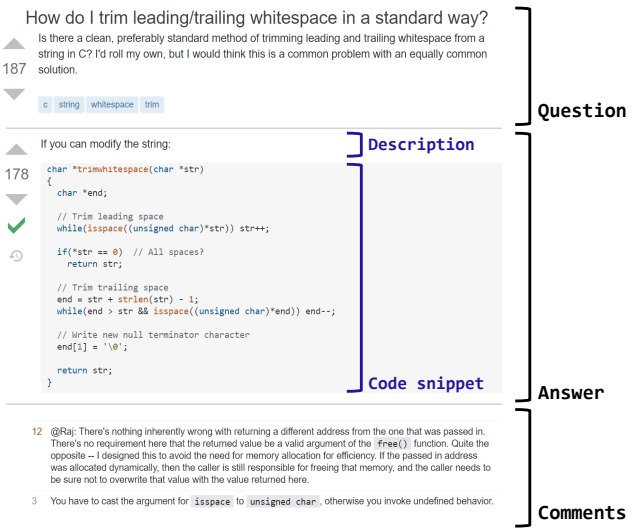

Figure 1: Example Stack Overflow post (#122721). We divide a post into three parts: question, answer, and comments; the answer is further subdivided into code snippet and description (*i.e.*, narrative part excluding code snippets).

demonstrates the effectiveness of Dicos as it can cover a large proportion of the insecure posts discovered by the existing approaches while discovering more hidden insecure posts (see subsection 5.3).

We further detected propagated insecure code snippets in the latest versions of 2,000 popular C/C++ open-source software (OSS) using the existing code clone detection technique [21]. As a result, we confirmed that **151 OSS (8%)** were reusing insecure code snippets; we reported cases where the insecure code snippet could adversely affect the software to the vendors (see subsection 6.4).

This paper makes the following three contributions:

- We propose Dicos, an accurate approach for discovering insecure code snippets in Stack Overflow posts by leveraging user discussions in Stack Overflow.
- We extracted features that prevalent in security patches (*i.e.*, changes in security-sensitive APIs, control flows, and security-related keywords), which can be used for discovering insecure code snippets.
- We demonstrated the effectiveness of Dicos using 1,958,283 Stack Overflow posts; Dicos discovered 14,719 insecure code snippets with 91% precision and 93% recall.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background of discussions in Stack Overflow, provide a motivating example of this paper, and then describe an overview of Dicos.

2.1 Discussions in Stack Overflow

Developers actively discuss software development through Stack Overflow. Figure 1 presents an example post on Stack Overflow. The entire discussion flow can be categorized into three steps: (1)

asking a question, (2) answering the question, and (3) commenting or voting on both the questions and answers. The details of each step are as follows.

(1) **Asking a question:** The *questioner* posts a question to Stack Overflow. Questions belong to various categories: a user can ask about developing a source code that performs a specific functionality or about addressing an error that occurs when compiling specific software.

(2) **Answering the question:** Multiple users suggest answers to the posted question (*i.e.*, *answerers*). A user answers the question in a narrative form (*i.e.*, the "description" part in Figure 1), or often provides an actual source code snippet (*i.e.*, the "code snippet" part in Figure 1).

(3) **Commenting or voting:** Any Stack Overflow users can comment, vote, and score for both questions and answers. Users tend to add comments, especially when there are issues with answerers' code snippets (*e.g.*, a flaw was detected) [33].

Questions, answers, and comments can be edited at any time after the initial registration. All questions, answers, and comments are released publicly, including their revisions. Many developers refer to this information in their software development, or even reuse the source code contained in the answer post [7, 35].

2.2 Motivating example

In this paper, we focus on the problems that arise when code snippets within the post are insecure. As an example, we introduce Stack Overflow post #122721, a post on how to trim trailing spaces from strings in C language[1]. The first answer to this question was posted in September 2008; the post provided the "`trimwhitespace`" function, which can trim leading and trailing spaces in strings in C language (see Listing 1).

Listing 1: The original answer code snippet in post #122721.

```
1  char *trimwhitespace(char *str) {
2      char *end;
3      // Trim leading space
4      while(isspace(*str)) str++;
5
6      // Trim trailing space
7      end = str + strlen(str) - 1;
8      while(end > str && isspace(*end)) end--;
9
10     // Write new null terminator character
11     *(end+1) = 0;
12     return str;
13 }
```

However, multiple Stack Overflow users commented in the answer post that the following two code parts could be insecure:

(1) **"`isspace()`" may cause undefined behavior.** If the `isspace()` function (*i.e.*, line #4 in Listing 1) receives a negative value, the compiler changes the value to a sign-extended value. As a result, unexpected results would be obtained.

(2) **"`*str`" may cause a null pointer dereference.** If the length of "`*str`" is 0, the string "end" becomes "`str-1`", with which the value could be invalid pointer. This typically causes a crash or exit, *e.g.*, a race condition vulnerability.

[1] https://stackoverflow.com/questions/122721

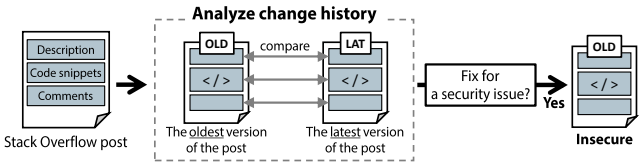

Figure 2: High-level overview of the workflow of Dicos.

After confirming the comments, the answerer edited the post to fix the insecure code snippet. The changes are shown in Listing 2. Specifically, they addressed the `isspace()` issue by casting `char` to `unsigned char` (see the related documentation [16]), as shown in lines #5 and #13 in Listing 2, and they added the exception handling code to lines #7-8 in Listing 2 to handle the `*str` issue.

Listing 2: A code change history for post #122721.

```
1  char *trimwhitespace(char *str) {
2      char *end;
3      // Trim leading space
4      -  while(isspace(*str)) str++;
5      +  while(isspace((unsigned char)*str)) str++;
6
7      +  if(*str == 0) // All spaces?
8      +     return str;
9
10     // Trim trailing space
11     end = str + strlen(str) - 1;
12     -  while(end > str && isspace(*end)) end--;
13     +  while(end > str && isspace((unsigned char)*end)) end--;
14
15     // Write new null terminator character
16     -  *(end+1) = 0;
17     +  end[1] = '0';
18     return str;
19 }
```

If this insecure code snippet is reused in a software program, the entire security and functionality of the software program can be compromised. Therefore, it is important to discover insecure code snippets in Stack Overflow posts in advance; thereafter, we can track software containing any discovered insecure code snippets, and recommend remediations, *e.g.*, fixing the insecure code.

However, the existing approaches are not designed to discover insecure code snippets with high accuracy. For instance, they cannot discover our motivating code snippet as insecure; this is because the code snippet does not contain any security-sensitive APIs (*e.g.*, strcpy, malloc, cryptographic functions) [5, 7, 34] or vulnerable keywords (*e.g.*, "CoinHive") [32]. Moreover, this code snippet is written in the C language, and is out of the scope of several existing approaches [5, 7, 34].

This insecure code snippet was reused in the latest versions of three popular open-source projects. One of them is the Linux kernel, in which this code snippet was reused with the `isspace()` issue; however, they confirmed to us that they already addressed these issues using an exception handling in a different source file. We have forwarded this issue to the development teams for the other two projects, and we are still waiting their responses.

2.3 Overview of Dicos

As explained in the motivating example, an insecure code snippet in Stack Overflow posts can adversely affect various software programs. To resolve such problems, we propose Dicos, an approach for discovering insecure code snippets in Stack Overflow posts. Figure 2 illustrates the high-level workflow of Dicos.

The key idea of Dicos for discovering insecure code snippets is leveraging user discussions in Stack Overflow. In general, an answerer edits their code snippets when they notice that their code has a flaw, such as a security issue [4, 24]. After editing, they leave all edit logs in their post.

Inspired by this process, we decided to use the change history of the post, as it provides significant hints for discovering insecure code snippets. Specifically, Dicos comprises the following three phases: (1) extracting the change history (*i.e.*, `diffs` between the oldest and latest revisions) from the post, (2) analyzing the `diffs` using selected three features, and (3) determining whether the post contains insecure code snippets.

Dicos first extracts the `diffs` from the Stack Overflow post. Although the goal of Dicos is to discover insecure code snippets, changelogs in descriptions and comments can also be used to discover insecure code snippets, thus focusing on all the changes to the description, code snippets, and comments (see Figure 1). Dicos then discovers insecure code snippets by analyzing whether the extracted `diffs` are intended to fix a security issue, based on the selected features (the feature selection process is introduced in section 3). If a `diff` is intended to patch a security issue, Dicos determines that the oldest code snippet in the post is insecure (*i.e.*, a code snippet without applying security patches). The detailed design of Dicos is provided in section 4.

3 FEATURE SELECTION

To discover insecure code snippets, we first examined utilized features in existing approaches and then picked only the effective ones based on an empirical study using CVE vulnerability patches. In particular, we examined various features that were previously used in related approaches: Stack Overflow insecure code snippet discovery approaches [5, 7, 32, 34] and unknown security patch detection approaches [14, 15, 18, 25].

Initial feature selection methodology. We reviewed the features used in the related approaches and selected a total of 12 features applicable to code snippets, descriptions, and comments; the list of features used in the existing approaches is summarized in Table 1. Among them, however, some features cannot be used for discovering vulnerable snippets especially from Stack Overflow (*e.g.*, number of commits); in addition, some other features are too general to distinguish vulnerable code snippets (*e.g.*, changes in files, functions, and lines); thus, we excluded them. Finally, we consider the following six features (*i.e.*, F1 to F6).

F1. **Changes in security-sensitive APIs:** Checking whether a code change has occurred in security-sensitive APIs. We selected security-sensitive APIs by referring to related approaches [5, 7, 34] and Common Weakness Enumeration (CWE) documents [17].

Table 1: List of the features that were used in related approaches for discovering insecure code snippets and unknown security patches.

Idx	Features	Approaches (considered features: O)							
		[18]	[7]	[32]	[34]	[14]	[25]	[5]	[15]
1	Changes in a file						O		
2	Changes in a function	O					O		
3	Changes in a line						O		
4	Changes in a conditional statement					O	O		
5	Changes in a control flow					O	O		O
6	Changes in a function call						O		
7	Changes in an operator						O		
8	Changes in a variable						O		
9	Changes in a security-related keyword	O	O	O				O	
10	Changes in a security-sensitive API		O		O			O	
11	Changes in a hunk count	O					O		
12	Number of commits	O							

F2. **Changes in security-related keywords:** Checking whether a change in the description or commit messages contains security-related keywords. We selected the keywords by referring to existing approaches [7, 9]. In addition, we analyzed the commit messages of 3,323 CVEs, which disclosed their code patches via GitHub, and manually collected words that were frequently included in the commit messages.

F3. **Changes in control flows (and conditional statements):** Checking whether a control flow change exists (*e.g.*, adding new conditional statements). As existing approaches have found that the control flow changes account for a large portion of the security patches [14, 15, 25], we also consider this feature in discovering insecure code snippets.

F4. **Changes in literals (*e.g.*, operator, constants):** Checking whether a change in code snippets contains literal changes (*e.g.*, modifying constant values).

F5. **Changes in identifiers (*e.g.*, variable):** Checking whether a change in code snippets contains AST identifier changes [31] (*e.g.*, changing local variables).

F6. **Changes in function calls:** Checking whether a change in code snippets contains a function call change; the function call can include security-sensitive APIs.

Large-scale empirical study using CVE vulnerabilities. To demonstrate the efficiency of the selected features, we conducted an empirical study using 3,323 C/C++ CVE vulnerabilities, which provide their patch commits via GitHub (referring to [13]). We analyzed all patches of the collected CVEs to measure the number of appearances of each feature. Additionally, to determine whether the selected features were prominent only in security patches, we extracted a total of 1,000 of the latest commits with 889 general code patches from the Linux kernel, Tensorflow, Redis, and Electron (*i.e.*, 250 commits from each repository), which are four of the most popular C/C++ repositories on GitHub based on the rank of the number of stars. We measured the appearances of the features in security patches, and compared the results with those from the general code patches (security-sensitive APIs and security-related keywords selected for the empirical study are listed in Table 7 in Appendix A and Table 9 in Appendix B, respectively). Consequently, Table 2 presents the measurement results.

Table 2: Results of the empirical study using 3,323 CVE vulnerability patches and 1,000 general code commits containing 889 code patches.

Index	Features	# of appearances of the features	
		CVE patches	General patches
F1	Changes in security-sensitive APIs	577 (17.36%)	38 (4.00%)
F2	Changes in security-related keywords	2,019 (60.76%)	114 (11.4%)∗
F3	Changes in control flows	2,493 (75.02%)	230 (25.87%)
F4	Changes in literals	1,454 (43.76%)	179 (20.13%)
F5	Changes in identifiers	2,170 (65.30%)	405 (45.56%)
F6	Changes in function calls (APIs)	2,791 (83.99%)	505 (56.81%)

∗ Security-related keywords were searched for in 1,000 commit messages.

First, we confirmed that *F2, F3, F5*, and *F6* appeared frequently in the security patches (*i.e.*, more than 60%). However, unlike *F2* and *F3*, which appeared at a much higher rate in security patches (60.76% and 75.02%, respectively) than in general code patches (11.4% and 25.87%, respectively), *F5* and *F6* frequently appeared in both general and security patches. Thus, we decided to use *F2* (changes in security-related keywords) and *F3* (changes in control flows) while excluding *F5* and *F6* when discovering insecure code snippets.

Next, we confirmed that *F1* (changes in security-sensitive APIs) appeared at a rate four times higher in security patches than in general code patches. Obviously, changes in security-sensitive APIs are likely to be security patches (*e.g.*, removing the `strcpy` function). Therefore, we decided to use feature *F1*. Lastly, we confirmed that *F4* (changes in literals) appeared at a rate more than twice as high in security patches as compared to general patches. However, surprisingly, we discovered that F4 mainly appears with either *F1* or *F3*; there were only eight security patches where F4 appeared without *F1* or *F3*. Since we already selected *F1* and *F3*, *F4* was excluded.

Consequently, we decided to use the following three features:

F1. Changes in security-sensitive APIs;

F2. Changes in security-related keywords;

F3. Changes in control flows.

Figure 3 shows the coverage of the three selected features in the collected CVE patches. In particular, we found that 92% of the collected CVE patches had at least one selected feature. The remaining CVE patches either did not have a commit message or no specific security patch pattern was revealed (*e.g.*, adding only a variable declaration statement); thus, the selected features failed to cover them. Furthermore, among the collected general patches, only eight patches (1%) contained all three selected features; there were 90 patches (10%) containing two features simultaneously. In other words, we concluded that the selected features can distinguish between security patches and general code patches with a high degree of accuracy.

4 INSECURE CODE SNIPPET DISCOVERY

In this section, we describe how Dicos discovers insecure code snippets. Dicos comprises the following three phases for discovering insecure code snippets: (1) extracting the change history of a post, (2) analyzing the change history using selected features, and (3) determining whether the post contains insecure code snippets.

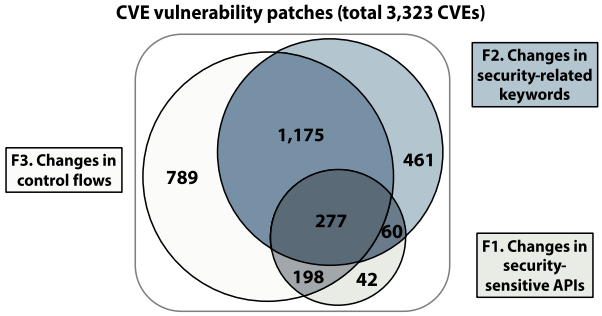

Figure 3: Illustration for the coverage of the three selected features (changes in security-sensitive APIs, changes in security-related keywords, and changes in control flows).

4.1 Extracting the change history of a post

As mentioned in subsection 2.3, Dicos discovers insecure code snippets by leveraging user discussions in Stack Overflow: when insecure code is detected in a particular post, the answerer takes a series of specific actions to resolve the issue.

In subsection 2.1, we categorize a Stack Overflow post into three parts: the question, answer, and comments. The answer part can be further categorized into two sub-parts: description and code snippet parts (see Figure 1). Among them, the *description*, *code snippets*, and *comments* parts are mainly changed when a security issue is detected in a code snippet. We summarize the characteristics of the changes for each part as follows.

(1) **Comments - *reporting security issues*:** If the code snippet in the answer post has security issues (*e.g.*, vulnerability), Stack Overflow users can notify the issues through comments, or even provide "suggested edit".

(2) **Code snippet - *fixing the security issues*:** When a security issue is found in the code snippet, the answerer modifies the insecure code snippet into a safe one.

(3) **Description - *introducing the changes*:** If answerers edit their posts, they often leave the related messages (*e.g.*, the reason for editing their post) in the description part.

Dicos first collects Stack Overflow posts, and then extracts all the change histories of each post. Because the Stack Overflow post dataset contains every revision of each post [2], this task can easily be conducted. Dicos then selects the oldest and the latest revision of the post, and then extracts the differences (*i.e.*, `diffs`) between the two revisions. Although a post may have been changed multiple times, we focus on the difference between the oldest and the latest version of a particular post, because we assume that changes between the oldest and latest revisions include all change patterns that exist between them. The `diffs` for comments and descriptions are extracted in the form of string differences.

Technical challenge: code snippet pairing problem. Here, one technical challenge arises: the code snippet pairing problem. In general, a post contains multiple code snippets, and a code snippet can contain several functions that perform specific functionalities.

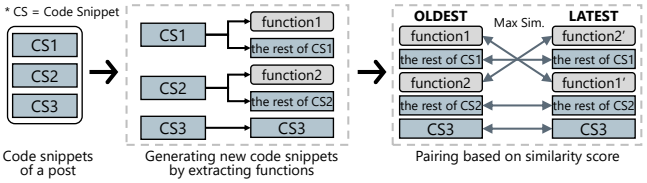

Figure 4: Overall flow of the code snippet pairing.

In the process of updating the post, a code snippet can be added or deleted, the order of code snippets can be changed, and only some of the contained functions can move to another code snippet. In this situation, the simple diffing method can extract the erroneous change history from a completely different code snippet pair.

To overcome this challenge, we present a novel code snippet pairing technique. The key idea is extracting functions contained in code snippets by cleaving them, and measuring similarity scores between all code snippets in the oldest and latest revisions. The overview of the pairing process is depicted in Figure 4.

Function extraction from code snippets. From every code snippet in the post, Dicos extracts all functions contained in the code snippet. In other words, Dicos splits the code snippet into finer functional units for the more attentive discovery of insecure code snippets. Let *CS* be a code snippet in a post; this can represent an extracted function. Dicos applies this step for the oldest and latest revision of the post, respectively.

Code snippet pairing. In the code snippet pairing process, Dicos first picks one *CS* from the oldest revision, then measures the similarity score to every *CS* in the latest revision. To do this, Dicos first splits each *CS* into a set of code lines (*i.e.*, splits with a newline character), removes spaces from each code line, and converts all the characters in each code line to lower cases (*i.e.*, applying normalization) for ignoring formatting-related changes that do not affect the semantic of the codes [11, 27, 30].

Let the normalized *CS* (*i.e.*, set of normalized code lines) of the oldest revision is CS_o and that of the latest revision is CS_l. Dicos then measures the similarity score (let Φ) between CS_o and CS_l by leveraging the Jaccard index [26] $(i.e., \Phi = \frac{|CS_o \cap CS_l|}{|CS_o \cup CS_l|})$. If Φ is greater than the pre-defined threshold θ value, Dicos determines that CS_o and CS_l as *a code snippet pair*; here, θ is selected as a small value by considering code changes (*e.g.*, less than 0.5).

After finding all code snippet pairs between the oldest and latest revisions, Dicos extracts the `diffs` between the paired code snippets by utilizing the known diffing command, such as `git diff`. The `diffs` for the code snippets are extracted in the form of code patches. Here, if a CS_o has a similarity value less than θ with all CS_l, we consider that the function has undergone a major change or has been deleted during post updates. To cover this case, Dicos searches older revisions from the latest revision in the reverse direction and processes the same approach, recursively. If Dicos does not find any code snippets with a similarity greater than θ to CS_o, even after traversing all revisions of the post, Dicos considers that CS_o was deleted during post updates, and the change history is obtained in which the entire code lines in CS_o was deleted.

4.2 Analyzing the extracted change history

In this part, we introduce how Dicos analyzes the change history.

Technical challenge: applying features to Stack Overflow. Here comes a technical challenge, which is to devise an efficient way to apply features extracted from known security patches to insecure code snippet discovery from Stack Overflow. We introduce the process of discovering insecure code snippets by applying the three features extracted in section 3 to each element (*i.e.*, description, comments, and code snippets) of a Stack Overflow post.

Analyzing descriptions and comments. When confirming that the `diffs` of descriptions and comments are related to a security patch, we can check whether security-related keywords (*F2*) are included in the `diffs`. However, this simple method can yield many false positives. For example, "fix" is one of the most frequently used keywords in security patches, but it can also be used to explain a non-security change (*e.g.*, one of the commit messages from Redis[2] is "Fix typos in comments and improve readability").

Instead, we classified the selected keywords into three categories: nouns, verbs, and modifiers. Thereafter, Dicos checks if a security-related keyword *pair* is included within *each sentence* of the `diffs` in post description or comments; here, a security-related keywords pair is defined either (*noun*, *verb*) or (*modifier*, *verb*). In other words, we determine that the keyword is matched only when the specific target and the behavior of the target are both related to security.

Checking the existence of a security-related keyword pair is conducted using a simple string inclusion operation. To prevent matching failures caused by the difference between uppercase and lowercase, Dicos replaces all characters in descriptions, comments, and keywords to lowercase when performing string matching operations (the selected security-related keywords and their classification are listed in Table 9 in Appendix B).

One consideration is that if the purpose of a specific post is to resolve a security issue, the description may contain security-related keywords from the time the post was created, even though the contained code snippet was safe. Therefore, Dicos only considers cases where a security-related keyword pair is included only in `diffs`: that is, if a security-related keyword pair exists in the post description when the post originated, Dicos determines that this post does not contain the security patch pattern. As a representative example, we introduce post #44184152[3] (*i.e.*, "How to avoid if/else if chain when classifying a heading into 8 directions?").

Listing 3: Patch for the insecure code snippet in post #44184152.

```
Dir GetDirForAngle(int angle)
{
    const Dir slices[] = { RIGHT, UP_RIGHT, UP, UP, UP_LEFT,
          LEFT, LEFT, DOWN_LEFT, DOWN, DOWN, DOWN_RIGHT, RIGHT };
    -  return slices[(angle % 360) / 30];
    +  return slices[(((angle % 360) + 360) % 360) / 30];
}
```

Listing 4: Added comments and descriptions in post #44184152.

- **User's comment:** ...this wouldn't work for negative inputs because angle % 360 returns a negative value when angle is negative.
- **Answerer's decription:** **Fixed** math to handle **negative** angles...

[2]https://github.com/redis/redis
[3]https://stackoverflow.com/questions/44183771

The original code snippet in the answer post was insecure because the "`angle % 360`" could be negative (see Listing 3). The code snippet was modified by a user's comment, and the answerer included details about the change in the description part (see Listing 4). Dicos can identify that this is a security-related change by checking whether a security-related keyword is included in the `diffs` of descriptions: the verb "fix" and the modifier "negative" are included in one sentence of the added description.

Analyzing code snippets. Dicos uses *F1* and *F3* (*i.e.*, changes in security-sensitive APIs and control flows) to analyze code snippets in Stack Overflow posts; the selected APIs are listed in Table 7 (for C/C++ posts) and Table 8 (for Android posts) in Appendix A.

To analyze changes in security-sensitive APIs (*F1*), Dicos considers the `diff` of code snippet that extracted in subsection 4.1, especially the deleted source code lines. If any delete code lines contain a security-sensitive API, Dicos determines that the change history of the post contains a security patch. As an example, we introduce post #700018[4] (*i.e.*, "Display the binary representation of a number in C?") where *F1* was detected.

Listing 5: A patch snippet contained in post #700018.

```
1  static char *binrep (unsigned int val, char *buff, int sz) {
2      + char *pbuff = buff;
3      ..
4      /* Special case for zero to ensure some output. */
5      if (val == 0) {
6          -   strcpy(buff, "0");
7          +   char *pbuff = buff;
8          +   *pbuff++ = '0';
9          +   *pbuff = '\0';
10         return buff;
11 ..}
```

The patch shown in Listing 5 was applied to prevent a possible buffer overflow; as the `strcpy` function has been deleted in line #6, Dicos can identify that the change is related to security.

Next, to analyze changes in control flows (*F3*), Dicos first extracts control flows and all conditional statements from both the paired code snippets (*e.g.*, we can use the robust parser Joern [31] to extract these pieces of information). Dicos then checks whether the `diffs` of the code snippet contains a change in control flows or conditional statements. Specifically, Dicos considers both (1) a change in the direction of control flows (*e.g.*, adding `if` statement) and (2) a change in the conditions of each conditional statement even though the direction of control flows remains unchanged (*e.g.*, changing a condition in an `if` statement). For example, we introduce post #744822 (*i.e.*, "How to compare ends of strings in C?") where *F3* was detected[5].

Listing 6: A patch snippet contained in post #744822.

```
1  int EndsWith (const char *str, const char *suffix) {
2      + if (!str || !suffix)
3      +     return 0;
4      size_t lenstr = strlen(str);
5      size_t lensuffix = strlen(suffix);
6  ..}
```

By applying this patch (Listing 6), the answerer tried to prevent possible errors by adding a null check for the `str` and `suffix` variables. Dicos can determined that this change is a security patch as the control flow was obviously changed.

[4]https://stackoverflow.com/questions/700018
[5]https://stackoverflow.com/questions/744822

4.3 Determining insecure code snippets

Dicos discovers insecure code snippets based on the analysis results using the selected features (*F1*, *F2* and *F3*) in subsection 4.2. For more accurate detection, Dicos defines *insecure posts* as posts in which *two or more features* appear simultaneously. This is because, we confirmed that the post in which only one feature appeared, especially for *F1* and *F3*, is more likely to be modified to simply change the functionalities, rather than resolving a security issue. Finally, Dicos determines that the code snippet contained in the discovered insecure posts is an *insecure code snippet*.

Unlike existing approaches, Dicos does not depend on a single feature but uses a combination of effective features and does not consider only specific security issues but can cover various security issues by using multiple security-related keywords and security-sensitive APIs along with considering a change in the control flows. Furthermore, the design of Dicos is applicable to any programming language, and all processes of Dicos can be conducted in an automated manner. Consequently, Dicos can discover insecure code snippets with high accuracy compared to existing approaches (see subsection 5.3).

5 EVALUATION

In this section, we evaluate Dicos. subsection 5.1 introduces the collected dataset and the implementation of Dicos. subsection 5.2 investigates how accurately Dicos can discover insecure code snippets in practice. We then compare Dicos with existing approaches [7, 23] in subsection 5.3, thereby demonstrating the effectiveness of Dicos. subsection 5.4 examines the effectiveness of the techniques utilized in Dicos, and subsection 5.5 measures the performance of Dicos. We evaluated Dicos on a machine with Ubuntu 18.04.4 LTS, an Intel i5-6600 CPU @ 3.30GHz, 24GB RAM, and a 1TB SSD.

5.1 Dataset and implementation

We first introduce the dataset collection methodology and the implementation of Dicos.

Dataset. We preferentially evaluate Dicos using the C, C++, and Android related posts, since the reuse of small pieces of code is prevalent in the software [7, 11, 27, 28]. It should be noted that the design of Dicos can be applied to any programming language. We used the *SOTorrent* dataset [2] which is available at Google BigQuery [19], as the *SOTorrent* dataset provides well-constructed Stack Overflow posts. We utilized the latest released version, *i.e.*, "2020-12-31" (4.4GB). We then extracted all Stack Overflow answer posts tagged with C, C++, and Android, which contain at least one code snippet, and extracted a total of 1,958,283 Stack Overflow answer posts (*i.e.*, 987,367 C/C++ and 970,916 Android posts). As Dicos focuses on the change history of each post, every post should contain at least one change history. Among all the posts collected, 668,520 (34%) satisfied this condition.

As a result, our dataset for the evaluation consisted of 668,520 posts, which contained an accumulation of 1,514,547 code snippets (*i.e.*, averaged two or three code snippets per a post).

Implementation. Dicos comprises the following two modules: a *post collector* and a *post analyzer*. The *post collector* collects Stack Overflow posts and extracts the change history in descriptions, comments, and code snippets for each post. In the code snippet pairing process, we set the θ value as 0.3 (see subsection 4.1).

The *post analyzer* analyzes the collected posts based on the selected features in section 3, and then discovers insecure code snippets. When analyzing the code snippets provided in the form of a C/C++ function, Dicos utilizes Ctags [6] and a Joern parser [31]. In particular, Dicos first extracts a function from the code snippets with Ctags, and then generates a code property graph of the extracted function using the Joern parser, from which it can obtain the control flows and contained conditional statements of the function. For the code snippets provided in the form of a set of code lines, Dicos identifies control flows and conditional statements based on regular expressions, because Ctags and the Joern parser can only be applied to a function unit. Unlike Ctags, which are applicable to all languages, the Joern parser can only be applied in C/C++ languages. Therefore, we utilized the combination of Ctags and regular expressions for analyzing Android code snippets.

Dicos is implemented on approximately 800 lines of Python code excluding for the external libraries. The source code of Dicos is available at https://github.com/hyunji-hong/DICOS-public.

5.2 Discovery accuracy of Dicos

To evaluate accuracy, we analyzed the insecure code snippet discovery results of Dicos on real-world Stack Overflow posts.

Methodology. We applied Dicos to our dataset of 668,520 Stack Overflow posts, and measured the accuracy of Dicos. Specifically, we verified whether the insecure posts discovered by Dicos actually contain insecure code snippets, and whether the posts that were determined as secure by Dicos do not contain insecure code snippets. To evaluate the accuracy of Dicos, we used the following seven metrics: true positives (TP), false positives (FP), true negatives (TN), false negatives (FN), precision $\left(\frac{\#TP}{\#TP + \#FP}\right)$, recall $\left(\frac{\#TP}{\#TP + \#FN}\right)$, and accuracy $\left(\frac{\#TP + \#TN}{\#TP + \#FP + \#TN + \#FN}\right)$.

Accuracy measurement. In our experiments, Dicos discovered **12,458 insecure posts** (a total of 14,719 insecure code snippets) out of 668,520 collected posts; 8,941 insecure posts tagged with C/C++, and the remaining 3,517 insecure posts tagged with Android.

Among the 12,458 insecure posts, we observed that 788 insecure posts contained all three selected features (*F1*, *F2*, and *F3*, see section 3), and the remaining 11,670 insecure posts contained two of the selected features. In fact, it is very challenging to verify all discovery results manually. Instead, we initially selected the top posts with the highest number of votes and measured the accuracy of Dicos by analyzing the selected posts. This is because we determined that they had a greater impact on the software development; such posts may obtain more users' attention and are more likely to spread to other software. In addition, we analyzed a group consists of randomly selected insecure code snippets discovered by Dicos to avoid giving a biased result in the number of votes. Finally, the five groups selected for accuracy measurement are as follows.

Table 3: Accuracy measurement result of Dicos for C/C++ posts.

ID	#Posts	#TP	#FP	#TN	#FN
G1	731	704	27	N/A	N/A
G2	200	171	29	N/A	N/A
G3	100	82	18	N/A	N/A
G4	200	N/A	N/A	151	49
G5	100	N/A	N/A	92	8
Total	1,331	957	74	243	57
Precision					**0.93**
Recall					**0.94**
Accuracy					**0.90**

Table 4: Accuracy measurement result of Dicos for Android posts.

ID	#Posts	#TP	#FP	#TN	#FN
G1	57	53	4	N/A	N/A
G2	200	175	25	N/A	N/A
G3	100	80	20	N/A	N/A
G4	200	N/A	N/A	167	33
G5	100	N/A	N/A	93	7
Total	657	308	49	260	40
Precision					**0.86**
Recall					**0.89**
Accuracy					**0.86**

G1. All posts with three selected features
G2. Top 200 posts with two selected features
G3. Randomly selected 100 posts with two features
G4. Top 200 posts with only one feature
G5. Top 100 posts without features

Note that G1, G2, and G3 are insecure posts discovered by Dicos (*i.e.*, for measuring TP and FP); G4 and G5 are posts that Dicos determined to be safe (*i.e.*, for measuring TN and FN). We manually verified the discovery results; the manual verification was performed by two researchers with the ability to determine whether a code snippet is insecure by reviewing the post and its change history, which took ten days. The accuracy measurement results are presented in Table 3 (for C/C++), Table 4 (for Android), and Table 5 (for both languages).

We confirmed that Dicos showed **91% precision**, **93% recall**, and **89% accuracy** for insecure posts discovery (see Table 5); in particular, Dicos showed 93% precision, 94% recall, and 90% accuracy for C/C++ insecure posts discovery (see Table 3), and 86% precision, 89% recall, and 86% accuracy for Android insecure posts discovery (see Table 4). The accuracy measurement results of C/C++ and Android showed comparable patterns in the groups from G2 to G5, but there was a big difference in the verification results for G1. Note that changes of security-sensitive APIs predominantly appeared in Android posts while control flow changes hardly occurred. Consequently, the number of posts contained in G1 was small in Android cases, and most of the posts belonging to G1 are TPs (*i.e.*, insecure),

Table 5: Integrated accuracy measurement results for C, C++, and Android posts.

ID	#Total Posts	#TP	#FP	#TN	#FN
G1	788	757	31	N/A	N/A
G2	400	346	54	N/A	N/A
G3	200	162	38	N/A	N/A
G4	400	N/A	N/A	318	82
G5	200	N/A	N/A	185	15
Total	1,988	1,265	123	503	97
Precision					**0.91**
Recall					**0.93**
Accuracy					**0.89**

Table 6: Comparison results of insecure posts discovery of Dicos, T_{and} [7], and T_{cpp} [23].

Category	Android posts (up to Mar. 2016)		C/C++ posts (up to Sep. 2018)	
	Dicos	T_{and} [7]	Dicos	T_{cpp} [23]
#Discovered insecure posts	2,454	278	7,241	69
#Discovered insecure posts (containing change history)	2,454	62	7,241	36
#Commonly discovered insecure posts	50		22	
Coverage of the other tool's results (containing change history)	**0.81** (50/62)	**0.02** (50/2,454)	**0.61** (22/36)	**0.003** (22/7,241)

thus C/C++ cases showed slightly better accuracy than Android cases.

Although Dicos precisely discovered insecure posts in most cases, it reported several false results. We found that the main reasons for FPs are variable name changes in the conditional statements, and comments that incorrectly reported that a code snippet was insecure (*e.g.*, the comment of post #2736841[6] contained the following message: "you have a memory leak ... my apologies, I am mistaken."). The cause of FNs is when a security patch pattern other than the selected features is discovered in the code snippet (*e.g.*, a type-casting-related vulnerability), without mentioning any security-related keywords. In addition, when the security issue was resolved by changing only the data flows, Dicos produced false negatives. Strict feature selection to reduce FPs will increase FNs, and selecting more features to reduce FNs will cause more FPs. We believe that the features used by Dicos work effectively and maintains a good balance in terms of precision and recall.

5.3 Comparison with the existing approach

In this part, we compared the insecure code snippet discovery results of Dicos with those of existing approaches [7, 23], to demonstrate the effectiveness of Dicos.

Tool selection. We reviewed several approaches that discovered insecure code snippets from Stack Overflow [5, 7, 8, 23, 32, 34]. For the in-depth comparison, we should be able to use their tools or experimental results. Thus, we excluded existing approaches that do not publicly provide tools or experimental results [5, 32, 34] (despite our requests). Finally, we decided to compare the discovery results of Dicos to that of the following approaches: Fischer *et al.* [7] (T_{and}), which attempted to discover Android insecure posts, and Verdi *et al.* [23] (T_{cpp}), which attempted to discover C++ insecure posts from Stack Overflow.

Methodology. We first examine the total number of insecure posts discovered by each approach (*i.e.*, Dicos, T_{and}, and T_{cpp}). Then, we investigate the coverage of Dicos to their discovery results (*i.e.*, the number of commonly discovered insecure posts); here we only consider the posts to which the methodology of Dicos can be applied (*i.e.*, the posts should contain at least one change history). The experimental results are presented in Table 6.

Comparison results (Android). T_{and} [7] examined 1,165,350 Android answer posts in Stack Overflow (datasets up to March 2016), and then discovered 420 insecure code snippets from **278 unique posts** of which 62 posts containing change history. When we applied Dicos to the same dataset, Dicos discovered **2,454 insecure posts**, which is *9 times more* than reported by T_{and}.

Among the 62 insecure posts containing change history discovered by T_{and}, Dicos discovered 50 out of 62 posts (81%) as insecure; it is worth noting that T_{and} covered only 2% of the Dicos discovery results. The 12 posts that Dicos did not discover are either the latest revision of the post remains in a vulnerable state (*i.e.*, there is no security patch in the change history) or only one feature is detected (*i.e.*, security-sensitive APIs) in the post.

Because T_{and} only utilizes security-sensitive APIs while Dicos additionally considers security-related keywords and control flow changes along with the security-sensitive APIs, there was a considerable difference in the number of discovered insecure posts. The comparison results affirmed that the approach of Dicos, which considers the combination of three effective features, could discover more insecure posts in a wider range than the existing approach considering only security-sensitive APIs.

Comparison results (C/C++). T_{cpp} [23] reviewed 72,483 C++ code snippets in Stack Overflow (datasets up to September 2018), and then discovered 99 insecure code snippets from **69 unique posts** of which 36 posts containing change history. We confirmed that Dicos discovered **7,241 insecure posts** in the same dataset, which is 105 times more than discovered by T_{cpp}.

When T_{cpp} attempted to discover insecure code snippets, they manually analyzed code snippets based on the selected CWE types, such as CWE-476 (*i.e.*, null pointer dereference). Because their approach relied on manual analysis and considered only a few CWE types, they failed to discover many insecure code snippets that are actually prevalent on Stack Overflow.

In contrast, Dicos showed substantially better discovery coverage than the existing approach. Among the 36 posts with change history discovered by T_{cpp}, we confirmed that Dicos could discover 22 of them (61%) as insecure posts. The 14 posts that Dicos failed to

[6] https://stackoverflow.com/questions/2736841

discover did not show a specific pattern that could be determined as insecure posts, similar to the cause of FNs of Dicos, as mentioned in subsection 5.2. The results demonstrated that Dicos, an effective feature-based insecure code snippet discovery approach, can discover insecure code snippets more widely and accurately than the existing approach.

5.4 Effectiveness of the utilized techniques

In subsection 4.1, we introduced the code snippet pairing technique utilized in Dicos, which pairs the most similar two code snippets between two post revisions.

To demonstrate its effectiveness, we compared the result of our proposed pairing technique with the result of simple pairing technique based on the code snippet number; notably, sequential numbers were assigned to all code snippets in each post of the collected dataset. Hence, we assume that the simple pairing technique pairs the code snippets with the same number between two post revisions. We randomly selected 2,000 C/C++ insecure posts, and then measured the similarity scores using the Jaccard index [26] between code snippet pairs detected by each technique, respectively. The results are illustrated in Figure 5.

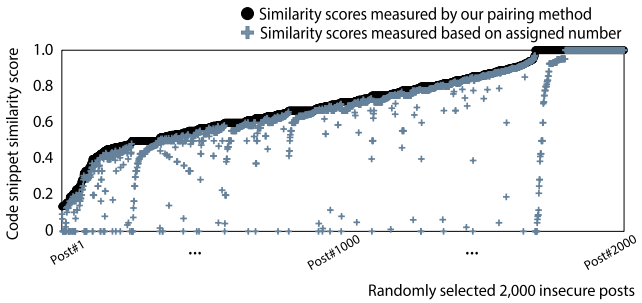

Figure 5: Similarity scores measured by (1) our proposed code snippet pairing method and (2) simple method based on the assigned sequential numbers of code snippets.

It is worth noting that the similarity scores detected by our proposed pairing technique showed a much higher similarity score than that of the simple sequential number-based pairing technique. For 222 posts, both techniques provided the same similarity score. However, for the remaining 1,778 posts, our pairing technique provided greater similarity scores than that of the simple pairing technique. This suggests that, even though there are more similar code snippet pairs, pairings can be made between completely different code snippets if they are simply paired based on a sequential number. In conclusion, our pairing method, which pairs the two code snippets with the highest similarity, is effective in the situation that the order and number of code snippets are frequently changed during the post update process.

5.5 Performance of Dicos

In our setup, it took a total of 20 hours to download the Google BigQuery dataset, select posts with change history, and extract the oldest and latest revision of each post. In addition, it took a total of ten days to discover insecure posts; this includes the time needed to extract the change history from a post, check whether the three selected features are contained in the change history of the collected 668,520 posts, and determine whether a post is insecure. On average, Dicos took approximately 1.4 s to determine whether a post contains an insecure code snippet, which is sufficient to discover insecure code snippets using a large-scale dataset.

6 FINDINGS

From our experiments, we confirmed Dicos discovered 14,719 insecure code snippets from 12,458 posts. In this section, we provide the analysis results related to the following four questions:

- **Q1.** Are *older* posts more likely to provide *insecure* code snippets? (subsection 6.1)
- **Q2.** Are *accepted* answer posts *more secure* than non-accepted posts? (subsection 6.2)
- **Q3.** What *types* of insecure code snippets were discovered? (subsection 6.3)
- **Q4.** What is the *status of reusing insecure code snippets* in popular open-source software? (subsection 6.4)

6.1 Creation time of a post and vulnerabilities

To answer the first question, we analyzed the correlation between the post creation time and the security of the post. In particular, we examined the year distribution of secure and insecure posts from 2008 to 2020. The results are depicted in Figure 6.

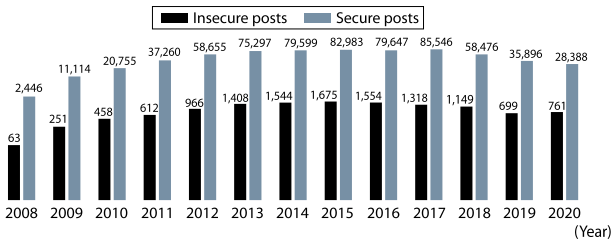

Figure 6: Year distributions of secure and insecure posts discovered by Dicos (logarithmic scale).

Interestingly, we confirmed that the proportion of insecure posts accounted for approximately 2% each year regardless of how old or new the post is. In other words, our experimental results implies that there is no clear correlation between post creation time and security. Incidentally, the fact that insecure posts are constantly being uploaded to Stack Overflow suggests the need for an automated approach that can accurately discover insecure posts on Stack Overflow, such as Dicos.

6.2 Acceptance of a post and security

In general, users are more likely to think that accepted answers are more reliable (*e.g.*, secure) than non-accepted answers [29]. To answer such a common thought, we investigated the relations between the acceptance of an answer post and its security. Figure 7 illustrates the results.

From the results, we confirmed that the ratio of insecure posts was almost the same between accepted (1.67%) and non-accepted (1.99%) posts. This, presumably, occurred because the security of

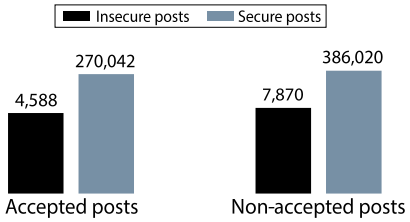

Figure 7: Ratio of insecure posts between accepted and non-accepted posts discovered by Dicos (logarithmic scale).

the code snippet did not need to be considered in the process of accepting an answer post by the questioner. The results affirmed that developers need to verify a code snippet even from an accepted post whether it contains a insecure code snippet; Dicos can help this verification process.

6.3 Types of insecure code snippets

For 788 insecure posts with all three selected features, we manually examined the types of insecure code snippets (*i.e.*, 880 code snippets). This task was mainly conducted on the discovered security-related keywords, and additionally, we referred to user comments, descriptions of the answerer, and the code change history. Figure 8 illustrates the examined results.

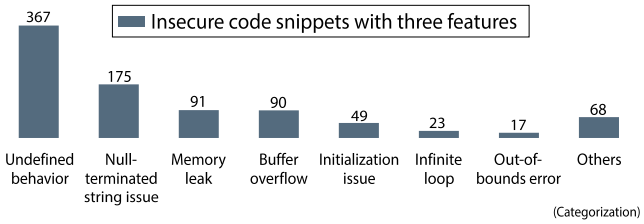

Figure 8: Types of discovered insecure code snippets.

The most prevalent type of insecure code snippets was `undefined behavior`, accounting for 42% of the total; the representative example was introduced in subsection 2.2. Perhaps, given the nature of Stack Overflow to answer specific questions, answerers focus more on the functional aspects of their code snippets, not consider all possible exceptions. For this reason, we determined that undefined behavior accounted for the highest proportion. In addition, various types of insecure code snippets existed, such as null-terminated string issues, memory leaks and buffer overflows. This result suggests the following fact: unlike previous approaches that could only discover specific types of insecure code snippets, Dicos can discover various types of insecure code snippets with the help of three effective features.

6.4 Reusing insecure code snippets in the wild

Finally, to answer the last question (Q4), we investigated how widespread the insecure code snippets discovered by Dicos are in real-world popular open-source software (OSS) projects.

Methodology. As the searching target pool, we collected the latest versions of 2,000 popular C/C++ OSS projects from GitHub (ranked by the number of stars), including OS (*e.g.*, Linux), databases (*e.g.*, Redis), and media (*e.g.*, FFmpeg) related projects in May 2021. We decided to leverage the existing large-gap code clone detection technique (*i.e.*, SourcererCC [21]); this is because we decided that a large-gap code clone detector would be effective as many parts of the insecure code snippets (*e.g.*, variable names) could be changed during the code reuse process. However, as SourcererCC is a code clone detection technique, not for the vulnerable code clones, it can report several false results [11]. Therefore, we manually analyzed the primary detection results and only considered the case where the insecure code snippet is actually propagated to popular OSS projects. We applied the same experimental setting for SourcererCC that used in their paper (*i.e.*, the θ is selected as 0.8).

Detection result. Consequently, we confirmed that 27 insecure code snippets, discovered by Dicos, were reused in the latest versions of **151 popular OSS projects (8%)**. Of these, the most notable example related to insecure code snippet reuse is the Linux kernel case, which is introduced in subsection 2.2. For the cases of discovered dangerous code snippets that can affect the security of the entire software, we have reported to the corresponding vendors[7]. Our experimental results indicate that: (1) code snippets are actually reused in various OSS projects, and (2) a considerable number of insecure code snippets are included in the latest version of popular OSS projects. Reusing insecure code snippets can open up an attack vector in the affected software. As the first step in the prevention of such undesirable situations, we can apply Dicos for more attentive insecure code snippet detection.

7 RELATED WORKS

In this section, we introduce a number of related works.

Discovering insecure code snippets. Fischer *et al.* [7] provided a security analysis for security-related Android code snippets. They manually checked whether the code snippet was insecure, and demonstrated the impact of the use of insecure code snippets in real-world Android applications. Yanfang *et al.* [32] proposed ICSD, a tool for detecting insecure posts based on the utilized APIs, methods, and social coding properties of each post. Zhang *et al.* [34] analyzed potential API usage violations in the code snippets. Chen *et al.* [5] labeled Android code snippets as secure or insecure, and analyzed their distributions, such as their view counts or the number of duplicates. Finally, Verdi *et al.* [23] detected C++ insecure code snippets in Stack Overflow based on manual inspection.

However, existing approaches are not efficient in discovering insecure code snippets in terms of discovery coverage. They cannot discover insecure code snippets other than Java or Android (*i.e.*, language-restricted) and that are not related to security-sensitive APIs (*i.e.*, feature-restricted). In addition, because they do not consider the semantics of the code, their results have many FPs and FNs (*i.e.*, low accuracy) [30]. Furthermore, unless insecure code snippets are discovered in an automated manner, an approach is not suitable for checking for Stack Overflow posts that are constantly being added.

[7]Since we have not received confirmation from most vendors, we omit the detailed OSS list with insecure code snippets.

Detecting unknown security patches. Perl *et al.* [18] proposed `VCCFinder` to identify potentially vulnerable commits using coarse-grained feature set. Their features focus on changes from commits or repository metadata rather than changes in the source code. Wang *et al.* [25] identified unknown security patches from code commits with the finer-grained feature set (*e.g.*, considering 61 features). Machiry *et al.* [15] proposed `SPIDER` to analyze safe patches by considering control flow changes. Kangjie *et al.* [14] proposed `CRIX` to detect missing-check bugs in kernels. However, these approaches may yield false alarms owing to the ineffective feature selection. This is because, they consider only the fragmentary characteristics of the security patch (*e.g.*, control flow changes [15]), are focused more on external characteristics rather than the code itself [18], select too general and excessive features that are also prevalent in general code patches [25], and are applicable only in limited environment settings [14].

8 DISCUSSION

In this section, we discuss several considerations related to Dicos.

The number of used features. Part of the conclusion is that considering multiple features is more accurate than considering a single feature in discovering insecure code snippets. The G4 group in subsection 5.2 is composed of 200 posts containing only one of each feature. When Dicos used a single feature to discover an insecure post, the precision was as follows: 25% for *F1*, 27% for *F2*, and 27% for *F3*. By contrast, when Dicos considered multiple features to discover an insecure post, the precision was much higher than when using a single feature (*e.g.*, 96% for three features, see Table 5). A single feature-based approach not only fails to discover insecure code snippets where patch patterns other than the selected features occur but can also lead to a misinterpretation of a secure code snippet as being insecure. Conversely, the approach of selecting too many features and determining whether the selected many features appear simultaneously in a code snippet yields a false negative. Therefore, we determined that the approach of Dicos maintains a good balance in terms of discovery precision and recall.

Practical usage: porting to Stack Overflow. Currently, Stack Overflow does not provide any notification or information about insecure code snippets. In this situation, the information in the insecure code snippet discovered by Dicos can be used in the following two ways. First, whenever a change occurs after a post is uploaded to Stack Overflow, the Dicos can be used to verify that the change is intended to resolve a security issue; if the change is a security patch, Stack Overflow can add a mark to the post such as a security warning. Next, among registered posts, Stack Overflow can provide the related information on insecure posts (*e.g.*, patch information), discovered by Dicos, in the form of a database. By providing the insecure post information as such ways, Stack Overflow can provide more secure code snippets and increase credibility, and users can avoid reusing insecure code snippets in their software.

Limitations and future work. First, although Dicos showed a wider discovery coverage than previous approaches by utilizing a combination of effective features, it is still difficult to discover insecure code snippets that appear with patterns other than the selected features. Second, some of the insecure code snippets we found on Stack Overflow and real world OSS projects could be triggered, but not all. We are attempting to trigger the discovered insecure codes by referring to related approaches (*e.g.*, [12]) and websites (*e.g.*, [22]), and will contribute to the security of the OSS ecosystem by reporting threatening insecure codes that can be triggered. Finally, our approach can be applicable to other online Q&A fora such as *Quora* and *MSDN forum*. Since Dicos discovers insecure posts by analyzing the post change history, it can be applied to other Q&A fora that provide post change history; for example, as *Quora* provides the change history via `Post log`, and thus, Dicos can be applied to discover insecure posts within *Quora*.

9 CONCLUSION

Reusing code snippets from online Q&A fora such as Stack Overflow can cause security problems when developers reuse the code snippets without fully understanding the code implications. In response, we presented Dicos, which is an accurate approach by examining the change history of Stack Overflow posts for discovering insecure code snippets. We confirmed that Dicos successfully discovered 14,719 insecure code snippets from 1,958,283 Stack Overflow posts with 91% precision and 93% recall. Equipped with insecure code snippet discovery results from Dicos, the credibility of Stack Overflow can be improved by addressing discovered insecure code snippets, and further, this can create a safe code snippet reuse environment. The source code of Dicos is available at https://github.com/hyunji-hong/DICOS-public.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers and our shepherd for their valuable comments to improve the quality of the paper. We also thank you for the dedicated help of program chairs. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security, No.2019-0-01343 Regional Strategic Industry Convergence Security Core Talent Training Business, and No.IITP-2021-2020-0-01819 ICT Creative Consilience program).

REFERENCES

[1] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 289–305.

[2] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. 2018. SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts. In *Proceedings of the 15th international conference on mining software repositories*. 319–330.

[3] BetterProgramming. 2020. Why Code Snippets From Stack Overflow Can Break Your Project. https://betterprogramming.pub/why-code-snippets-from-stack-overflow-can-break-your-project-ced579a48ddb

[4] Aaditya Bhatia, Shaowei Wang, Muhammad Asaduzzaman, and Ahmed E Hassan. 2020. A Study of Bug Management Using the Stack Exchange Question and Answering Platform. *IEEE Transactions on Software Engineering* (2020).

[5] 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)*. IEEE, 536–547.

[6] Ctags 2021. *Universal Ctags*. Ctags. https://github.com/universal-ctags/.

[7] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In *2017 IEEE*

Symposium on Security and Privacy (SP). IEEE, 121–136.
[8] Felix Fischer, Huang Xiao, Ching-Yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. 2019. Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography. In *2019 28th USENIX Security Symposium (Security)*. 339–356.
[9] Md Rakibul Islam and Minhaz F Zibran. 2021. What changes in where?: an empirical study of bug-fixing change patterns. *ACM SIGAPP Applied Computing Review* 20, 4 (2021), 18–34.
[10] Seulbae Kim and Heejo Lee. 2018. Software Systems at risk: An empirical study of cloned vulnerabilities in practice. *Computers & Security* 77 (2018), 720–736.
[11] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.
[12] Seongkyeong Kwon, Seunghoon Woo, Gangmo Seong, and Heejo Lee. 2021. OCTOPOCS: Automatic Verification of Propagated Vulnerable Code Using Reformed Proofs of Concept. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 174–185.
[13] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.
[14] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *2019 28th USENIX Security Symposium (Security)*. 1769–1786.
[15] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1562–1579.
[16] Microsoft. 2019. Microsoft Build C6328. https://docs.microsoft.com/en-us/cpp/code-quality/c6328?view=msvc-160
[17] MITRE. 2021. CWE-676: Use of Potentially Dangerous Function. https://cwe.mitre.org/data/definitions/676.html
[18] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *2015 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 426–437.
[19] Google Cloud Platform. 2021. Google Bigquery StackOverflow Data. https://cloud.google.com/bigquery/public-data.
[20] Reddit. 2018. Docker for Windows won't start if Razer Synapse 3 is running. https://www.reddit.com/r/docker/comments/815l9n/docker_for_windows_wont_start_if_razer_synapse_3/
[21] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big Code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 1157–1168.
[22] Offensive Security. 2021. Exploit Database. https://www.exploit-db.com/.
[23] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples. *IEEE Transactions on Software Engineering* (2020).
[24] Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. 2018. How Do Users Revise Answers on Technical Q&A Websites? A Case Study on Stack Overflow. *IEEE Transactions on Software Engineering* 46, 9 (2018), 1024–1038.
[25] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. 2019. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 485–492.
[26] Wikipedia. 2021. Jaccrd index. https://en.wikipedia.org/wiki/Jaccard_index.
[27] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *2021 30th USENIX Security Symposium (Security)*. 3041–3058.
[28] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 860–872.
[29] Yuhao Wu, Shaowei Wang, Cor-Paul Bezemer, and Katsuro Inoue. 2019. How do developers utilize source code from stack overflow? *Empirical Software Engineering* 24, 2 (2019), 637–673.
[30] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *2020 29th USENIX Security Symposium (Security)*. 1165–1182.
[31] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy (SP)*. IEEE, 590–604.
[32] Yanfang Ye, Shifu Hou, Lingwei Chen, Xin Li, Liang Zhao, Shouhuai Xu, Jiabin Wang, and Qi Xiong. 2018. ICSD: An Automatic System for Insecure Code Snippet Detection in Stack Overflow over Heterogeneous Information Network. In *2018 Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*. 542–552.
[33] Haoxiang Zhang, Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. 2021. Are Comments on Stack Overflow Well Organized for Easy Retrieval by Developers? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–31.
[34] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are Code Examples on an Online Q&A Forum Reliable?: A Study of API Misuse on Stack Overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.
[35] Tianyi Zhang, Di Yang, Crista Lopes, and Miryung Kim. 2019. Analyzing and Supporting Adaptation of Online Code Examples. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 316–327.

A SECURITY-SENSITIVE API

Table 7 and Table 8 show lists of security-sensitive APIs utilized by Dicos. The APIs were selected by referring to existing approaches [5, 7, 34] and the CWE-676 document [17].

Table 7: Security-sensitive APIs for C/C++ posts.

C/C++ security-sensitive APIs
`strcpy, strncpy, strcat, strncat, system, memcpy, memset, malloc, gets, vfork, realloc, pthread_mutex_lock, free, chroot, strlen, vsprintf, sprintf, scanf, fscanf, sscanf, vscanf, vsscanf, vfscanf, snprintf, atoi, strtok, strcmp, strncmp, strcasecmp, strncasecmp, memcmp, signal, va_arg.`

Table 8: Security-sensitive APIs for Android posts.

Android security-sensitive APIs
`hostnameverifier, trustmanager, sslcontext, cipher, webview, setseed, messagedigest, secretkey, keystore, pbekeyspec, nextbytes, signature, keyfactory, connectionspec, sslsocketfactory, ivparameterspec`

B SECURITY-RELATED KEYWORD

Table 9 lists the selected security-related keywords and their categorization. We selected these security-related keywords by referring to existing approaches [7, 9] and by analyzing commit messages of known CVE patches (see section 3).

Table 9: Selected security-related keywords categorized as nouns, modifiers, and verbs.

Category	Security-related Keywords
Nouns	`vulnerab, fault, defect, sanit, mistake, flaw, bug, infinite, loop, secur, overflow, error, remote, mitigat, realloc, heap, privilege, underflow, attack, DoS, denial-of-service, initiali, xss, leak, patch, authori, corruption, crash, memory, null, injection, out-of-bounds, use-after-free, dereferenc, buffer, hack, segment, authentication, exploit.`
Modifiers	`incorrect, vulnerab, harm, undefine, unpredict, unsafe, secur, malicious, dangerous, critical, bad, unprivileged, negative, stable, invalid.`
Verbs	`flaw, hack, fix, change, modify, exploit, mitigat, leak, realloc, invoke, inject, ensure, reject, initiali, fail, authori, update, attack, trigger, lock, corrupt, crash, prevent, avoid, access, cause, overflow, terminat.`