An Empirical Study of C++ Vulnerabilities in Crowd-Sourced Code Examples

Morteza Verdi, Ashkan Sami , Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh

Abstract—Software developers share programming solutions in Q&A sites like Stack Overflow. The reuse of crowd-sourced code snippets can facilitate rapid prototyping. However, recent research shows that the shared code snippets may be of low quality and can even contain vulnerabilities. This paper aims to understand the nature and the prevalence of security vulnerabilities in crowd-sourced code examples. To achieve this goal, we investigate security vulnerabilities in the C++ code snippets shared on Stack Overflow over a period of 10 years. In collaborative sessions involving multiple human coders, we manually assessed each code snippet for security vulnerabilities following CWE (Common Weakness Enumeration) guidelines. From the 72,483 reviewed code snippets used in at least one project hosted on GitHub, we found a total of 69 vulnerable code snippets categorized into 29 types. Many of the investigated code snippets are still not corrected on Stack Overflow. The 69 vulnerable code snippets found in Stack Overflow were reused in a total of 2859 GitHub projects. To help improve the quality of code snippets shared on Stack Overflow, we developed a browser extension that allow Stack Overflow users to check for vulnerabilities in code snippets when they upload them on the platform.

Index Terms—Stack Overflow, Software Security, C++, SOTorrent, Vulnerability Migration, GitHub, Vulnerability Evolution

1 Introduction

A major goal of software development is to deliver high quality software in timely and cost-efficient manner. Code reuse is an accepted practice and an essential approach to achieve this premise [1]. The reused code snippets come from many different sources and in different forms, e.g., third-party library [2], open source software [3], and Question and Answer (Q&A) websites such as Stack Overflow [4], [5]. Sharing code snippets and code examples is also a common learning practice [6]. Novices and even more senior developers leverage code examples and explanations shared on platforms like Stack Overflow, to learn how to perform new programming tasks or use certain APIs [1], [7], [8], [9]. Multiple studies [10], [11], [12] have investigated knowledge flow and knowledge sharing from Stack Overflow answers to repositories of open source software hosted in GitHub. They report that code snippets found on Stack Overflow can be toxic, i.e., of poor quality, and can potentially lead to license violations [12]. An important aspect of quality that has not been investigated in details by the research community is security. If vulnerable codes snippets are migrated from Stack Overflow to applications, these applications will be prone to attacks.

Most studies published on security aspects of code snippets posted on Stack Overflow focused on Java and Python; overlooking C++ which is the fourth most popular programming language [13]. C++ is the language of choice for embedded, resource-constrained programs. It is also extensively used in large and distributed systems. Vulnerabilities in C++ code snippets are therefore likely to have a major impact. However, to the best of our knowledge, no study has examined the security aspects of C++ Stack overflow code snippets. This paper aims to fill this gap in the literature. More specifically, we aim to understand the nature and the prevalence of security vulnerabilities in code examples shared on Stack Overflow. To achieve this goal, we empirically study C++ vulnerabilities in code examples shared in Stack Overflow along the following two dimensions:

- **Prevalence.** We review the C++ vulnerability types contained in a Stack Overflow data-set named SOTORRENT [14], [15] and analyze their evolution over time; in particular their migration to GitHub projects.
  From 72,483 C++ code snippets reused in at least one GitHub project we found 69 vulnerabilities belonging to 29 different types of vulnerabilities.
- **Propagation.** We investigate how the vulnerable code snippets were reused in GitHub repositories.
  The 69 identified vulnerable code snippets are used in 2589 GitHub files. The most common vulnerability propagated from Stack Overflow to GitHub is CWE-150 (Improper neutralization of space, meta, or control space).

To assist developers in reusing code from stack Overflow safely, we developed a Chrome extension that allow checking for vulnerabilities in code snippets when they are uploaded on Stack Overflow.

M. Verdi is with Shiraz University, Iran. E-mail: m.verdi@shirazu.ac.ir
A. Sami (Corresponding Author) is with Shiraz University, Iran. E-mail: sami@shirazu.ac.ir
J. Akhondali is with Shiraz University, Iran. E-mail: jafar.akhondali@yahoo.com
F. Khomh is with Polytechnique Montreal University, Quebec Canada. E-mail: foutse.khomh@polymtl.ca
G. Uddin is with Polytechnique Montreal University, Quebec Canada. Email: giasu@cs.mcgill.ca
A. Karami Motlagh is with Chamran University, Iran. E-mail: alireza.karami.m@gmail.com

TABLE 1: Research contributions made in this paper to understand The *prevalence* and *propagation* of C++ vulnerabilities in crowd-Sourced code examples

Type	Research Contribution	Research Advancement
Prevalence: Empirical Evidence from Stack Overflow	Evidence of the Prevalence of Vulnerable C++ Code in Stack Overflow. We analyzed C++ code snippets contained in answers posted on Stack Overflow and identified the vulnerabilities that they contain.	Reusability in software is a very high-profile issue, which has been highlighted by many studies [1] [2] [3] [4], [5] [6]. In this study, we try to highlight the programming language of C++ and its use in Software environment, because C++ language is still widely used in large industrial systems [16], [17]. Our study has been conducted to improve the quality of C++ code snippets shared on online forums like Stack Overflow.
Propagation: Empirical Evidence from Stack Overflow and GitHub	Evidence of the propagation of Vulnerable C++ Code snippets from Stack Overflow to GitHub Repositories. We tracked all the vulnerable C++ code snippets found on Stack Overflow to their reusing projects on GitHub. We conducted a survey of GitHub developers who copied Vulnerable Code from Stack Overflow to their GitHub repositories.	One of the challenges in software is the reuse of vulnerable codes to accelerate the development of a software product, which ultimately leads to a decrease in software quality [8], [9]. Through this study, we aim to inform programmers and developers about the risks of reusing vulnerable code snippets from Stack Overflow.

The remainder of this paper is organised as follows. Section 2 provides background information about code reuse and discusses the related literature. Section 3 introduces our research questions and data collection and data processing. Section 4 and Section 5 discusses obtained results, while Section 6 discusses the implications of our findings. Section 7 discusses threats to the validity of our study and Section 8 concludes the paper; outlining some avenues for future works.

2 Background and Related Work

In this section, we provide background information about security vulnerabilities and review the related literature.

2.1 CWE (Common Weakness Enumeration

CWE is a community-developed list of common software security weaknesses. It serves as a common reference, a measuring stick for software security tools, and as a baseline for weakness identification, mitigation, and prevention efforts. It is regarded as an universal online dictionary of weaknesses that have been found in computer software. The purpose of CWE is to facilitate the effective use of tools that can identify, find and resolve bugs, vulnerabilities and exposures, in computer software before the programs are distributed to the public.

2.2 Reusing of Code Shared in Stack Overflow

Stack Overflow is regarded as the most popular question and answer website for software developers [15]. Software developers benefit from SO posts, while programming [8], [12], [18], [19], [20], and read about the technologies and tools needed for development [21], [22], [23]. Thus, research on Stack Overflow is of high importance in software community.
Developers create and maintain software by standing on the shoulders of others [24]; they reuse components and libraries, and mine the Web for information that can help them in their tasks [25]. For help with their code, developers often turn to programming question and answer (Q&A) communities, most visible of which is Stack Overflow [26], [27].

Xia et al. [11] show that a large number of open source systems reuse outdated third-party libraries which can lead to harmful effects to the software because they may introduce security flaws in the software.

Abdalkareem et al. [1] examined F-Droid repositories, and identified clones between Stack Overflow posts and Android apps. They observed that copied code from SO posts can have an adverse effect on the quality of applications.

Yang et al. [10] analyzed 909k non-fork Python projects hosted on GitHub, which contain 290M function definitions, and 1.9M Python snippets captured in Stack Overflow and performed a quantitative analysis of block-level code cloning intra and inter Stack Overflow and GitHub.

Akanda Nishi et al. [28] studied code duplication between two popular sources of software development information: the Stack Overflow Q&A site and software development tutorials, to understand the evolution of duplicated information overtime.

An et al. [20] investigated clones between 399 Android apps and Stack Overflow posts. They found 1,226 code snippets which were reused from 68 Android apps. This reused of code snippets resulted in 1,279 cases of potential license violations.

2.3 Security challenges of Stack Overflow code snippets

Several studies have reported the presence of insecure code in highly up-voted, and accepted answers on Stack Overflow [8], [29], [30]. However, these studies did not investigated C++ code snippets. Yet, C++ is the fourth most popular programming language. The use of insecure code snippets has been linked to multiple software attacks in which user credentials, credit card numbers, and other private information were stolen [31]. C++ is reported to

TABLE 2: Comparison between our study and prior studies

Theme	Our Study	Prior Study	Comparison
Reusability of C++ Posts in Stack Overflow	In this study we investigated the reusability of C++ code snippets from Stack Overflow answer posts to GitHub projects.	Reusability in Stack Overflow contain copying code in other open sources application, license violation in Stack Overflow and use in open source projects such as GitHub repositories [8] , [19], [25], [26]. The studies [10], [11], [12], [20] included reusability in Java and Android application, [1], [40] in Python, [28] in Third party code, [41] in IDE, [42] in API documentation, [43] in Php.	This study indicates the reusability of Stack Overflow C++ codes to GitHub projects and the prevalence of these codes in GitHub repositories that until now not mentioned.
Security of C++ Posts in Stack Overflow	Software security is a very broad and, at the same time, extremely difficult to detect specially for C++ programming language. We analyze C++ code snippets in Stack Overflow answer posts.	Studies [8], [29] in java Script and android application, [7], [30], [35] in java and [23] in python showed the Stack Overflow have a security vulnerability in their code snippets that uses in applications, open source projects, and APIs.	We carefully scrutinized the vulnerability and expressed each of them with a CWE vulnerability label of the C++ code in Stack Overflow answer posts so far no study has been conducted in this area.
Security in GitHub Repositories	This study shows all possible vulnerable C++ GitHub projects used Stack Overflow vulnerable code snippets.	Study about analysis of security in GitHub projects contain secure coding, sentiment analysis, security issues [36], [37], [38], [39].	No studies exists specifically addresses the C++ vulnerable GitHub projects that migrated from Stack Overflow codes.

be prone to misuses (e.g., memory corruption bugs) that can easily lead to vulnerable code and exploitable applications [32], [33], [34].
Fischer et al. [8] found insecure code snippets from Stack Overflow copied into 196,403 Android applications, published on Google Play.
Zhang et al. [35] investigated the quality of Stack Overflow code snippets by examining the misuse of API calls. They reported that approximately 31% of their analysed code snippets possibly incorporate API misuses that could lead to failures and–or resource leakages.

2.4 Security issues in GitHub

Rahman et al. [36] detected seven types of security smells that are indicative of security weaknesses in IaC scripts and identified 21,201 occurrences of security smells that include 1326 occurrences of hard-coded passwords.
Zahedi et al. [37] examined issue topics in GitHub repositories and found that only 3% of them were related to security. The majority of these security issues were cryptography issues.
Pletea et al. [38] examined security-related discussions on GitHub, and report that they represent approximately 10% of all discussions on GitHub. They also report that security related discussions are often associated with negative emotions.
Acar et al. [39] conducted an experiment with active GitHub users to examine the validity of recruiting convenience samples in security-related study. They observed that neither the self-reported status of participants (i.e., as student or professional developers) nor the security background of the participants correlated with their capacity to complete security tasks successfully.

3 Research Questions and Data Collection

3.1 Research Questions

We explore the following Research Questions (RQs):
RQ1: How prevalent are C++ vulnerabilities in Stack Overflow code snippets?
Previous work on other programming languages revealed the existence of vulnerable code in Stack Overflow [7] [44]. To understand the existence and distribution of insecure C++ codes in Stack Overflow, we reviewed and analyzed C++ answer posts throughout the ten years of Stack Overflow history stored in SOTorrent that had link to GitHub projects.
RQ2: How are the vulnerable C++ code exampled shared in Stack Overflow reused in GitHub repositories?
Knowledge sharing by code reuse routinely occurs between Stack Overflow and GitHub. The effect of vulnerable code migration to GitHub projects have not been investigated in any programming languages so far. Detected C++ vulnerable Stack Overflow code snippets might have migrated to GitHub and ended up deployed in the field. This research question aims to examine the extent of this phenomenon.

3.2 Data collection

In this section, we describe the data collection and analysis approaches that we used to answer our two research questions. Figure 1 shows a general overview of our data processing approach. We will describe each step in our data processing approach. The corresponding data and scripts are available at [45].

To study Stack Overflow posts evolution and their relation with GitHub, SOTorrent data-set Version 2018-09-23 has been used. This version of SOTorrent contains posts from 2008 until 2018. In total there are 41,472,536 question and answer posts and 109,385,095 post version with 206,560,269 post block versions containing 6,039,434 links to software projects. There are 3,861,573 links to public GitHub repositories.

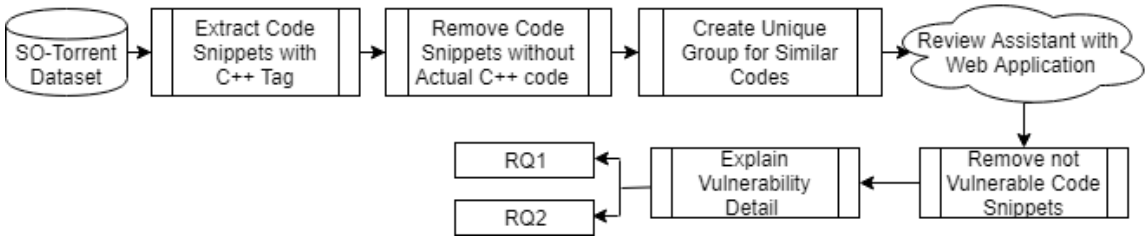


Fig. 1: Overall steps in case study

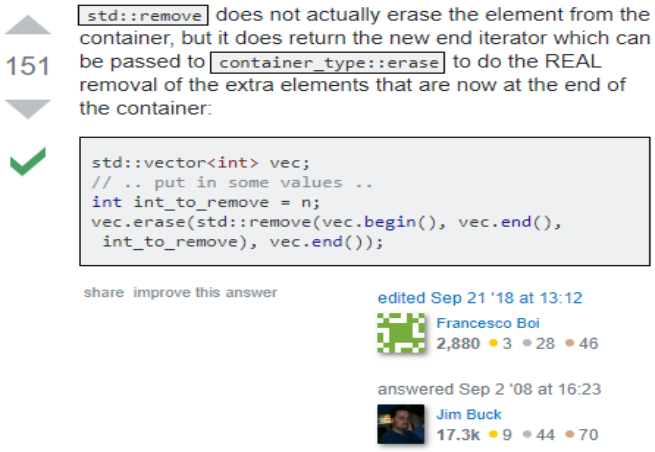


Fig. 2: Structure of SOTorrent posts

SOTorrent provides access to the version history of Stack Overflow content for ten years. As shown in Figure 2, whole post, individual text or code snippet can be accessed independently. According to Figure 3, SOTorrent connects code snippets from Stack Overflow posts to other platforms by aggregating URLs from surrounding text blocks and comments and by collecting references from GitHub files to Stack Overflow posts [14], [15].

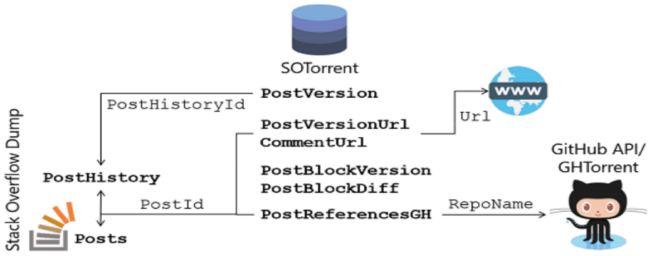


Fig. 3: Connection of SOTorrent table to other resources

3.3 Data Preprocessing

The total count of questions with C++ tag was 583,415 out of 16,389,567. These questions had 1,074,990 answers, including previous and modified versions. A total of 1,738,346 version histories existed. As shown in Figure 5, at least 50% of the posts in the Stack Overflow only had one edit in their context.

Our study aimed to analyze code snippets that migrated to GitHub; therefore, answers without code snippets were removed. Answers with one or more code snippets summed up to 1,032,696. The count of answers that migrated to GitHub and had C++ tag was 1,770. These 1,770 C++ answers have been referenced in 14,779 GitHub files. These 14,779 references came from 8,172 unique projects. The distribution of GitHub files in GitHub projects is shown in Figure 4. A vulnerability might exist in older versions of a code snippet in an answer, and a developer might copied that vulnerable code snippet into a GitHub file at that time. So we had to also analyze older versions of code snippets too. Including older versions of the code snippets, of these answers there were 121,892 possible cases of migrations from code snippets to GitHub.

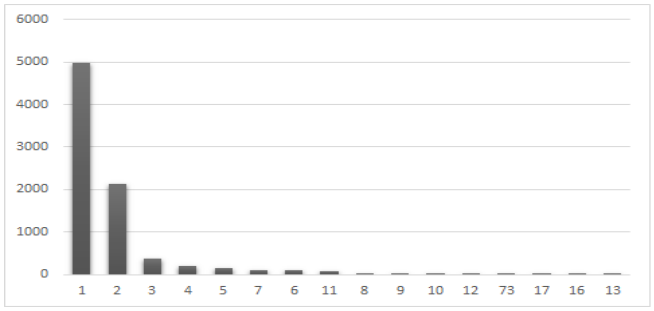


Fig. 4: Distribution of GitHub URLs in GitHub projects

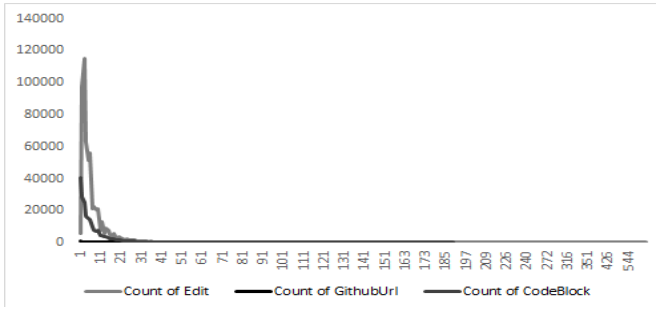


Fig. 5: Count of Edits, GithubUrls and Code Blocks for answers post

3.4 Data cleaning

Not all code snippets in SOTorrent were actually C++ codes. Figure 7 shows a tagged code snippet supposed to be C++. Other examples of pseudo codes or plain texts tagged as code snippet could be found.

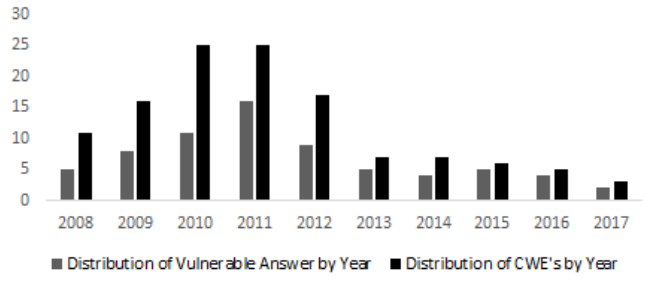


Fig. 6: Distribution of CWE's and Vulnerable Answers by year

Add a list of your source files:

```
SOURCES = $(wildcard $(SRC)/*.cpp)
```

and a list of corresponding object files:

```
OBJS = $(addprefix $(TGT)/, $(notdir $(SOURCES:.cpp=.o)))
```

and the target executable:

```
$(TGT)/myapp: $(OBJS)
    $(CXX) $(LDFLAGS) $(OBJS) -o $@
```

Fig. 7: Example of code snippet in answer with no real c++ code, but only configuration of 'makefile' - Answer id 13109884

Syntaxnet, a natural language processing tool was used to detect code snippets that had actual C++ codes. Syntaxnet is one of the most accurate parsers available [46]. The main difference between Syntaxnet and other NLP tools is that Syntaxnet does not use the meaning of the sentence, but also considers the words independent of each other. Among 121,892 possible code snippets only 72,483 code snippets were actually C++ code snippets included in 1,325 answers.

At SOTorrent, each change in question or answer in Stack Overflow is stored as a set of records but links to GitHub projects are only provided just at the answer level. Therefore, we must review all the code snippets within SOTorrent for vulnerabilities but to investigate the migration we have to follow the link to GitHub at the answer level of the code snippet. SourcererCC [47] was used to find cloned code snippets with exact similarity (Type-1 clone) and 2,056 different sets of similar code snippets within SOTorrent were identified. This makes vulnerability assessments more efficient. On the other hand, when a vulnerability is found in a clone, all the similar clones have the same vulnerability. Afterwards, we can find out possible dangerous migrations in links to GitHub projects provided at the answer level of all the similar code snippet.

4 Prevalence of C++ Vulnerabilities in Stack Overflow Code Examples (RQ1)

4.1 Approach

In order to make the review process more efficient and systematic we created a web application having a simple interface with language-specific syntax highlighting. The web-based review application could mark code snippets as vulnerable, assign one or more CWE tags for each code snippet and view all similar code of a same answer at once.

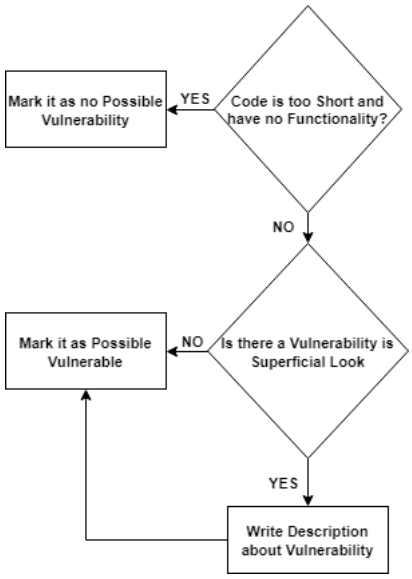


Fig. 8: Flowchart of code reviewing in first step

Three experienced master students (first, third and sixth authors) in C++ security issues were chosen to review each code snippets. As shown in previous section, only 2,056 unique code snippets needed to be reviewed. The reviewers tagged the vulnerabilities by appropriate CWE's.
At the first step of manual inspection process, the goal was to reduce data-set size without losing accuracy. Thus, all code snippets that were certainly not vulnerable were removed, As shown in Figure 8, our three reviewers inspected the code snippets and marked any possible vulnerable code snippet as vulnerable code. If a vulnerability within a code snippet was noticeable within first round of review, They would write a short description explaining why they thought the code snippet might be vulnerable. The specific steps that were followed are described below. This process took 868 hours. In contrast, code snippets not having a specific functionality or which were only used for teaching purposes (and did not have vulnerabilities) were removed. During the review process, reviewers were directly in contact with each other and solved their disagreement through discussions. According to Fleiss' Kappa [48] we calculate Cohen's Kappa [48] score agreement between three raters 0.26 that is a fair agreement. After this first stage of thorough code review, 498 possible vulnerable code snippets were detected. The first round of review was presented to author 2, a professor in software security for validation. A group meeting of 12 graduate students who previously had system and/or software security courses at graduate level finalized the first round of review.
The second round of review process was more robust and followed a self-established guidelines. In order to find vulnerabilities in answers, reviewers needed to indulge deeper into the process and have a better understanding of code snippets and its evolution. Based on knowledge obtained from first round of review, we established a set of guidelines

explained bellow to find as many vulnerabilities in the code snippets as possible and not miss any.

1) **Read the corresponding question to answer with the probable vulnerable code snippet**: To have a better understanding of the reasons why developers shared the code snippet on Stack Overflow.
2) **Read last version of answer, its description and analyze evolution of code over time**: To find out whether the vulnerability has been fixed or evolved within the various versions.
3) **Read comments of answers**: To find out if the vulnerability has been reported through comments of the post. As an example, in Figure 9, 1st and 2nd comments indicated a vulnerability, and 3rd and 4th comments indicated deprecated answer. Source code of answer is also included in listing 1.

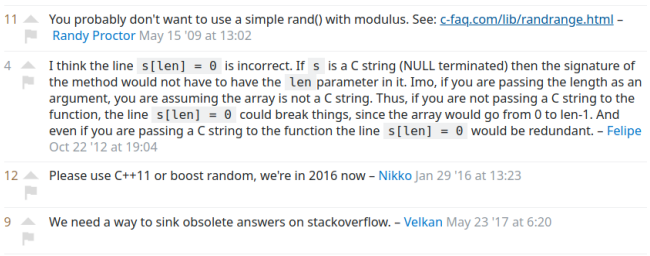


Fig. 9: Comments to vulnerable Answer in Stack overflow with id (440240)

4) **Look for deprecated or dangerous functions in code snippet**: For example 'rand()' function is obsoleted since C++11 [49] and it is not recommended for random-number generation and cryptographic operations.
5) **Check the arguments passed to the functions in the code snippet**: Types of arguments and their values are very important. For example, an out-of-bound large unsigned integer passed to a function that accepts signed integers may interpret the value as a negative number which results in an undefined behaviour or a program crash.
6) **Check function usages based on official documentations**: For referencing and proper documentation of found vulnerabilities, official documentations were extensively used throughout the review process. For example in listing 2, return value of malloc was not checked.
7) **Look for logical vulnerabilities in code snippets**. Usually security is not the first priority of answerers in Stack Overflow. Answerers focus more on functionality than security. For example in listing 5, the goal is to read a vector, but no bounds checking is performed. Using a larger value than index bound can happen either by a programming mistake or could be the doing of an attacker.

After the second round of review process, the identified vulnerable code snippets were confirmed and tagged based on CWE's. One or multiple CWE tag(s) were assigned to each code snippet. These tags allowed us to track the evolution of the security of the code snippets throughout the evolution of Stack Overflow.

4.2 Results

Overall, the distribution of all C++ answers from 2008 to 2018 is shown in Figure 11. If one hypothesizes that Stack Overflow usage reflects the popularity of the programming language, C++ has been the most popular programming language in 2013, and its usage declined after that. The distribution of Stack Overflow answers linked to GitHub projects by year, also shown in Figure 11, again shows that in 2013, C++ had the most migrations to GitHub projects. From our manual reviews of the code snippets, we found 99 vulnerable code snippets residing in 69 answers. By looking at the distribution of vulnerable answers, we find that most vulnerable answers were created in 2011 (as shown in Figure 10). The frequency of CWE's in code snippets is presented in Figure 12. CWE-1006 and CWE-754 are the most frequent ones. The list of CWEs that have been found is explained in Table 3. A complete description of CWE's can be found at [50]. In the following, we present some examples of vulnerabilities found in the inspected code snippets.

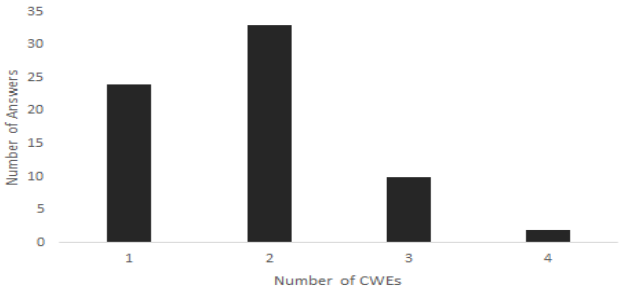


Fig. 10: Frequency of CWEs in Each Answer

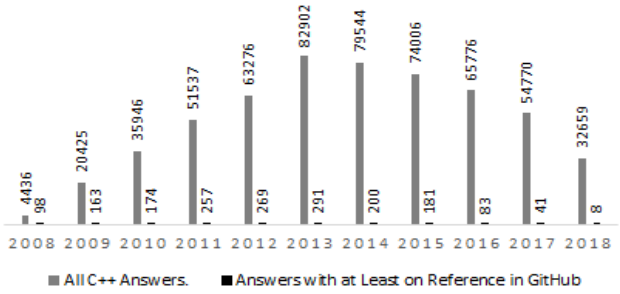
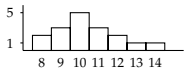
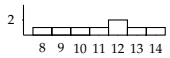
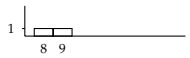
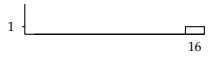
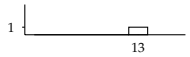
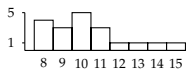
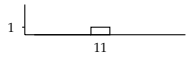
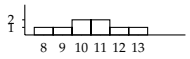
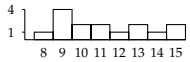
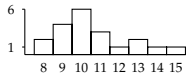
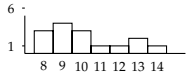
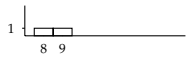
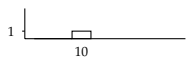
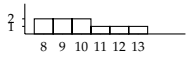
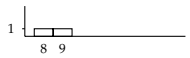
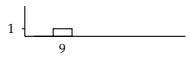
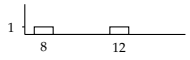
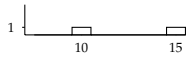


Fig. 11: Distribution of answers in C++ by year

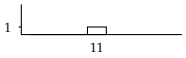
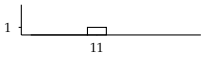
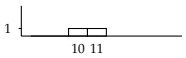
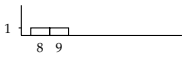
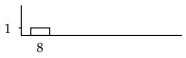
Listing 1: Generate random string in C++ - Answer id 440240 in Stack overflow, shows vulnerability due to use rand function with incorrect using method, (CWE-1006, CWE-477, CWE-193, CWE-754)

```
void alphanum [ gen_random (char *s, const int len)] {
  static const char alphanum[] =
    "0123456789"
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    "abcdefghijklmnopqrstuvwxyz";
    for (int i = 0; i < len; ++i) {
        s[i] = alphanum[ rand()% ( sizeof(alphanum)-1)];
    }
    s[len] = 0;
}
```

TABLE 3: The different types of CWE C++ vulnerabilities and their frequency as we observed in our dataset of Stack Overflow Answers. Each tick in X-axis denotes the last one/two letters of a year, e.g., 8 for 2008 to 16 for 2016.

CWE	Title and Description	Frequency by Year
20	**Improper input validation:** When software does not validate input properly, an attacker is able to craft the input in a form that is not expected by the rest of the application.	
78	**OS command injection:** Constructs all or part of an OS command using externally-influenced input from an upstream component, that could modify the intended OS command when it is sent to a downstream component.	
116	**Improper encoding or escaping of output** A structured message is prepared to communicate with another component, but encoding or escaping of the data is either missing or done incorrectly.	
121	**Stack base buffer overflow** The situation is where the buffer is rewritten in the stack (like, a local variable or, rarely, a parameter to a function).	
125	**Out-of-bounds Read** The software reads data past the end, or before the beginning, of the intended buffer.	
131	**Incorrect calculation of buffer size** Does not correctly calculate the size to be used when allocating a buffer, which could lead to a buffer overflow.	
134	**Use of externally controlled format string** Have been used a function that accepts a format string as an argument, but the format string originates from an external source.	
754	**Improper Check for Unusual or Exceptional Conditions** This vulnerability occurs based on the assumption that events or specific circumstances never happen, such as low memory conditions, lack of access to resources.	
158	**Improper neutralization of null byte or null character** The input is received from a upstream component, but it does not neutralize or incorrectly neutralizes when null bytes are sent to a downstream component.	
190	**Integer overflow or wraparound** Perform a calculation that can produce an integer overflow or wraparound, when the calculation is used for resource management or execution control.	
193	**Off by one error** A product calculates or uses an incorrect maximum or minimum value that is 1 more, or 1 less, than the correct value.	
789	**Uncontrolled memory allocation** Memory is allocated based on invalid size and allowing arbitrary amounts of memory to be allocated.	
252	**Unchecked return value** The return value is not checked by a method or function, which may create an unexpected state.	
1006	**Bad Coding Practices** These weaknesses are deemed to cause exploitation's that are not vulnerable by self but indicate that the application is not developed carefully.	
1019	**Validate input Weaknesses** Weaknesses are related to the design and architecture of a system's input validation components that could lead to a degradation of the quality of data flow in a system.	
415	**Double free** Called free() twice on the same memory address, potentially leading to modification of unexpected memory locations.	
426	**Untrusted search path** The application searches for critical resources using an externally-supplied search path that can point to resources that are not under the application's direct control.	
476	**Null pointer deference** A NULL pointer dereference occurs when dereference a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.	
477	**Use of obsolete function** The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained.	
628	**Function call with incorrect specific arguments** The product calls a function, procedure, or routine with arguments that are not correctly specified, leading to always-incorrect behavior and resultant weaknesses.	
120	**Classic buffer overflow** Been copied an input buffer to an output buffer without verifying that the size of the input buffer is less than the size of the output buffer.	
676	**Use of potentially dangerous function** Invoked a potentially dangerous function that could introduce a vulnerability if it is used incorrectly.	
682	**Incorrect calculation** Perform a calculation that generates incorrect or unintended results that are later used in security-critical decisions or resource management.	
686	**Function call with incorrect argument type** A function, procedure, or procedure is called up with arguments that are not properly specified, resulting in always mistaken behavior and resulting weaknesses.	

Continuation of Table 5: The different types of CWE C++ vulnerabilities and their frequency as we observed in our dataset of Stack Overflow Answers. Each tick in X-axis denotes the last one/two letters of a year, e.g., 8 for 2008 to 16 for 2016.

CWE	Title and Description	Frequency by Year
710	**Improper adherence to coding standards** Not followed certain coding rules for development, which can lead to resultant weaknesses.	
150	**Improper neutralization of escape, meta, or control sequence** The software receives input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements when they are sent to a downstream component.	
758	**Reliance on Undefined unspecific or implementation defined behavior˜** Used an API function, data structure, in a way that relies on properties that are not always guaranteed to hold for that entity.	
232	**Improper handling of undefined values** Does not handled or incorrectly handled when a value is not defined or supported for the associated parameter, field, or argument name.	
835	**loop with unreachable exit condition** The program contains an iteration or loop with an exit condition that cannot be reached, i.e., an infinite loop.	
369	**Divide by zero** Typically occurs when an unexpected value is provided to the product, or an error occurs that is not properly detected.	
413	**Improper resource locking** The software does not lock or does not correctly lock a resource when the software must have exclusive access to the resource.	

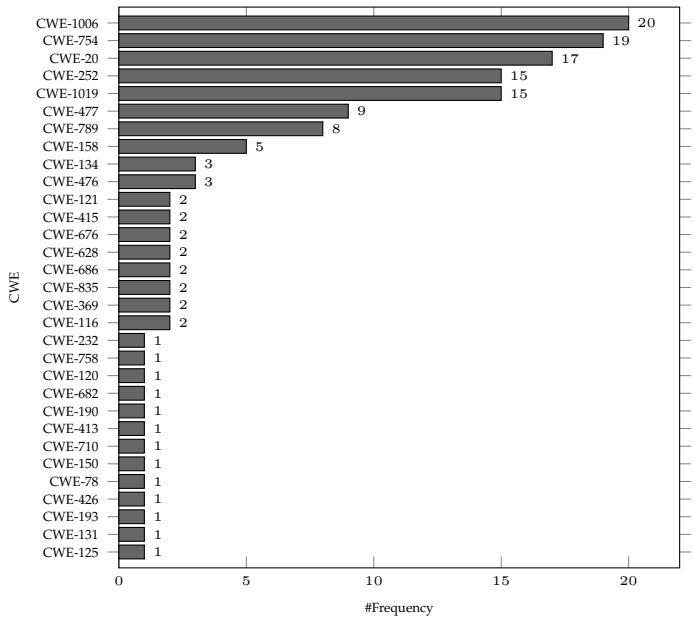


Fig. 12: Frequency of CWEs in code snippets

This code snippet of answer 440240 shown in Listing 1 can be dangerous. Functions with count parameters like 'len' should take into account the terminating 'NULL' as an extra character. But this function actually writes into the character 'len+1' when executing s[len] = 0. That is CWE- 193: Off-by-one-error vulnerability [51] that may lead to unpredictable behaviour, memory corruption and application crash. If one pays special attention and always passes the length a number at maximum one less than the desired, the function works correctly, otherwise we have off-by-one error.

Line 's[i] = alphanum[rand() % sizeof(alphanum) - 1]' is faulty since size of 'alphanum' is '63', where the last character in the string indexed 69^{th} is 'NULL'. Therefore, once in a while a NULL may be included in the generated 'random' string. This vulnerability can be categorized as 'CWE-754: Improper check for unusual or exceptional conditions' [52], where an improper number may be used as a return of a function leading to a crash or other unintended behaviours. Another appropriate category is 'CWE-1006: Bad coding practices'. Stated differently, a generated random string with this algorithm may include 'NULL' in the middle of string.

Moreover, 'rand()' is an obsolete function in C and C++. So another vulnerability category is 'CWE-477: Use of obsolete function' a major degradation in software quality. Another vulnerability exists within the code since the developer did not use a random seed before calling the function. Thus, the generated random number is not 'random' at all. Moreover, 'rand() % mod' is not a good practice since it returns lower bits which are not again random [53].

Listing 2: Execute functor in given thread in QT - Answer id 21653558 in Stack overflow, shows vulnerability due to use malloc function without checking return special condition, (CWE-1006, CWE-252, CWE-789, CWE-476)

```
class FunctorCallEvent: public QMetaCallEvent {
public:
  template <typename Functor>
   FunctorCallEvent(Functor && fun, QObject * receiver) :
     QMetaCallEvent(new QtPrivate::QFunctorSlotObject<Functor,
          0, typename QtPrivate::List_Left<void, 0>::Value, void>
       (std :: forward<Functor>(fun)), receiver, 0,
            0,0,(void**) malloc (sizeof(void*)));
```

Another vulnerability is shown in Listing 2 of answer 21653558. The code snippet in this answer uses 'malloc' to allocate memory and passes its pointer to a function in QT library that requires a valid pointer. malloc return pointer may be set to NULL in case of malloc failure. Thus, the return pointer from malloc must be checked even if the amount of memory requested is small [54]. In this example, return value of malloc is not checked. This vulnerability is called CWE-252 [55]; Unchecked return value. In case of malloc failure null pointer dereference occurs.

Listing 3: Execute command and get output - Answer id 478960 in Stack overflow, Execute function in given thread in QT - Answer id 21653558 in Stack overflow, shows vulnerability due to OS command injection because user input are involved, (CWE-78, CWE-1019)

```
std :: string exec(const char* cmd) {
  std :: shared_ptr<FILE> pipe(popen( cmd , "r"), pclose);
  if (!pipe) return "ERROR";
  char buffer[128];
  std :: string result = "";
  while (!feof(pipe.get())) {
      if (fgets(buffer, 128, pipe.get()) != NULL)
          result += buffer;
      }
      return result;
  }
```

The function shown in Listing 3 is vulnerable to code injection (OS command injection) attacks since user inputs commands are inputted and not checked. In other words, any command with privilege level of the program can be executed without any errors or warnings.

Listing 4: Set the global LUA_PATH variable programmatically - Answer id 4156038 in Stack overflow, shows vulnerability due to second arg in this function may contain multiple path separated by ";" (CWE-754, CWE-252, CWE-426)

```
int setLuaPath( lua_State* L, const char* path ){
  lua_getglobal( L, "package" );
  lua_getfield( L, -1, "path" );
  std :: string cur_path = lua_tostring )( L, -1 );
  cur_path.append( ';' );
  cur_path.append( path );
  lua_pop( L, 1 );
  lua_pushstring( L, cur_path.c_str() );
  lua_setfield( L, -2, "path" );
  lua_pop( L, 1 );
  return 0;
}
```

Listing 4 deals with system path programmatically, or different paths the program searches. The operation is dangerous and should be performed carefully. For example, 'path' in this function may contain multiple paths separated by ';'. For instance, '/usr/share/lua;/foo/bar/evil/path'. Having an untrusted search path within the paths produces the probability of arbitrary code execution with privilege of the program and redirection to a wrong file potentially triggering a crash. The vulnerability is called CWE-426: Untrusted search path. For more on this vulnerability, please refer to [56]. The search may lead to execution of programs, which in turn lead to unusual or exceptional conditions; i.e. CWE-754: Improper check for unusual or exceptional conditions. Moreover, all the return values of the functions in the code snippet are not checked. Thus, the snippet also has CWE-252: Unchecked return values.

Listing 5: Set Byte vector to integer type - Answer id 41031865 in Stack overflow, shows vulnerability due to out of bound read and the lack of checking the size of the variable, (CWE-20, CWE-125, CWE-1019)

```
template<typename T>
static T get_from_vector(const std::vector<uint8_t>& vec,
      const size_t current_index ){
T result;
  uint8_t *ptr = (uint8_t *) &result;
  size_t idx = current_index + sizeof(T);
  while(idx > current_index)
    *ptr++ = vec[--idx];
  return result;
}
```

In Answer 41031865 shown in Listing 5, 'current_index + sizeof(T)' can become larger than size of 'vec' due to CWE-1019: Validate inputs vulnerability. In addition, when index exceeds the limit, information leakage can occur or CWE-125; the vulnerability 'Out of bound read' is present.

Listing 6: Checks if string ends with .txt - Part of answer id 20447331 in Stack overflow, all defined functions have vulnerability have fail if input string that contain a null value, (CWE-158, CWE-1019)

```
bool ends_with (std :: string const &a, std :: string const &b) {
  auto len = b.length();
  auto pos = a.length() - len;
  if (pos < 0)
    return false;
  auto pos_a = &a[pos];
  auto pos_b = &b[0];
  while (*pos_a)
    if (*pos_a++ != *pos_b++)
     return false;
  return true;
}

bool ends_with_string (std :: string const& str, std :: string const&
      what) {
  return what.size() <= str.size()
      && str.find(what, str.size() - what.size()) != str.npos;
}
```

In answer (20447331) shown in Listing 6 on how to validate whether a file name ends with ".txt" or not, this answer includes code of functions and their benchmarks for six methods in the original code snippet in answer post. The vulnerability for other functions defined in this code snippet is exactly the same as the two vulnerable functions. However if filename in function includes a NULL character, all of above methods will fail. This is a common trick to bypass web application firewalls and file uploaders.
Example:
Validating 'shell.txt\0.php' will return True for all of above functions.

5 Propagation of C++ Vulnerable Code from Stack Overflow to GitHub (RQ2)

5.1 How frequently are the vulnerable code examples from Stack Overflow copied to GitHub? (RQ2.1)

• **Approach.**
To detect the vulnerable code snippets that migrated to GitHub projects, it may seem plausible to use clone detection tools like SourcererCC [47]. However, the most effective clone detection tools work only for Java applications, e.g., Oreo [57]. The ones that can detect C++ clones only work at file or class level. For Java, SourcererCC can find cloned procedures but the same capability is not implemented for C++. The majority of vulnerable code segments that we found are functions or a part of a function. Therefore, we had to use some heuristics to search and find

similar codes in linked GitHub projects. To find vulnerable clones, we searched for the signatures of the code snippets in Stack Overflow by looking at the sequences of keywords that can uniquely find them within GitHub projects. The used heuristics are explained below:

We take motivation from previous work [58], [59] and use a rule-base approach to detect security flaws. We used rules because unlike keyword-based searching, rules are less susceptible to false positive [58], [59]. We select an ordered sequence of keywords that with or without them, the vulnerable code can be found within the linked GitHub projects. For each code version, we chose a unique set of words to identify that specific version in GitHub projects. For example, to detect vulnerable GitHub projects that used the code snippet shown in listing 1, we searched 'RAND()' keyword and for older version chose RAND() and)-1 keywords. To evaluate this method, we randomly selected five GitHub files from from the 69 vulnerable answers and reviewed them manually. Every link to Github has been tagged either with vulnerable (YES) or not vulnerable (NO). This section was chosen as our BASELINE method. We then run the our set of keywords using the discussed algorithm on the 287 GitHub files, which reported whether there is a vulnerability in the GitHub file or not.

We then changed the criteria for choosing the keywords for the algorithm and chose the keywords randomly without changing the number of words selected for each code snippets. In order to select random keywords, at first the comments of the code snippet has been removed, then the code snippet has been tokenized. We then removed reserved keywords from code snippet and only chose words with more than three characters as a candidate keyword.

The following results are presented for two methods (Chosen-Keyword and Random-Keyword) and compare it with the baseline method.

Baseline-Method	Label-Algorithm	Chosen-Keyword	Random-Keyword
YES	NO	34	51
NO	NO	10	160
NO	YES	29	41
YES	YES	223	35

TABLE 4: Two Method Results

• **Results.**

For each method we calculate Recall, Precision, F1-Measure and Accuracy:

$$\text{RECALL} = \frac{TruePositive}{TruePositive + FalseNegative}$$

$$\text{PRECISION} = \frac{TruePositive}{TruePositive + FalsePositive}$$

$$\text{F1 Measure} = 2 * \left(\frac{Precision * Recall}{Precision + Recall}\right)$$

$$\text{ACCURACY} = \frac{TruePositive + TrueNegative}{TruePositive + FalsePositive + FalsePositive + FalseNegative}$$

The most important weakness of the Random-Keyword is the use of keywords that do not indicate existence of security vulnerabilities in GitHub code. According to listing 7 the sequence chosen for Chosen-Keyword is based on the fact that by navigating and searching ordered keywords, the vulnerability was detected mean while in the Random-Keyword method this criterion is determined without consideration of vulnerable part of the code and its sequence which will lead into incorrect results.

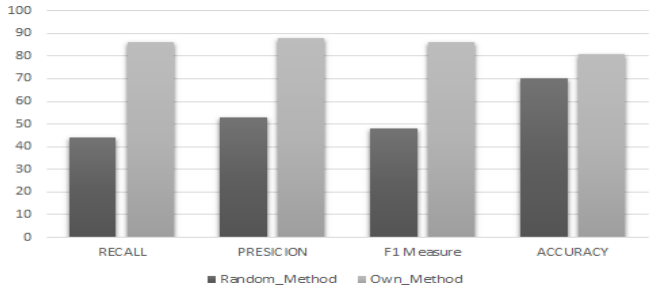


Fig. 13: Comparing Results for Two Method

Listing 7: Selected keywords from answer id: 4156038

```
!"Path"
!lua_tostring(
```

Total count of GitHub files for 69 vulnerable answers was 2859 GitHub links that has been showen in table 5 separated by CWE definition. After run previous method, our algorithm found 287 GitHub vulnerable files can potentially have security flaws. Samples of False-Positive and False-Negative Chosen-Keyword heuristic and Random-Keyword heuristic results can be found in online material.

TABLE 5: CWEs detection in GitHub Repositories with chosen-keyword Algorithm

GitHub Count	Confirm Count	CWE Title
1539	4	CWE-835-Loop with unreachable Exit condition
703	37	CWE-20-Improper input validation
653	72	CWE-754-Improper check for unusual or exceptional condition
324	187	CWE-1006-Bad coding practice
250	5	CWE-158-Improper neutralization of null byte or null character
212	2	CWE-369-Divided by zero
151	141	CWE-150-Improper neutralization of escape, meta, or control sequence
118	0	CWE-628-Function call with incorrectly specific argument
89	14	CWE-252-Unchecked return value
73	2	CWE-134-Use of externally controlled format string
54	4	CWE-476-Null pointer dereference
53	4	CWE-789-Uncontrolled memory allocation
41	12	CWE-477-Use of obsolete function
20	1	CWE-676-Use of potentially dangerous function
20	0	CWE-232-Improper handling of undefined values
14	2	CWE-121-Stack base buffer overflow
7	0	CWE-415-Double free
5	1	CWE-78-Improper neutralization of special elements used in an os command
5	0	CWE-413-Improper resource locking
5	5	CWE-116-Improper encoding or escaping of output
5	0	CWE-193-Off by one error
3	3	CWE-682-Incorrect calculation
3	0	CWE-686-Function call with incorrect argument type
3	0	CWE-120-Buffer copy without checking size of input
3	0	CWE-131-Incorrect calculation of buffer size
3	1	CWE-710-Improper adherence to coding standard
1	0	CWE-426-Untrusted search path

5.2 How frequently are the copied vulnerable code examples fixed in the GitHub repositories? (RQ2.2)

We created a new web application to make the review process more efficient and systematic. With the review system, we assessed whether the vulnerable Stack Overflow code snippets that were migrated to GitHub were either fixed or still contained the vulnerability.

• **Results.** Among 287 GitHub files that must be checked, vulnerabilities were corrected in 34 files and other 253 GitHub file still had vulnerabilities. For instance as can be seen in Listing 8, the two CWE-789 [60] and CWE-252 [55] were corrected.

Listing 8: Part of code was Fixed and improved in GitHub File for answer id 2654860 in Stack Overflow

```
//Improve and adapted version of http://stackoverflow.com/a/
    2654860
```

```
void save_bmp(string filename, uchar4* ptr, const int width, const
      int height)
{
const int num_elems = width*height;
unsigned char* img = (unsgined char*)malloc(3* num_elems);
int i =0;
 .....
}
```

As shown in listing 9 boundary was limited and mentioned in comments of source code in GitHub file and CWE-125 [61], CWE-category-1019 [62] and CWE-20 [63] were corrected.

Listing 9: Code was Fixed in GitHub File mentioned in comment for answer id 41031865 in Stack Overflow

```
//check if we can read sizeof(T) bytes starting the next index
check_lenght(vec.size(), sizeof(T), current_index + 1);
T result;
auto* ptr = reinterpret_cast<uint8_t*>(&result);
for (size_t i = 0; i < sizeof(T); ++i)
{
    *ptr++ = vec[current_index + sizeof(T) -i];
}
return result;
}
```

5.3 How did the GitHub developers react when informed of the vulnerable code examples? (RQ2.3)

• **Approach.**
To inform developers about vulnerability in their repositories, a script has been used to make an issue on the repository. The script has been fed with our code review result. Developers have been notified with information including:

- **Description:** The vulnerability in the code snippet is explicitly expressed.
- **Example:** An attack scenario is provide to justify why the vulnerability is dangerous and how it may lead to exploitation.
- **Mitigation Scenario:** The mitigation scenario is included to inform the developer on how to fix the vulnerability.
- **Reference:** An authenticated reference is provided to show vulnerability is labelled based on objective and factual judgements.

• **Results.** In addition, a set of questions related to the vulnerability with multiple choice or Likert scale answers were given. We received 15 response from 174 issues that were sent.

As shown in Figure 14 GitHub owner comments about existence vulnerability in their code.

Fig. 14: User opinion about exists vulnerability in their code

Figure 15 shows the response of some users about vulnerability in their projects. As shown in Figure16, the user's opinion about the usefulness of the automatic detection of the vulnerability was discussed.

In the end, we asked the users about the best way to know about the vulnerability shows in Figure 17.

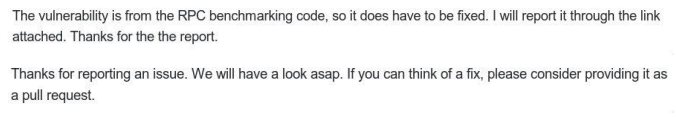


Fig. 15: User response about security vulnerability in their code

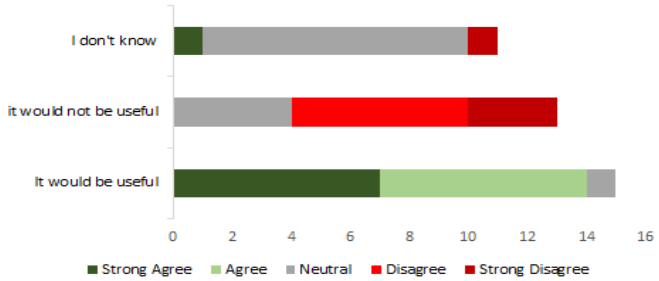


Fig. 16: User opinion about automated vulnerability analysis useful for future development

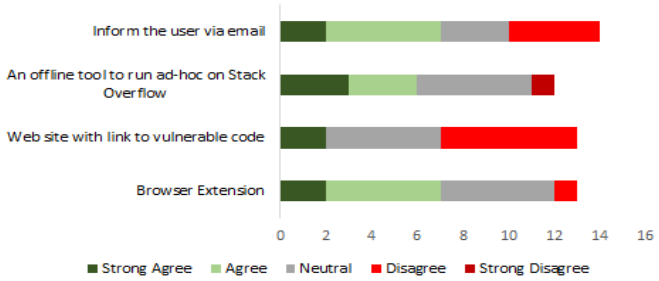


Fig. 17: How inform developer of potential vulnerabilities in code examples

6 Implications of Findings

Of the fifteen responses we received out of a 117 issues to the GitHub owners, 40% pointed to the possibility of vulnerabilities in the code provided their input data is not dynamic and 13.3% acknowledge the vulnerability in the code but are reluctant to fix it. Most practitioner agree to implement a automated security mechanism to detect vulnerabilities in the code. They expressed the use of the browser extension [45] and offline tool to run ad-hoc on Stack Overflow methods more effective than other methods to inform users to vulnerabilities in the code examples.

To inform users about existence vulnerability in the code, we have developed a browser extension. Suppose that a developer needs to create a random alpha-numeric string in C++ for their task in their program. The developer searches in Stack Overflow for a possible solution. The search shows a question with ID 440133 as the top match. The importance of the question is determined in Stack Overflow based on how developers perceive the question. This asker of this question offered a bounty reward of 100 to the accepted answer. Consequently, the question received many answers. The accepted answer (ID 440240) has 263 scores (upvote - downvote) and it was viewed more than 174,000 times as of today. Therefore, a new developer looking for a solution for this task is expected to be convinced to use the solution provided in the answer. However, the provided solution has one of the security vulnerabilities as we discussed in Section 4.2 (Listing 1). Therefore, the provided solution, if used as is, will introduce potential C++ security vulnerability in the developer's software. Our browser extension aims to prevent developers from reusing such vulnerable code snippets, as well as to recommend them of better alternatives, i.e., non-vulnerable code snippets in other Stack Overflow posts. As we

TABLE 6: Questions asked in primary survey.

NO	Question
1	**Which of the following situation for our issue was true?** *(eight options)*
2	**Please justify your choice above** *(text box)*
3	**What would be the best way to inform developers of potential vulnerabilities in code examples shared in Stack Overflow** *(5-point Likert scale for each opinion)*
4	**Do you have any other suggestions to design automated techniques to assist developers to handle security vulnerabilities while using code from online forums? Please write in couple of sentences below** *(text box)*
5	**Could automated vulnerability analyses of code snippets in online forums be useful in your future development tasks?** *5-point Likert scale for each opinion*
6	**Would you like to informed of such a tool that could be developed in future?** *(3-point Likert scale for each opinion)*
7	**Get developers email address to expand the vulnerability detection tool** *(text box)*

recall from our survey of GitHub developers, such a browser extension was also desired by the survey respondents.

In Figure 18, we show a screenshot of our developed extension. The extension gets activated when a developer visits a Stack Overflow post. The extension consults our database of vulnerable C++ code snippets in Stack Overflow to determine whether the provided solution in the post is vulnerable. If the provided solution is indeed found vulnerable, the extension then shows a warning message to the developer with an explanation of why the code snippet is vulnerable (see 18). The extension then recommends non-vulnerable similar code snippets from other Stack Overflow posts, so that the developer can reuse those safe code snippets instead of the vulnerable code snippet.

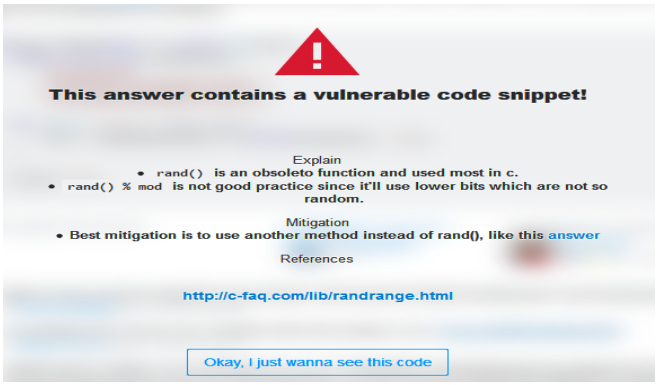


Fig. 18: Browser Extension for Code Snippet with AnswerId 440240 in Stack Overflow

7 THREATS TO VALIDITY

We now discuss the threats to validity of our study following the guidelines for case study research.

Construct validity threats: Concern the relation between theory and observation. In our study, threats to the construct validity are mainly due to measurement errors. We use the state-of-the-art clone detection tool, SourcererCC [47], to identify similar code between the Stack Overflow C++ code snippets. We use the tokenization at file level in SourcererCC and set the similarity degree 100 percent to find all exact C++ code in Stack Overflow all other setting have default setting. So far, SourcerecCC has not implemented the block-level tokenization implementation for C++, so we used our heuristic to detect vulnerable C++ code snippets from Stack Overflow to GitHub that can miss some vulnerable codes. Other concern is related to false negatives that Syntaxnet [46] may have produced.
Internal validity threats: Any misrepresentation in the topic areas we used SOTorrent dataset would effect our analysis. For instance, migrated Stack Overflow code snippets that SOTorrent may have missed to link to GitHub projects were automatically omitted from our analysis. Moreover, several posts are not correctly tagged, so C++ snippets that are not correctly tagged are also not included in our analysis. These missed snippets would just extend our analysis to a broader perspectives but does not have any effect on the results obtained.
External validity threats: Concern the possibility to generalize our results. The findings in this paper was focused on Stack Overflow and does not generalize to other Q&A websites.

8 CONCLUSION

In this paper, we have analyzed vulnerabilities in C++ code snippets shared on Stack Overflow and their migration to GitHub projects. This is the first study that address the security issues of C++ code examples shared on Stack Overflow. We have investigated security vulnerabilities in the C++ code snippets shared on Stack Overflow over a period of 10 years. From the 72,483 reviewed code snippets used in at least one project hosted on GitHub, we found a total of 69 vulnerable code snippets categorized into 29 types. Bad coding practices, improper check for unusual or exceptional conditions and improper input validation were most prevalent types of vulnerabilities. The 69 vulnerable code snippets found in Stack Overflow were reused in a total of 2859 GitHub projects. Information about the detected vulnerabilities were presented to developers of the studied projects. Although they acknowledged the vulnerabilities, many of them are still not corrected today.

REFERENCES

[1] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow: An exploratory study on android apps. *Information and Software Technology*, 88:148–158, 2017.
[2] Wayne C Lim. Effects of reuse on quality, productivity, and economics. *IEEE software*, 11(5):23–30, 1994.
[3] Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe. Where does this code come from and where does it go?-integrated code history tracker for open source systems. In *Proceedings of the 34th International Conference on Software Engineering*, pages 331–341. IEEE Press, 2012.
[4] Christoffer Rosen and Emad Shihab. What are mobile developers asking about? a large scale study using stack overflow. *Empirical Software Engineering*, 21(3):1192–1223, 2016.
[5] Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 191–201. ACM, 2015.
[6] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, 2012.
[7] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305. IEEE, 2016.

[8] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017.

[9] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics*, pages 87–94. IEEE, 2002.

[10] Di Yang, Pedro Martins, Vaibhav Saini, and Cristina Lopes. Stack overflow in github: any snippets there? In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 280–290. IEEE, 2017.

[11] Pei Xia, Makoto Matsushita, Norihiro Yoshida, and Katsuro Inoue. Studying reuse of out-dated third-party code in open source projects. *Information and Media Technologies*, 9(2):155–161, 2014.

[12] Chaiyong Ragkhitwetsagul, Jens Krinke, Matheus Paixao, Giuseppe Bianco, and Rocco Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 2019.

[13] TIOBE Company. TIOBE Index for January 2019. https://www.tiobe.com/tiobe-index/, 2019. [Online; accessed 19-January-2019].

[14] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. Sotorrent: Reconstructing and analyzing the evolution of stack overflow posts. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 319–330. ACM, 2018.

[15] Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of stack overflow code snippets. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 191–194. IEEE Press, 2019.

[16] C Cowan. Automatic adaptive detection and prevention of buffer overflow attacks. In *Proc. of the 7USENIX Security Conf., San Antonio, Texas*, 1998.

[17] Matt Bishop, Michael Dilger, et al. Checking for race conditions in file accesses. *Computing systems*, 2(2):131–152, 1996.

[18] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. Jumping through hoops: Why do java developers struggle with cryptography apis? In *Proceedings of the 38th International Conference on Software Engineering*, pages 935–946. ACM, 2016.

[19] Shaowei Wang, Tse-Hsun Chen, and Ahmed E Hassan. Understanding the factors for fast answers in technical q&a websites: An empirical study of four stack exchange websites. In *Proceedings of the 40th International Conference on Software Engineering*, pages 884–884, 2018.

[20] Le An, Ons Mlouki, Foutse Khomh, and Giuliano Antoniol. Stack overflow: a code laundering platform? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 283–293. IEEE, 2017.

[21] Akond Rahman, Asif Partho, Patrick Morrison, and Laurie Williams. What questions do programmers ask about configuration as code? In *Proceedings of the 4th International Workshop on Rapid Continuous Software Engineering*, pages 16–22. ACM, 2018.

[22] Anton Barua, Stephen W Thomas, and Ahmed E Hassan. What are developers talking about? an analysis of topics and trends in stack overflow. *Empirical Software Engineering*, 19(3):619–654, 2014.

[23] Akond Rahman, Effat Farhana, and Nasif Imtiaz. Snakes in paradise?: insecure python-related coding practices in stack overflow. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 200–204. IEEE Press, 2019.

[24] Margaret-Anne Storey, Christoph Treude, Arie van Deursen, and Li-Te Cheng. The impact of social media on software engineering practices and tools. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 359–364. ACM, 2010.

[25] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1589–1598. ACM, 2009.

[26] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest q&a site in the west. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 2857–2866. ACM, 2011.

[27] Eduardo C Campos, Martin Monperrus, and Marcelo A Maia. Searching stack overflow for api-usage-related bug fixes using snippet-based queries. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 232–242. IBM Corp., 2016.

[28] Manziba Akanda Nishi, Agnieszka Ciborowska, and Kostadin Damevski. Characterizing duplicate code snippets between stack overflow and tutorials. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 240–244. IEEE Press, 2019.

[29] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 372–383. IEEE, 2018.

[30] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. How reliable is the crowdsourced knowledge of security implementation? In *Proceedings of the 41st International Conference on Software Engineering*, pages 536–547. IEEE Press, 2019.

[31] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61. ACM, 2012.

[32] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. Its4: A static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267. IEEE, 2000.

[33] Andrew C Myers and Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241. ACM, 1999.

[34] Stephen Turner. Security vulnerabilities of the top ten programming languages: C, java, c++, objective-c, c#, php, visual basic, python, perl, and ruby. *Journal of Technology Research*, 5:1, 2014.

[35] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. Are code examples on an online q&a forum reliable? 2018.

[36] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering*, pages 164–175. IEEE Press, 2019.

[37] Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. An empirical study of security issues posted in open source projects. In *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[38] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on mining software repositories*, pages 348–351. ACM, 2014.

[39] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L Mazurek, and Sascha Fahl. Security developer studies with github users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security ({SOUPS} 2017)*, pages 81–95, 2017.

[40] Eric Horton and Chris Parnin. Gistable: Evaluating the executability of python code snippets on github. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 217–227. IEEE, 2018.

[41] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[42] Christoph Treude and Martin P Robillard. Augmenting api documentation with insights from stack overflow. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 392–403. IEEE, 2016.

[43] Tommi Unruh, Bhargava Shastry, Malte Skoruppa, Federico Maggi, Konrad Rieck, Jean-Pierre Seifert, and Fabian Yamaguchi. Leveraging flawed tutorials for seeding large-scale web vulnerability discovery. In *11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17)*, 2017.

[44] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 372–383. IEEE, 2018.

[45] https://github.com/paper-materials-crowd-sourced/materials, 2019.

[46] Google Company. Announcing SyntaxNet: The Worlds Most Accurate Parser Goes Open Source. https://ai.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html, 2019. [Accessed 29 Jun. 2019].

[47] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1157–1168. IEEE, 2016.
[48] Justus J. Randolph. Free-Marginal Multirater Kappa: An Alternative to Fleiss FixedMarginal Multirater Kappa. https://files.eric.ed.gov/fulltext/ED490661.pdf, 2005. [Accessed 27 Jun. 2019].
[49] Walter E. Brown. Deprecating rand() and Friends. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3775.pdf, 2013. [Accessed 27 Jun. 2019].
[50] Community developed list of common software security weaknesses. CWE Mitre. https://cwe.mitre.org/. [Accessed 27 July. 2018].
[51] Community developed list of common software security weaknesses. CWE 193. https://cwe.mitre.org/data/definitions/193.html. [Accessed 27 July. 2018].
[52] Community developed list of common software security weaknesses. CWE 754. https://cwe.mitre.org/data/definitions/754.html. [Accessed 27 July. 2018].
[53] http://c-faq.com/lib/randrange.html. [Accessed 12 July. 2018].
[54] https://docs.microsoft.com. [Accessed 12 July. 2018].
[55] Community developed list of common software security weaknesses. CWE 252. https://cwe.mitre.org/data/definitions/252.html. [Accessed 30 July. 2018].
[56] Community developed list of common software security weaknesses. CWE 426. https://cwe.mitre.org/data/definitions/426.html. [Accessed 30 July. 2018].
[57] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V Lopes. Oreo: Detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 354–365. ACM, 2018.
[58] Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, pages 112–122. IEEE Press, 2017.
[59] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 145–158. ACM, 2007.
[60] Community developed list of common software security weaknesses. CWE 789. https://cwe.mitre.org/data/definitions/789.html. [Accessed 30 July. 2018].
[61] Community developed list of common software security weaknesses. CWE 125. https://cwe.mitre.org/data/definitions/125.html. [Accessed 30 July. 2018].
[62] Community developed list of common software security weaknesses. CWE 1019. https://cwe.mitre.org/data/definitions/1019.html. [Accessed 30 July. 2018].
[63] Community developed list of common software security weaknesses. CWE 20. https://cwe.mitre.org/data/definitions/20.html. [Accessed 30 July. 2018].