Raising Security Awareness using Cybersecurity Challenges in Embedded Programming Courses

Tiago Espinha Gasiba
Samra Hodzic
tiago.gasiba@siemens.com
samra.hodzic@siemens.com
Siemens AG
Munich, Bavaria, Germany

Ulrike Lechner
Universität der Bundeswehr München
Neubiberg, Germany
ulrike.lechner@unibw.de

Maria Pinto-Albuquerque
Instituto Universitário de Lisboa (ISCTE-IUL), ISTAR
Lisboa, Portugal
maria.albuquerque@iscte-iul.pt

Abstract

Security bugs are errors in code that, when exploited, can lead to serious software vulnerabilities. These bugs could allow an attacker to take over an application and steal information. One of the ways to address this issue is by means of awareness training. The Sifu platform was developed in the industry, for the industry, with the aim to raise software developers' awareness of secure coding. This paper extends the Sifu platform with three challenges that specifically address embedded programming courses, and describes how to implement these challenges, while also evaluating the usefulness of these challenges to raise security awareness in an academic setting. Our work presents technical details on the detection mechanisms for software vulnerabilities and gives practical advice on how to implement them. The evaluation of the challenges is performed through two trial runs with a total of 16 participants. Our preliminary results show that the challenges are suitable for academia, and can even potentially be included in official teaching curricula. One major finding is an indicator of the lack of awareness of secure coding by undergraduates. Finally, we compare our results with previous work done in the industry and extract advice for practitioners.

Keywords: secure coding, software quality, embedded programming, training, cybersecurity challenge, education, security bug

1 Introduction

Security vulnerabilities originate in programming errors, that, when left in code, can be exploited by malicious parties and lead to severe security incidents. Examples of vulnerabilities are, e.g., Shellshock, HeartBleed, POODLE, and DirtyCOW. The presence of these errors in software code is an indicator of poor code quality [34]. Consequences of security breaches include, among others, leakage of confidential information, denial of service, and privilege escalation [21]. Over the last years, the number of vulnerabilities in software has been steadily increasing and the corresponding financial consequences, e.g. for affected companies, is exceeding several billion dollars [2]. One reason that might justify this increase in vulnerabilities is the ever increasing complexity of code and systems.

In an industrial context, several methods exist to address code quality, e.g. using Static Application Security Testing (SAST) Tools [24], IDE plugins, performing threat and risk analysis, and code reviews. In [18], McIntosh et al. discovered that code review coverage and expertise participation have a significant link with software quality. While these methods are typically used in the industry, another important aspect should also be considered, namely students from academia, as these will be the next generation workforce. Furthermore, students might not have access to all of these methods, or might even lack training in cybersecurity.

In this work we look at raising cybersecurity awareness on secure programming for students in an academic setting. Our work is based on exploring and using CyberSecurity Challenges (CSC) in the academia. CyberSecurity Challenges is a novel serious game [8] which aims to raise awareness on secure coding of software developers in the industry [11]. These games are well investigated, and are showing very promising results in an industrial setting. However, these games have, until now, not been explored in an academic setting.

The current research extends previous work [10] by developing C++ challenges in an industrial secure coding awareness platform (Sifu platform). Although other providers exist that offer programming courses, the Sifu platform targets especially industrial environments. The design of the new challenges hereby presented differ from challenges previously implemented in the Sifu platform since they target embedded programming courses. Furthermore, evaluation is done in an academic setting, which allows to compare industry and academia. In particular, this work compares the perceived benefits of using the awareness platform in academia and industry. The results of this comparison might differ due to the different background of students and professional software developers. Therefore we aim to evaluate the suitability of CSC games in the academia, provide details on the implementation of embedded programming challenges, and aid universities to adopt the platform to assist teaching secure programming of embedded systems. The industry also benefits from the latter, as these students will be better prepared for the industry demands in terms of secure software development.

In the following we introduce three C++ challenges with different security vulnerabilities. In these challenges, the security vulnerabilities that will be introduced are, according to the authors' experiences in the industry, likely to happen in embedded systems programming. These are: side-channel vulnerability, invalid memory access, and race condition. Along with the challenge description, we will address the learning goal of the challenge to raise awareness on secure coding. We also give details on how the Sifu platform can assess the vulnerability in the code by means of automatically testing that a participant's solution follows secure coding guidelines. The main contributions of this work are the following:

- design of three defensive C++ programming challenges in the industry which aim to raise secure coding awareness
- base the challenges on security vulnerabilities common in embedded systems
- details on how to test that solutions from a participant follow secure coding
- preliminary results on the acceptance of the challenges by students in academia
- insight into implementation and improvements for practitioners

The current work's main target is to use the Sifu platform, which was developed in the industry, extend it, and analyze how it applies in academia. We aim to answer the following research questions:

RQ1 How can the Sifu platform be used in academia?
RQ2 How to decide whether a challenge based on side-channel, invalid memory access, and race condition was correctly solved?
RQ3 How do students perceive the challenges presented through the Sifu platform?

In section 2, we present previous work related to our research. section 3 discusses our approach to design defensive C++ challenges. We first briefly describe the Sifu platform, which is used to integrate these challenges to be used for training. The main part of this section is to introduce the implemented challenges and which security vulnerabilities they contain. We also present our methodology for assessing the presence of the security vulnerabilities in the code. Additionally, we discuss the evaluation of the developed challenges and platform in the academia. The results are presented and discussed in section 5. Three groups of results are provided: evaluation of the challenge assessment methods, analysis of two test runs including a survey and semi-structured interview, and comparison with our results to previous results for the industrial context. Finally, section 6 summarizes our work and briefly discusses possible next steps.

2 Related Work

Previous research shows that software developers lack secure programming awareness and skills [12, 28]. Gasiba et al. [14] introduced CyberSecurity Challenges (CSCs) to raise secure coding awareness of software developers in the industry. CSCs are serious games that refine the popular CTF format and adapt it to the industry. Gasiba et al. [9] researched the constraints and requirements for delivering a cybersecurity challenge which can cover secure coding from an industry perspective. One important outcome of this research is that the challenges should focus on the defensive perspective, and not on the offensive. In their work, they introduced a new platform [10], which the authors call Sifu. The platform performs an automatic assessment of challenges in terms of compliance to secure coding standards and guidelines. It uses an artificial intelligence coach, which guides the participant throughout the challenge. Their work presents results that indicate that the Sifu platform's CSC events are adequate for raising secure coding awareness of software developers in the industry. Parker et al. [25] designed a similar training, however their platform is not focused on a specific programming language, and includes offensive challenges.

Tabassum et al. [27] and Whitney et al. [30], present the importance of secure coding guidelines and standards in the software development life-cycle. Given the lack of knowledge on secure coding, developers tend to search online resources for answers and solutions. However, Kurachi et al. [33] and Zhang et al [17] show that these solutions are not always adequate and blind usage of these solutions can lead to additional problems.

Static analysis is an evolving approach to evaluate programs based exclusively on their source code without running them. Clang is a C/C++/Objective-C open-source compiler [7]. It is a continually developing initiative sponsored by large companies such as Apple, Microsoft, and Google. Clang is currently a popular method for designing new static analyzers. Its modular architecture is one of Clang's strength [5]. The Sifu platform makes use of this technology provided by Clang to implement security assessments of challenges.

In [1, 3, 4, 6], the researchers compare different open-source static analysis tool available for C/C++. The authors have developed their C/C++ applications and introduced various vulnerabilities in the application. They use these applications to check the tool's capabilities to detect the introduced vulnerabilities. The researchers have also presented a study comparing commercial static code analysis tools for detecting vulnerabilities in a software source code.

In our work, we use the semi-structured interviews methodology as given by Wilson et al. [31]. We also use the definition of awareness as given by Hänsch et al. [15]. In their work, they specify awareness as having three components: perception, protection and behavior. Perception relates to knowledge of threats, protection relates to knowing available mechanisms to protect against these threats, and finally behavior relates to actual individual behavior, e.g. as in actively writing secure code.

3 Embedded Challenges

In this work, we address three different cybersecurity challenges, targeting specific security vulnerabilities. This work's challenges are Sorting - Time Side Channel, Complex Factory (invalid memory access), and TOCTOU Race Condition. All challenges were implemented in the C++ programming language and integrated into the Sifu platform. Table 1 presents these challenges along with security vulnerabilities and guidelines contained in them. In subsection 3.1, we briefly introduce the Sifu Platform. Next, we describe implementation details of the assessment of the cybersecurity vulnerabilities of the three challenges. We also discuss the evaluation of the implementation of the challenges. Finally we refer to the setup and results of the empirical study performed with participants from academia. Notice that the proposed vulnerability detection methods run additionally to several already existing vulnerability detection mechanisms in the Sifu platform.

3.1 Sifu Platform

Sifu is a web-based CyberSecurity Awareness Platform [11]. This platform was developed in the industry with the aim to raise software developers' awareness on secure coding. In the industry, this platform is typically embedded in a serious game called CyberSecurity Challenges. The platform contains several exercises (challenges) that are presented to participants in the form of a project, e.g. in C/C++. These challenges contain one or more vulnerabilities in the code. The task of the player is rewrite the code such that it does not contain the vulnerability, while still performing the intended functionality. All interactions between the player and the platform takes place through the web interface.

Once the player has made changes to the code, he or she can submit the code to the backend, which will analyse the submitted codeand provide feedback to the player. The goal of the analysis performed in the backend is to assess the presence of cybersecurity vulnerabilities in the code submitted by the player in terms of secure coding guidelines. The goal of the feedback is, depending on the results of the cybersecurity assessment, to either indicate to the player that the challenge has been solved, or to guide the player to the correct solution by means of hints.

Figure 1 shows the main components of the Sifu platform's backend. The automatic assessment of challenges is performed using several components: pre-processor, compilers, static and dynamic application security tools, unit tests and run-time application security tests. An artificial intelligence component collects the results of these tools, performs the cybersecurity assessment and generates hints. The Sifu platform can be deployed in a local intranet server or in a cloud environment, enabling remote awareness workshops possible.

Table 1. Secure coding guidelines disregard list

Challenge	Rule	Severity	Likelihood	Description	Line number
Complex Factory	MEM31-C [19]	Medium	Probable	Free dynamically allocated memory when no longer needed	No destructor
	EXP35-CPP [19]	High	Probable	Do not read uninitialized memory	25
	EXP45-CPP [19]	High	Probable	Do not access an object outside of its lifetime	18
	MEM51-CPP [19]	High	Likely	Properly deallocate dynamically allocated resources	33
	CTR50-CPP [19]	High	Likely	Guarantee that container indices and iterators are within the valid range	18, 25
	ARR31-C [19]	High	Probable	Ensure size arguments for variable length length arrays are in a valid range	6
	CWE-315 [20]	Medium	Likely	Double free	33
	CWE-416[20]	High	Likely	Use after free	18, 25
Sorting	CWE-208 [20]	High	Likely	Observable Timing Discrepancy	7-12
TOC-TOU	CWE-367 [20]	High	Probable	Time-of-check Time-of-use (TOCTOU) Race Condition	7-12

For further information and details on the platform, we refer the reader to [10].

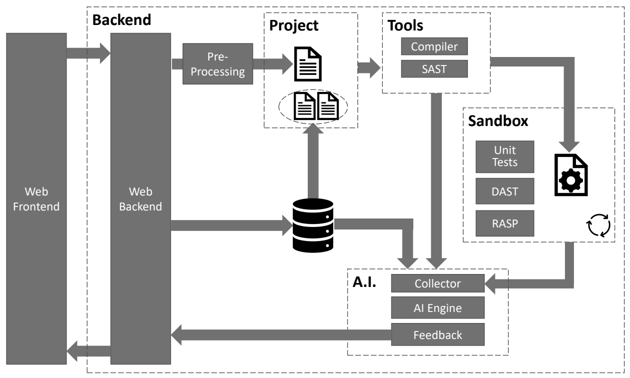


Figure 1. Architecture of the Sifu Platform

3.2 Sorting - Time Side Channel

The goal of this challenge is to raise awareness on the player on a well-known security vulnerability called the *time side channel* (TSC) [26]. A time side-channel occurs when the execution time of an algorithm is different for different inputs of the same size. In general, side-channels, such as time, can leak data and cause security problems [23]. This type of security vulnerability is critical in embedded systems and programming, since execution time is closely related with power consumption. Note that time side-channels can be easily observed and measured directly on hardware, such as in an embedded system through an oscilloscope, and measuring power-consumption profiles. This type of vulnerability can originate in both a poor implementation of hardware components, or non-constant time algorithms implemented in software. A typical example of a software algorithm that is vulnerable to time side-channels, is a string comparison function which returns, i.e. stops comparing the two strings, whenever the first difference is found. The run time of the algorithm thus depends on the initial number equal characters. However, any algorithm that does not run in constant time can leak information.

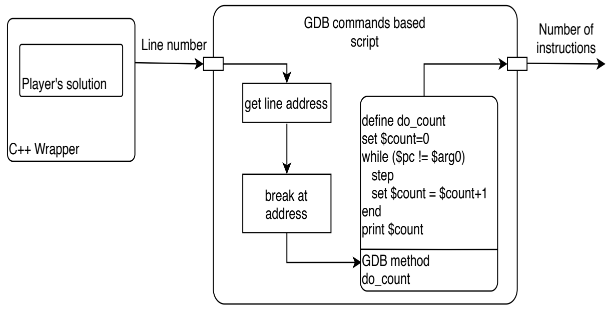
Listing 1. sort.cpp

```
1  #include <vector>
2  using namespace std;
3
4  // This function sorts a vector of int
5  // Goal: implement the function
6  void sort(vector<int> &list) {
7    size_t i, j;
8    for (i = 0; i < list.size(); i++){
9      for (j = 0; j < list.size()-1; j++){
10       // ...
11     }
12   }
13 }
```

In this work, we focus on a different algorithm - a array sorting algorithm. The secure coding guideline which addresses this vulnerability is given by CWE-208: "Observable Timing Discrepancy" [20].

The source code of the challenge that is presented to the participant is shown in Listing 1. To solve the challenge, the player needs to implement a constant-time sorting algorithm, i.e. a sorting algorithm where the execution time only depends on the number of elements to be sorted and not on their individual values or positions.

To address the assessment of the existence of a timing side-channel vulnerability, in code submitted by a player in the Sifu platform, we have searched for existing tools and libraries that could assist in this task. Two requirements that this tool or library must follow is that 1) it should be independent of an embedded system, and 2) it should be easily run in any environment, such as in the cloud. A possible solution to this problem is to perform the evaluation by means of an embedded system simulator. This, however, is not practical since the assessment could incur a considerable delay, making a practical usage in the Sifu platform difficult. Furthermore, we could not find any library that could easily be used in practice to detect this vulnerability. Therefore, we propose to use the GNU Debugger, by means of a dedicated python program, as shown in Figure 2. In this solution, GDB is used to count the number of steps that the entire sorting function needs to execute, i.e. from start to end. We assume that number of steps required to run the code is highly correlated with the time it takes to execute it, and ignore any possible issues relative to cache misses. One of the main advantages of this solution is that the measurement is no longer dependent on actual CPU run time, and can be deployed in any environment. While this method possibly takes more time to perform the assessment than running the code in bare-metal, it will take considerable less time than running an entire embedded hardware simulator. However, the true result might be dependent on the number of clock-cycles of each assembly instruction.



```
define do_count
set $count=0
while ($pc != $arg0)
  step
  set $count = $count+1
end
print $count
```

Figure 2. Count instructions diagram

Figure 2 shows an overview on how to perform the cybersecurity assessment of the participant's code. The code submitted by the participant is embedded in a C/C++ project which contains a wrapper that calls the sorting function. This small project is then compiled. A python script starts a GDB session which is used to debug the compiled project. This script interacts with GDB in the following three steps: 1) create a breakpoint at function call, 2) execute the function step-wise, and 3) step the code until it returns from the function. During the function's step-wise execution, the total number of steps (iterations) is recorded in an internal GDB variable. To assess the TSC, the C++ wrapper calls the sorting function with at least two different input vectors of the same size, e.g., using a sorted array and an unsorted array. The python script measures the number of steps that each function call takes to sort the vector. Comparing the number of steps in both cases is used to assess the presence of a time side-channel vulnerability. In our experiments, we have performed the step-wise measurements using the GDB *step* and *stepi* commands. The step command executes each line of code, while the *stepi* executes each assembly instruction.

3.3 Complex Factory

The main goal of this challenge is to raise the awareness on invalid memory access. However, this challenge contains several additional vulnerabilities, as shown in table Table 1. The secure coding guidelines and vulnerabilities were chosen base on their likelihood [19] and adaptability to the challenge.

Listing 2 shows the code that is presented to the player. The code implements a C++ class that stores complex numbers in an internal buffer of a given maximum size. The maximum size of the buffer it set at construction time, when an instance of the class is created. Table 1 also details the line numbers where the code's vulnerabilities are present, together with the corresponding SEI-CERT secure coding guidelines [19] and vulnerabilities as defined by the Common Weakness Enumeration (CWE) [20].

Listing 2. ComplexFactory.cpp

```
1  #include "FCplx.h"
2  using namespace std;
3
4  /* Constructor allocates a
5     container with MAX elements */
6  FCplx::FCplx(int _max): max(_max)
7  {
    pos = 0;
    container = new complex<int>[max];
}

/* Stores a complex number in the
   container and returns a reference
   to it */
complex<int>& FCplx::create(int x, int y)
{
  complex<int> a = complex<int>(x,y);
  container[pos++] = a;
  return a;
}
/* Returns a reference to an element
   stored in the container
   index 1 returns first element */
complex<int>& FCplx::get(int index){
  return container[index - 1];
}

/* Frees the allocated array. After
   calling this method no further method
   calls are be allowed */
void FCplx::empty()
{
  delete container;
}
```

To detect code that disregards the secure coding guidelines presented in Table 1, two distinct methods are used: GCC sanitize flags and security tests. Sanitize flags are part of dynamic application security testing, and are provided to the compiler at compile time. They instruct the compiler to generate extra checks that are performed during code execution. The following are the sanitize flags which we use: *address*, *leak*, and *undefined behavior*. Address flag adds extra run-time checks on memory addressing, leak flag adds extra run-time checks related to memory allocation, and undefined behavior flag adds extra run-time checks on undefined behavior, as defined by the C++ standard.

The second method used to detect code vulnerabilities is through security testing. Security tests are used to test specific corner cases, and they try to expose a vulnerability during run-time. One example of a security test, e.g. to test CWE-315 (double free), is to delete the class variable twice. A secure solution should catch this problem and react accordingly; however, a poorly implemented solution will cause a double free error, which in turn will trigger the leak sanitizer. For each secure coding guideline, one or more security test is implemented which tries to trigger the corresponding vulnerability.

3.4 Race Condition

The goal of this challenge is to raise the awareness on race condition vulnerabilities. A race condition can occur when two or more concurrent processes try to access a shared resource, whereby at least one process tries to modify the shared resource. The time between the resource is read and the resource is modified is known as race window or critical section. Consequences of exploiting this types of vulnerability include denial-of-service, and privelege escalation.

While race conditions typically occur in shared memory, we propose a challenge based on files. In this case the shared resource is a file and the vulnerability is called a time-of-check, time-of-use (TOCTOUC). The security vulnerability is listed in the CWE database under CWE-367 [20]. Listing 3 shows the challenge that is presented to the player. To solve the challenge, the player needs to perform two steps: check if a file exists and, if it exists, modify its permissions. The race window occurs between the time that the existence of the file is checked until the attributes are modified. The problem is that a malicious user can change the file between the two operations and, therefore the code changes the permissions of the wrong file. Previous research that addresses this vulnerability on real-time embedded systems is [29, 32]. Note: one possible solution to this challenge is by using the fchmod, instead of the chmod function.

Listing 3. set_permissions.cpp

```
/* Check if the file exists, and change
   the mode of the file. Return true if
   everything was successful */
bool setPerm(char *fName, mode_t mode){
  // Check if the file exists
  FILE *f_ptr;
  // Change the mode
  if (chmod(fName, mode) == -1) {
    // Handle error ...
    return false;
  }
  return true;
}
```

This type of vulnerability can potentially be detected by means of static application security testing tools. However, to the best of our knowledge, except for commercial solutions, there is no open-source tool that can detect this type of vulnerability [22]. Furthermore, static application security testing tools are known to be unreliable [22, 24]. In light of this, we propose a simple solution based on an attack script and a wrapper function, as shown in Figure 3. The wrapper function calls the participants' solution multiple times while, in parallel, an attacker script is running and swapping the two files in an endeless loop. Both the wrapper and the attacker script stop executing when one of the following conditions is met: (1) the permissions are changed in the wrong file, or (2) a maximum number of iterations is achieved. Condition two will occur if there is no vulnerability in the code of the player, or the race condition could not be achieved, i.e. the vulnerability was not detected after all the iterations.

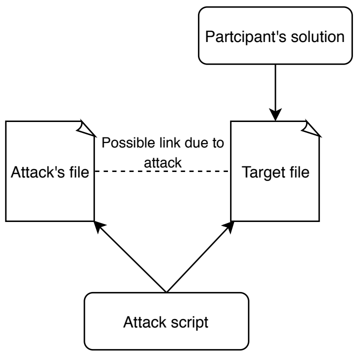


Figure 3. Top-level challenge structure

Note that, since the race condition is not reliably triggered, several tries need to be performed. We expect that the more iterations are performed, the higher the probability to detect the the vulnerability. In section 4 we present the results of evaluation on the trade-off between the detection probability and the total number of iterations.

Another possible solution to the detection problem is by artificially modifying the player's code and injecting a delay in the code's critical section. Although this solution can potentially increase the reliability to detect the vulnerability, it requires, however, a modification of the source code submitted by the participant. Furthermore, the method and implementation details to achieve this reliably is currently an open topic.

4 Evaluation

Validation of the presented cybersecurity assessment methods was performed through computer experimentation. Additionally, the challenges were deployed in an academic environment, followed by semi-structured interviews and a survey. In this section we give details on the evaluation process.

4.1 Evaluation of Design

For the Time Side-Channel challenge, an evaluation took place by means of two possible implementations of the sorting algorithm: with and without a time side channel. The algorithm containing the vulnerability was a standard bubble-sort algorithm, while the implemented solution was a bubble sort algorithm modified to swap elements with the same index, thus increasing run-time and avoiding the time side-channel vulnerability. The input to the sorting algorithm consisted of three random permutations of an integer vector size of 5 elements. The process was repeated 1000 times for both algorithms and for the *step* and *stepi* GDB commands respectively. Furthermore, the total execution required to get an answer from the backend was measured.

For the Race Condition challenge, we focus on the evaluation of the detection probability of the vulnerability. To compute this probability, we ran the python script 1000 times, and recorded the number of iterations required to detect the vulnerability. Based on these results, we can compute the cumulative density function of the detection probability $c(n)$ by normalizing the number of times that the vulnerability was detected in less than n cycles.

Since the Complex Factory challenge uses standard detection mechanisms, we have not performed an evaluation step for the security assessment.

All the tests were conducted on a PC with the following specifications: Ubuntu 18.04.4 LTS on an Intel i5-3427U CPU running at 1.80GHz with four cores and 8 GB of RAM.

4.2 Challenge Deployment in Academia

We have deployed the challenges in the Sifu Platform and asked several students, without previous industry experience, to evaluate them during two runs. Table 2 shows details on the demographics of these runs.

The participants were first given a short introduction to the platform, and were given instructions on how to use it. Next, the participants were given time to check and familiarize with the platform, and to solve the challenges. After solving the challenges, a semi-structure interview took place. Participation in the semi-structured interview was not mandatory, and the collected answers were anonymized.

The participants were also asked to answer a small survey consisting of eleven questions. The questions that were asked in the survey are shown in Table 3. Survey feedback answers were gathered using Google Forms. The answers to the questions were based a 5 point Likert scale [16] for agreement, i.e. *strongly disagree*, *disagree*, *neutral*, *agree*, and *strongly agree*. The questions are adopted from [11] and extended to target the academia. The adoption of the same questions allows, in the results section, to compare the answers from academia to answers from the industry.

Table 3. Survey questionnaire

	Survey Question
Q1	Paying attention to secure coding increases my code quality
Q2	University teaching includes awareness in secure coding
Q3	I learned new techniques and principles of secure software development
Q4	I know how to use the information about secure coding guidelines
Q5	I understand the importance of secure coding guidelines
Q6	Focusing on the challenges improves my practical secure coding skills
Q7	I have learned about new issues that I would like to check in my own code
Q8	I know where I can find more information about secure coding guidelines
Q9	The learning goals of the challenges were clearly explained
Q10	The help from the virtual coach was adequate

In total, six participants responded to the planned semi-structured interview. The semi-structured interview questions were based on the following questions: *what is the most significant advantage in participating in these challenges*, *what did not go well and you would like to change*, and *do you think that secure coding awareness increases the code quality overall*.

5 Results

This section show the results on the evaluation of the proposed vulnerability assessment schemes for different the challenges. This section also shows the analysis of the survey questions, and result from the semi-structured interviews. We also present a comparison of our survey results with two similar surveys by Gasiba et al. which were held in an industrial context.

5.1 Sorting - Time Side Channel

Figure 4 presents the results when using *step* (right plot) and *stepi* (left plot) GDB commands respectively. Both instructions, step and stepi, show only a line (i.e. single value) when there is no time side-channel vulnerability present in the code. This result comes as a consequence of having the same number of instructions for every input. Furthermore, the vertical lines show the worst and best case for no TSC. The observable difference in instruction count comes as a consequence that the stepi instruction needs to execute every assembly instruction of the source code.

The stepi GDB command takes more iterations than the step GDB command to perform the assessment of the vulnerability. This result is expected, since the stepi command executes the code one assembly instruction at a time, while the step command executes the code one source code line at a time. Due to their nature, the stepi command is more precise than the step command; however this represents a trade-off between precision and speed, since the stepi command is more than twenty two times slower than the step command.

Figure 4 also shows that, for the stepi GDB command, an increase in the number iterations between 31.8% and 95.3% between code without TSC and code with TSC is observed. The same figure shows that for the step command, the increase in the number of iterations is between 44.4% and 157% for code without TSC and code with TSC.

The second aspect to check the design is the reasonable delay after the participant has submitted a solution.

Table 2. Participants' information

No.	**Start Date**	**End Date**	**Participants**	**Where**	**Age Range**	**Field of Study**	**Educational Level**
1	5 Nov 2020	10 Nov 2020	12: Germany	Online	20-28	Electrical engineering, Computer engineering, Informatics	Bachelor's degree Master's degree
2	16 Nov 2020	23 Nov 2020	4: Germany	Online	22-25	Computer engineering, Communications engineering	Master's degree

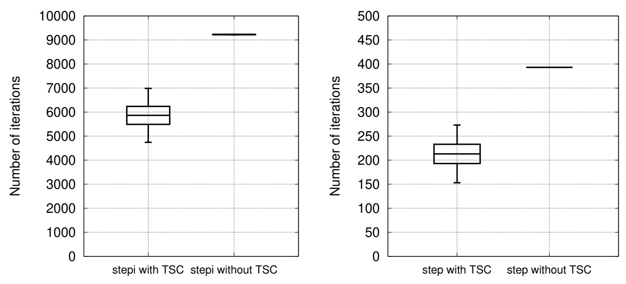


Figure 4. Step instruction comparison

We have measured the execution time of this implementation when running it in the Sifu platform. Table 4 shows the captured measurements together with 1st and 3rd quartile. Same cases were covered as in the Figure 4.

Table 4. Execution time measurement

Test case	Mean exec time [s]	Q1	Q3
Step with TSC	1.81	1.76	1.86
Step without TSC	2.19	2.18	2.20
Stepi with TSC	3.74	3.44	3.91
Stepi without TSC	5.43	5.39	5.48

5.2 Race Conditions

In the TOC-TOU Race Condition Challenge, we evaluated the detection probability of the attacking script. The results are shown in Figure 5. The x-axis of this figure shows the number of tried, i.e. iterations, performed to detect the vulnerability, while the y-axis shows the probability of detection.

This graph shows that, to achieve a detection rate of 99%, more than 3000 iterations are required. Since the execution time for a single round of this assessment method is very small (in the order of microseconds), we recommend to implement more than 10,000 iterations in a practical scenario. Our results show that, this value is reasonable and does not lead to considerable delay in the backend.

Figure 5 can be used as a guideline by practitioners who wish to implement this detection mechanism in their own deployments. Since the actual running time depends on the speed and on the CPU where the test is run, we recommend that practitioners start with the value 10,000 and run their own evaluation of the number of iterations that are required to achieve a detection probability of 99% or higher.

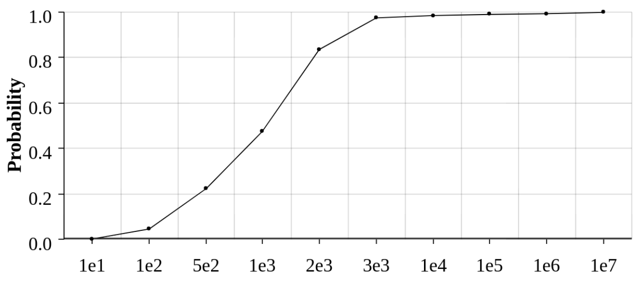


Figure 5. Success probability related to number of tries

5.3 Survey Results

Figure 6 shows the results of the survey which was based on the questions introduced in Table 3. Our preliminary results show, except for Q1 and Q2, an overall agreement with the survey question. The questions that include the highest agreement are Q6, Q7, Q9, and Q10. This means that the participants agree that the assistance provided by the virtual coach is adequate, thus providing a good indicator of the suitability of the proposed vulnerability detection methods. Participants also agree that focusing on secure coding challenges improves their security coding skills, therefore also their awareness on the secure coding guidelines. Finally, the participants have learned new vulnerabilities that they would like to check in their own code. This gives a good indicator that these types of games can motivate a positive behaviour towards secure coding.

The aspects there received the lowest amount of agreement, though still overall positive, are related with Q4 and Q8. This is not surprising since the Sifu platform is developed for the industry, and motivates the participants to find additional information on secure coding using internal resources; the same cannot be said for the academia, where further information should be searched in online forums or scientific publications. It is however surprising that the participants claim that playing the challenges allows them to know where to find more information about secure coding, and how to use this information.

We also observe that, in general, there is still a high level of neutral answers. The reason for this is not well understood and further research is necessary to understand this issue. Also, for Q1, it is surprising that the participants are not entirely sure if paying attention to secure coding increases their code quality. Although one possible reason for this might be the lack of specific training in secure coding and code quality, this aspect needs more search. Finally, another surprising result is

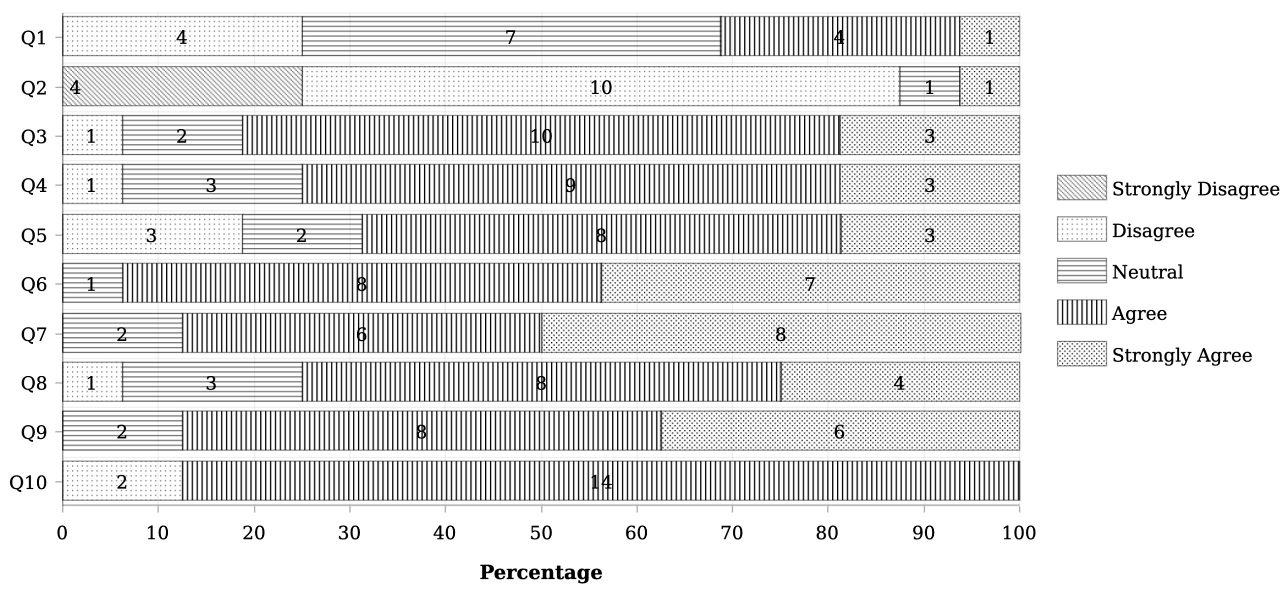


Figure 6. Survey Results

the one obtained in Q2, where the participants claim that university teaching does not include awareness on secure coding. This result is an indicator that this awareness campaigns are necessary to be held in the industry, and a specialized course that covers secure coding and secure coding guidelines might be beneficial. Attending this course would better prepare the students for a future career in the industry.

5.4 Comparison to Previous Work

Table 5 shows a comparison of the current work, and the work by Gasiba et al. (see [13], and [10]). The main difference is that, in the current work, we evaluate the CyberSecurity Challenges in the academia, while previous work evaluates these in an industrial context. Both the current work as the previous work show a general agreement in relation to all the survey questions. However, one important difference is the fact that Q5 has a higher agreement level for the industry as in the academia. We think that the main reason for this difference might be related with the fact that the usage of secure coding guidelines are typically mandated by security policies in the industry, e.g. as a result of requirements from cybersecurity standards, while this is not the case in academia. We also observe that Q4 and Q8 exhibit a large amount of neutral answers for both academia and also for the industry. The reason behind this is not entirely understood and requires further investigation.

In terms of disagreement, we observe that Q3, Q7, and Q8 have a similar amount of disagreement in both the academic setting and also in the industrial setting. One common positive point in all the surveys, industry and academia, is the fact that the support by the coach is seen as an important factor. The factor that has the largest amount of difference between the academia and industry is Q8 - knowing where to find more information; Based on our experience, we think that this might be related with the fact that in an industrial environment, there are internal procedures related with secure software development life-cycle which establish where further information can be obtained. This might justify the higher value for academia and lower value for the industry.

Surprisingly there is about 8% of negative answers in relation to Q9 - the goals of the challenge are clearly stated - for the industry, while 0% for academia. This effect might be related with the diversity of software developers in the industry; however, further investigation is needed to understand this discrepancy.

In summary, although there are some differences observed in the results of academia vs industry, overall there is a large agreement for all the survey questions. This fact gives a good indication that the Sifu platform and its challenges are well suited for both academia and industry.

Table 5. Comparison with previous work

Present work				Gasiba et al. [13]				Gasiba et al. [10]			
Question	**Negative**	**Neutral**	**Positive**	**Question**	**Negative**	**Neutral**	**Positive**	**Question**	**Negative**	**Neutral**	**Positive**
Q1	25.0%	43.7%	31.3%								
Q2	87.6%	6.2%	6.2%								
Q3	6.2%	12.5%	81.3%	Q1.1	12.5%	7.1%	80.4%	X1	0.0%	10.0%	90.0%
Q4	6.2%	18.8%	75.0%	Q6.1	8.9%	28.6%	80.3%				
Q5	18.7%	12.5%	68.8%	Q10.9	0.0%	5.3%	94.7%	X9	0.0%	0.0%	100.0%
Q6	0.0%	6.2%	93.8%	Q7.1	3.6%	14.3%	82.1%	F2	0.0%	4.0%	96.0%
Q7	0.0%	12.5%	87.5%	Q8.1	10.7%	5.4%	83.9%				
Q8	6.3%	18.7%	75.0%	Q9.1	12.5%	31.2%	55.4%	X8	0.0%	10.0%	90.0%
Q9	0.0%	12.5%	87.5%	Q11.1	8.9%	8.9%	82.2%	F8	8.0%	8.0%	84.0%
Q10	12.5%	0.0%	87.5%	Q13.1	1.8%	12.5%	85.7%	X6	0.0%	0.0%	100.0%
16 participants				56 participants				25 participants			

5.5 Semi-Structured Interviews

After using the Sifu platform and solving each of the proposed challenges, participants were interviewed. Analysis of the participants' answers resulted in their classification into three groups: *Benefits*, *Application in Embedded systems*, and *Drawbacks*. Table 6 shows the top 10 quotes from participants, along with a mapping to the individual groups. The group *Benefits* covers the user experience and what the participants thought was the most beneficial outcome of the challenges. Group *Application in Embedded systems* was related to our primary goal, which was not explicitly mentioned, with a reason to see if the participants see a possible application of this platform for training secure coding in embedded systems. Finally, the group *Drawbacks*, covers negative aspects experienced by the participants. The feedback that we have collected allows to understand the strong and weak points of the platform, and also to improve the it further.

The grouping of certain answers and comments answered how participants perceive the platform and see a possible application in embedded programming. The main outcome of the semi-structured interviews are as follows. For the perceived benefits of the platform, the fact that it is used as a game was positively perceived. Furthermore, playing the games exposes the participants to secure coding tasks, which is also perceived as being beneficial. In therms of the embedded programming group, some participants suggested that the platform might be used in a standard course at the university, which might be beneficial for those who work on security critical systems, e.g. in critical infrastructures.

Although most of the feedback was positive, we collected some drawbacks. The two main negative points

Table 6. Quotes from Participants

No	Quote from Participant	Group
1	I believe that accounting for security while coding is beneficial in writing more efficient code.	Benefits
2	Learning really useful strategy to make significantly more quality code	
3	The best thing is that platform acts as a game	
4	I think it is good to be exposed to an important aspect of coding which is handling or accounting for security issues or potential vulnerabilities	
5	Suitable for training secure coding on embedded systems	Embedded Programming
6	This could be used as a training platform for some courses at my university	
7	This will be a real advantage for users who are working on security critical systems.	
8	The hints given by the chat bot were not always accurate or precisely leading to the nature of the problem in the code.	Drawbacks
9	The user interface is minimalistic for nowadays standards.	
10	Make a more sophisticated user interface and experience	

are related with the user interface, and with the precision of the feedback given by the virtual coach. Related to quotes 9 and 10, interviewees gave a more detailed answer in our discussion. One interviewee said that having a standard debugging tool would ease the challenges and increase the learning factor. As well to compete with other online learning platforms, more details on the design have to be applied.

The collected positive and optimistic answers indicate that participating in these challenges can potentialy lead to raising secure coding awareness in academia. The answers are also encouraging towards validating the suitability of the proposed vulnerability assessment methods.

5.6 Discussions

In this work we have presented the implementation and evaluation of cybersecurity challenges for the Sifu platform. The implementation of the challenges covered technical aspects on how to evaluate the presence of a given vulnerability in source code, while the evaluation was performed in several ways: computer experiments, survey, semi-structured interviews and comparison to previous work.

In terms of technical implementation, we have discussed a mechanism to detect time side-channel, and race conditions. The main idea for the detection of time side-channel is to use a standard debugger, such as GDB, together with a wrapper and a python script to control the debugger. By stepping through the code, we can evaluate the number of steps that the algorithm would require to run from beginning to the end. In the results section we present practical advice for practitioners who want to implement this method, in particular we discuss about the trade-off between precision and execution time. Notice that we propose a simplified method to assess the presence of a time-side channel. In practice, other effects will impact the running time of the algorithm, such as number of cycles per instruction, hyper-threading, cache misses, and CPU internal pipelines, and therefore the presence of time-side channels. However, although the proposed method is simple, it allows to raise awareness on time-side channels through a serious game. Furthermore, the reason why we observe highly correlated results between the "step" and "stepi" method is related to the fact that the implemented algorithm mostly uses mathematical operations and no function calls.

For the race condition vulnerability, we propose to write a script that attacks the function written by the participant. We discuss the trade-off between probability of detecting the vulnerability and execution time. The reason why execution time is crucial, is that the Sifu platform is an interactive platform. The higher the delay in the backend, the less interactive the platform becomes, which could lead to problems with user experience.

One major finding in our survey results is the fact that the students claimed a lack of awareness on secure coding during courses at the university. While this result cannot be generalized to every university and every course, it does raise the need to address this topic in general, as the students of today are the workforce of the future. The survey results also indicate that the Sifu platform might be suitable to address this awareness in secure coding at the university. In particular, we have received positive indications that the platform could be integrated into the standard teaching curricula. A comparison with two previous surveys performed in an industrial setting was also discussed. Both the studies performed in the academia as also in the industry show encouraging results on the suitability of the platform as a means to raise awareness on secure coding. However, there are small differences between the surveys that indicate small discrepancies. We believe that a practitioner who wishes to deploy or refine these types of challenges in the academia, can find valuable information in these studies, to guide in their decision making.

5.7 Threats to Validity

Possible threats to our conclusions include: number of participants, study field and experience, and participant bias. Our preliminary results are mostly positive. This might be related to the relatively low number of participants to the survey, since it was not mandatory. Therefore, some negative comments might not have been been captured. Participants have different study fields, backgrounds, and are at different levels in their studies. Although the number of participants is limited, it is in line with comparable empirical studies. Therefore, some participants might find the challenges easy, while other might have more difficulties to solve them. This might lead to different answers to the survey and also to the semi-structured interview and, therefore. A more detailed analysis and research on the answers given by the different groups would be required to understand possible bias effects.

Finally, since the participants are aware of the purpose of the study, a positive bias cannot be discarded. In particular, participants might respond to the survey in a way that they think the authors expect them to answer. Nevertheless, the results obtained in this work are in alignment with previous work that was done in an industrial environment. Therefore, we do not think that considerable different conclusions from the ones presented in this work would be obtained by increasing the number of survey participants.

6 Conclusions

Security bugs, when exploited, often can lead to serious software vulnerabilities. Nowadays, secure coding guidelines exist to teach software developers and make them aware of software vulnerabilities and how to write secure code that avoids these vulnerabilities. However, not all software developers are knowledgeable about these or secure coding standards in general. This is true for the industry and, the present study also finds out that this might also be true in academia.

To address this issue, this work extends previous research conducted on the Sifu platform. We introduce three challenges with security vulnerabilities common in embedded programming that can be integrated into university teaching curricula. This paper consists of two main parts: 1) a brief description of how to implement the challenges in terms of evaluation of the presence of the vulnerability in the participant's source code, and 2) an evaluation of the challenge design and evaluation of the challenges in an academia setting.

In the first part, three different C++ challenges were implemented on Sifu's platform. The implementation process and decision-making were briefly explained. A critical part of our work is how to test particular security vulnerabilities that are presented in the challenges. We give a detailed explanation of the architecture and the implementation of testing particular vulnerabilities.

In the second part, we evaluate the design from a technical view, and an empirical view. Our results indicate that the proposed methods can be used to detect the challenges' vulnerabilities. Additionally we give practical advice on the implementation of the challenges. Through two trial runs in academia, we collected answers on a survey and performed a semi-structured interview. Our preliminary results give a good indication that the challenges are adequate to raise secure coding awareness for students. Additionally the results for the semi-structured interview give a positive indication on the suitability of the challenges for academia, and give good insight into future improvements, e.g. on user experience. Finally, we have performed a comparison with previous studies, thus comparing results from industry to results from academia. While the majority of the results indicate a good agreement between industry and academia, the small differences between both can serve as a guide to practitioners who wish to deploy the Sifu platform in an academic setting.

As further steps, the authors would like to investigate the usage of popular open-source static code analysis tools, and tapping system calls to improve the platform's detection mechanisms. Furthermore, the authors would like to address these tools' usage to improve the platform's hint mechanism.

Acknowledgments

The authors would like to thank the survey participants for their useful and insightful discussions and for their participation in the test run. This work is partially financed by national funds through FCT - Fundação para a Ciência e Tecnologia, I.P., under the projects FCT UIDB/04466/2020 and UIDP/04466/2020. Furthermore, the fourth author thanks the Instituto Universitário de Lisboa and ISTAR, for their support.

References

[1] Richard Amankwah, Patrick Kudjo, and Samuel Yeboah. 2017. Evaluation of Software Vulnerability Detection Methods and Tools: A Review. *International Journal of Computer Applications* 169 (07 2017), 22–27. https://doi.org/10.5120/ijca2017914750

[2] Apextechservices. 2017. NotPetya: World's First $10 Billion Malware. https://tinyurl.com/y6mkok57

[3] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2017. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, Timisoara, Romania, 161–168.

[4] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu. 2018. A Comparison of Static Analysis Tools for Vulnerability Detection in C/C++ Code. Romanian National Authority for Scientific Research and Innovation.

[5] Bence Babati, Gábor Horváth, Viktor Májer, and Norbert Pataki. 2017. Static Analysis Toolset with Clang. In *Proceedings of the 10th International Conference on Applied Informatics*. ACAI, Eger, Hungary, 23–29.

[6] Hanmeet Kaur Brar and Puneet Jai Kaur. 2015. Article: Comparing Detection Ratio of Three Static Analysis Tools. *International Journal of Computer Applications* 124, 13 (August 2015), 35–40. Published by Foundation of Computer Science (FCS), NY, USA.

[7] The Clang Community. 2020. Clang: a C language family frontend for LLVM. https://clang.llvm.org/index.html

[8] Ralf Dorner, Stefan Gobel, Wolfgang Effelsberg, and Josef Wiemeyer. 2016. *Serious Games: Foundations, Concepts and Practice*. Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-40612-1

[9] Tiago Gasiba, Kristian Beckers, Santiago Suppan, and Filip Rezabek. 2019. On the Requirements for Serious Games geared towards Software Developers in the Industry. In *Conference on Requirements Engineering Conference*, Daniela E. Damian, Anna Perini, and Seok-Won Lee (Eds.). IEEE, Jeju, South Korea, 286–296. https://doi.org/10.1109/re.2019.00038

[10] Tiago Gasiba, Ulrike Lechner, and Maria Pinto-Albuquerque. 2020. Sifu - A CyberSecurity Awareness Platform with Challenge Assessment and Intelligent Coach. *Special Issue of Cyber-Physical System Security of the Cybersecurity Journal* 3, 1 (10 2020), 1–23.

[11] Tiago Gasiba, Ulrike Lechner, and Maria Pinto-Albuquerque. 2021. CyberSecurity Challenges for Software Developer

Awareness Training in Industrial Environments. *16. Internationale Tagung Wirtschaftsinformatik* (2021), 1–17.

[12] Tiago Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Daniel Mendez Fernandez. 2020. Awareness of Secure Coding Guidelines in the Industry - A first data analysis. In *TrustCom 2020: International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, Guangzhou, China, 1–8.

[13] Tiago Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Anmoal Porwal. 2020. Cybersecurity Awareness Platform with Virtual Coach and Automated Challenge Assessment. In *6th Workshop On The Security Of Industrial Control Systems & Of Cyber-Physical Systems (CyberICPS)*. Springer, Guildford, UK, 1–16.

[14] Tiago Gasiba, Ulrike Lechner, Filip Rezabek, and Maria Pinto-Albuquerque. 2020. Cybersecurity Games for Secure Programming Education in the Industry: Gameplay Analysis. In *First International Computer Programming Education Conference (ICPEC 2020) (OpenAccess Series in Informatics (OASIcs), Vol. 81)*, Ricardo Queirós, Filipe Portela, Mário Pinto, and Alberto Simões (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10:1–10:11.

[15] Norman Hänsch and Zinaida Benenson. 2014. Specifying IT security awareness. In *2014 25th International Workshop on Database and Expert Systems Applications*. IEEE, Munich, Germay, 326–330.

[16] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *Current Journal of Applied Science and Technology* 7 (2015), 396–403.

[17] Ryo Kurachi, Hiroaki Takada, Masato Tanabe, Jun Anzai, Kentaro Takei, Takaaki Iinuma, Manabu Maeda, and Hideki Matsushima. 2018. Improving secure coding rules for automotive software by using a vulnerability database. In *2018 IEEE International Conference on Vehicular Electronics and Safety (ICVES)*. IEEE, Madrid, Spain, 1–8.

[18] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[19] Carnegie Mellon. 2020. SEI CERT Coding Standards. https://wiki.sei.cmu.edu/confluence/display/seccode Online. Accessed 19 November 2020.

[20] MITRE. 2020. Common Weakness Enumeration: CWE. https://cwe.mitre.org/ Online. Accessed 25 November 2020.

[21] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. 2013. Dismal code: Studying the evolution of security bugs. In *{LASER} 2013 ({LASER} 2013)*, Vol. 49. USENIX, New York, United States, 37–48.

[22] Jonathan Moerman, Sjaak Smetsers, and Marc Schoolderman. 2018. Evaluating the performance of open source static analysis tools. *Bachelor Thesis, Radboud University, The Netherlands* 24 (2018), 1–66.

[23] Reza Montasari, Amin Hosseinian-Far, Richard Hill, Farshad Montaseri, Mak Sharma, and Shahid Shabbir. 2018. Are Timing-Based Side-Channel Attacks Feasible in Shared, Modern Computing Hardware? *International Journal of Organizational and Collective Intelligence* 8 (04 2018). https://doi.org/10.4018/IJOCI.2018040103

[24] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. 2018. Myths and facts about static application security testing tools: an action research at Telenor digital. In *International Conference on Agile Software Development*. Springer, Cham, Switzerland, 86–103.

[25] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. 2016. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, New York, United States, 690–703.

[26] François-Xavier Standaert. 2010. Introduction to side-channel attacks. In *Secure integrated circuits and systems*. Springer, Boston, MA, 27–42.

[27] Madiha Tabassum, Stacey Watson, Bill Chu, and Heather Richter Lipford. 2018. Evaluating Two Methods for Integrating Secure Programming Education. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. ACM, New York, United States, 390–395. https://doi.org/10.1145/3159450.3159511

[28] Mohammad Tahaei and Kami Vaniea. 2019. A Survey on Developer-Centred Security. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, Stockholm, Sweden, 129–138.

[29] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2017. Automatic Detection and Validation of Race Conditions in Interrupt-Driven Embedded Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/3092703.3092724

[30] Michael Whitney, Heather Richter Lipford, Bill Chu, and Tyler Thomas. 2017. Embedding Secure Coding Instruction Into the IDE: Complementing Early and Intermediate CS Courses With ESIDE. *Journal of Educational Computing Research* 56 (05 2017), 073563311770881. https://doi.org/10.1177/0735633117708816

[31] Chauncey Wilson. 2014. *Semi-Structured Interviews*. Elsevier, Boston, 23–41. https://doi.org/10.1016/B978-0-12-410393-1.00002-8

[32] Chen Yan, Xu Xiaofeng, Li Xiaochao, and Guo Donghui. 2010. Race Condition and Its Analysis Approach of Real-time Embedded Systems. *Journal of Computer Research and Development* 47, 7, Article 1201 (2010), 9 pages. http://crad.ict.ac.cn/EN/abstract/article_1751.shtml

[33] 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, New York, United States, 886–896.

[34] Z. Zhioua, Y. Roudier, S. Short, and R. B. Ameur. 2016. Security Guidelines: Requirements Engineering for Verifying Code Quality. In *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*. IEEE, Beijing, China, 80–85. https://doi.org/10.1109/REW.2016.028