Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2023.036695
Article

SMINER: Detecting Unrestricted and Misimplemented Behaviors of Software Systems Based on Unit Test Cases

Kyungmin Sim, Jeong Hyun Yi and Haehyun Cho*

School of Software, Soongsil University, Seoul, 06978, Korea
*Corresponding Author: Haehyun Cho. Email: haehyun@ssu.ac.kr
Received: 09 October 2022; Accepted: 15 January 2023

Abstract: Despite the advances in automated vulnerability detection approaches, security vulnerabilities caused by design flaws in software systems are continuously appearing in real-world systems. Such security design flaws can bring unrestricted and misimplemented behaviors of a system and can lead to fatal vulnerabilities such as remote code execution or sensitive data leakage. Therefore, it is an essential task to discover unrestricted and misimplemented behaviors of a system. However, it is a daunting task for security experts to discover such vulnerabilities in advance because it is time-consuming and error-prone to analyze the whole code in detail. Also, most of the existing vulnerability detection approaches still focus on detecting memory corruption bugs because these bugs are the dominant root cause of software vulnerabilities. This paper proposes SMINER, a novel approach that discovers vulnerabilities caused by unrestricted and misimplemented behaviors. SMINER first collects unit test cases for the target system from the official repository. Next, preprocess the collected code fragments. SMINER uses pre-processed data to show the security policies that can occur on the target system and creates a test case for security policy testing. To demonstrate the effectiveness of SMINER, this paper evaluates SMINER against Robot Operating System (ROS), a real-world system used for intelligent robots in Amazon and controlling satellites in National Aeronautics and Space Administration (NASA). From the evaluation, we discovered two real-world vulnerabilities in ROS.

Keywords: Security vulnerability; test case generation; security policy test; robot operating system; vulnerability assessment

1 Introduction

Software vulnerabilities that threaten software companies and end users have been a constant threat for the past 25 years. If a software vulnerability is exploited by an attacker before a software vendor has detected the vulnerability, such a situation would make numerous users under a serious threat until the vulnerability is mitigated. Therefore, tremendous research efforts have been conducted to automatically detect and mitigate security vulnerabilities because manual auditing of source code to find software vulnerabilities is a daunting, error-prone, costly, and time-consuming task. To be specific, automatically discovering memory corruption bugs such as fuzzing has been one of the most active areas in the research community because memory corruption bugs are still the dominant root causes of reported security vulnerabilities [1–3]. Commonly, the goal of approaches for detecting software vulnerabilities is to effectively find vulnerabilities without the manual efforts of security analysts. To this end, such automated approaches employ various analysis techniques and algorithms that explore complex execution paths of software systems by themselves. For example, feedback-based fuzzers such as American Fuzzy lop (AFL) use smart algorithms that automatically tailor input data based on the coverage which is the amount of code tested by a fuzzer.

However, discovering the other types of vulnerabilities in an automated way is a field that has not been explored yet in depth [4]. As we can observe in the cases of Apache Struts [5], Shellshock [6], and Heartbleed [2], security design flaws do not affect the functionality of software systems, but they can cause serious security accidents such as remote code executions [7–9]. However, these vulnerabilities are difficult to detect and there have not been many research efforts conducted to develop effective, automated approaches that can find such security vulnerabilities [10]. The lack of approaches for detecting such vulnerabilities based on design flaws motivated us to develop a practical solution that verifies whether a software system has implementation errors without huge manual efforts.

Designing and implementing a fully automated solution for finding design or implementation flaws that affect the security of software systems is a challenging issue. According to 2022 Top 25 The Open Web Application Security Project (OWASP) Prating Scale for the list of software weaknesses, incorrect design and implementation defects are ranked high. This is mainly because such defects cannot be discovered by using rigorous unit tests or by performing fuzz testing as far as a software system has no functional errors and memory errors, while other security vulnerabilities (e.g., memory corruptions) can be detected by checking whether a program makes a crash or not. In addition, the manual finding of such flaws in large software systems requires tremendous cost and time.

To alleviate the problem of detecting design or implementation flaws, we propose a semi-automated approach called SMINER. It can help to perform assessing security vulnerabilities that can be caused by unrestricted and misimplemented behaviors of software systems. In this work, we define unrestricted and misimplemented behaviors as follows. An unrestricted behavior occurs when a system does not properly limit or restrict operations that must be controlled securely. On the other hand, misimplemented behavior is a result of the invalid or inappropriate implementation of functionalities from the perspective of security. It also refers to unexpected behavior that occurs under a particular situation or program state where the behavior should not happen. Therefore, a misimplemented behavior does not affect the functionalities of a system in general but can bring erroneous results related to the security feature.

The architecture of SMINER consists of three steps: pre-processing, security policy test, and generating test cases. First, SMINER goes through a pre-processing process that collects basic unit test cases implemented for testing each functionality of a software system. SMINER, then, analyzes them after tokenizing them. Specifically, it disassembles code snippets into classes, functions, comments, and variables. These classified data will be assembled to generate test cases for discovering unrestricted and misimplemented behaviors of a target system. We identify such behaviors of a target system based on intuitively recognizable features which can be provided by disassembled test suites. Therefore, if only poorly developed unit test cases that do not have rich information are available, SMINER may not provide enough intuitions to users to identify detailed functionalities of the target system. Next, to discover security issues that may occur in the system, security policy testing modules showing security policies are set by users. Finally, SMINER generates new test cases for detecting and discovering unrestricted and misimplemented behaviors. As such, by utilizing well-implemented unit test cases and features captured from disassembled test cases, SMINER can avoid huge manual effort for analyzing a target system for generating effective test cases to detect unrestricted and misimplemented behaviors.

We performed qualitative and quantitative evaluations to demonstrate what types of vulnerabilities SMINER can detect and whether SMINER can discover actual vulnerabilities against a real-world system or not. In our evaluation, we used SMINER for testing the Robot Operating System (ROS) employed by a lot of academic projects and commercial products (e.g., intelligent robots in Amazon and satellites in National Aeronautics and Space Administration (NASA)) [11]. As result, we discovered two vulnerabilities in ROS which show the effectiveness of test cases generated by SMINER. In summary, we make the following three contributions to this work.

- We define unrestricted and misimplemented behaviors and demonstrate how they can bring severe security vulnerabilities.
- We propose a novel approach, named SMINER, that can generate test cases to discover security vulnerabilities based on unrestricted and misimplemented behaviors of software systems, avoiding huge manual effort.
- We evaluate SMINER against a real-world system (ROS) to show the effectiveness of test cases generated by SMINER. We revealed two actual vulnerabilities in the system and reported them.

2 Background

We have been facing a ton of security vulnerabilities such as remote code executions, and sensitive information leaks in various software systems. In general, the main reasons causing such security vulnerabilities are two-fold: (1) memory corruption errors (e.g., out-of-bound memory accesses, uninitialized memory uses); and (2) security design flaws. In this section, we present the root causes and how these vulnerabilities can be exploited.

2.1 Memory Corruption Errors

Memory corruption errors such as buffer overflow, out-of-bound access, use-after-free, and double-free are critical security issues impacting programs developed in unsafe languages (i.e., C and C++).

Most of them are caused by programming errors. However, these errors frequently occur due to the size and complexity of real-world software. By exploiting such vulnerabilities, an attacker might be able to change an address to execute malicious code snippets or accesses sensitive data in memory [12]. As such, memory safety violations remain a clear and present danger to computer security. They are still the top rank of dangerous software vulnerabilities for the past 27 years and ranked 5th in the 2020 Top 25 Most Dangerous Software Weaknesses [13].

2.2 Severe Security Vulnerabilities Caused by Design Flaws

We introduce three real-world cases where a design flaw became a critical security threat to numerous users.

2.2.1 Case 1: Vulnerabilities in Apache Struts

Apache Struts framework, used for Java Web Application development based on the The Model-View-Controller (MVC) framework, possessed severe vulnerabilities because it did not control unrestricted behaviors on this framework [14]. Apache Struts framework supports Object-Graph Navigation Language (OGNL) expressions that are used to get or set property values of Java objects. Furthermore, Apache Struts can execute the OGNL expressions received from a HyperText Transfer Protocol (HTTP) request. Therefore, their execution of them should have been controlled carefully because OGNL expressions can access internal objects containing sensitive data of the system. However, unfortunately, the absence of such control mechanisms in the framework gave rise to many serious vulnerabilities that allowed arbitrary code executions. Attackers exploited these vulnerabilities by injecting exploit code in the HTTP request tag by using OGNL expressions to leak credential files or to open a backdoor.

2.2.2 Case 2: Shellshock

Shellshock is a set of vulnerabilities discovered in Unix Bash Shell, which can execute arbitrary commands [15]. The vulnerability allows an attacker to gain unauthorized access and arbitrary code execution privileges by executing arbitrary commands on environment variables that should not be available. Attackers started exploiting the Shellshock vulnerability after a few hours of the initial disclosure and millions of actual attacks related to the vulnerability were recorded within only a few days [3].

This case demonstrates that a design flaw can be a significant cybersecurity threat affecting numerous users. Also, the fact, that it took 22 years for the vulnerabilities to be discovered after the first release of the Bash version affected by them, demonstrates assessing design flaws in terms of security is of great importance and an essential task for eliminating potential cybersecurity threats.

2.2.3 Case 3: Heartbleed

Heartbleed is a critical security vulnerability in OpenSSL cryptographic software library that provides the Secure Sockets Layer (SSL)/Transport Layer Security (TLS) encryption used to secure the Internet. By exploiting the vulnerability, 64KB of memory can be read because the match between the payload included in the heartbeat request message and the length of the payload is not checked [10,16]. Heartbeat allows communication connections to be maintained without renegotiating connections each time in TLS and Datagram Transport Layer Security (DTLS) protocols. When the server responds to the heartbeat, it returns as many as the length of the payload received as a request from memory. Therefore, sensitive information such as private keys stored in the system memory can be leaked. This security design flaw made a great number of websites vulnerable.

2.3 Automated Vulnerability Assessment and Software Testing

In this section, we introduce existing automated vulnerability detection approaches and how they can detect vulnerabilities.

2.3.1 Fuzzing

Fuzzing is the state-of-the-art technique for discovering memory vulnerabilities in programs, causing a logic flaw or memory corruption, existing in the program. Fuzzing has two types of mutational fuzzing and generative fuzzing is a useful approach to finding memory-related vulnerabilities [17]. A mutational algorithm fuzzing [18–20] is a method of testing a program by randomly changing a given input value. Because mutation fuzzing generates an input value without an understanding of the model of input structure, it is easy and fast to implement the fuzzing. But, it is difficult to generate a valid input value for the target program.

Otherwise, a generative fuzzing approach [6,21] is a more effective fuzzing that generates effective input values based on modeling the structure of input data. Also, there are many research instrumenting target programs to maximize the effectiveness of fuzzing [22,23], and fuzzing operating systems [24,25].

In order to expose bugs through testing, the fuzzing must be able to distinguish between unexpected (buggy) program behavior. To identify it, Sanitizer [13] is used to recognize these various unexpected behaviors. Undefined Behavior Sanitizer (UBSan), one of them, uses a pointer (misaligned pointer or null pointer) or overflow (signed integer or between floating-point types) to catch the undefined behavior of a program. But, UBsan has some limitations. Binary must have enough debug info so that the runtime could figure out the source file or function name to match against the suppression. Also, check groups (like undefined) cannot be used in suppressions files, only fine-grained checks are supported.

2.3.2 Symbolic Execution

Symbolic execution [26,27] is a means to test a program by analyzing symbols that can execute all path conditions in the program. Symbolic execution has been used to detect vulnerabilities by providing effective input value and by tracking the path conditions by a generative fuzzing approach [28]. Symbolic execution extends the code coverage of fuzzing in that symbolic path conditions can be obtained even in complex programs. Thus, it is used to find the crash existing in the program by fuzzing.

2.3.3 Case 3: Automated Test Case Generation

We can define a test case as a set of actions executed on a system to verify that the system meets software requirements as designed and the system has no undefined behaviors. Therefore, we need to carefully implement test cases to verify that the various functions within the system perform as expected. However, designing effective test cases requires a significant amount of manual effort with costs. Many research efforts have been made to create automated and cost-effective test cases to find a variety of software errors on specific targets to address this issue [14,29–31]. Albeit such research efforts, to the best of our knowledge, there has been little attention on finding unrestricted and misimplemented behaviors that can corrupt access control policies, data integrity, and confidentiality. In the following section, we define such abnormal behaviors of software systems and introduce the goal of this work to find them by using a semi-automated approach that leverages unit test cases implemented by software developers.

3 Overview

Software vulnerabilities can lead us to perform unauthorized actions. Still, the most common security vulnerability class is caused by memory corruption bugs. Because such bugs that may introduce security vulnerabilities can crash programs, ideally, they can be found during software development and testing. In addition, as we presented in Section 2, design flaws that do not affect the functionalities of software but can cause incorrect, unexpected, or unintended behavior may bring severe security vulnerabilities such as remote code execution threatening numerous users. These vulnerabilities are difficult to detect due to the lack of automated assessment tools, and thus, can be hidden for a very long time as we observed in the Shellshock case (Section 2.2.2).

Goal. In this work, we aim to discover security vulnerabilities based on unrestricted and misimplemented behaviors of software systems. To this end, we propose a semi-automated approach, named SMINER, that performs vulnerability assessments via the security policy with test cases of targeted programs.

Unrestricted and Misimplemented Behaviors. We define two abnormal behaviors of software systems that the design of a system or implementation of the design flaws can bring.

(1) **Unrestricted Behavior:** An unrestricted behavior occurs when a system does not properly restrict, or limit actions requested by itself or from outside. For example, the Structured Query Language (SQL) Injection maliciously injects and executes arbitrary SQL queries, manipulating the database to behave abnormally, which can lead to serious data leakage incidents. It is a vulnerability caused by not properly inspecting acceptable input types and values.
(2) **Misimplemented Behavior:** A misimplemented behavior or unexpected behavior is prescribed to be an unpredictable action that occurs under specific circumstances or statuses of a program. Also, a misimplemented behavior can be a result of the invalid or inappropriate implementation of functionalities from the perspective of security. Memory corruption bugs such as use-after-free can cause misimplemented behaviors, which are beyond the scope of this paper. We focus on detecting misimplemented behaviors caused by inappropriate implementations. To take an example, we consider a case where a program encrypts wrong data (other than data that must be encrypted) when it performs cryptographic operations as a misimplemented behavior.

System Overview. We design SMINER to automatically generate test cases for inspecting systems by using testing code snippets implemented by developers for testing a system's functionalities. The core idea of SMINER is to employ basic unit test cases to generate complicated and fuzzy test cases. In general, unit test cases do not contain error-making code but code that must work properly in an expected way.

With such fuzzy test cases automatically generated by SMINER, we aim to discover undefined and misimplemented behaviors in a target system. Fig. 1 illustrates the architecture of SMINER, which consists of three steps:

(1) **Pre-processing:** SMINER collects basic unit test cases for a system and the collected code is pre-processed—SMINER parses the data into classes, functions, variables, and comments.
(2) **Security Policy Test:** SMINER assembles photo-processed information to generate test cases with help of users. Users set security policies that should not be violated in a system by using assembled data.
(3) **Generating Test Cases:** SMINER generates test cases based on the system model. They can be a complex set of intertwined codes with various functions of a system, or a set of data to intensively test the functionality of the system.

Evaluation. In Section 5, we perform qualitative and quantitative evaluations to demonstrate what types of vulnerabilities SMINER can detect and whether SMINER can detect vulnerabilities against a real-world system or not.

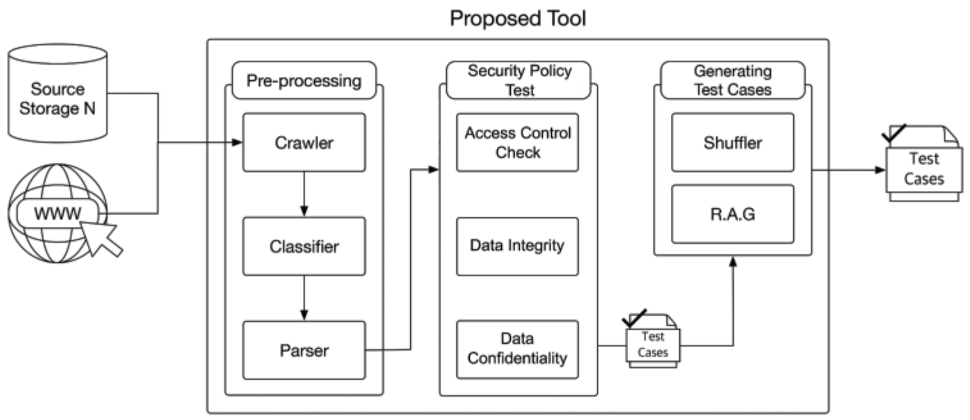

Figure 1: The test case generation process of SMINER

4 SMINER

In this section, we present the design of SMINER for discovering vulnerabilities based on unrestricted and undefined behaviors existing in software systems. For discovering such vulnerabilities, SMINER automatically generates complicated and fuzzy test cases based on basic unit test cases of a system that are implemented by developers.

4.1 Pre-Processing Unit Test Cases

SMINER first collects unit test cases of a target system from its official repository. Typically, most software systems include such basic code snippets for testing each functionality. After collecting unit test cases, SMINER also collects information on where the code came from, such as the Uniform Resource Locator (URL) where it was located, and paths stored inside the source storage because such information can be used. For example, suppose that we brought the code template through https://github.com/ros/ros_tutorials/tree/noetic-devel/rospy_tutorials/001_talker_listener/talker.py. In this case, we can figure out from the URL that this code template targets the noetic versions of ROS and is a unit test for testing functions between a talker and a listener from the perspective of the talker.

Next, SMINER pre-processes collected code snippets. To be specific, it classifies code snippets into classes, functions, comments, and variables. These classified data will be assembled and manipulated to generate test cases for discovering unrestricted and misimplemented behaviors of a target system. SMINER aims to identify such behavior of a target system by modifying intuitively recognizable features of unit test cases, rather than analyzing logical flows or semantics of each unit test. As an example, when a function is called with specific variables in a unit test, SMINER generates test cases calling the function, randomly modifying values in the variables. Comments in each unit test case are also provided to users to display an intended functionality of a target system. Based on such information, users can find the intended functionalities of a target system without manually analyzing it or unit test cases. It is worth noting that SMINER cannot provide enough information from poorly developed unit test cases that do not have proper comments for explaining which functionalities are being tested by them. In such cases, SMINER is only able to provide information such as names of functions or unit tests to users, and thus, could not be very effective.

4.2 Security Policy Test

Indeed, any security policies are strongly linked to system functions. Functional testing involves executing many security mechanisms. Security vulnerabilities can potentially affect the availability of applications. However, test functions do not include test security aspects. In most cases, the distribution of security policies is not automated, and the accuracy of implementation must be verified or tested. The security policy test is a practical way to ensure that it is correctly implemented with a certain level of confidence in an information or networking system. Therefore, SMINER shows a security policy that may occur from the security perspective of the target system using pre-processed data. Users can understand and check whether the target system is properly implemented through the security policy.

The categories for applying security policies in a system are very diverse. Among them, data security is the practice of protecting digital information from unauthorized access, damage, or theft throughout its entire lifecycle. A concept that encompasses all aspects of information security, from the physical security of hardware and storage devices to management and access control, to the logical security of software applications. In this paper, we focus on data security among the characteristics of the system and aim to proactively prevent security issues that may arise from unrestricted and misimplemented behavior. The data security policy determines who has access and what types of actions are allowed for each user of an object. The security policy consists of three types.

Access Control Check. Access control checks identify detailed security or authorization functions such as role-based access to functions, user access rights, and functional partitioning. Functions such as manager authority and authority elevation operating in the system should also be checked whether authorization has been granted through control.

Data Integrity. Data should be protected from unauthorized alteration or alteration. Data integrity protects against the security risk of manipulation in which someone intercepts and changes unauthorized information. In addition to protecting data stored within the network, additional security may be required to ensure data integrity when data enters the system from untrusted sources. If the data input to the system comes from a public network, it should be generally encrypted to protect the data from being sniffed and interpreted, and to ensure that the data has not been changed.

Data Confidentiality. Confidentiality means disclosing information only to authorized users. Complete reporting of the content of transmitted data prevents unauthorized persons from accessing the actual content of the information. Confidentiality is guaranteed and the disclosure of unsolicited information can be prevented. It can mainly protect the behavior related to data interception, and it is necessary to check whether encryption, the most popular method of use, is used.

SMINER uses pre-processed information and security policies set by users to show whether a system violates the security policies or not. As an example, if a user wants to check the data integrity, he/she can check the security policy by implementing a simple function that checks whether data has been modified. Therefore, when data is transmitted in a test case generated by SMINER and the value of the received data is different, it means that the data integrity is broken. The function is inserted when SMINER generates test cases to report results to users. This process requires manual implementations, but we believe that the manual effort is not huge because it is very straightforward to implement such checkers. We leave this limitation as future work.

4.3 Generating Test Cases

Basically, SMINER generates a test case by using the pre-processed information. However, in order to maximize the chances of discovering unrestricted and misimplemented behaviors, SMINER generates test cases for testing various functionalities at a time rather than providing a set of test cases that can only be used to investigate a particular functionality.

To be specific, SMINER uses the following two policies for generating test cases. The first policy is to manipulate a single unit test case with randomly generated data types and values. The second policy is to randomly combine multiple unit test cases implemented by developers into a test case, that SMINER generates, to evaluate a target system under unexpected execution flows. SMINER randomly decides the number of unit test cases for generating a test case. Also, SMINER allows users to decide specific unit test cases that are going to be combined in a test case to test security policies set by users.

5 Evaluation

In this section, we implement SMINER and evaluate it against a real-world software system, Robot Operating System (ROS). We perform quantitative and qualitative evaluations on SMINER. To be specific, there are two research questions that we are going to address as follows.

- **RQ1.What types of vulnerabilities can be discovered by SMINER?**
- **RQ2.Can SMINER discover real-world vulnerabilities by using test cases?**

5.1 Implementation

We implemented a proof-of-concept of SMINER. SMINER generates test cases against a target system by using unit test cases implemented by the developers of the system. Currently, SMINER is only able to analyze unit test cases implemented in Python programming language, but SMINER is not limited to generating test cases for a system implemented in other programming languages such as C and C++. We leave support for analyzing unit test cases implemented in other programming languages as future work.

5.2 Experimental Setup

In this section, we describe the experimental setup to provide detailed experiment information on our SMINER.

Setup: Our evaluations were performed on a machine running Ubuntu Linux 14.04 64-bit with a 3.40 GHz Intel Skylake processor and 16 GB of RAM.

Target System: In this work, we use SMINER for discovering vulnerabilities against various versions of the Robot Operating System (ROS). ROS is an open-source meta-operating system (a collection of software frameworks) for robotics. Currently, ROS is employed in a lot of academic projects and commercial products [32].

ROS implements hardware abstraction, low-level device control, and frequently used functions provided by general operating systems, and provides inter-process message delivery and package management functions. It also provides tools and libraries to work with multiple systems and has a considerable level of complexity, including distributed computation, multi-threading, event-driven programming, and other concepts at the core of the system. Therefore, a high level of expertise is required to understand the system. ROS proceeds through message communication between nodes.

Nodes must register their information in the master upon startup. The master has the information on each node and plays an important role in managing the node. However, the master may also fail, and if an unrestricted or misimplemented behavior occurs on the master, it may adversely affect the whole system. Remote code execution vulnerabilities including CVE-2016-10681 were also found in ROS, a representative meta OS for robots. In addition, many MiR robots were exposed to vulnerabilities due to design flaws in the calculation graph package provided by ROS. ROS has a number of well-designed unit test cases for checking basic functionalities, services, and interactions between nodes. By using SMINER, we generated our test cases to discover unrestricted or misimplemented behaviors in ROS.

5.3 Types of Vulnerabilities That Can be Discovered by SMINER

We first evaluate what types of vulnerabilities can be discovered by SMINER. This assessment provides a specific scope that SMINER can assist in detecting vulnerabilities. Table 1 illustrates that detectable vulnerability types through executing test cases generated by SMINER.

Table 1: Vulnerability types that can be detected by SMINER. (✓indicates a detectable vulnerability type)

Vulnerability types		Detected with SMINER
Code injection vulnerability	Cross-site scripting (XSS)	✓
	Carriage return line feed (CSLF) injection	✓
	SQL injection	✓
	Expression language injection	✓
	Insecure randomness injection	✓
Data validation vulnerability	Insufficient session-ID length	✓
	Authentication validation	✓
	Improper data validation	✓
Exception handling	Missing error handling	✓
	Null dereference	✓
	Unchecked error condition	✓
Memory corruption	Buffer overflow	×
	Use-after-free	
	Double free	
Session management	Session hijacking	×
	Cross site request forgery (CSRF)	
Other	Access control	×
	Configuration error	

The vulnerabilities in the code injection vulnerability category allow malicious code to be inserted into the system and executed by writing a payload. These vulnerabilities exist when a program does not restrict or check input values properly. SMINER can provide a test code using various inputs with predefined values or symbols used for triggering such vulnerabilities.

The data validation vulnerability type arises when functions for validating data types are mis-implemented. SMINER generates test cases where it intentionally triggers improper uses of data types against a target system. Furthermore, SMINER can generate test cases that execute multiple arbitrary functions in random order. Such test cases also can be used to discover the data validation vulnerability type because they may use or manipulate data structures with random functions that are not implemented for the data structures.

Vulnerabilities in the exception-handling type occurs when there are no appropriate exception-handling functions implemented in a system. An exception can crash or lead to another vulnerability if an exception handler is not implemented in a system. SMINER provides test cases that use a variety of data, as well as a code that can test the interactions of different features of the system. These test codes can cause exceptions in the system, which can help detect vulnerabilities associated with exception handling.

In summary, Table 1 shows that SMINER is able to detect code injection, data validation, and exception handling types of vulnerabilities but SMINER cannot be effectively used to discover other types of vulnerabilities. Note that, while memory corruption vulnerabilities are beyond the scope of this paper, it is possible to several types of memory corruption bugs such as out-of-bound accesses through test cases generated by SMINER.

5.4 Discovering Vulnerabilities in ROS

We demonstrate the effectiveness of SMINER by presenting evaluation results against all released versions of ROS. To detect vulnerabilities based on unrestricted and misimplemented behaviors in ROS, we used SMINER to generate fuzzy test cases and execute them on ROS, monitoring it.

5.4.1 Generating Test Cases

Listing 1: Test case with multiple functions in a random order

```
[code]
def listener():
    rospy.init_nodes('listener', anonymous = True)
    rospy.Subscriber('chatter', String, callback)
    rospy.spin()

def talker():
    pub = rospy.Publisher('chatter', String, queue_size = 10)
    rospy.init_node('talker', anonymous = Ture)
if __name__ == "__main__":
    listener()
    listener()
    talker()
```

Listing 2: Test case with randomly generated arguments

```
def talker(data):
    pub = rospy.Publisher('chatter', String, queue_size = 10)
    rospy.init_node('talker', anonymous = Ture)
      pub.publish(data) # Send Message
  rate.sleep()
# 'RAG_List' have randomly generated argument candidates by SMiner
if __name__ == "__main__":
    for data in RAG_List:
        talker(data)
```

First off, SMINER collected 113 unit-test cases from the official source code repository and the web community of ROS. SMINER then generates test cases for discovering vulnerabilities. Lis. 1 and Lis. 2 show code snippets of test cases generated by SMINER. As in each listing, SMINER provides a brief description of modules that are being evaluated in each test case.

5.4.2 Execution Time of Individual Test Cases and Throughput

To evaluate the execution times of individual test cases, we chose three versions out of 13 different versions of ROS. The versions of the ROS used are Noetic, Melody, Lunar, Kinetic, Jade, Indigo, Hydro, Groovy, Fuerte, Electric, Diamondback, C, and Box. Then, we executed test cases generated by SMINER as shown in the previous section. Fig. 2 shows the distribution of execution times required to run each test case. On the three versions of ROS, each test case generated by SMINER runs for roughly 5 s on average. On the other hand, Fig. 3 demonstrates the throughput (execs/hour) of test cases to find vulnerabilities in the three versions of ROS. The throughput was measured every hour for six hours.

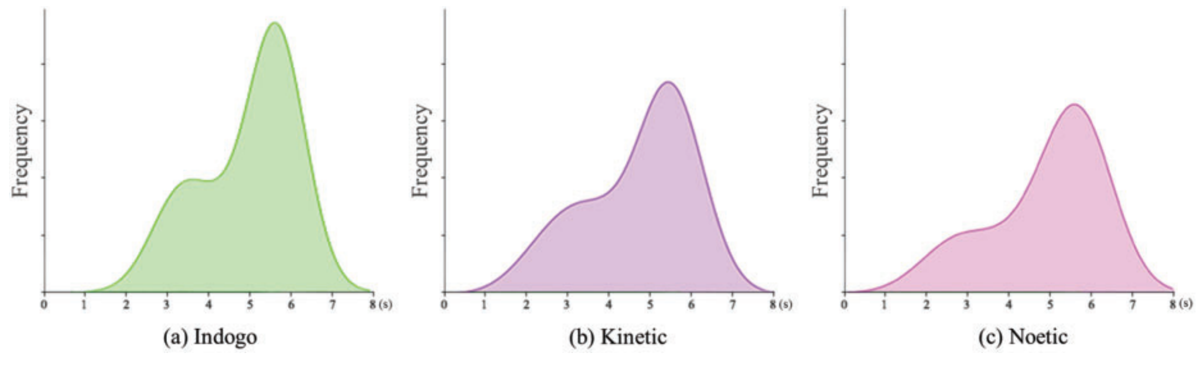

Figure 2: Test case execution time per test case in 3 different versions of ROS

From the numerous test codes SMINER generated, we identified one unrestricted behavior and one misimplemented behavior, by which we could reveal two vulnerabilities in the (extensible Markup Language (XML) Remote Procedure Call) XMLRPC module and (Transmission Control Protocol (TCP) based Robot Operating System protocol) TCPROS module of ROS. Also, we confirmed that the vulnerabilities are exploitable from the first released version of ROS to the latest version.

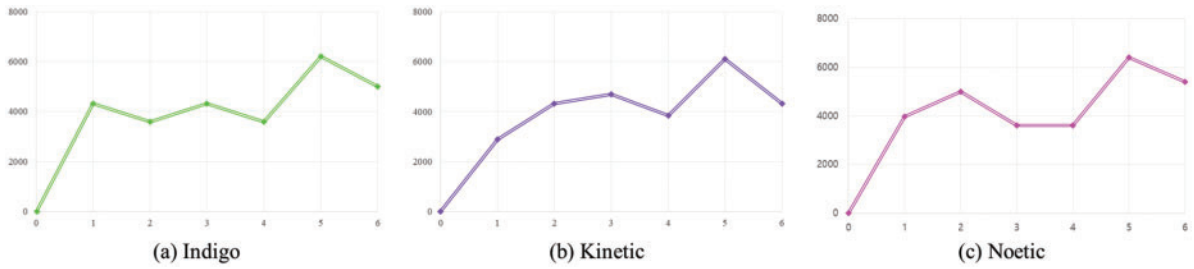

Figure 3: Test case execution throughput (execs/hour) measured every hour for six hours

Listing 3: Process of registering new information of nodes in the master

```
node_ref = self.nodes.get (caller_id, None)
bumped_api = None
if node_ref is not None:
    if node_ref.api == caller_api:
        return node_ref, False
    else:
        bumped_api = node_ref.api
        self.thread_pool.queue_task (bumped_api,
    shutdown_node_task, (bumped_api, caller_id, " new node
    registered with same name"))
node_ref = NodeRef (caller_id, caller_api)
self.nodes [caller_id] = node_ref
return (node_ref, bumped_api ! = None)
```

5.4.3 Unrestricted Behavior in XMLRPC

We detected an unrestricted behavior by executing a test case that registered the same name as a listener. In ROS, talker and listener nodes must be registered in the master node to communicate with each other. In Lis. 1, a node with the name "listener" is initially registered in the master, and then, another node that has the same name "listener" sends a registration request to the master. Via the test case, we found that the previously registered listener node is being unregistered.

We manually analyzed the root cause of this vulnerability and discovered a problem in the XMLRPC communication module that is used in the master node for initiating connections between nodes (e.g., talker and listener). When the master node receives a request from a node that wants to be a talker and a listener, the master does not perform any authentication process for registering the node as a listener or talker.

Lis. 3 shows a code snippet that registers a new node in the XMLRPC module. When a registration request comes from a node, only the api value corresponding to the node address is checked. If the previous api value and the requested api value are different, the XMLRPC module sets the requested api to bumped_api without any further checks. Therefore, the previously operated service will be canceled and a new connection will be started, if a new node that has the same name as the previously registered node requested to be a listener. Consequently, in ROS, a malicious node can intercept data traffic between any two nodes by being a listener.

5.4.4 Misimplemented Behavior in TCPROS

Listing 4: Checking md5sum and string message type code used in ROS

```
def _get_message_type(info):
    message_type = _message_types.get(info.md5sum)
    if message_type is None:
        try:
            message_type = genpy.dynamic.generate_dynamic(info.datatype, info.msg_def)[ info.datatype]
            if (message_type._md5sum ! = info. md5sum):
                print('WARNING: For type [%s] stored md5sum [%s] does not match message definition [%s].\n Try: "rosrun rosbag fix_msg_defs.py old_bag new_bag.""%(info.datatype, info.md5sum, message_type._md5sum),file = sys.tderr)
        _message_types[info.md5sum] = message_type
    return message_type
class String(genpy.Message):
    _md5sum = "992ce8a1687cec8c8bd883ec73ca41d1"
    _type = "std_msgs/String"
    _has_header = False
    _full_text = """"string data""""
    __slots__ = ['data']
    _slot_types = ['string']
```

Lis. 2 is a test case that sends randomly generated arguments to the "talker" node. By executing this test case and monitoring its execution results, we found a misimplemented behavior in ROS: a hash function for checking the integrity of messages was not properly used.

In ROS, the TCPROS module is used for transmitting messages and services based on TCP/IP. In the message header of all messages transferred by the TCPROS module, there is a md5sum field for checking the integrity of a message. However, in the TCPROS module, a function for verifying the integrity of messages was not carefully implemented.

To be specific, Lis. 4 shows the function that uses the md5sum hash in the TCPROS module. In the function, only the message type is hashed to create md5sum instead of hashing the message body containing the actual data of a message. Therefore, even if messages are different, the md5sum values of the messages are the same. As a result, in ROS, it is possible to forge each message by bypassing the verification process.

6 Limitations

SMINER is a novel approach that automatically generates test cases for discovering unrestricted and misimplemented behaviors based on basic unit test cases provided by developers, but we find several limitations and leave them as future work.

First, SMINER generates test cases by relying on only publicly available unit test cases. If there are no unit test cases implemented for testing functionality, SMINER is not able to generate a test case regarding the function. Also, SMINER collects unit test cases from a variety of sources, such as the web and code storage, and thus, unit test cases for testing the same function can be collected. SMINER cannot identify and filter them because they have different names of identifiers and different forms of code configurations. This limitation can make the testing process inefficient by producing redundant test cases. In addition, since SMINER generates only test cases, we should develop an approach for monitoring unrestricted and misimplemented behaviors to further automate the whole vulnerability detection process.

7 Related Work

In this section, we describe existing research related to vulnerability detection in various areas of study.

Symbolic Execution. Symbolic execution [33] is a widely used analysis technique to find symbols executing all of the path conditions in the program. Symbolic execution can be used to find a crash in programs or get the effective seeds used for fuzzing to discover a memory-related vulnerability. However, because symbolic execution is executed repeatedly to find the valid value of the path condition, it requires huge resources.

Machine Learning for Discovering Vulnerabilities. Much research using machine learning is proposed to hunt potential vulnerabilities in source code. Russell et al. [34] propose a fast and scalable vulnerability detection tool based on deep feature representation learning. It used C, C ++ source code from Static Analysis Tool Exposition (SATE) IV, GitHub, and Debian as a learning dataset. It also used Convolutional Neural Network (CNN) for sentence sentiment classification and Recurrent Neural Network (RNN) for feature-level source vulnerability classification. They demonstrated that the approach is a promising way to detect software vulnerabilities.

Automated Test Case Generation. In general, a test case consists of a series of actions that execute on a system to check whether the system meets software requirements as designed and that the system does not have undefined behavior, by which we can prevent errors or glitches in the system. There have been many research efforts on the creation of automated and cost-effective test cases to find various software errors in a specific target [14,29–31]. Gregory [14] extracts defects from the Google Gson (Gson) project to collect the system's defects and corrected versions of the code, as well as developer-written test cases exposing each defect. For each defect, we use the Evo Suite framework to generate a test for each affected class under test and evaluate the effectiveness of the resulting suite. A study by Mohammadi et al. [29] presents a test case-based approach to automatically detect XSS vulnerabilities due to the use of incorrect encoding functions. Syntax-based, Semantics-based, and Vector Representations (SySerVR) [30] focuses on obtaining program representations that can accommodate syntactic and semantic information using deep learning to detect program vulnerabilities. Shahriar et al. [31] identified seven criteria for analyzing security test tasks, comparing and contrasting the superior security test approaches available. Unlike the previous studies, our system has a specific goal of identifying unrestricted and misimplemented behaviors of a target system by checking access control errors, data integrity, and confidentiality based on test cases generated by using existing unit test cases.

8 Conclusion

In this paper, we proposed SMINER to detect vulnerabilities based on unrestricted and misimplemented behaviors of software systems. SMINER utilizes unit test cases implemented by developers of a target system and generates complicated and fuzzy test cases by using them. We discovered two real-world vulnerabilities while evaluating SMINER against Robot Operating System (ROS) and reported them.

Funding Statement: This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT), and Future Planning under Grant NRF-2020R1A2C2014336 and Grant NRF-2021R1A4A1029650.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Böhme, V. -T. Pham, M. -D. Nguyen and A. Roychoudhury, "Directed greybox fuzzing," in *24th ACM Conf. on Computer and Communications Security (CCS)*, Dallas, TX, pp. 2329–2344, 2017.
[2] Heartbleed, The Heartbleed Bug, https://heartbleed.com/, 2021.
[3] W. You, P. Zong, K. Chen, X. Wang, X. Liao *et al.,* "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *24th ACM Conf. on Computer and Communications Security (CCS)*, NY, United States, pp. 2139–2154, 2017.
[4] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Int. Symp. on Empirical Software Engineering and Measurement*, Alberta, Canada, pp. 97–106, 2011.
[5] O. Pieczul and S. N. Foley, "Runtime detection of zero-day vulnerability exploits in contemporary software systems," in *Data and Applications Security and Privacy XXX*, Trento, Italy, pp. 347–363, 2016.
[6] NIST, Shellshock vulnerability, https://nvd.nist.gov/vuln/detail/CVE-2014-6271, 2021.
[7] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik and T. Holz, "REDQUEEN: Fuzzing with input-to-state correspondence," in *2019 Network and Distributed System Security Symp.*, San Diego, CA, 2019.
[8] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie *et al.,* "Hawkeye: Towards a desired directed grey-box fuzzer," in *25th ACM Conf. on Computer and Communications Security (CCS)*, Toronto, Canada, pp. 2095–2108, 2018.
[9] S. Chen, L. Fan, G. Meng, T. Su, M. Xue *et al.,* "An empirical assessment of security risks of global android banking apps," in *2020 IEEE/ACM 42nd Int. Conf. on Software Engineering (ICSE)*, Seoul, South Korea, pp. 1310–1322, 2020.
[10] R. Vanciu and M. A-Antoun, "Finding architectural flaws using constraints," in *28th IEEE/ACM Int. Conf. on Automated Software Engineering*, Silicon Valley, CA, USA, pp. 334–344, 2013.
[11] NASA, 2020 CWE top 25 most dangerous software weaknesses, https://www.nasa.gov/viper/lunar-operations, 2021.
[12] S. A. Baddar, A. Merlo and M. Migliardi, "Anomaly detection in computer networks: A state-of-the-art review," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, vol. 5, pp. 29–64, 2014.
[13] CWE. 2020 CWE top 25 most dangerous software weaknesses, https://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html, 2021.

[14] G. Gregory, “Detecting real faults in the gson library through search-based unit test generation,” in *Int. Symp. on Search Based Software Engineering*, Montpellier, France, pp. 385–391, 2018.

[15] H. Han, D. Oh and S. K. Cha, “CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines,” in *Annual Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, 2019.

[16] P. Vivekanandan, “A Type-based formal specification for cryptographic protocols,” *Journal of Internet Services and Information Security*, vol. 8, no. 4, pp. 16–33, 2018.

[17] R. B. Basnet, R. Shash, C. Johnson, L. Walgren and T. Doleck. “Towards detecting and classifying network intrusion traffic using deep learning frameworks,” *Journal of Internet Services and Information Security*, vol. 9, no. 4, pp. 1–17, 2019.

[18] S. K. Cha, M. Woo and D. Brumley, “Program-adaptive mutational fuzzing,” in *36th IEEE Symp. on Security and Privacy*, San Jose, CA, USA, pp. 725–741, 2015.

[19] T. Petsios, J. Zhao, A. D. Keromytis and S. Jana, “Slowfuzz: Automated domainindependent detection of algorithmic complexity vulnerabilities,” in *ACM Conf. on Computer and Communications Security (CCS)*, Dallas, TX, pp. 2155–2168, 2017.

[20] D. She, K. Pei, D. Epstein, J. Yang, B. Ray *et al.,* “Neuzz: Efficient fuzzing with neural program smoothing,” in *40th IEEE Symp. on Security and Privacy*, San Francisco, CA, vol. 1, pp. 803–817, 2019.

[21] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren *et al.,* “Optimizing seed selection for fuzzing,” in *23rd USENIX Security Symp. (Security)*, San Diego, CA, pp. 861–875, 2014.

[22] M. Cho, S. Kim and T. Kwon, “Intriguer: Field-level constraint solving for hybrid fuzzing,” in *26th ACM Conf. on Computer and Communications Security (CCS)*, London, UK, pp. 515–530, 2019.

[23] I. Haller, A. Slowinska, M. Neugschwandtner and H. Bos. “Dowsing for overflows: A guided fuzzer to find buffer boundary violations,” in *22nd USENIX Security Symp. (Security)*, Washington, D.C., United States, pp. 49–64, 2013.

[24] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin *et al.*, “IoTFuzzer: Discovering memory corruptions in IoT through appbased fuzzing,” in *2018 Annual Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, 2018.

[25] P. Chen, J. Liu and H. Chen, “Matryoshka: Fuzzing deeply nested branches,” in *26th ACM Conf. on Computer and Communications Security (CCS)*, London, UK, pp. 499–513, 2019.

[26] J. He, M. Balunovic, N. Ambroladze, P. Tsankov and M. Vechev, “Learning to fuzz ´ from symbolic execution with application to smart contracts,” in *26th ACM Conf. on Computer and Communications Security (CCS)*, London, UK, pp. 531–548, 2019.

[27] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu *et al.,* “Tainting-assisted and context-migrated symbolic execution of android framework for vulnerability discovery and exploit generation,” *IEEE Transactions on Mobile Computing*, vol. 19, pp. 2946–2964, 2019.

[28] H. Jingxuan, B. Mislav, A. Nodar, T. Petar and V. Martin, “Learning to fuzz from symbolic execution with application to smart contract,” in *2019 ACM SIGSAC Conf. on Computer and Communications Security*, New York, United States, pp. 531–548, 2019.

[29] M. Mohammadi, B. Chu and H. R. Lipford, “Detecting cross-site scripting vulnerabilities through automated unit testing,” in *2017 IEEE Int. Conf. on Software Quality, Reliability and Security (QRS)*, Praha, Czech Republic, pp. 364–373, 2017.

[30] L. Zhen, “Sysevr: A framework for using deep learning to detect software vulnerabilities,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 2244–2258, 2021.

[31] H. Shahriar and M. Zulkernine, “Automatic testing of program security vulnerabilities,” in *33rd Annual IEEE Int. Computer Software and Applications Conf.*, Seattle, WA, USA, vol. 2, pp. 550–555, 2019.

[32] J. M. O’Kane, *A Gentle Introduction to ROS*. SC, USA: CreateSpace Independent Publishing Platform, 2014. [Online]. Available: https://jokane.net/agitr/agitr-letter.pdf

[33] S. Poeplau and A. Francillon, “Symbolic execution with symcc: Don’t interpret, compile!,” in *29th USENIX Security Symp.*, Berkeley, CA, United States, pp. 181–198, 2020.

[34] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer *et al.,* “Automated vulnerability detection in source code using deep repr1esentation learning,” in *17th IEEE Int. Conf. on Machine Learning and Applications (ICMLA)*, Cancun, Mexico, pp. 757–762, 2018.