International Journal of Information and Communication Sciences
2023; 8(1): 1-11
http://www.sciencepublishinggroup.com/j/ijics
doi: 10.11648/j.ijics.20230801.11
ISSN: 2575-1700 (Print); ISSN: 2575-1719 (Online)

Automatic Vulnerability Detection in Tizen Applications with Dynamic Symbolic Execution

Sobhan Safdarian, Mohammad Hossein Asghari, Maryam Mouzarani

Department of Electrical and Computer Engineering, Isfahan University of Technology, Isfahan, Iran

Email address:

sobhan.safdarian@ec.iut.ac.ir (Sobhan Safdarian), mhossein.asghari@ec.iut.ac.ir (Mohammad Hossein Asghari), mouzarani@iut.ac.ir (Maryam Mouzarani)

To cite this article:

Sobhan Safdarian, Mohammad Hossein Asghari, Maryam Mouzarani. Automatic Vulnerability Detection in Tizen Applications with Dynamic Symbolic Execution. *International Journal of Information and Communication Sciences*. Vol. 8, No. 1, 2023, pp. 1-11.
doi: 10.11648/j.ijics.20230801.11

Received: November 19, 2022; **Accepted**: January 26, 2023; **Published**: February 9, 2023

Abstract: Security of Internet-of-Things (IoT) systems is important due to their widespread usage in everyday life. Much research has been performed on analyzing the security of IoT communication protocols and operating systems. However, few studies have focused on analyzing the security of IoT applications and automatic detection of vulnerabilities in them. In these studies, the code of IoT applications and operating systems are analyzed statically to detect vulnerabilities. To the best of our knowledge, there is no dynamic analysis solution suggested for vulnerability detection in such applications, although this method is more accurate than static analysis. In fact, IoT applications are executed in special-purpose hardware, which makes their dynamic analysis more difficult than ordinary applications. In this paper, we propose a technical solution that combines static and dynamic analysis methods to automatically detect vulnerability in applications of Tizen IoT operating system. We consider Native and Web Tizen applications and present an automatic vulnerability detection method for each type of application. Our focus is on detecting buffer overflow and XSS vulnerability classes in Native and Web applications, respectively. We have evaluated the effectiveness of our method using a group of native and web test programs. The results of our experiments show that our solution is able to detect the vulnerability in these programs effectively.

Keywords: Tizen, Dynamic Symbolic Execution, Native, Web, Vulnerability, IoT, C++, JavaScript

1. Introduction

Internet of Things (IoT) has affected many aspects of modern human life. Devices we use for our daily activities such as shopping, communication, sport, and helping disabled people are all examples of the ever-increasing usage of IoT in our daily human life. Therefore, the security of IoT devices becomes important to protect human activities, privacy, and data against malicious intruders. A considerable part of an IoT device is its software and thus in order to make it secure, we must take enough care of the security of its running applications.

IoT devices use various architectures and operating systems, such as Windows IoT core [1], Amazon FreeRTOS [2], Tizen [3], Contiki [4], etc. This diversity complicates the security analysis of IoT systems. Also, some IoT applications are used for real-time purposes and so they do not apply common security measures to protect against well-known attacks to increase their performance. In addition, IoT devices are implemented using cheaper and lighter hardware compared to ordinary systems. Thus, they are not able to incorporate usual memory protection mechanisms which have a huge overhead [5].

Application analysis for vulnerability detection is conducted by two approaches: dynamic and static. The static approach reviews the application code to find vulnerabilities. This approach is sound because it covers all the execution paths of the program in its analysis. However, the results are conservative and contain many false positive alarms [6]. On the other hand, the dynamic approach executes the application and analyzes its behavior. The tools with this approach report no false positive alarms because they run the program with actual data and analyze the exact behavior of applications. The drawback of dynamic analysis is that it cannot guarantee to cover all application paths and states, and thus there are false negative alarms in its reports. Dynamic

symbolic execution [7] is a hybrid analysis method that combines static and dynamic analysis to profit high coverage and accuracy of each method, respectively. In this paper, we use this method for analyzing and detecting vulnerabilities in Tizen IoT applications.

In recent years, the security of communication protocols in IoT systems has been studied by some researchers, such as [8], [9], and [10]. There are also limited studies regarding the static analysis of IoT OS and applications [11], [12] and [5]. These works emphasize different vulnerability types in current IoT operating systems and applications alongside the lack of proper protection measures against such vulnerabilities. Although, to the best of our knowledge, there is yet no proposed solution for dynamic analysis of these applications. The dynamic analysis of IoT applications is more challenging than other applications because they are executed on special-purpose hardware that might not contain enough facilities to debug and analyze the application runtime behavior. This paper presents a method for dynamic analysis of applications in Tizen IoT operating system. We use the dynamic symbolic execution method in our analysis to achieve a proper path coverage of the program. We present the details of how to overcome the challenges of performing dynamic symbolic analysis of applications in Tizen devices. We demonstrate the effectiveness of our proposed method by employing the implemented tool for analyzing two sample Tizen applications. The implemented method is publicly available in our GitHub repository [26] with sample test scenarios.

We present the detailed method of dynamically analyzing two types of Tizen applications, namely native and web. Analysis of native applications is conducted by connecting the analyzer to the *gdbserver* in Tizen OS and statically analyzing the binary code of the target application to locate probably vulnerable function calls, such as *strcpy()*, and determine input arguments of those functions. Then, *Symbion* symbolic execution engine [13] is employed to calculate the path constraints on input data for executing a path that reaches located probably vulnerable function calls. After solving the constraints using a constraint solver, we generate consistent input values that are large enough to execute the intended paths and cause buffer overflow in considered function calls.

Tizen web applications are written in HTML, JavaScript and CSS that may execute some API functions in this OS. Since the analysis of such applications and detection of XSS vulnerability do not directly depend on the result of execution of Tizen API functions, and we could not employ a dynamic symbolic execution framework in that operating system for our analysis of web applications, we analyze the web application runtime behavior by executing it in an environment outside of Tizen. We use *ExpoSE* [14] tool that is developed to symbolically execute NodeJS applications for our analysis. Our proposed method extracts the JavaScript code of a web application and prepares it to be analyzed by ExpoSE. In fact, it creates a new version of the application code that can be processed by ExpoSE to detect vulnerabilities in it. To do so, our proposed method first analyzes the HTML and JavaScript codes of the test application statically to determine its HTML input/output entries and probably vulnerable JavaScript statements as *hotspots*. We add a conditional statement before a *hotspot* that checks if the data entering the *hotspot* might contain attack strings. In this way, we lead ExpoSE to calculate vulnerability constraints for input data in addition to path constraints in the paths containing *hotspots*. Afterward, the prepared code is given to ExpoSE in order to analyze the possibility of executing the application in an execution path containing *hotspots* with input values that lead to an attack.

The proposed method is evaluated using a group of native and web test programs that are executable in Tizen operating system. The results of our experiments demonstrate the effectiveness of our solution for detecting buffer overflow and XSS vulnerability in these programs.

This paper has the following structure: Section 2 reviews the previous related works. Tizen operating system and its application types are introduced in section 3. Section 4 describes the proposed vulnerability detection method for native applications of Tizen. Our proposed method for detecting vulnerability in Tizen web applications is presented in section 5. Section 6 evaluates the implemented solution and finally, section 7 presents the conclusion and possible directions for future works.

2. Related Works

Most studies in the field of IoT application security analysis have focused on operating system security and static application code analysis. For example, in [11], a tool named UnsafeFunsDetector is used to statically analyze the source code of some IoT operating systems such as Contiki, TinyOS, and openWSN to find a number of unsafe functions in their code. As another example, Firmalice is a static analyzer that searches through the binary statements of IoT firmware to find any hardcoded credentials that might reveal a backdoor in the firmware [15].

To the best of our knowledge, there has been no solution for dynamic analysis of Tizen applications until now. However, there are some general frameworks and tools, such as Avatar [16] and Qilling [17], to dynamically analyze native applications of IoT devices. Avatar is a simulator that physically connects to IoT devices and analyzes their system events [16]. It hooks the intended events from the physical system to transfer their data to the simulator and analyze them. The limitation of this simulator is that it requires a JTAG port in the physical IoT system that is not supported by all IoT devices.

Qilling is another example that simulates an entire system from the OS image and its file system [17]. This simulator reviews the system functions of the device file system to hook the intended system functions and make their behavioral analysis possible for the analyzer. This simulator requires the OS image of the intended device, which is not necessarily available for all devices. Tizen's producers have

released its image but in our experience of running this image in Qilling, it got stuck in the initial boot step and could not proceed further. On the other hand, Qilling runs the application image on the analyzer device, so it doesn't analyze the real application behavior as it runs in an IoT device. Notably, none of these simulators have a framework for dynamic symbolic execution and such a framework must be installed in these simulators.

There are also some solutions for dynamic symbolic execution of web applications, such as SymJS [18] and JSSeek [19], which are not available to be used in our solution. SymJS tries to create test data that covers the highest percentage of execution paths in a web application using symbolic analysis. Also, this tool automatically extracts the application events to simulate them and execute the relevant codes responding to these events. Since this is a private tool, we could not use it in our solution. JSSeek [19] is another tool that performs symbolic analysis of JavaScript applications. This tool uses symbolic analysis to find different errors in JavaScript applications, such as undefined variable error.

Some solutions are presented for analyzing JavaScript environments, such as NodeJS, but none of them were ever used in the analysis of web Tizen OS applications. For example, ExpoSE [14] performs dynamic symbolic execution on NodeJS applications. This tool explores different paths of an application by generating appropriate input values. We cannot use this tool directly in the analysis of Tizen web applications because of different data structures and functions in JavaScript applications that are executed in browsers alongside the dependency of JavaScript codes on the HTML codes of web pages.

3. Tizen

Tizen is an open-source operating system used in a wide range of IoT devices, such as wearable devices, televisions, and smartphones [3]. Hence, it is known as "the OS of Everything" [12]. This OS is based on the Linux kernel, which makes it similar to other Linux versions in many aspects, such as process management, memory management, file system, and system calls. In this paper, we consider Tizen version 5.5.

Tizen OS supports three application types:

1. Native Applications: C and C++ applications.
2. Web Applications: JavaScript, HTML, and CSS-based applications that are executed in a web engine.
3. NET Applications: Xamarin and Visual Studio-based applications.

Also, Tizen OS allows programmers to write applications that pack web and native applications in a single application known as Hybrids.

Native applications are executed directly in the Linux kernel platform. When we compile a native application by default using the Tizen IDE, it will have the NX and PIE mechanisms enabled, but the Canary mechanism is disabled. However, as shown in [20], it is possible to bypass the active security mechanisms of this OS. NX and PIE are two security mechanisms used in well-known operating systems to prevent memory corruption attacks [25].

The web Tizen applications are executed using the OS's web engine. XSS and HTML Injection are the vulnerabilities that might occur in these applications. Web Tizen applications are able to access Tizen system calls and APIs. For example, the *tizen* object is defined for JavaScript codes of these applications for gaining access to different system functions or sensor values. Therefore, exploiting the vulnerabilities in Tizen web applications might lead to disclosure of confidential information and unauthorized access to the device.

4. Detecting Buffer Overflow Vulnerability in Tizen Native Applications

Unsafe usage of functions, such as *strcpy*, *gets*, *scanf*, etc., in Tizen native applications can lead to buffer overflow. Different static and dynamic analysis methods have been introduced to detect this vulnerability in various applications. Dynamic symbolic execution is one of the popular vulnerability detection methods that generates proper input data for executing different application paths by calculating the constraints on input data for executing each path. This method suffers from the path explosion problem, which means the number of possible paths in an application grows exponentially and analysis of all execution paths becomes impossible in practice [21].

Therefore, in our proposed solution, we first perform static analysis to find execution paths in the program that contains possible vulnerable function calls, e.g., *strcpy*, and then we limit the scope of dynamic symbolic execution to those paths to avoid path explosion problem.

```
1. int main(int argc, char *argv[]) {
2. char buffer[40];
3. char input[100];
4. if (argc > 1) {
5. if (!strcmp(argv[1], "-s")) {
6. fgets(input, 100, stdin);
7. if (input[1] == 'G' && input[5] == 'M') {
8. strcpy(buffer, input);
9. }
10. }
11. } else {
12. printf("[-] Usage with flag (-s)\n");
13. }
14. return 0;
15. }
```

Figure 1. *Sample of a Native Tizen Application.*

As an example, consider Figure 1, which is a simple Tizen native application with buffer overflow vulnerability in line 8. This application receives two input values: one is the *main*

function argument and another is a console input value received via *fgets* function in line 6 that is saved in the *input* string. There is no buffer overflow in this stage because of the string input length control of line 6. Later in line 8, the *strcpy* function copies the *input* string into the *buffer* string if the value of the *input* string is equal to "XGXXXM*", in which X means an arbitrary character. Here a buffer overflow occurs because this function does not check the length of the strings.

To detect the buffer overflow vulnerability in this code, we first use the *angr* [22] framework for our static analysis and extract the application's control-flow graph to detect any calls of possible vulnerable functions. Thus, we find the execution path in which *strcpy* function is called. Then, we perform dynamic symbolic execution using *Symbion* plugin of the *angr* framework and calculate the path constraints on the input values that lead to the desired path. For this example, the main argument should be "-s" and the second and sixth characters of the console input value should be "G" and "M" respectively to reach the *strcpy* function call.

These constraints are delivered to Z3 constraint solver in *angr* to generate consistent input values accordingly. Then, we move on to the fuzzing step and increase the length of generated input data while considering the path constraint on it to generate new input data that executes the program's desired path and causes buffer overflow in the *strcpy* function. Then, we execute the application with these new data and report the vulnerability in case of a crash.

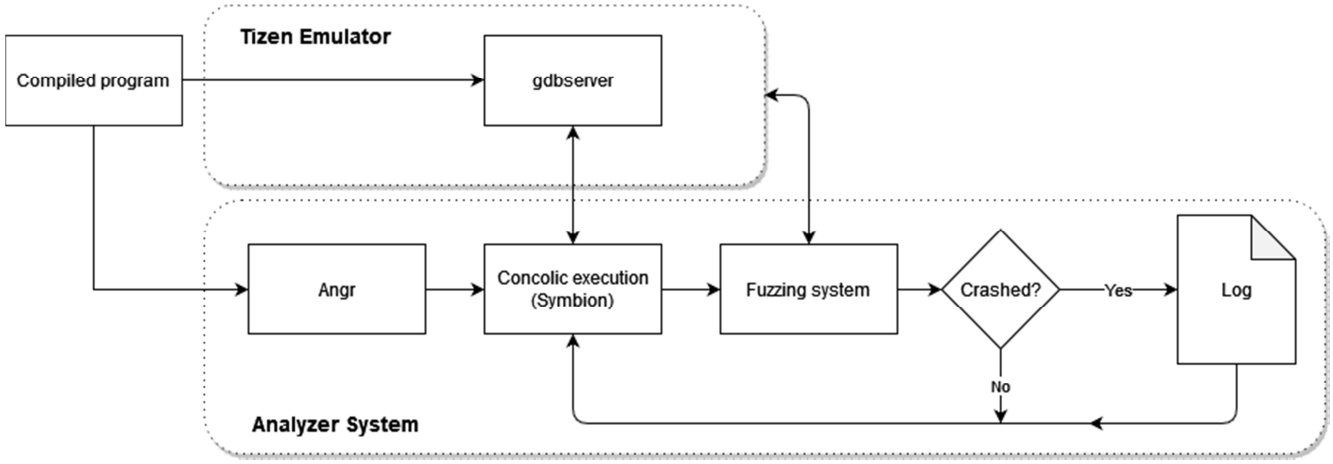

Figure 2. *Architecture of the Proposed Method for Analyzing Native Tizen Applications.*

Notably, the dynamic symbolic execution engine or the static analysis tool cannot be installed in the Tizen emulator, which is one of our main challenges in the static analysis and dynamic symbolic execution of the Tizen application. Therefore, we install and launch the test application in Tizen and analyze it by establishing a remote connection to the *gdbserver* of Tizen. Figure 2 illustrates the architecture of our solution for analyzing Tizen native applications. As shown in this figure, the compiled test application is installed and launched in Tizen to be analyzed dynamically. Also, the binary code of the application is given to the *angr* framework for static analysis. To perform dynamic symbolic execution, we use *Symbion* in our machine and connect remotely to the *gdbserver* in Tizen. All the commands used in our solution, including those for compiling and installing the application by sdb[1] and the Tizen configuration commands to debug applications using *gdbserver*, are presented in a script file in our Github repository [26].

Figure 3 illustrates the output of our implemented solution for analyzing the sample program. In this figure, SIGSEGV status code demonstrates that we could successfully generate long input data that is consistent with the path constraints to execute the program and cause buffer overflow in it.

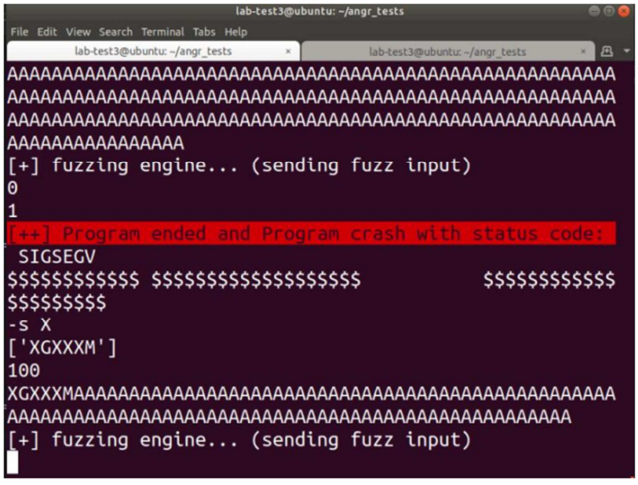

Figure 3. *Output of a Native Application's Analysis.*

5. Vulnerability Detection in Tizen Web Applications

Dynamic symbolic execution is more challenging for Tizen web applications as there is no framework or platform available in Tizen that enables us to symbolically execute web applications in that operating system. Meanwhile, since Tizen

1 The communication bridge between the developer and the Tizen system

web applications are written in HTML, JavaScript, and CSS languages and they do not depend heavily on Tizen web engine, we can execute them outside of Tizen without losing many functionalities. The only limitation is for calling Tizen APIs. With our proposed solution, it is possible to handle them by defining Tizen APIs in the target NodeJS code so that they are considered as input entries that return a symbolic variable.

Therefore, we extract the codes of Tizen web applications and execute them symbolically in our ordinary machine in a framework named *ExpoSE* [14]. At the time of writing this article, ExpoSE is the best available tool for us which has a proper performance but only analyzes NodeJS codes. This makes some difficulties in analyzing JavaScript codes that are executed in browsers due to the differences between JavaScript and NodeJS codes. For example, web JavaScript codes use data structures, such as DOM or Document Object Model, for ease of access to web page elements. These structures are not defined for NodeJS and thus are not recognizable by ExpoSE.

Also, web JavaScript is capable of handling the events of a web page. For example, you can write a procedure in JavaScript in response to the user clicking on an HTML element. These events do not exist in NodeJS and also, they are not defined for *ExpoSE*. In fact, JavaScript codes running on browsers are closely related to HTML codes. They might read some values from HTML pages or write new values to them.

Figure 5 demonstrates how a web application is analyzed using our solution. Our python application extracts the HTML and JavaScript codes of the intended web application and prepares an equivalent NodeJS version of it that is processable by *ExpoSE*. Afterward, the ExpoSE conducts the dynamic symbolic analysis of this code to determine the existence of any possible vulnerable points.

In the following, we present the details of how ExpoSE works.

5.1. ExpoSE

This tool is based on Jalangi [23], which receives a NodeJS code with predefined symbolic variables as its input and generates some values for these symbolic variables that lead to the execution of different paths as its output. Actually, we must manually determine the symbolic variables of an application code before using this tool.

```
1. var S$ = require('S$');
2. var t = S$.symbol('X', '');
3. if (t.includes('What')) {
4. throw 'Reachable 1';
5. }
```

Figure 4. *A Piece of Code for Testing the ExpoSE Tool.*

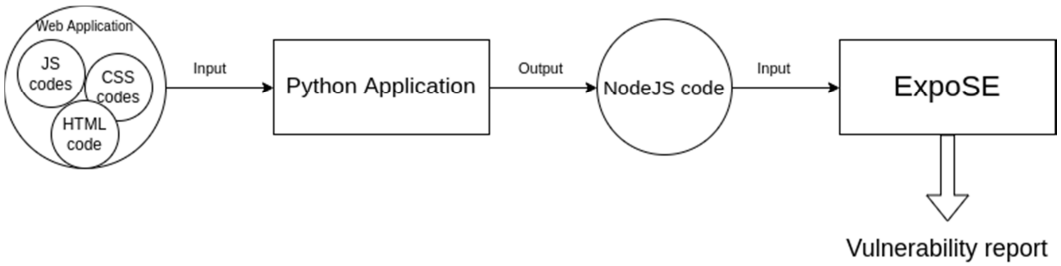

Figure 5. *Architecture of the Proposed Method for Analyzing Tizen Web Applications.*

For example, in Figure 4, lines 3 to 5 show a piece of the NodeJS code of an application. Lines 1 and 2 are added to this code to make it analyzable by *ExpoSE*. Line 1 adds the ExpoSE library and line 2 defines variable *t* using the symbol library function in *ExpoSE* as a symbolic variable named *X*. This code has two different paths based on the value of *t*, and ExpoSE finds the value needed for each path considering the symbolic *t*.

Figure 6 shows the output of *ExpoSE* after dynamic symbolic execution of this application in which there are two different values for variable *t* to execute each path.

Test Case	Time	Alternatives	Error Count
{"_bound":0,"X":""}	4.75s	1	0
{"X":"What","_bound":1}	3.09s	0	1

Figure 6. *ExpoSE output for Sample code in Figure 4.*

5.2. Sample Application

In order to explain the details of our solution, we first present a sample Tizen web application that is based on one of Tizen Studio-based web applications. Figure 7 and Figure 8 show HTML and JavaScript codes of this application, and Figure 9 shows how this application has been executed in a wearable Tizen OS device.

```
1.  <HTML><body>
2.  <script src="js/main.js"></script>
3.  First name: <input type="text" id="frm1" name="fname" value="firstname"
    onChange="myFunction(this.value)">
4.  <div id="textbox" class="contents"></div>
5.  <p id="demo"></p>
6.  <a href="main.HTML">click</a>
```

Figure 7. *The HTML Code of a Sample Web Tizen Application.*

```
1.  window.onload = function () {
2.   var textbox = document.querySelector('.contents');
3.   textbox.addEventListener("click", function(e)){
4.   box = document.querySlector('#textbox');
5.   box.innerHTML = box.innerHTML = = "Basic"?
6.   "Sample":"Basic";
7.   }};
8.   };
9.  function myFunction(val) {
10.  document.getElementById("demo").innerHTML = "Hello" + val; }
```

Figure 8. *The js/main.js Code of a Sample Web Tizen Application.*

Figure 9. *Image of the Sample Application in a Wearable Device.*

In this application, line 3 of the HTML code, in Figure 7, binds the event of changing the input HTML element to a function named *myFunction*. This function is defined in line 9 of the JavaScript code, Figure 8, and it is called whenever the input element changes. The input value of this element would be used directly by *myFunction* to generate an output string. Thus, it is possible to inject some JavaScript codes into the input tag text value and execute malicious commands. For example.

Figure 10 shows the result of executing the sample web application by injecting string <img src=x onerror=alert("XSS")> in the input tag text value. This shows that the application is vulnerable to XSS and HTML injection attacks.

Figure 10. *XSS Vulnerability in Figure 7 and 8.*

5.3. Dynamic Symbolic Execution Process for Tizen Web Applications

As mentioned in the previous sections, to use ExpoSE for dynamic symbolic execution of Tizen web applications, we have to first prepare the application code to be processable by this tool. Therefore, we analyze the JavaScript and HTML codes of the web application through text processing and pattern matching with regular expressions and create an equivalent NodeJS version of the codes. The proposed solution is implemented as a python program and is publicly available in our GitHub repository [26].

The implemented solution first defines necessary data structures such as DOM at the beginning of the equivalent NodeJS code so that NodeJS is able to recognize these objects. Some of these objects are application input entries. For example, the right side of the following statement is an application input entry:

var a=document.getElementById("theID").textContent;

Therefore, the DOM data structure is defined in a way so that it considers the received input data from these entry points as symbolic. In addition, the proposed solution determines output points in the code which are also DOM objects. For example, the left side of the following statement is an output point:

document.getElementById("theID").innerHTML=a;

When something is stored in an output point of the application, we insert a conditional statement that checks if the stored data may contain attack strings. *ExpoSE* calculates this condition as a path constraint when analyzing the final code. In other words, *ExpoSE* attempts to generate proper data that contains attack strings and causes executing a specific path and reaching the *hotspots*. If *ExpoSE* generates such input data, it means that the program is vulnerable to considered attacks.

5.4. Employing the Proposed Solution for the Sample Application

In the case of the sample application in Figures 7 and 8, our solution analyzes the code in the only JavaScript file js/main.js. It reads each line of this code and transfers it with or without changes into an equivalent NodeJS code, which is partly shown in Figure 11. This process is explained in detail in the following.

For anonymous functions in the JavaScript code, our method generates a random name to make the analysis simpler. For example, the first line of Figure 8 defines an anonymous function and assigns it to the *window.onload* method. Therefore, the first line of NodeJS equivalent code considers the random function name instead of this anonymous function and assigns it to *window.onload*, as shown in line 103 of Figure 11. Then, it defines a new function with the same name in the NodeJS code according to its actual definition in the original JavaScript code, as shown in lines 105 to 115. Due to the *hoisting* feature of JavaScript [24], using a function before defining it is not a problem.

```
103  window.onload = oddlnM19;
104
105  function oddlnM19() {
106      var textbox = document.querySelector('.contents');
107      function MvYuBgU1(e) {
108          box = document.querySelector('#textbox');
109          box.innerHTML == "Basic" ? "Sample" : "Basic";
110          if (String(box.innerHTML == "Basic" ? "Sample" : "Basic")
             .includes("&lt;img src=x onerror=alert(1) /&gt; <img
             src=x onerror=alert(1) />")) {
111              throw Error("XSS!");
112          }
113      }
114      textbox.addEventListener("click", MvYuBgU1);
115  }
116
117  function myFunction(val) {
118      "Hello " + val;
119
120      if (String("Hello " + val).includes("&lt;img src=x
         onerror=alert(1) /&gt; <img src=x onerror=alert(1) />")) {
121          throw Error("XSS!");
122      }
123
124  }
125  function html_event_X17SM9(s) {
126      this_ = s;
127      myFunction(this_.value);
128  }
129  TizEx_events_js.push([html_event_X17SM9, new S$.symbol
     ("onchange1", {})]);
130
131
132  window.onload();
133
```

Figure 11. *Part of the generated equivalent NodeJS code.*

For the input/output entries in the JavaScript code, our python script keeps a list of these entries to add a new conditional statement to the target NodeJS code for detecting XSS vulnerability whenever some data is stored in one of these entries. In line 2 of Figure 8, the *textbox* variable becomes an input/output entry, and thus it is added to this list.

In line 3 of this code, there is another anonymous function and the same operation is performed for this function as shown in lines 107 to 113 in Figure 11. The *addEventListener* function in this line is not defined in NodeJS. This function in web JavaScript codes analyzes the occurrence of a specific event and executes a function in response, which might have vulnerabilities in its body. This function should also be defined

in final NodeJS code in a similar manner to *document.getElementById* for NodeJS. Also, these events should be simulated in order to analyze their call-back function bodies. The simulation of application events requires the calling of event-related functions in different orders. Therefore, at the end of the final NodeJS code, we first shuffle the defined events and execute their related functions multiple times with some symbolic values as their inputs.

In line 4 of Figure 8, the *box* variable becomes an input/output entry and is added to the list of input/output entries. Line 5 of this code stores some data into *box.innerHTML* which is one of the HTML elements of this page. In this line, *box.innerHTML* is used twice; one, as the right side of the assignment operation in *box.innerHTML=="basic"* condition. This conditional statement gets the *innerHTML* value of the *box* tag and compares it with the string *"Basic*". Based on the result of this comparison, either *"Basic"* or *"Sample"* string is stored into the other *box.innerHTML* on the left side of the assignment operation. Therefore, the left *box.innerHTML* of this statement is an output entry. In this example, constant strings *"Sample"* and *"Basic"* are stored in an output entry, and thus XSS and HTML injection attacks are not possible here. In real-world applications, these strings might depend on the user input data, and there would be a chance for these attacks. For this line, a conditional statement is inserted into the NodeJS code, as shown in lines 110 to 112 of Figure 11, to check if the code is vulnerable to XSS attack.

The same happens in the first line of *myFunction* body in lines 9 to 10 of Figure 8 and the result in the final NodeJS code is shown in lines 117 to 124 of Figure 11. Here, there is a chance of XSS and HTML Injection because variable *val* depends on the user input.

Note that not all events of a web page are defined in its JavaScript codes, some of these events might be defined in the attributes of its HTML elements, such as *onChange*. For example, we have the *myFunction* function call in line 3 of the sample program's HTML code, Figure 7, in response to the *onChange* event. Therefore, we move on to the HTML page codes and their related events after analyzing the JavaScript codes. The application inserts all event codes of a specific event into a function, changes *this* variable to *this_*, and allocates a symbolic value to *this_* at the beginning of the function, as shown in lines 125 to 128 of Figure 11. Finally, it adds this new function to the array of event functions and treats them similarly to JavaScript events. The reason for assigning *this_* to the input of the function, instead of directly assigning it to a symbolic variable, is to be consistent with events that are defined in JavaScript code using *addEventListener*. Those functions have an input that corresponds to a related event. In this way, the newly created function can be considered as a call-back function for an event.

The final NodeJS code is processed by *ExpoSE* to perform dynamic symbolic execution. If *ExpoSE* generates input data that is consistent with the constraints of paths containing *hotspots*, this means that the program is vulnerable and the generated data can be used to attack the program. Figure 12 illustrates the *ExpoSE* output after processing the sample NodeJS.

6. Evaluation

We have evaluated our implemented solution using two groups of benchmark programs. The experiments are performed in a system with Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz CPU, 16G of RAM and Ubuntu 20.04 operating system. To study the efficiency of our solution in detecting buffer overflow in native applications, we have used NIST SARD benchmark C programs [27] and compiled them to be executed in Tizen. The programs in this benchmark contain one vulnerable statement and one or two similar secure statements that copies some data into a heap or stack buffer using a *strcpy*, *memcpy*, etc function call. Thus, a precise vulnerability detection solution would achieve one true positive and one or two true negative results for each test program. Since there are simple path constraints in these programs, we have made them more complicated by adding an additional if statement to the vulnerable paths. In addition, instead of copying constant data into a heap or stack buffer, we have copied an input string, entered by the user as a command-line argument, into that buffer. A vulnerable function in one of these benchmark programs is presented in Figure 13 as an example, and our added if statement is underlined in line 16. The same if statement has been similarly added to all benchmark programs.

Test Case	Error
{".contentshtml":"",".contentstext":"",".contentsvalue":"","onchange1_elements_value_0_t":"string","onchange1_elements_value_0":" a","#textboxhtml":"","#textboxtext":"","#textboxvalue":"","_bound":2}	Error: XSS!
{".contentshtml":"",".contentstext":"",".contentsvalue":"","#textboxhtml":"Basic","#textboxtext":"","#textboxvalue":"","onchange1_elements_value_0_t":"string","onchange1_elements_value_0":" ","_bound":4}	Error: XSS!

Figure 12. *ExpoSE output after analyzing the sample NodeJS code.*

```
1  void CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_34_bad(char * source)
2  {
3      char * data;
4      CWE121_Stack_Based_Buffer_Overflow__CWE805_char_declare_memcpy_34_unionType myUnion;
5      char dataBadBuffer[50];
6      char dataGoodBuffer[100];
7      /* FLAW: Set a pointer to a "small" buffer. This buffer will be used in the sinks as a destination * buffer in
        various memory copying functions using a "large" source buffer. */
8      data = dataBadBuffer;
9      data[0] = '\0'; /* null terminate */
10     myUnion.unionFirst = data;
11     {
12         char * data = myUnion.unionSecond;
13         {
14             /* POTENTIAL FLAW:
15             Possible buffer overflow if the size of data is less than the length of source */
16             if(source[0] == '7' && source[1] == '/' && source[2] == '4' && source[3] == '2'
                && source[4] == 'a' && source[5] == '8' && source[75] == 'a')
17             {
18                 memcpy(data, source, strlen(source)*sizeof(char));
19             }
20             data[100-1] = '\0'; /* Ensure the destination buffer is null terminated */
21             printLine(data);
22         }
23     }
24 }
```

Figure 13. *A vulnerable function in a benchmark program.*

Table 1 presents the results of analyzing NIST SARD benchmark programs by our implemented solutions. The columns in this table represent, from left to right, the number of test programs with stack or heap buffer overflow vulnerability, the total number of true positives, the total number of true negatives, the total number of false positives, and the total number of false negatives in the reports of our solution for each group. Figure 14 also represents the average time of analyzing a benchmark program with a specific vulnerable statement by our implemented solution. As the results demonstrate, our solution could detect all vulnerabilities in these programs effectively.

Table 1. *The results of evaluating our solution using vulnerable C benchmark programs.*

#Test cases	#True positives	#True negatives	#False positives	#False negatives
85 (stack based)	85	115	0	0
90 (heap based)	90	116	0	0

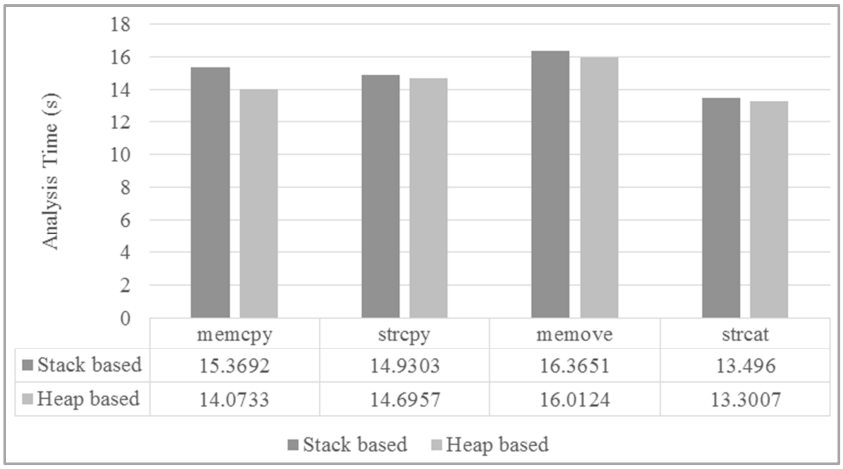

Figure 14. *Average analysis time of a test program with a specific vulnerable data copy operation in a stack or heap buffer, i.e. memcpy, strcpy, memove and strcat.*

To evaluate our solution for analyzing Tizen web applications, we have designed a group of vulnerable HTML and Javascript codes, as we could not find appropriate benchmark programs. There are various scenarios for XSS vulnerability occurrence and increasing levels of path complexity in these programs. These programs alongside a script to re-execute the experiment exist in the *testcase* directory of our solution source code.

Table 2 presents the results of this experiment. The columns in this table represent, from left to right, the name of each test program, the analysis time by our solution in seconds, the number of test cases generated to test the program, if our solution could detect the vulnerability with a true positive report and the constraints and details about the vulnerable statement in the program. As an example, XSS arises inside an event callback function in the test program named "Web4" and the path constraint on the input string to reach the vulnerable statement is "*input[0] = 'a'*". Our

solution has generated 4 input values to execute the program with and successfully detect this vulnerability.

As shown in this table, our solution could detect the vulnerability in all test programs with various complicated constraints. It is worth mentioning that the majority of our analysis time belongs to *ExpoSE*, thus the efficiency of our solution would be improved if we substitute our symbolic execution engine with a faster one in the future.

Table 2. *The results of evaluating the proposed solution using vulnerable HTML and JavaScript codes.*

Test Program	Time(s)	#test cases	True Positive	Constraints
Web1	7.59	2	yes	-
Web2	7.70	2	yes	- XSS inside an event call-back function
Web3	12.90	4	yes	- input[0] = 'x'
Web4	14.20	4	yes	- XSS inside an event call-back function - input[0] = 'a'
Web5	20.71	6	yes	- input[0] = 'a' - input[2] = 'b'
Web6	17.00	6	yes	- XSS inside an event call-back function - ! input.includes('<')
Web7	25.63	9	yes	- XSS inside an event call-back function - input[1] = 'b'
Web8	25.54	9	yes	- input[1] = 'b'
Web9	23.40	6	yes	- XSS inside an event call-back function - input[0] = 'a' - input[1] = 'b'
Web10	24.58	7	yes	- XSS inside an event call-back function - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x'
Web11	30.08	8	yes	- XSS inside an event call-back function - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x' - ! input.includes('XYZ')
Web12	37.52	10	yes	- XSS inside an event call-back function - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x' - ! input.includes('XYZ') - input[5] = 'j'
Web13	24.43	7	yes	- XSS inside an event call-back function - There is a variable, initially set to 0, it is incremented inside the same event call-back function. XSS arises when its value is not 0. - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x'
Web14	28.00	7	yes	- XSS inside an event call-back function - There is a variable, initially set to 0, it is incremented inside a separate event call-back function. XSS arises when its value is greater than 0. - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x'
web15	41.42	14	yes	- XSS inside an event call-back function - There is a variable, initially set to 0, it is incremented inside a separate event call-back function. XSS arises when its value is greater than 0. - input[0] = 'a' - input[1] = 'b' - input.slice(-1) = 'x' - ! input.includes('XYZ')

7. Conclusion and Future Works

This paper presented a dynamic symbolic analysis solution for native and web applications of Tizen OS. In this solution, dynamic symbolic analysis of native applications is performed by executing the application in the Tizen OS environment via connecting to the *gdbserver* of Tizen and by the use of *Symbion* tool. The analysis of web applications is conducted outside of Tizen OS due to the lack of dynamic symbolic execution frameworks for web applications deployed on this OS, alongside the possibility of executing their JavaScript and HTML codes in other web engines. Therefore, we process the application's JavaScript and HTML codes and turn them into an equivalent NodeJS code for dynamic symbolic execution using *ExpoSE*. This paper

fully described the dynamic symbolic execution process for native and web applications while demonstrating the effectiveness of the proposed solution for detecting vulnerabilities in two sample applications and a group of benchmark programs.

In the future, we intend to expand our method to discover other types of vulnerabilities such as race conditions and use-after-free in Tizen applications. Also, when it comes to web applications, the implemented solution is limited to ES5 or older versions. Therefore, it does not support newer syntax versions of JS or any related libraries, such as JQuery. We will define these libraries and syntaxes in the future so that our solution could analyze new versions of JavaScript codes.

References

[1] Windows IoT core, https://docs.microsoft.com/en-us/windows/iot/, Accessed 18 January 2022.

[2] Amazon FreeRTOS, https://aws.amazon.com/freertos/, Accessed 18 January 2022.

[3] Samasung Tizen, https://www.tizen.org, Accessed 18 January 2022.

[4] Contiki, https://www.contiki-ng.org/, Accessed 18 January 2022.

[5] Mullen G., Meany L.: Assessment of Buffer Overflow Based Attacks On an IoT Operating System. IEEE 2019 Global IoT Summit (GIoTS) pp. 1-6 (2019).

[6] Chess B., McGraw G.: Static Analysis for Security. IEEE security & privacy, vol. 2, pp. 76-79 (2004).

[7] Godefroid P., Klarlund N., Sen K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp. 213-223 (2005).

[8] Chaabouni N., Mosbah M., Zemmari A., Sauvignac C., Faruki P.: Network Intrusion Detection for IoT Security Based on Learning Techniques. IEEE Communications Surveys & Tutorials, vol. 21, pp. 2671-2701 (2019).

[9] Rizvi S., Orr R., Cox A., Ashokkumar P., Rizvi M.: Identifying the attack surface for IoT network. Internet of Things, vol. 9 (2020).

[10] Rathore S., Kwon B. W., HyukPark J.: BlockSecIoTNet: Blockchain-based decentralized security architecture for IoT network. Network and Computer Applications, vol. 143, pp. 167-177 (2019).

[11] Alnaeli S., Sarnowski M., Sayedul Aman M., Abdelgawad A., Yelamarthi K.: Vulnerable C/C++ Code Usage in IoT Software Systems. 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), pp. 348-352 (2016).

[12] Abraham A.: Hacking Tizen: The OS of Everything. In: Proceedings of the HITBSecConf-Hack In The Box Security Conference, Amsterdam, The Netherlands, pp. 26-29 (2015).

[13] Gritti F., Fontana L., Gustafson E., Pagani F., Continella A., Kruegel C., Vigna G.: SYMBION: Interleaving Symbolic with concrete execution. IEEE Conference on Communications and Network Security (CNS), pp. 1-10, (2020).

[14] Loring B., Mitchell D., Kinder J.: ExpoSE: Practical Symbolic Execution of Standalone JavaScript. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software., pp. 196-199 (2017).

[15] Shoshitaishvili, Wang Y., Hauser R., Krugel C., Vigana G.: Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. NDSS, vol. 1, pp. 1-1 (2015).

[16] Muench M., Nisi D., Francillon A., Balzarotti D.: Avatar 2: A multi-Target Orchestration Platform. InProc. Workshop on Binary Anal. Res. (colocated with NDSS Symp.), vol. 18, pp. 1-11 (2018).

[17] Qiling Framework, https://qiling.io. Accessed 18 January 2022.

[18] Li G., Andreasen E., Ghosh I.: SymJS: Automatic Symbolic Testing of JavaScript Web. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 449-459, (2014).

[19] JSSeek, http://glasnost.itcarlow.ie/~softeng4/C00137906/index.html. Accessed 18 January 2022.

[20] Basiri M., Mouzarani M.: Assessing the Resistance of the Internet of Things Applications to Memory Corruption Attacks. EAI SaSeIoT (2021).

[21] Baldoni R., Coppa E., D'ELIA D. C., Demetrescu C., Finocchi I.: A Survey of Symbolic Execution Techniques. ACM Computing Surveys (CSUR) 51. 3, pp. 1-39, (2018).

[22] angr. https://angr.ir. Accessed 18 January 2022.

[23] Sen K., Kalasapur S., Brutch T., Gibbs S.: Jalangi: A Tool Framework for Concolic Testing, Selective. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 615-618, (2013).

[24] MDN Web Docs: JS hoisting. https://developer.mozilla.org/en-US/docs/Glossary/Hoisting. Accessed 18 January 2022.

[25] Nicula Ș., Zota R. D: Exploiting stack-based buffer overflow using modern day techniques. Procedia Computer Science, vol. 160, pp. 9-14 (2019).

[26] TizenSecurity. https://github.com/SoftwareSecurityLab/TizenSecurity. Accessed 18 January 2022.

[27] NIST Software Assurance Reference Dataset. https://samate.nist.gov/SARD/. Accessed 5 August 2022.