Automated Source Code Instrumentation for Verifying Potential Vulnerabilities

Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee(✉)

Department of Computer Science and Engineering, Korea University,
Seoul, South Korea
{hongzhe,jaesangoh,hakjoo_oh,heejo}@korea.ac.kr

Abstract. With a rapid yearly growth rate, software vulnerabilities are making great threats to the system safety. In theory, detecting and removing vulnerabilities before the code gets ever deployed can greatly ensure the quality of software released. However, due to the enormous amount of code being developed as well as the lack of human resource and expertise, severe vulnerabilities still remain concealed or cannot be revealed effectively. Current source code auditing tools for vulnerability discovery either generate too many false positives or require overwhelming manual efforts to report actual software flaws. In this paper, we propose an automatic verification mechanism to discover and verify vulnerabilities by using program source instrumentation and concolic testing. In the beginning, we leverage CIL to statically analyze the source code including extracting the program CFG, locating the security sinks and backward tracing the sensitive variables. Subsequently, we perform automated program instrumentation to insert security probes ready for the vulnerability verification. Finally, the instrumented program source is passed to the concolic testing engine to verify and report the existence of an actual vulnerability. We demonstrate the efficacy and efficiency of our mechanism by implementing a prototype system and perform experiments with nearly 4000 test cases from Juliet Test Suite. The results show that our system can verify over 90 % of test cases and it reports buffer overflow flaws with *Precision* = 100 % (0 FP) and *Recall* = 94.91 %. In order to prove the practicability of our system working in real world programs, we also apply our system on 2 popular Linux utilities, Bash and Cpio. As a result, our system finds and verifies vulnerabilities in a fully automatic way with no false positives.

Keywords: Automatic instrumentation · Security sinks · Security constraints · Vulnerability verification

This research was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP)(R0190-15-2011, Development of Vulnerability Discovery Technologies for IoT Software Security).

© IFIP International Federation for Information Processing 2016
Published by Springer International Publishing Switzerland 2016. All Rights Reserved
J.-H. Hoepman and S. Katzenbeisser (Eds.): SEC 2016, IFIP AICT 471, pp. 211–226, 2016.
DOI: 10.1007/978-3-319-33630-5_15

1 Introduction

Even though security experts are making best efforts to ensure the software security, the number of software vulnerabilities is still increasing rapidly on a yearly basis, leaving great threats to the safety of software systems. According to the Common Vulnerabilities and Exposures(CVE) database [1], the number of CVE entries has increased from around 1000 CVEs yearly in 2000 to over 8000 yearly in 2015. The discovery and removal of vulnerabilities from software projects have become a critical issue in computer security. Nowadays, because of enormous amount of code being developed as well as limited manpower resource, it becomes harder and harder to audit the entire code and accurately address the target vulnerability.

Security researchers have devoted themselves into developing static analysis tools to find vulnerabilities [9]. The large coverage of code and access to the internal structures makes these approaches very efficient to find potential warnings of vulnerabilities. However, they often approximate or even ignore runtime conditions, which leaves them a great amount of false positives.

Recently, more advanced static analysis methods are proposed [4,5,15]. They either encode insecure coding properties such as missing checks, un-sanitized variables and improper conditions into the analyzer for vulnerability discovery, or they model the known vulnerability properties and generate search patterns to detect unknown vulnerabilities. Even though these approaches can find vulnerabilities using search patterns and exclude the majority of code needed to be inspected, they still require security-specific manual efforts to verify the vulnerability at the very end, which is neither efficient for vulnerability discovery nor feasible for non-security specialists to use it.

According to the previous efforts of researchers, finding exact vulnerabilities in a fully automatic way has been challenging. To automate the overall process of vulnerability detection and verification, we classify the potential vulnerable security sinks into basic 4 types and apply security constraint rules(corresponding to each type) to automatically instrument vulnerability triggering probes into the source code in order to verify vulnerabilities. In this paper, we propose an automatic mechanism to detect and verify software vulnerabilities from C code by using program source instrumentation and concolic testing. In the beginning, we leverage CIL [3] (C intermediate language) to statically analyze the source code including extracting the program CFG(control flow graph), locating the sensitive security sinks and backward tracing the sensitive variables. Subsequently, we classify the security sinks into 4 basic types and perform automated program instrumentation to insert security probes according to different properties of the security sink types, ready for the vulnerability verification. Finally, the instrumented program source is passed to the concolic(CONCrete + symbOLIC) testing engine [10,11] to report and verify the existence of an actual vulnerability. We here focus on buffer overflow vulnerabilities since this type of vulnerability is the major cause for malicious intensions such as invalid memory access, denial of service(system crach) and arbitrary code execution. We demonstrate the efficacy and efficiency of our mechanism by implementing a prototype system and perform experiments with 4000 buffer overflow test cases from Juliet Test Suite [14]. The results show that our prototype system gets a detection result with $Precision = 100\,\%$ and $Recall = 94.91\,\%$. In order to prove the practicability of our system working in real world programs, we also apply our mechanism on Linux utilities such as Bash and Cpio. As a result, our system finds and verifies vulnerabilities in a fully automatic way with no false positives.

Main contributions of our study are described as follows:

- **Fully Automated Verification for Vulnerability Discovery**. We propose, design and implement a fully automated verification mechanism to detect and verify vulnerabilities with zero interference of manual efforts, which can expand the system usability to more general users such as non-security specialist.
- **Memory Space Analysis(MSA) for Verifying Security Requirements**. We verify the existence of vulnerabilities by generating triggering inputs which violate security constraints($\overline{SC}$). The memory space analysis(MSA) enables us to track the size of buffer space at runtime and set the $\overline{SC}$ conditions accurately. It decreases the false positives for vulnerability verification.

2 Related Work

Source code auditing have been actively researched by security researchers for software assurance and bug finding. Previous researchers have proposed different approaches for static source code auditing and have developed source code static analysis tools for vulnerability discovery.

Flawfinder [9] applies a pattern matching technique to match the security sinks in the source code and report them as vulnerabilities. Even though these approaches can analyze large amount of source code and report vulnerabilities fast, they generate too many false positives due to a lack of analysis about program data flow and control flow information.

Chucky [15] statically taints the source code and identifies missing conditions linked to security sinks to expose missing checks in source code for vulnerability discovery. VCCFinder [4] proposes a method of finding potentially dangerous code with low false positive rate using a combination of code-metric analysis and meta-data from code repositories. It also trains a SVM classifier by the vulnerability commits database to flag code as vulnerable or node. Yamaguchi et al. [5] models the known vulnerability properties and generate search patterns for taint-style vulnerabilities. The generated search patterns are then represented by graph traversals which is used for vulnerability mining in a code property graph database [6]. Even though these approaches can find vulnerabilities using search patterns and exclude the majority of code needed to be inspected, they still require security-specific manual efforts to verify the vulnerability at the very end, which is neither efficient for vulnerability discovery nor feasible for general users(nonspecialist) to use it. Driven by this, there is an urgent need to build a fully automatic system to accurately catch vulnerabilities within reasonable time as well as the expansion of usability to more general users.

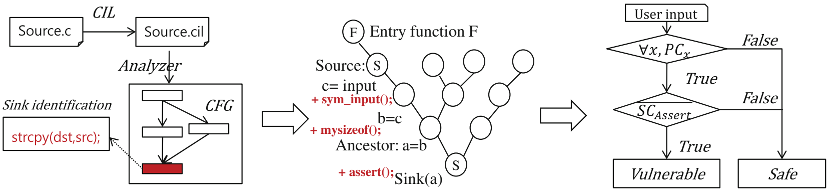
Based on the weakness of the above discussion, we are looking into an automatic and efficient way to do vulnerability verification. Symbolic execution has been proposed to do program path verification but it cannot resolve complex programs with enormous amount of path constraints [12]. Concolic testing [10,11] was proposed to improve symbolic execution in order to make it more practical in real world programs. KLEE [11] was developed to automatically generate high-coverage test cases and to discover deep bugs and security vulnerabilities in a variety of complex code. CREST-BV [10] has shown a better performance than KLEE in branch coverage and the speed of test case generation. However, these approaches suffer from path explosion problem which stops them from scaling to large programs.

CLORIFI [8] is the closest research to this paper. It proposes a method to detect code clone vulnerabilities by the combination of static and dynamic analysis. However, it has not been fully automated and it still requires manual efforts to do source instrumentation for concolic testing, which is still a tedious task. In this paper, we propose, design and implement a fully automated verification mechanism to detect and verify vulnerabilities. In our mechanism, we do vulnerability verification using concolic testing after an automated sink detection and source code instrumentation process, which reduces false positives. We also applies the runtime buffer memory space analysis(MSA) to track the real size of a buffer space which helps us to improve the accuracy of vulnerability discovery.

3 Proposed Mechanism

Discovery of vulnerabilities in a program is a key process to the development of secure systems. In order to find exact vulnerabilities in a fast and accurate way and to reduce the tedious manual efforts for vulnerability verification, we propose a fully automated mechanism to detect and verify software vulnerabilities by taking advantage of both static analysis and concolic testing.

Before we go into detailed description of our approach, the general process is illustrated in Fig. 1. Our mechanism mainly consists of 3 phases which are **code transformation**, **automated instrumentation**, and **vulnerability verification**. In the phase of code transformation, we first leverage the library of CIL to parse the source code into CIL program structures such as function definitions, variables, statements, expressions and so on, and calculate the control flow graph(CFG) information of each function. The reason why we do CIL code transformation is to simplify code structures for efficient static analysis. We then identify the security sinks(potential vulnerable) to get potential vulnerable points. In the second phase, we apply backward data tracing on sensitive variables of each sink to find the variable input location. Then, we perform automatic program instrumentation and prepare the testing object for vulnerability verification. In the last phase of automated instrumentation, we verify each potential security sink to report vulnerabilities using concolic testing.



(1) Code transformation (2) Automated instrumentation (3) Vulnerability verification

Fig. 1. General overview of our approach.

3.1 Using the CIL

CIL (C Intermediate Language) [3] is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs. The reason why we use CIL is that it compiles all valid C programs into a few core constructs without changing the original code semantics. Also CIL has a syntax-directed type system that makes it easy to analyze and manipulate C programs. Moreover, CIL keeps the code location information which enables us to pinpoint the exact location when reporting vulnerabilities(an example is shown in Fig. 5).

By using the provided APIs of CIL, we are able to flatten complex code structures into simple ones (e.g., all looping constructs are reduced to a single form, all function bodies are given explicit return statements). Subsequently, we extract and calculate the control flow graphs(CFGs) of each function using some wrapped module of CIL. Each created node in CFG corresponds to a single instruction in the source code, referred from Sparrow [7], a stable and sound source code analysis framework. Treating the most basic code unit(instruction level) as a CFG node can help us precisely and rapidly address the sensitive sinks and variables as well as providing convenience for the backward sensitive data tracing which will be explained in detail in the following sections.

3.2 Identification of Security Sinks and Sensitive Variables

Since most of buffer overflow vulnerabilities are caused by attacker controlled data falling into a security sink [13], it is crucial to first dig out security sinks. In this paper, we focus on the security sinks which often lead to buffer overflow vulnerabilities. Before that, we explain the definition of security sinks and how they can be classified according to argument properties.

Security Sinks: Sinks are meant to be the points in the flow where data depending from sources is used in a potentially dangerous way. Typical security-sensitive functions and memory access operations are examples of security sinks. Several typical types of security sinks are shown in Table 1.

As we can see from the Table 1, basically, security sinks are either security sensitive functions or a buffer memory assignment operation by index. Further more, according to the number of arguments and whether there is a format string("%s") argument inside a security sensitive function, we classify the security sensitive functions into 3 types in order to generalize the automatic process of backward tracing and program instrumentation which will be explained in following parts. Along with the last type(buffer assignment), we classify the security sinks of buffer overflow vulnerability into 4 types.

Table 1. Sink classification and variable location

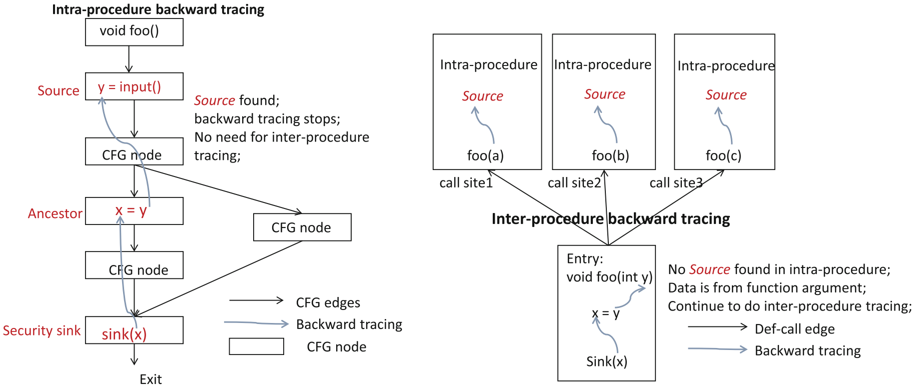
Sink type	Description	Argument format	Sensitive functions	Variable positions
Type 1	functions with two arguments	fn(dst,src)	strcpy, wcscpy strcat, wcscat	2
Type 2	functions with three arguments	fn(dst,src,n)	memcpy, memmove, strncpy strncat, wcsncpy, wcsncat	2,3
Type 3	function with format strings	fn(dst,n,"%s",src)	snprintf, swprintf	2,4
Type 4	memory operations	buffer[i] = expr	dstbuf[i] = 'A'	index:i

- Type 1: Security sensitive functions with 2 arguments: a destination buffer pointer(*dst*) and a source buffer pointer(*src*). The typical argument format is: *fn(dst,src)* and the sensitive variable is the 2nd variable(*src*) in the argument list. The instances of sinks include: *strcpy, wcscpy, strcat and wcscat.*
- Type 2: Security sensitive functions with 3 arguments: a destination buffer pointer(*dst*), a source buffer pointer(*src*) and a number of bytes integer(*n*). The typical argument format is *fn(dst,src,n)* and sensitive variable is the 2nd variable(*src*) and 3rd argument(*n*) in the argument list. The instances of sinks include: *memcpy, memmove, strncpy, strncat, wcsncpy and wcsncat.*
- Type 3: The security sensitive functions with format string argument: a destination buffer pointer(*dst*), a number of bytes integer(*n*), a format string argument and a source buffer pointer(*src*). The typical argument format is *fn(dst,n,format,src)* and the sensitive variable is the 2nd variable(*n*) and the 4th argument(*src*). The instances of sinks include: *snprintf* and *swprintf.*
- Type 4: The buffers are assigned by some value using buffer array index. This case causes buffer overrun when the index is out of buffer size bound. The typical format is: *buffer[index] = expr* and sensitive variable is the *index.* A instance of this type of sink is: *dstbuf[i] = 'A'.*

After the classification of security sinks, we identify the security sinks as potential vulnerabilities using a fast pattern matching approach over the CIL structures of the source code and extract sensitive variables needed to be backwardly traced in the following step based on the table above.

3.3 Backward Data Tracing

Since instrumentation points are needed before performing automated instrumentation, we propose backward data tracing to find the program input place



(1) Intra-procedure tracing

(2) Inter-procedure tracing

Fig. 2. Backward data tracing.

and treat it as an instrumentation point. Backward data tracing finds the program input location for the corresponding sensitive variables in the sink which reduces the whole input search space of a program. This will help us greatly improve the efficiency for the vulnerability verification. We perform intra- and inter-procedure backward tracing based on the nodes of the control flow graph extracted from CIL. Figure 2 shows the concept of intra-procedure and inter-procedure backward tracing respectively.

Concepts and Definitions. As shown in Fig. 2(1), starting from a security sink, we begin to backwardly trace the corresponding sensitive variable until we reach the source of the sensitive data. In order to understand the process of backward tracing, there are several terms that we need to know.

Source: *Source* is the original definition point of a variable. It is the node where the value of the variable does not depend on any other variable. For instance, the starting points where un-trusted input data is taken by a program. The *Source* is one the following 2 cases.

- $v_0 = gets()$; Assignment node where the left variable is assigned by a user input function such as gets(), scanf() and read(). We also maintain a user input function list.
- $v_0 = 5$; Assignment node where the left variable is assigned by a constant.

Ancestor: The ancestor A of a node N is described as: the traced sensitive data of N gets its value from A. Ancestor nodes are intermediate nodes while sensitive data gets propagated. The ancestor node of a certain variable v_0 could be one of the 4 cases below:

- $v_0 = expression(v_1)$; Node where variable assigned by expression
- $v_0 = f(m, n, ...)$; Node where variable assigned by function return value

- $f(v_0)$; Node where variable assigned inside a function call
- *void* $f(char\ v_0)$; Node for function declaration

The Description of Procedure. As shown in Fig. 2, the intra-procedure backward tracing starts from the initial sensitive variable in the security sink such as $sink(x)$ in Fig. 2(1) and recursively find its ancestor and identify a new variable needed to be traced. The intra-procedure backward tracing stops when the current node is the *Source* of the traced sensitive variable(whole backward tracing also stops) or it stops when the current node is the function entry node. In the later case, *Source* cannot be found in the current function and the data flow comes from argument passing of the function call, so we need further do inter-procedure backward tracing to find the *Source* of the sensitive variable. The inter-procedure tracing(see Fig. 2(2)) starts from intra-procedure tracing. It then checks the return node of intra-procedure backward tracing. If the node is *Source*, the procedure returns and backward tracing ends. If the node is the function entry node, the procedure finds out all the call sites of the current function and identifies the corresponding sensitive variable in each call site. Then it applies intra-procedure backward tracing on sensitive variables in each call site as a new starting point. The whole backward tracing stops and exits when the *Source* of the sensitive variable in the security sink is found.

3.4 Program Source Instrumentation

After backward tracing, we get security sinks and *Sources* of the corresponding sink and store them into a list of sinks(sink_list) and a list of Sources(source_list) accordingly. We also establish a sink-source mapping between the 2 lists which helps us to correctly find the corresponding source for a certain sink. To instrument the program source, we make security probes(assertions) based on our predefined security requirements (Table 2) right before the *security sink* and replace the source input statement with symbolic values. To automate the overall process of source code instrumentation, we generalize the security constraint rules for the 4 basic types of vulnerability to automatically instrument bug triggering probes into the source code in a more generalized way.

Program Constraints(PC) and Security Constraints(SC): Program constraints are generated by following a specific branch according to conditional statements in the program. Program inputs which satisfy a set of program constraints are meant to execute a certain path of the program. Security constraints are clearly high-level security requirements, which is also used as our baseline to make security probes before the security sinks. For instance, the length of the string copied to a buffer must not exceed the capacity of the buffer. We define security requirements for security sinks such us security-sensitive functions and buffer assignment with index based on the condition that related arguments must satisfy to ensure the security of software systems. In our work, we have collected 14 library functions from Juliet Test Suite which are well known to be “insecure” for buffer overflows as security-sensitive functions and generated security requirements for them. Table 2 shows part of our predefined security constraints for security sinks. When there are inputs which satisfy program constraints but violates security constraints($PC \wedge \overline{SC}$) at a certain point during the execution, the program is considered to be vulnerable. To suggest a way to extend the method to cover other types of vulnerabilities, we will investigate the vulnerability features and define more sinks and the corresponding security requirements for vulnerability types such as integer overflows, format strings and divide-by-zeros.

Table 2. Security requirements for security sinks

Security sinks	Security requirement	Description
strcpy(dst,src)	dst.space > src.strlen	Space of dst should be bigger than the string length of src
strncpy(dst,src,n)	(dst.space ≥ n) ∧ (n ≥ 0)	Space of dst should be bigger or equal to the positive integer n
strcat(dst,src)	dst.space > dst.strln + src.strlen	Space of dst should be bigger than the total string length of dst and src
getcwd(buf,size)	(buf.space ≥ size) ∧ (size ≥ 0)	Space of buf should be bigger or equal to the positive integer size
fgets(buf,size,f)	(dst.space ≥ size) ∧ (size ≥ 0)	Space of dst should be bigger or equal to the positive integer size
buf[i] = expr.	buf.space > i	Space of buf should be bigger than the index value i

Instrument Input Location and the Security Sink: To instrument the input location, we first iterate over all the element in the *source_list*, if the current *Source* takes the data from user input such as command line, network data, or a file, we replace the user input data with symbolic input values. For example, "$a = readfile()$;" will be replaced by "$a = sym\_input()$;".

To instrument the security sink, we insert an assertion statement right before the security sink based on the security rules defined in Table 2. For example, before the sink $strcpy(dst, src)$, we insert an assertion statement $assert(dst.space > strlen(src))$. However, there is a problem here. We can easily get the length of a string by using "$strlen()$" function(C library function), but the space of a buffer is hard to be determined at runtime as well as at compile time. Someone may say, we can always get the size of a buffer by "$sizeof()$". This is not correct when we measure the size of the buffer memory space. E.g., if we want to get the real space that dst points to, we cannot use "$sizeof(dst)$"

because it will always return the actual size of a pointer itself which is the number 4 at 32 bit architectures. In order to get the real buffer size of a buffer pointer, we propose a pointer analysis approach to correctly get the buffer size at program runtime.

Memory Space Analysis(MSA) at Runtime: As we can see from Table 2, in the security requirement rules, we have to get the "buffer.space" value to accurately set the bug triggering condition and instrument the sinks. However, the common static analysis of source code cannot get the runtime updating information about memory size of a buffer pointer. This usually will result in an inaccurate assertion of SC violating condition, which makes the system generate possible false positives when reporting vulnerabilities. The runtime memory space analysis enables us to accurately track the real size of a pointer's buffer space and helps us to correctly instrument the program so as to ensure high accuracy of vulnerability verification. We make a library called "libmysizeof" and it provides 3 major functions: *mysizeof_store()*, *mysizeof_propagate()* and *mysizeof_getspace()*. We insert *mysizeof* functions in the corresponding place in the source code. The steps are shown below:

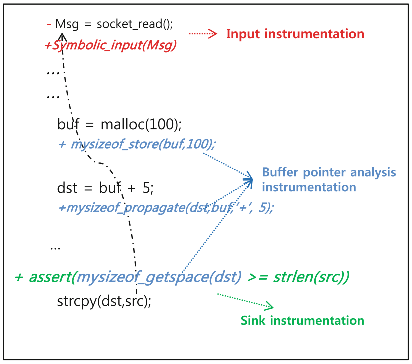
- Iterate over all the instructions in the source code and identify buffer allocation statement such as *buf = malloc(n);* and *char buf[100];*. Then, we store the buffer pointer name and the corresponding size in a global map M by inserting *mysizeof_store()* function at the current location.
- Identify each pointer propagation instruction such us *p_2 = p_1 + 5;*. The *mysizeof_propagate()* will propagate the size of *p_1* to get the size of *p_2* according to the operation and store the size of *p_2* into the map *M*. We then insert this *mysizeof_propagate()* function at the current location.
- When we need to get the runtime size of a pointer's buffer space, we insert *mysizeof_getspace()* function at a certain location in the source code to get the correct buffer space.

After inserting "mysizeof" functions, we can get the accurate size of a pointer's buffer space at runtime. The buffer size information can then be used in the assertions. For instance, $assert(mysizeof\_getspace(dst) > strlen(src))$. Figure 3 shows an example of instrumenting pointer analysis functions along with input and sink instrumentation.

Until here, we prepare a testing source object from the program input to the potential vulnerable sinks. This object is usually a small part of the whole program source which helps us to release the burden of next stage.

3.5 Vulnerability Verification Using Concolic Testing

We apply concolic testing in our mechanism to verify the potential vulnerabilities. The general principle of the verification is to find inputs which satisfy all the program constraints(PCs) but violate the security constraints(SCs) as shown in Fig. 1. Symbolic execution and concolic execution have been widely used in software testing and some have shown good practical impact, such as KLEE [11] and



```
- Msg = socket_read();
+Symbolic_input(Msg)
...
...
  buf = malloc(100);
  + mysizeof_store(buf,100);

  dst = buf + 5;
  +mysizeof_propagate(dst,buf,'+', 5);

  ...
+ assert(mysizeof_getspace(dst) >= strlen(src))
  strcpy(dst,src);
```

Fig. 3. Automatic instrumentation.

CUTE [2]. However, they suffer from path explosion problem which makes them cannot scale well to large real world programs. In our scenario, the backward tracing module helps us to find the program inputs which are related to the sensitive data instead of the whole program input space. This can mitigate the path explosion problem mentioned before. Our approach for concolic testing to verify potential vulnerabilities mainly follows a general concolic testing procedure [10]. However, the difference is that we focus on generating an input to execute the vulnerable branch instead of generating inputs to traverse every possible paths of the program. Hence, it is more cost efficient when doing concolic testing.

4 Experimental Results

4.1 Implementation

System Prototype: We have implemented our mechanism by developing a prototype system. Our system consists of 3 phases: **code transformation**, **automated instrumentation**, and **vulnerability verification**. Its architecture is described in Fig. 4. The system is used to discover buffer overflow vulnerabilities in software projects.[1]

Environment Setup: We performed all experiments to test our automatic vulnerability detection and verification system on a desktop machine running Linux Ubuntu 14.04 LTS (3.3 GHz Intel Core i7 CPU, 8 GB memory, 512 GB hard drive).

Dataset: For the experiment, we prepared 2 kinds of datasets - Juliet Test Suite and Linux utilities. First one is Juliet Test Suite provided by US National Security Agency(NSA) which has been widely used to test the effectiveness of vulnerability detection tools. To test our tools, we prepared 3,969 files for stack based buffer overflow vulnerabilities, each of which belongs to 4 basic types

[1] Our program and sample scripts are available at http://cssa.korea.ac.kr/clorifi.

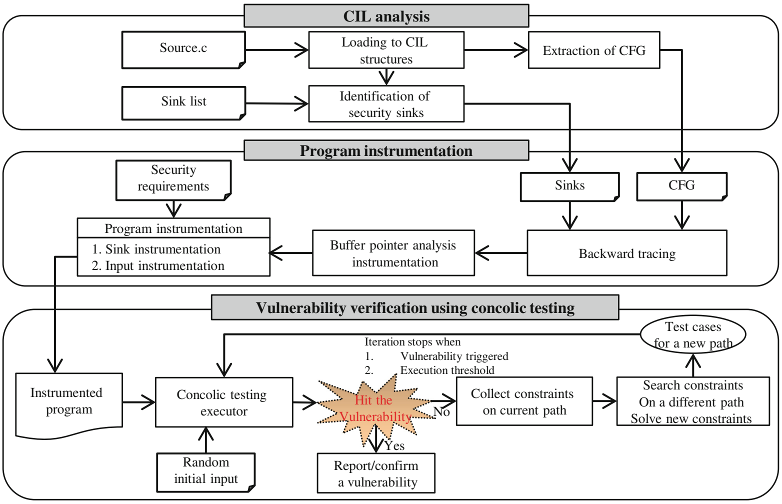

CIL analysis
Source.c
Loading to CIL structures
Extraction of CFG
Sink list
Identification of security sinks
Program instrumentation
Security requirements
Sinks
CFG
Program instrumentation
1. Sink instrumentation
2. Input instrumentation
Buffer pointer analysis instrumentation
Backward tracing
Vulnerability verification using concolic testing
Test cases for a new path
Iteration stops when
1. Vulnerability triggered
2. Execution threshold
Instrumented program
Concolic testing executor
Hit the Vulnerability
No
Yes
Collect constraints on current path
Search constraints On a different path Solve new constraints
Random initial input
Report/confirm a vulnerability

Fig. 4. The system architecture.

based on our sink classification in Table 1. The second dataset is 2 famous Linux utilities, Bash and Cpio, which is used to prove the practicability of our system.

4.2 Experimental Results

We have conducted our experiments with two types of dataset.

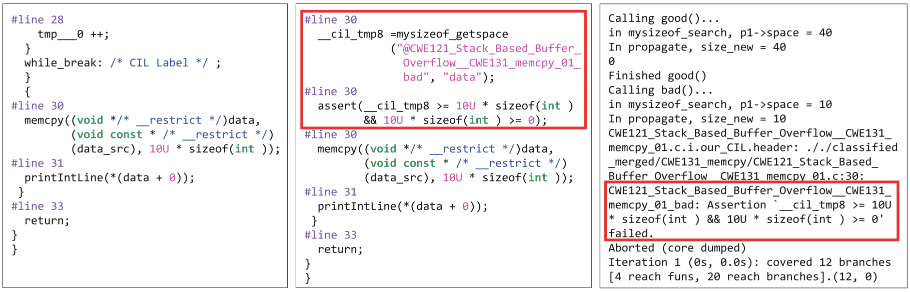
Juliet Test Suite. We tested our approach with 3,969 testcases of Stack Based Buffer Overflow vulnerabilities of Juliet Test Suite. The number of samples covered in Table 3 states the number of cases that we verified (over 90 % in total). Our processing time for 3,969 testcases is nearly 20 min, which includes about 2 min of instrumentation and 17 min of concolic testing. Table 3 shows the number of testcases processed, number of good and bad sinks, and elapsed time of instrumentation and concolic testing for each type.

We checked the rest of the cases that we couldn't verify and found out that our frontend parser cannot handle non-standard coding style such as wide string L"AAA"(222 cases). Besides, our tool failed to correctly instrument the source code when handling function pointers(76 cases) and complex use of pointers such us pointer's pointer(105 cases).

A working example is shown in Fig. 5. Sub figures in Fig. 5 indicate the sequence of our mechanism. Figure 5(1) shows the result of CIL transformation of a file named CWE121_Stack_Based_Buffer_Overflow__CWE131_memcpy_01.c. Then we show the instrumentation result of this case in Fig. 5(2). The assertion $assert(dst.space \geq n) \wedge (n \geq 0)$ is automatically inserted before the sink *memcpy*. Figure 5(3) shows the execution result of concolic testing. In this step, it actually verifies the vulnerability by checking whether the execution of program violates

Table 3. Type of vulnerabilities and base information of experiment

Type	Number of Samples Covered	Elapsed Time (Instrumentation)	Elapsed Time (Concolic Testing)	Number of Bad Sinks	Number of Good Sinks
1	2,065/2,217	78.816 s	547.879 s	2,217	3,126
2	695/732	25.893 s	175.986 s	732	1,008
3	292/296	10.535 s	68.504 s	296	408
4	715/724	26.150 s	197.690 s	724	1,048
Total	3,767/3,969	141.394 s	990.059 s	3,969	5,590



```
#line 28
    tmp___0 ++;
  }
  while_break: /* CIL Label */ ;
  }
  {
#line 30
  memcpy((void */* __restrict */)data,
         (void const * /* __restrict */)
         (data_src), 10U * sizeof(int ));
#line 31
  printIntLine(*(data + 0));
 }
#line 33
  return;
}
}
```

(1) CIL transformation of source code

```
#line 30
  __cil_tmp8 =mysizeof_getspace
             ("@CWE121_Stack_Based_Buffer_
               Overflow__CWE131_memcpy_01_
               bad", "data");
#line 30
  assert(__cil_tmp8 >= 10U * sizeof(int )
         && 10U * sizeof(int ) >= 0);
#line 30
  memcpy((void */* __restrict */)data,
         (void const * /* __restrict */)
         (data_src), 10U * sizeof(int ));
#line 31
  printIntLine(*(data + 0));
 }
#line 33
  return;
}
}
```

(2) Automated Instrumentation

```
Calling good()...
in mysizeof_search, p1->space = 40
In propagate, size_new = 40
0
Finished good()
Calling bad()...
in mysizeof_search, p1->space = 10
In propagate, size_new = 10
CWE121_Stack_Based_Buffer_Overflow__CWE131_
memcpy_01.c.i.our_CIL.header: ./../classified
_merged/CWE131_memcpy/CWE121_Stack_Based_
Buffer_Overflow__CWE131_memcpy_01.c:30:
CWE121_Stack_Based_Buffer_Overflow__CWE131_
memcpy_01_bad: Assertion `__cil_tmp8 >= 10U
* sizeof(int ) && 10U * sizeof(int ) >= 0'
failed.
Aborted (core dumped)
Iteration 1 (0s, 0.0s): covered 12 branches
[4 reach funs, 20 reach branches].(12, 0)
```

(3) Vulnerability Verification

Fig. 5. Snapshot results of different phases

the assertion or not. By using this mechanism, our approach can detect and verify the vulnerabilities in the Juliet Test Suite in a fully automatic way.

The Comparative Detection Results. We show the number of false positives regarding to each type of sink in Table 4. As we can see, when we apply our system with MSA, we get no false positives while there are some false positives(233 in total) when MSA is not applied. The memory space analysis technique finds the real size of the buffer space at runtime and accurately set the condition for violating security constraint(bug triggering condition). When the MSA is not applied, the real runtime memory size of a pointer in the violation condition can only be set by approximation, which results in false positives. MSA can help our system completely reduce false positives.

We also compare our system with Flawfinder [9] which is a source code auditing tool widely used in security communities. We measure the precision ($\frac{TP}{TP+FP}$), recall($\frac{TP}{TP+FN}$) and F1_Value($\frac{2P*R}{P+R}$) for each tool: (1) our system with MSA; (2) our system without MSA; (3) Flawfinder. As shown in Fig. 6, our system applied with MSA gets the highest *Precision* of 100 % which means 0 false positives. In terms of *Recall*, Our system with MSA gets *Recall* of 94.91 %. Flawfinder has the highest *Recall* value, however, its *Precision* is quite low. *F1_value* is a more comprehensive indicator for evaluation, our system with MSA gets the highest *F1_value* of 97.43 %. For the false negatives, our tool failed to correctly instrument the source code when handling the cases involving function pointers and complex use of pointers such as pointer's pointer(total 181 cases). This makes the

Table 4. False positives with or w/o MSA

Sink type	# of FP with MSA	# of FP without MSA
1	0	102
2	0	58
3	0	25
4	0	48
Total	0	233

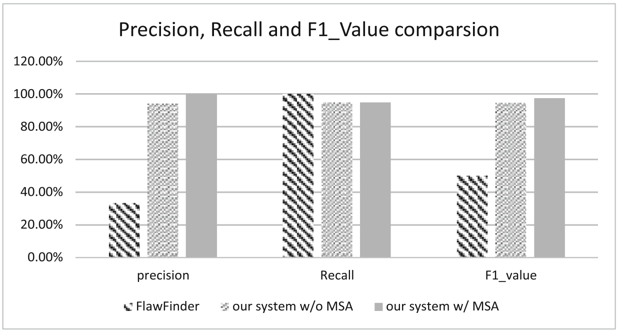
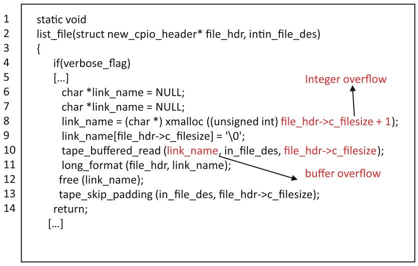


Fig. 6. Detection performance and comparison

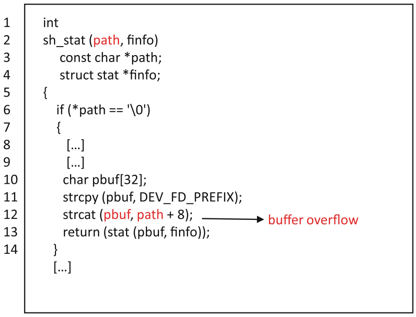
tool cannot trigger the failure of the assertion when a real vulnerability exists, which contributes to false negatives.

Case1: Cpio-2.6(CVE-2014-9112). We also demonstrate the effectiveness of our system by real open source projects. Figure 7 shows a vulnerability in program Cpio-2.6 which is a program to manage archives of files. This vulnerability is caused by an integer overflow induced buffer overflow. at line 8, the numeric



```
1   static void
2   list_file(struct new_cpio_header* file_hdr, intin_file_des)
3   {
4       if(verbose_flag)
5       [...]
6          char *link_name = NULL;
7          char *link_name = NULL;
8          link_name = (char *) xmalloc ((unsigned int) file_hdr->c_filesize + 1);
9          link_name[file_hdr->c_filesize] = '\0';
10         tape_buffered_read (link_name, in_file_des, file_hdr->c_filesize);
11         long_format (file_hdr, link_name);
12         free (link_name);
13        tape_skip_padding (in_file_des, file_hdr->c_filesize);
14      return;
      [...]
```

Fig. 7. CVE-2014-9112 vulnerability from Cpio-2.6



```
1    int
2    sh_stat (path, finfo)
3        const char *path;
4        struct stat *finfo;
5    {
6       if (*path == '\0')
7       {
8          [...]
9          [...]
10        char pbuf[32];
11        strcpy (pbuf, DEV_FD_PREFIX);
12        strcat (pbuf, path + 8);
13        return (stat (pbuf, finfo));
14      }
       [...]
```

Fig. 8. CVE-2012-3410 vulnerability from Bash-4.2

operation can cause an integer overflow and results in 0 bytes allocation for "link_name". The buffer overflow is at line 10 when the program is trying to write "c_filesize" number of bytes to 0 space buffer. We apply our system to automatically report out this vulnerability in a fully automatic way by generating an input which makes "$filesize\_c = 0xfffffff$".

Case2: Bash-4.2(CVE-2012-3410). We also apply our system to Bash-4.2 and successfully verifies the vulnerability in Fig. 8. Our system identifies the sink "strcat", backwardly tracing the user input and set the violating condition of security constraint by automatic instrumentation. The system reports out this vulnerability with a triggering input "$path = /dev/fd/aaaa...aa(35'a's)$".

5 Conclusion

In this paper, we propose, design and implement an automated verification mechanism for vulnerability discovery. Different from other source code auditing methods, our mechanism needs no human interference with extremely low false positive rate and it can expand the system usability to non-security specialist. It takes source code as input, detects and verifies the existence of vulnerability in a fully automatic way. What's more, the memory space analysis(MSA) enables us to set violating condition of security requirements accurately so as to report vulnerabilities with higher accuracy. We developed a prototype system and conducted several experiments with Juliet test cases and also real open source projects. The results show that our system can detect and verify vulnerabilities with low false positives within reasonable time.

However, there are concerns and limitations as well. To the current stage, our system focuses on buffer overflow vulnerability. In future research, we will study the features of other kinds of vulnerability and expand the vulnerability type coverage. Moreover, due to the incapability of handling complex data types such as nested structures in C code, function pointers and pointer's pointer, the system is limited to be working on programs with relatively small amount of source code. The source code analysis and automatic instrumentation will be further generalized to cover large programs.

References

1. MITRE group.: Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/
2. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for C. In: ACM International Symposium on Foundations of Software Engineering, pp. 263–272 (2005)
3. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
4. Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Acar, Y.: Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In: Proceedings of the 22nd ACM CCS, pp. 426–437 (2015)

5. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K.: Automatic inference of search patterns for taint-style vulnerabilities. In: IEEE Symposium of Security and Privacy, pp. 797–812 (2015)
6. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: IEEE Symposium of Security and Privacy, pp. 590–604 (2014)
7. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K.: Selective context-sensitivity guided by impact pre-analysis. ACM SIGPLAN Not. **49**(6), 475–484 (2014)
8. Li, H., Kwon, H., Kwon, J., Lee, H.: CLORIFI: software vulnerability discovery using code clone verification. Pract. Experience Concurrency Comput. **28**(6), 1900–1917 (2015)
9. Wheeler, D.: Flawfinder (2011). http://www.dwheeler.com/flawfinder
10. Kim, M., Kim, Y., Jang, Y.: Industrial application of concolic testing on embedded software: Case studies. In: IEEE International Conference on Software Testing, Verification and Validation, pp. 390–399 (2012)
11. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symposium on Operating Systems Design and Implementation, vol. 8, pp. 209–224 (2008)
12. Zhang, D., Liu, D., Lei, Y., Kung, D., Csallner, C., Wang, W.: Detecting vulnerabilities in C programs using trace-based testing. In: IEEE/IFIP International Conference on Dependable Systems and Networks, pp. 241–250 (2010)
13. Di Paola, S.: Sinks: Dom Xss Test Cases Wiki Project. http://code.google.com/p/domxsswiki/wiki/Sinks
14. Boland, T., Black, P.E.: Juliet 1.1 C/C++ and Java test suite. J. Comput. **45**(10), 89–90 (2012)
15. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: Exposing missing checks in source code for vulnerability discovery. In: ACM CCS, pp. 499–510 (2013)