**Automated Source Code Instrumentation for Verifying Potential Vulnerabilities**Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee(B)  
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 *P recision* = 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  
  
**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 *P recision* = 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(*SC*). The memory space analysis(MSA) enables us to track the size of buffer space at runtime and set the *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 214 H. Li et al. 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  
S  
S  
F  
a=b  
Sink(a)  
Source:  
c= input  
Entry function F  
b=c  
Ancestor:  
**+ assert();  
+ sym\_input();  
+ mysizeof();  
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  
216 H. Li et al. 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 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 interprocedure 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 218 H. Li et al.  
– *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 ∧ 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 3good practical impact, such as KLEE [11] and  
**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 racing 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 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 ≥ n*)*∧*(*n ≥* 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 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.   
**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 |

**Fig. 5.** Snapshot results of different phases  
**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 ( *T PT P* +*F P* ), recall( *T PT P* +*F N* ) and F1 Value( 2*PP*+*∗RR*) 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 *P recision* 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 *P recision* is quite low. *F* 1 *value* is a more  
comprehensive indicator for evaluation, our system with MSA gets the highest *F* 1 *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  
224 H. Li et al. **Table 4.** False positives with or w/o MSA tool cannot trigger the failure of the assertion when a real vulnerability exists, which contributes to false negatives.

|  |  |  |
| --- | --- | --- |
| 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  
**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 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* = 0*xfffffff*”.  
**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*as*)”.  
**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)  
226 H. Li et al.  
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)

**Instrumentasi Source code Otomatis untuk Memeriksa Potensi Kerentanan** Hongzhe Li, Jaesang Oh, Hakjoo Oh, dan Heejo Lee (B)   
Departemen Ilmu Komputer dan Teknik, Universitas Korea,   
Seoul, Korea Selatan   
*{Hongzhe,* jaesangoh, hakjoo oh, *heejo}* @ korea.ac.kr   
**Abstrak.** Dengan tingkat pertumbuhan tahunan yang pesat, kerentanan perangkat lunak membuat ancaman besar terhadap keamanan sistem. Secara teori, mendeteksi dan menghapus kerentanan sebelum kode dikerahkan dapat sangat memastikan kualitas perangkat lunak dilepaskan. Namun, karena banyaknya kode yang dikembangkan serta kurangnya sumber daya manusia dan keahlian, kerentanan berat tetap tersembunyi atau tidak dapat diungkapkan secara efektif. Perangkat audit source code saat ini untuk penemuan kerentanan menghasilkan terlalu banyak kesalahan positif atau memerlukan upaya manual yang berlebihan untuk melaporkan kekurangan perangkat lunak yang sebenarnya. Dalam makalah ini, kami mengusulkan mekanisme verifikasi otomatis untuk menemukan dan memverifikasi kerentanan dengan menggunakan instrumentasi sumber program dan pengujian konsentrat. Pada awalnya, kami memanfaatkan CIL untuk menganalisis secara statik source code termasuk mengekstraksi program CFG, menemukan sink keamanan dan melacak belakang variabel sensitif. Selanjutnya, kami melakukan instrumentasi program otomatis untuk memasukkan probe keamanan yang siap untuk verifikasi kerentanan. Akhirnya, sumber program yang diinstruksikan dilewatkan ke mesin uji concolic untuk memverifikasi dan melaporkan adanya kerentanan yang sebenarnya. Kami menunjukkan efikasi dan efisiensi mekanisme kami dengan menerapkan sistem prototipe dan melakukan eksperimen dengan hampir 4.000 uji kasus dari Juliet Test Suite. Hasilnya menunjukkan bahwa sistem kami dapat memverifikasi lebih dari 90% kasus uji dan melaporkan buffer   
kelemahan meluap dengan *P recision* = 100% (0 FP) dan *Recall* = *94.* 91%. Untuk membuktikan kemampuan praktis sistem kami dalam program dunia nyata, kami juga menerapkan sistem kami pada 2 utilitas Linux yang populer, Bash dan Cpio. Akibatnya, sistem kami menemukan dan memverifikasi kerentanan secara otomatis tanpa kesalahan positif.   
**Kata kunci:** instrumentasi otomatis *·* Keamanan sinks *·* Keamanan   
  
**1. Perkenalan** Meskipun para ahli keamanan melakukan upaya terbaik untuk memastikan keamanan perangkat lunak, jumlah kerentanan perangkat lunak masih meningkat dengan cepat setiap tahunnya, sehingga menimbulkan ancaman besar terhadap keamanan sistem perangkat lunak. Menurut Common kerentanan dan Eksposur (CVE) database [1], jumlah entri CVE telah meningkat dari sekitar 1000 CVEs tahunan pada tahun 2000 menjadi lebih dari 8000 tahunan pada tahun 2015. Penemuan dan penghapusan kerentanan dari proyek-proyek perangkat lunak telah menjadi isu kritis Dalam keamanan komputer Saat ini, karena   
Sejumlah besar kode dikembangkan dan juga sumber daya tenaga kerja yang terbatas, menjadi semakin sulit untuk mengaudit seluruh kode dan secara akurat mengatasi kerentanan target.   
Peneliti keamanan telah mengabdikan diri dalam mengembangkan alat analisis statis untuk menemukan kerentanan [9]. Cakupan besar kode dan akses ke struktur internal membuat pendekatan ini sangat efisien untuk menemukan potensi peringatan kerentanan. Namun, mereka sering memperkirakan atau bahkan mengabaikan kondisi runtime, yang membuat mereka memiliki banyak kesalahan positif.   
Baru-baru ini, lebih maju metode analisis statis diusulkan [4, 5, 15]. Mereka juga mengkodekan sifat pengkodean yang tidak aman seperti pemeriksaan yang hilang, variabel yang tidak disterilkan dan kondisi yang tidak benar ke dalam penganalisis untuk penemuan kerentanan, atau mereka memodelkan sifat kerentanan yang diketahui dan menghasilkan pola pencarian untuk mendeteksi kerentanan yang tidak diketahui. Meskipun pendekatan ini dapat menemukan kerentanan menggunakan pola penelusuran dan mengecualikan sebagian besar kode yang perlu diperiksa, namun masih memerlukan upaya manual khusus keamanan untuk memverifikasi kerentanan pada akhirnya, yang tidak efisien untuk penemuan kerentanan atau kemungkinan untuk non- Pakar keamanan menggunakannya.   
Menurut upaya para peneliti sebelumnya, menemukan kerentanan yang tepat secara otomatis sepenuhnya menantang. Untuk mengotomatisasi keseluruhan proses deteksi dan verifikasi kerentanan, kami mengklasifikasikan sink keamanan potensial yang rentan ke dalam 4 tipe dasar dan menerapkan aturan kendala keamanan (sesuai dengan masing-masing jenis) terhadap kerentanan instrumen secara otomatis yang memicu probe ke source code untuk memverifikasi kerentanan. Dalam makalah ini, kami mengusulkan sebuah mekanisme otomatis untuk mendeteksi dan memverifikasi kerentanan perangkat lunak dari kode C dengan menggunakan instrumentasi sumber program dan pengujian concolic. Pada awalnya, kami memanfaatkan CIL [3] (C intermediate language) untuk statis menganalisis source code termasuk penggalian program CFG (grafik kontrol aliran), menemukan   
Keamanan sensitif sinks dan mundur menelusuri variabel sensitif. Selanjutnya, kami mengklasifikasikan wastafel keamanan menjadi 4 tipe dasar dan melakukan instrumentasi program otomatis untuk memasukkan probe keamanan sesuai dengan sifat yang berbeda dari jenis wastafel keamanan, yang siap untuk verifikasi kerentanan. Akhirnya, sumber program diinstrumentasi diteruskan ke concolic (beton + simbolik) mesin pengujian [10, 11] melaporkan dan memverifikasi keberadaan kerentanan yang sebenarnya. Kami di sini fokus pada kerentanan buffer overflow karena jenis kerentanan ini adalah penyebab utama terjadinya intensitas berbahaya seperti akses memori yang tidak valid, penolakan layanan (crach system) dan eksekusi kode yang sewenang-wenang. Kami menunjukkan efikasi dan efisiensi mekanisme kami dengan menerapkan sistem prototipe dan melakukan eksperimen dengan 4000 kasus uji luapan luapan dari Juliet   
Test Suite [14]. Hasil penelitian menunjukkan bahwa sistem prototipe kami mendapat hasil deteksi dengan *P recision* = 100% dan *Recall* = *94.* 91%. Untuk membuktikan kemampuan praktis sistem kami dalam program dunia nyata, kami juga menerapkan mekanisme kami pada utilitas Linux seperti Bash dan Cpio. Akibatnya, sistem kami menemukan dan memverifikasi kerentanan secara otomatis tanpa kesalahan positif. Kontribusi utama penelitian kami adalah sebagai berikut:   
- **Sepenuhnya Otomatis Verifikasi Kerentanan Penemuan.** Kami mengusulkan, merancang dan menerapkan mekanisme verifikasi otomatis untuk mendeteksi dan memverifikasi kerentanan tanpa gangguan upaya manual, yang dapat memperluas kegunaan sistem kepada pengguna yang lebih umum seperti spesialis non-keamanan.   
- **Memory Analisis Ruang (MSA) untuk Persyaratan Verifikasi Keamanan.**   
Kami memverifikasi keberadaan kerentanan dengan menghasilkan memicu input yang melanggar kendala keamanan *(SC).* Analisis ruang memori (MSA) memungkinkan kita untuk melacak ukuran ruang buffer pada saat runtime dan mengatur kondisi *SC* akurat. Ini mengurangi false positive untuk verifikasi kerentanan.   
**2 Pekerjaan Terkait** Audit source code telah diteliti secara aktif oleh periset keamanan untuk mengetahui kepastian perangkat lunak dan temuan bug. Peneliti sebelumnya telah mengusulkan pendekatan yang berbeda untuk pengkodean source code statis dan telah mengembangkan alat analisis sumber kode statis untuk penemuan kerentanan.   
Flawfinder [9] menerapkan teknik pencocokan pola untuk mencocokkan sinks keamanan dalam source code dan melaporkannya sebagai kerentanan. Meskipun pendekatan ini dapat menganalisis sejumlah besar source code dan melaporkan kerentanan   
Dengan cepat, mereka menghasilkan terlalu banyak false positive karena kurangnya analisis tentang aliran data program dan arus informasi kontrol.   
Chucky [15] statis taints source code dan mengidentifikasi kondisi yang hilang terkait dengan keamanan sinks untuk mengekspos cek hilang dalam source code untuk penemuan kerentanan. VCCFinder [4] mengusulkan metode untuk menemukan kode berpotensi berbahaya dengan rendah tingkat positif palsu menggunakan kombinasi analisis kode-metrik dan meta-data dari repositori kode. Ini juga melatih pengklasifikasi SVM oleh kerentanan melakukan database untuk menandai kode sebagai rentan atau node. Yamaguchi dkk. [5] model sifat kerentanan dikenal dan menghasilkan pola pencarian untuk kerentanan noda-gaya. Pola pencarian yang dihasilkan kemudian diwakili oleh traversals grafik yang digunakan untuk pertambangan kerentanan dalam database grafik properti kode [6]. Meskipun pendekatan ini dapat menemukan kerentanan menggunakan pola penelusuran dan mengecualikan sebagian besar kode yang perlu diperiksa, namun masih memerlukan upaya manual khusus keamanan untuk memverifikasi kerentanan pada akhirnya, yang tidak efisien untuk penemuan kerentanan dan juga tidak memungkinkan pengguna umum (Nonspecialis) untuk menggunakannya. Didorong oleh ini, ada kebutuhan mendesak untuk membangun 214 H. Li et al. Sistem otomatis untuk menangkap kerentanan secara akurat dalam waktu yang wajar serta perluasan kegunaan kepada pengguna yang lebih umum.   
Berdasarkan kelemahan pembahasan di atas, kami mencari cara otomatis dan efisien untuk melakukan verifikasi kerentanan. Eksekusi simbolik telah diusulkan untuk melakukan verifikasi jalur Program tetapi tidak dapat menyelesaikan program kompleks dengan sejumlah besar kendala path [12]. Pengujian Concolic [10, 11] diusulkan untuk meningkatkan pelaksanaan simbolis dalam rangka untuk membuatnya lebih praktis dalam program dunia nyata. Klee [11] dikembangkan untuk secara otomatis menghasilkan tinggi cakupan uji kasus dan untuk menemukan bug dalam dan kerentanan keamanan dalam berbagai kode yang kompleks. CREST-BV [10] telah menunjukkan kinerja yang lebih baik daripada Klee dalam cakupan cabang dan kecepatan uji kasus generasi. Namun, pendekatan ini menderita masalah ledakan jalur yang menghentikannya dari penskalaan   
Untuk program besar.   
CLORIFI [8] adalah penelitian paling dekat dengan makalah ini. Ini mengusulkan metode untuk mendeteksi kerentanan kloning kode dengan kombinasi analisis statis dan dinamis. Namun, hal itu belum sepenuhnya otomatis dan masih memerlukan upaya manual untuk melakukan instrumentasi sumber untuk pengujian concolic, yang masih merupakan tugas yang membosankan.   
Dalam makalah ini, kami mengusulkan, merancang dan menerapkan mekanisme verifikasi otomatis untuk mendeteksi dan memverifikasi kerentanan. Dalam mekanisme kami, kami melakukan verifikasi kerentanan menggunakan pengujian concolic setelah deteksi sink otomatis dan proses instrumentasi source code, yang mengurangi false positive. Kami juga menerapkan analisis ruang memori buffer runtime (MSA) untuk melacak ukuran sebenarnya dari ruang penyangga yang membantu kita memperbaiki keakuratan penemuan kerentanan.   
**3 Mekanisme yang Diusulkan** Penemuan kerentanan dalam sebuah program merupakan proses kunci bagi pengembangan sistem keamanan. Untuk menemukan kerentanan yang tepat dengan cara yang cepat dan akurat dan untuk mengurangi upaya manual yang membosankan untuk verifikasi kerentanan, kami mengusulkan mekanisme otomatis untuk mendeteksi dan memverifikasi kerentanan perangkat lunak dengan memanfaatkan analisis statis dan pengujian konsolidatif.   
Sebelum kita membahas secara rinci pendekatan kita, proses umum diilustrasikan pada Gambar. 1. Mekanisme kami terutama terdiri dari 3 fase yang **transformasi kode,** **instrumentasi otomatis,** dan **verifikasi kerentanan.** Pada fase transformasi kode, pertama-tama kita memanfaatkan library CIL untuk mengurai source code ke dalam struktur program CIL seperti definisi fungsi, variabel, pernyataan, ekspresi dan sebagainya, dan menghitung informasi flow flow control (CFG) dari masing-masing fungsi. . Alasan mengapa kita melakukan transformasi kode CIL adalah menyederhanakan struktur kode untuk analisis statik yang efisien. Kami kemudian mengidentifikasi petugas keamanan (rentan potensial) untuk mendapatkan poin rentan potensial. Pada tahap kedua, kita menerapkan penelusuran data ke belakang pada variabel sensitif masing-masing sink untuk menemukan lokasi input variabel. Kemudian, kami melakukan instrumentasi program otomatis dan menyiapkan objek uji untuk verifikasi kerentanan. Pada tahap terakhir instrumentasi otomatis, kami memverifikasi setiap wastafel keamanan potensial untuk melaporkan kerentanan menggunakan pengujian concolic.

(1) Transformasi kode (2) Instrumentasi otomatis (3) Verifikasi kerentanan   
S   
S   
F   
A = b   
Sink (a)   
Sumber:   
C = masukan   
Fungsi masuk F   
B = c   
Leluhur:   
**+ Menegaskan ();**  **+ Sym\_input ();**  **+ Mysizeof ();**  **Ara.** **1.** Gambaran umum dari pendekatan kami.   
**3.1 Menggunakan CIL** CIL (C Intermediate Language) [3] adalah representasi tingkat tinggi bersama dengan satu set alat yang memungkinkan analisis mudah dan transformasi sumber-ke-sumber program C. Alasan mengapa kita menggunakan CIL adalah mengkompilasi semua program C yang valid menjadi beberapa konstruksi inti tanpa mengubah kode asli semantik. Juga CIL memiliki sistem tipe yang sesuai dengan sintaksis yang memudahkan analisis dan manipulasi program C. Selain itu, CIL menyimpan informasi lokasi kode yang memungkinkan kita menentukan lokasi yang tepat saat melaporkan kerentanan (contohnya   
Ditunjukkan pada Gambar. 5).   
Dengan menggunakan API CIL yang tersedia, kita dapat meratakan struktur kode kompleks menjadi struktur yang sederhana (misalnya, semua konstruksi perulangan dikurangi menjadi bentuk tunggal, semua badan fungsi diberi pernyataan pengembalian eksplisit). Selanjutnya, kita mengekstrak dan menghitung grafik aliran kontrol (CFG) dari masing-masing fungsi menggunakan beberapa modul terbungkus CIL. Setiap node dibuat dalam CFG sesuai dengan instruksi tunggal dalam source code, dirujuk dari Sparrow [7], kerangka analisis source code yang stabil dan suara. Mengobati unit kode yang paling dasar (level instruksi) sebagai simpul CFG dapat membantu kita secara tepat dan cepat mengatasi sink dan variabel yang sensitif serta memberikan kemudahan untuk melacak data sensitif ke belakang yang akan dijelaskan secara rinci pada bagian berikut.   
**3.2 Identifikasi Sinks Keamanan dan Variabel Sensitif** Karena sebagian besar kerentanan buffer overflow disebabkan oleh penyerang dikendalikan data yang jatuh ke wastafel keamanan [13], sangat penting untuk pertama menggali keamanan sinks.   
Dalam tulisan ini, kami fokus pada security sink yang sering menyebabkan kerentanan buffer overflow. Sebelum itu, kami menjelaskan definisi sink keamanan dan bagaimana mereka dapat diklasifikasikan menurut properti argumen.   
**Keamanan Sinks:** Sinks dimaksudkan untuk menjadi poin dalam aliran dimana data tergantung dari sumber digunakan dengan cara yang berpotensi berbahaya. Fungsi peka keamanan yang khas dan operasi akses memori adalah contoh bak cuci keamanan. Beberapa jenis khas sinks keamanan ditunjukkan pada Tabel 1.   
Seperti yang bisa kita lihat dari Tabel 1, pada dasarnya, keamanan sinks adalah baik keamanan fungsi sensitif atau tugas operasi memori penyangga dengan indeks. Lebih jauh lagi, sesuai dengan jumlah argumen dan apakah ada formatnya   
216 H. Li dkk. String ("% s") argumen di dalam fungsi sensitif keamanan, kami mengklasifikasikan fungsi sensitif keamanan ke dalam 3 jenis untuk menggeneralisasi proses otomatis penelusuran mundur dan instrumentasi program yang akan dijelaskan di bagian berikut. Seiring dengan jenis terakhir (buffer assignment), kita mengklasifikasikan wastafel keamanan kerentanan buffer overflow menjadi 4 tipe.   
**Tabel** klasifikasi **1.** Sink dan lokasi variabel

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tipe wastafel | Deskripsi | Format argumen | Fungsi sensitif | Posisi variabel |
| Tipe 1 | Berfungsi dengan dua  Argumen | Fn (dst, src) | Strcpy, wcscpy strcat,  Wcscat | 2 |
| Tipe 2 | Berfungsi dengan tiga  Argumen | Fn (dst, src, n) | Memcpy, memmove,  Strncpy strncat,  Wcsncpy, wcsncat | 2,3 |
| Tipe 3 | Fungsi dengan format  Senar | Fn (dst, n, "% s", src) | Snprintf, swprintf | 2,4 |
| Tipe 4 | Operasi memori | Buffer [i] = expr | Dstbuf [i] = 'A' | Indeks: i |

- Tipe 1: Keamanan fungsi sensitif dengan 2 argumen: tujuan penyangga pointer *(dst)* dan pointer sumber penyangga *(src).* Format argumen khas: *fn (dst, src)* dan variabel sensitif adalah 2 variabel *(src)* dalam daftar argumen. The contoh sinks meliputi: *strcpy, wcscpy, strcat dan wcscat.*   
- Tipe 2: Keamanan fungsi sensitif dengan 3 argumen: tujuan penyangga pointer *(dst),* sumber penyangga pointer *(src)* dan sejumlah byte bilangan bulat *(n).*   
Format argumen khas *fn (dst, src, n)* dan variabel sensitif adalah 2 nd variabel *(src)* dan 3 argumen *(n)* dalam daftar argumen. The contoh sinks meliputi: *memcpy, memmove, strncpy, strncat, wcsncpy dan wcsncat.*   
- Tipe 3: Keamanan fungsi sensitif dengan argumen format string: tujuan penyangga pointer *(dst),* sejumlah byte bilangan bulat *(n),* argumen format string dan pointer sumber penyangga *(src).* Format argumen khas *fn (dst, n, format, src)* dan variabel sensitif adalah 2 variabel *(n)* dan argumen-4 *(src).* The contoh sinks meliputi: *snprintf* dan *swprintf.*

- Tipe 4: Buffer ditugaskan oleh beberapa nilai menggunakan indeks buffer array. Hal ini menyebabkan buffer overrun saat indeks berada di luar batas ukuran buffer. Format khas adalah: *penyangga [index] = expr* dan variabel sensitif *indeks.*   
Sebuah contoh dari jenis wastafel adalah: *dstbuf [i] = 'A'.*   
Setelah klasifikasi sinks keamanan, kami mengidentifikasi sink keamanan sebagai kerentanan potensial dengan menggunakan pendekatan pencocokan pola cepat di atas struktur CIL dari source code dan mengekstrak variabel sensitif yang harus dilacak ke belakang pada langkah berikut berdasarkan tabel di atas.   
**3.3 Mundur Menelusuri Data** Karena titik instrumentasi diperlukan sebelum melakukan instrumentasi otomatis, kami mengusulkan penelusuran data ke belakang untuk menemukan masukan program dan memperlakukannya sebagai titik instrumentasi. Penelusuran data ke belakang menemukan lokasi masukan program untuk variabel sensitif yang sesuai di wastafel yang mengurangi keseluruhan ruang pencarian masukan sebuah program. Ini akan membantu kami meningkatkan efisiensi verifikasi kerentanan. Kami melakukan penelusuran mundur intra dan inter-procedure berdasarkan simpul grafik aliran kontrol yang diambil dari CIL. Gambar 2 menunjukkan konsep intra-prosedur dan interprocedure mundur melacak masing-masing.   
**Konsep dan Definisi.** Seperti ditunjukkan pada Gambar. 2 (1), mulai dari wastafel keamanan, kita mulai backwardly melacak variabel sensitif yang sesuai sampai kita mencapai sumber data sensitif. Untuk memahami proses   
Jejak ke belakang, ada beberapa istilah yang perlu kita ketahui.   
**Sumber:** *Sumber* adalah titik definisi asli dari variabel. Ini adalah simpul dimana nilai variabel tidak bergantung pada variabel lainnya. Misalnya, titik awal di mana data masukan yang tidak dipercaya diambil oleh sebuah program. *Sumber* adalah salah satu yang berikut 2 kasus.   
- *v* 0 = *mendapat* (); Penugasan node dimana variabel kiri diberikan oleh fungsi input pengguna seperti gets (), scanf () dan read (). Kami juga memelihara daftar fungsi masukan pengguna.   
- *v* 0 = 5; Simpul tugas dimana variabel kiri diberikan dengan konstanta.   
**Leluhur:** Nenek moyang *A* dari simpul *N* digambarkan sebagai: data sensitif yang ditelusuri dari *N* mendapat nilai dari *A.* Nenek moyang adalah node intermediate sementara data sensitif diperbanyak. Simpul nenek moyang dari variabel *v* tertentu 0 bisa   
Jadilah salah satu dari 4 kasus di bawah ini:   
- *v* 0 = *ekspresi* *(v* 1); Node dimana variabel ditugaskan oleh ekspresi   
- *v* 0 = *f* *(m, n, ...);* Node dimana variabel ditugaskan oleh nilai kembali fungsi 218 H. Li et al.   
- *f* *(v* 0); Node dimana variabel ditugaskan di dalam pemanggilan fungsi   
- *void f* *(char v* 0); Node untuk deklarasi fungsi   
**Deskripsi Prosedur.** Seperti ditunjukkan pada Gambar. 2, intra-prosedur mundur menelusuri dimulai dari variabel yang sensitif awal di wastafel keamanan seperti *sink* *(x)* pada Gambar. 2 (1) dan secara rekursif menemukan nenek moyang dan mengidentifikasi variabel baru yang dibutuhkan untuk ditelusuri. Intra-prosedur mundur menelusuri berhenti ketika node saat ini adalah *Sumber* variabel sensitif ditelusuri (seluruh tracing mundur juga berhenti) atau berhenti ketika node saat ini adalah fungsi entri simpul.   
Dalam kasus kemudian, *Sumber* tidak dapat ditemukan dalam fungsi saat ini dan aliran data berasal dari argumen lewat dari fungsi panggilan, jadi kita perlu lebih melakukan antar-prosedur mundur tracing untuk menemukan *Sumber* variabel sensitif.   
Antar-prosedur tracing (lihat Gambar. 2 (2)) dimulai dari intra-prosedur tracing.   
Kemudian memeriksa kembali simpul pelacakan intra-prosedur ke belakang. Jika node adalah *Sumber,* kembali prosedur dan ujung melacak mundur. Jika node adalah node entri fungsi, prosedur mengetahui semua situs panggilan dari fungsi saat ini dan mengidentifikasi variabel sensitif yang sesuai di setiap situs panggilan. Kemudian diterapkan penelusuran intra-procedure ke belakang pada variabel sensitif di setiap situs panggilan sebagai titik awal yang baru. Seluruh mundur menelusuri berhenti dan keluar ketika *Sumber* variabel sensitif di wastafel keamanan ditemukan.   
**3.4 Instrumentasi Sumber Program** Setelah tracing mundur, kita mendapatkan sinks keamanan dan *Sumber* yang sesuai wastafel dan menyimpannya ke dalam daftar sinks (daftar sink) dan daftar Sumber (daftar sumber) sesuai. Kami juga membuat pemetaan sumber-sink antara 2 daftar yang membantu kami menemukan sumber yang sesuai untuk wastafel tertentu. Untuk instrumen sumber program, kita membuat probe keamanan (pernyataan) berdasarkan persyaratan keamanan yang telah ditetapkan (Tabel 2) tepat sebelum *wastafel keamanan* dan mengganti pernyataan masukan sumber dengan nilai-nilai simbolik. Untuk mengotomatisasi keseluruhan proses instrumentasi source code, kami menggeneralisasi aturan kendala keamanan untuk 4 tipe dasar kerentanan terhadap pemicu bug instrumen secara otomatis ke source code dengan cara yang lebih umum.   
**Kendala Program (PC) dan Kendala Keamanan (SC):** kendala Program dihasilkan dengan mengikuti cabang tertentu sesuai dengan pernyataan bersyarat dalam program ini. Masukan program yang memenuhi seperangkat kendala program dimaksudkan untuk melaksanakan jalur program tertentu. Kendala keamanan jelas merupakan persyaratan keamanan tingkat tinggi, yang juga digunakan sebagai dasar kami untuk membuat probe keamanan sebelum perangkat keamanan sinks. Misalnya, panjang string yang disalin ke buffer tidak boleh melebihi kapasitas buffer. Kami mendefinisikan persyaratan keamanan untuk bak keamanan seperti fungsi peka keamanan dan penyangga dengan indeks berdasarkan kondisi yang harus dipenuhi oleh argumen terkait untuk menjamin keamanan sistem perangkat lunak. Dalam pekerjaan kami, kami telah mengumpulkan 14 fungsi perpustakaan dari Juliet Test Suite yang dikenal "tidak aman" untuk buffer overflows sebagai fungsi peka keamanan dan menghasilkan persyaratan keamanan untuk mereka. Tabel 2 menunjukkan bagian dari kendala keamanan yang telah ditetapkan untuk sinks keamanan. Bila ada masukan yang memenuhi kendala Program tapi melanggar kendala keamanan *(PC* *∧* *SC)* pada titik tertentu selama eksekusi, program ini dianggap rentan. Untuk menyarankan cara untuk memperluas metode untuk menutupi jenis kerentanan lainnya, kami akan menyelidiki fitur kerentanan dan menentukan lebih banyak sink dan persyaratan keamanan yang sesuai untuk jenis kerentanan seperti overflow integer, string format dan   
Membagi-by-nol   
**Tabel 2.** Persyaratan Keamanan untuk sinks keamanan

|  |  |  |
| --- | --- | --- |
| Wastafel keamanan | Persyaratan keamanan | Deskripsi |
| Strcpy (dst, src) | *dst.space>* src.strlen | Ruang dst harus lebih besar  Dari pada panjang string dari src |
| Strncpy (dst, src, n) | (dst.space *≥* n) *∧* (n *≥* 0) | Ruang dst harus lebih besar  Atau sama dengan yang positif  Bilangan bulat n |
| Strcat (dst, src) | *dst.space>* dst.strln + src.strlen | Ruang dst harus lebih besar  Dari total panjang string  Dari dst dan src |
| Getcwd (buf, ukuran) | (buf.space *≥* ukuran) *∧* (ukuran *≥* 0) | Ruang buf harus lebih besar  Atau sama dengan yang positif  Ukuran integer |
| Fgets (buf, size, f) | (ukuran *≥ dst.space)* *∧* (ukuran *≥* 0) | Ruang dst harus lebih besar  Atau sama dengan yang positif  Ukuran integer |
| Buf [i] = expr. | *buf.space>* i | Ruang buf harus lebih besar  Dari nilai indeks i |

**Instrumen Masukan Lokasi dan Sink Keamanan:** Untuk instrumen lokasi input, pertama kita iterate atas semua elemen dalam *daftar sumber,* jika *Sumber* saat mengambil data dari input pengguna seperti baris perintah, jaringan data, atau file, kita ganti Data input pengguna dengan nilai input simbolik. Misalnya, *“a* = *readfile* ();” akan diganti dengan *“a* = *masukan sym* ();”.   
Untuk instrumen wastafel keamanan, kami memasukkan pernyataan penegasan tepat sebelum wastafel keamanan berdasarkan aturan keamanan yang didefinisikan dalam Tabel 2. Misalnya, sebelum wastafel *strcpy* *(dst, src),* kami menyisipkan *menegaskan* pernyataan pernyataan *(dst.space> strlen* *(src)).* Namun, ada masalah disini. Kita bisa dengan mudah mendapatkan panjang string dengan menggunakan *“strlen* ()” fungsi (C fungsi library), tapi ruang penyangga sulit untuk ditentukan pada saat runtime serta pada waktu kompilasi. Seseorang mungkin mengatakan, kita selalu bisa mendapatkan ukuran buffer oleh *“sizeof* ()”.   
Ini tidak benar bila kita mengukur ukuran ruang memori penyangga. Misalnya, jika kita ingin mendapatkan ruang nyata yang poin *dst* untuk, kita tidak bisa menggunakan *“sizeof* *(dst)”* karena akan selalu kembali ukuran sebenarnya dari pointer itu sendiri yang merupakan nomor 4 di arsitektur 32 bit. Untuk mendapatkan ukuran penyangga yang sebenarnya dari buffer pointer, kami mengusulkan sebuah pendekatan analisis pointer untuk mendapatkan ukuran buffer dengan benar pada runtime program.   
**Memori Analisis Ruang (MSA) di Runtime:** Seperti yang kita lihat dari Tabel 2, dalam aturan persyaratan keamanan, kami harus mendapatkan “buffer.space” Nilai untuk secara akurat mengatur bug memicu kondisi dan instrumen sinks. Namun, analisis statis umum source code tidak dapat memperbarui informasi tentang ukuran memori dari pointer penyangga. Hal ini biasanya akan menghasilkan pernyataan yang tidak akurat mengenai kondisi pelanggaran SC, yang membuat sistem menghasilkan kemungkinan kesalahan positif saat melaporkan kerentanan. Analisis ruang memori runtime memungkinkan kita untuk secara akurat melacak ukuran sebenarnya dari ruang buffer pointer dan membantu kita untuk benar instrumen program sehingga untuk memastikan akurasi yang tinggi verifikasi kerentanan. Kami membuat sebuah perpustakaan yang disebut “libmysizeof” dan menyediakan 3 fungsi utama: *toko mysizeof (),* *mysizeof menyebarkan ()* dan *getspace mysizeof ().* Kami memasukkan fungsi *mysizeof* di tempat yang sesuai dalam source code. Langkah-langkahnya ditunjukkan di bawah ini:   
- Iterate atas semua petunjuk dalam source code dan mengidentifikasi penyangga pernyataan alokasi seperti *buf = malloc (n);* dan *Char buf [100];* . Kemudian, kita menyimpan buffer nama pointer dan ukuran yang sesuai dalam peta *M* global dengan memasukkan fungsi *mysizeof toko ()* di lokasi saat ini.   
- Identifikasi setiap pointer propagasi instruksi seperti kita *p 2 = p 1 + 5;* . The *mysizeof propagate ()* akan menyebarkan ukuran *p 1* untuk mendapatkan ukuran *p 2* menurut operasi dan menyimpan ukuran *p 2* ke dalam peta *M.* Kami kemudian menyisipkan   
ini *mysizeof propagate ()* fungsi di lokasi saat ini.   
- Ketika kita perlu untuk mendapatkan ukuran runtime ruang buffer pointer, kita menyisipkan fungsi *mysizeof getspace ()* di lokasi tertentu dalam source code untuk mendapatkan ruang buffer yang benar.   
Setelah memasukkan fungsi "mysizeof", kita bisa mendapatkan ukuran buffer yang akurat pada saat runtime. Informasi ukuran buffer kemudian dapat digunakan dalam asersi. Misalnya, *menegaskan* *(mysizeof getspace* *(dst)> strlen* *(src)).*   
Gambar 3 menunjukkan contoh instrumenting fungsi analisis pointer bersama dengan masukan dan sinks instrumentasi.   
Sampai disini, kami menyiapkan objek sumber uji dari masukan program ke sink potensial yang rentan. Objek ini biasanya merupakan bagian kecil dari keseluruhan sumber program yang membantu kita melepaskan beban tahap selanjutnya.   
**3.5 Verifikasi Kerentanan Menggunakan Pengujian Concolic** Kami menerapkan pengujian concolic dalam mekanisme kami untuk memverifikasi potensi kerentanan. Prinsip umum verifikasi adalah mencari masukan yang memenuhi semua kendala program (PC) namun melanggar batasan keamanan (SCs) seperti yang ditunjukkan pada Gambar. 1. Eksekusi simbolik dan eksekusi concolic telah banyak digunakan dalam pengujian perangkat lunak dan beberapa telah menunjukkan dampak praktis 3good, seperti Klee [11] dan   
**Ara.** **3.** instrumentasi otomatis.   
LUCU [2]. Namun, mereka menderita masalah ledakan jalur yang membuat mereka tidak dapat melakukan skala baik terhadap program dunia nyata yang besar. Dalam skenario kami, modul balap terbelakang membantu kami menemukan masukan program yang terkait dengan data sensitif, bukan keseluruhan ruang masukan program. Hal ini dapat mengurangi masalah jalur ledakan yang disebutkan sebelumnya. Pendekatan kami untuk pengujian concolic untuk memverifikasi potensi kerentanan terutama mengikuti prosedur pengujian concolic umum [10].   
Namun, bedanya adalah kita berfokus pada menghasilkan masukan untuk mengeksekusi cabang yang rentan alih-alih menghasilkan masukan untuk melintasi setiap jalur program yang mungkin. Makanya, lebih hemat biaya saat melakukan uji coba.   
**4 Hasil Eksperimental**  **4.1 Implementasi**  **Sistem Prototype:** Kami telah menerapkan mekanisme kami dengan mengembangkan sistem prototipe. Sistem kami terdiri dari 3 fase: **Kode transformasi,** **instrumentasi otomatis,** dan **verifikasi kerentanan.** Arsitekturnya adalah   
Dijelaskan pada Gambar. 4. Sistem ini digunakan untuk menemukan kerentanan buffer overflow dalam proyek perangkat lunak. 1   
**Pengaturan lingkungan:** Kami melakukan semua percobaan untuk menguji deteksi kerentanan dan verifikasi sistem otomatis kami pada mesin desktop yang menjalankan Linux Ubuntu 14.04 LTS (3,3 GHz Intel Core i7 CPU, memori 8 GB, 512 GB hard drive).   
**Dataset:** Untuk percobaan, kita siap 2 jenis dataset - Juliet Test Suite dan Linux utilitas. Yang pertama adalah Juliet Test Suite yang disediakan oleh US National Security Agency (NSA) yang telah banyak digunakan untuk menguji keefektifannya   
Alat deteksi kerentanan Untuk menguji alat-alat kami, kami siap 3969 file untuk berdasarkan tumpukan kerentanan buffer overflow, yang masing-masing milik 4 tipe dasar berdasarkan klasifikasi wastafel kami pada Tabel 1. Dataset kedua adalah 2 utilitas Linux yang terkenal, Bash dan Cpio, yang digunakan untuk membuktikan kepraktisan sistem kami.   
**4.2 Hasil Eksperimental** Kami telah melakukan eksperimen kami dengan dua jenis dataset. **Juliet Test Suite.** Kami menguji pendekatan kami dengan 3.969 testcases kelemahan Stack Based Buffer Overflow dari Juliet Test Suite. Jumlah sampel yang tercakup dalam Tabel 3 negara jumlah kasus yang kami diverifikasi (lebih dari 90% total).   
Waktu pemrosesan kami untuk 3.969 testcases hampir 20 menit, yang mencakup sekitar 2 menit instrumentasi dan 17 menit pengujian concolic. Tabel 3 menunjukkan jumlah testcases diproses, jumlah baik dan buruk wastafel, dan waktu berlalu instrumentasi dan pengujian concolic untuk setiap jenis. Kami memeriksa sisa kasus yang tidak dapat kami verifikasi dan menemukan bahwa parser frontend kami tidak dapat menangani gaya pengkodean non-standar seperti string lebar L "AAA" (222 kasus). Selain itu, alat kita gagal untuk benar instrumen source code saat menangani fungsi pointer (76 kasus) dan penggunaan yang kompleks pointer seperti pointer pointer kita (105 kasus).   
Contoh kerja ditunjukkan pada Gambar. 5. Sub angka pada Gambar. 5 menunjukkan urutan mekanisme kami. Gambar 5 (1) menunjukkan hasil transformasi CIL dari sebuah file bernama CWE121 Stack Berbasis Buffer Overflow CWE131 memcpy 01.c. Kemudian kita tunjukkan hasil instrumentasi dari kasus ini pada Gambar. 5 (2). Penegasan *menegaskan* *(dst.space* *≥* *n)* *∧* *(n* *≥* 0) secara otomatis dimasukkan sebelum sinks *memcpy.* Gambar 5 (3) menunjukkan hasil pelaksanaan pengujian concolic. Pada langkah ini, sebenarnya memverifikasi kerentanan dengan memeriksa apakah eksekusi program melanggar asersi atau tidak. Dengan menggunakan mekanisme ini, pendekatan kami dapat mendeteksi dan memverifikasi kerentanan dalam Juliet Test Suite dengan cara sepenuhnya otomatis. **Tabel 3.** Jenis kerentanan dan informasi dasar percobaan

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mengetik | Jumlah  sampel Covered | Berlalu Waktu  (Instrumentasi) | Berlalu Waktu  (Concolic Testing) | Jumlah  Sinks buruk | Jumlah  Sinks baik |
| 1 | 2065/2217 | 78,816 s | 547,879 s | 2217 | 3126 |
| 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 | 3767/3969 | 141,394 s | 990,059 s | 3969 | 5590 |

**Ara.** **5.** Hasil Snapshot dari fase yang berbeda **The Perbandingan Deteksi Hasil.** Kami menunjukkan jumlah positif palsu mengenai setiap jenis sinks pada Tabel 4 . Seperti yang kita lihat, ketika kita menerapkan sistem kami dengan MSA, kami tidak mendapatkan positif palsu sementara ada beberapa positif palsu (233 total) ketika MSA tidak diterapkan. Teknik analisis ruang memori menemukan ukuran nyata dari ruang buffer pada saat runtime dan akurat mengatur kondisi karena melanggar kendala keamanan (bug memicu kondisi). Ketika MSA tidak diterapkan, ukuran memori runtime nyata dari sebuah pointer dalam kondisi pelanggaran hanya dapat ditetapkan oleh pendekatan, yang menghasilkan positif palsu. MSA dapat membantu sistem kami benar-benar mengurangi positif palsu. Kami juga membandingkan sistem kami dengan Flawfinder [ 9 ] yang merupakan alat kode audit sumber banyak digunakan dalam komunitas keamanan. Kami mengukur presisi ( *T PT P* + *FP* ), ingat ( *T PT P* + *FN* ) dan F1 Nilai ( 2 *PP* + *\* RR* ) untuk masing-masing alat: (1) sistem kami dengan MSA; (2) sistem kami tanpa MSA; (3) Flawfinder.  
  
  
Seperti ditunjukkan pada Gambar. 6 , sistem kami diterapkan dengan MSA mendapat tertinggi *P recision* dari 100% yang berarti 0 positif palsu. Dalam hal *Ingat* , sistem kami dengan MSA mendapat *Recall* dari 94 *.* 91%. Flawfinder memiliki tertinggi *Ingat* nilai, namun, yang *P recision* cukup rendah. *F* 1 *nilai* adalah lebih   
indikator yang komprehensif untuk evaluasi, sistem kami dengan MSA mendapat tertinggi *F* 1 *nilai* dari 97,43%. Untuk negatif palsu, alat kami gagal untuk benar instrumen source code ketika menangani kasus yang melibatkan pointer fungsi dan penggunaan kompleks pointer seperti pointer pointer ini (Total 181 kasus). Hal ini membuat 224 H. Li et al. **Tabel 4.** Salah positif dengan atau w / o MSA alat tidak dapat memicu kegagalan pernyataan ketika kerentanan nyata ada, yang memberikan kontribusi untuk negatif palsu.

|  |  |  |
| --- | --- | --- |
| jenis wastafel | # Dari FP dengan MSA | # Dari FP tanpa MSA |
| 1 | 0 | 102 |
| 2 | 0 | 58 |
| 3 | 0 | 25 |
| 4 | 0 | 48 |
| Total | 0 | 233 |

**Ara.** **6.** kinerja Deteksi dan perbandingan **Case1: cpio-2.6 (CVE-2014-9112).**

Kami juga menunjukkan efektivitas dari sistem kami oleh proyek-proyek open source yang nyata. Gambar 7 menunjukkan kerentanan dalam program cpio-2.6 yang merupakan program untuk mengelola arsip file. Kerentanan ini disebabkan oleh buffer overflow bilangan bulat melimpah diinduksi. pada baris 8, operasi numerik dapat menyebabkan overflow integer dan hasil dalam alokasi 0 byte untuk “d”. Buffer overflow pada baris 10 ketika program mencoba untuk menulis “c filesize” jumlah byte untuk 0 ruang buffer. Kami menerapkan sistem kami secara otomatis melaporkan keluar kerentanan ini dengan cara yang sepenuhnya otomatis dengan menghasilkan input yang membuat “ *filesize c* = 0 *xfffffff* ”.   
**Case2: Bash-4.2 (CVE-2012-3410).** Kami juga menerapkan sistem kami ke Bash-4.2 dan berhasil memverifikasi kerentanan pada Gambar. 8. Sistem kami mengidentifikasi wastafel “strcat”, backwardly menelusuri input pengguna dan mengatur kondisi melanggar kendala keamanan dengan instrumentasi otomatis. Sistem ini melaporkan keluar kerentanan ini dengan masukan “memicu *path* = */dev/fd/aaaa...aa* (35 *sebagai* )”.  
**5. Kesimpulan** Dalam tulisan ini, kami mengusulkan, desain dan menerapkan mekanisme verifikasi otomatis untuk penemuan kerentanan. Berbeda dari metode source code audit lainnya, mekanisme kita tidak membutuhkan campur tangan manusia dengan sangat rendah tingkat positif palsu dan dapat memperluas kegunaan sistem untuk spesialis non-keamanan.   
Dibutuhkan source code sebagai input, mendeteksi dan memverifikasi keberadaan kerentanan dalam cara yang sepenuhnya otomatis. Terlebih lagi, analisis ruang memori (MSA) memungkinkan kita untuk mengatur kondisi melanggar persyaratan keamanan secara akurat sehingga   
melaporkan kerentanan dengan akurasi yang lebih tinggi. Kami mengembangkan sistem prototipe dan melakukan beberapa percobaan dengan Juliet uji kasus dan juga proyek-proyek open source yang nyata. Hasil penelitian menunjukkan bahwa sistem kami dapat mendeteksi dan memverifikasi kerentanan dengan positif palsu yang rendah dalam waktu yang wajar.   
Namun, ada kekhawatiran dan keterbatasan juga. Untuk tahap saat ini, sistem kami berfokus pada kerentanan buffer overflow. Dalam penelitian masa depan, kita akan mempelajari fitur dari jenis lain dari kerentanan dan memperluas cakupan jenis kerentanan. Selain itu, karena ketidakmampuan menangani tipe data yang kompleks seperti struktur bersarang dalam kode C, pointer fungsi dan pointer pointer, sistem terbatas untuk dapat bekerja pada program dengan jumlah yang relatif kecil dari source code. Analisis source code dan instrumentasi otomatis akan lebih digeneralisasi untuk menutupi program besar.  
**Referensi** 1. Kelompok MITRE .: Common kerentanan dan Eksposur (CVE). https: //cve.mitre.   
org /   
2. Sen, K., Marinov, D., Agha, G .: Lucu: mesin pengujian unit concolic untuk C. Dalam:   
. ACM Simposium Internasional Yayasan Rekayasa Perangkat Lunak, pp 263-   
272 (2005)   
3. Necula, GC, McPeak, S., Rahul, SP, Weimer, W .: CIL: bahasa Menengah   
dan alat untuk analisis dan transformasi program C. Dalam: Nigel Horspool, R.   
(ed.) CC 2002. LNCS, vol. 2304, hlm. 213-228. Springer, Heidelberg (2002)   
4. Perl, H., Dechand, S., Smith, M., Arp, D., Yamaguchi, F., Rieck, K., Acar, Y .:   
Vccfinder: Menemukan kerentanan potensial di terbuka proyek-source untuk membantu kode   
audit. Dalam:. Prosiding 22 ACM CCS, pp 426-437 (2015) 226 H. Li et al. 5. Yamaguchi, F., Maier, A., Gascon, H., Rieck, K .: inferensi otomatis pencarian pola kerentanan noda-gaya. Dalam:. IEEE Simposium Keamanan dan Privasi, pp 797-812 (2015) 6. Yamaguchi, F., Golde, N., Arp, D., Rieck, K .: Modeling dan menemukan kerentanan dengan grafik properti kode. Dalam: IEEE Simposium Keamanan dan Privasi, . Pp 590-604 (2014) 7. Oh, H., Lee, W., Heo, K., Yang, H., Yi, K .: Selektif konteks sensitivitas dipandu oleh dampak pra-analisis. ACM SIGPLAN Tidak. **49** (6), 475-484 (2014) 8. Li, H., Kwon, H., Kwon, J., Lee, H .: CLORIFI: software penemuan kerentanan menggunakan kode verifikasi clone.  
  
  
  
  
  
  
  
  
Praktik. Mengalami Concurrency Comput. **28** (6), 1900-   
1917 (2015)   
9. Wheeler, D .: Flawfinder (2011). http://www.dwheeler.com/flawfinder   
10. Kim, M., Kim, Y., Jang, Y .: aplikasi industri pengujian concolic pada tertanam   
perangkat lunak: Studi kasus. Dalam: IEEE Konferensi Internasional tentang Pengujian Perangkat Lunak,   
. Verifikasi dan Validasi, pp 390-399 (2012)   
11. Cadar, C., Dunbar, D., Engler, D .: Klee: generasi tanpa bantuan dan otomatis   
tes tinggi cakupan untuk sistem yang kompleks program. Dalam: USENIX Simposium  
Desain Sistem operasi dan Implementasi, vol. 8, pp. 209-224 (2008)   
12. Zhang, D., Liu, D., Lei, Y., Kung, D., Csallner, C., Wang, W .: Mendeteksi kerentanan dalam program C menggunakan trace- pengujian berbasis. Dalam: IEEE / IFIP Internasional   
Konferensi diandalkan Sistem dan Jaringan, pp 241-250 (2010).   
13. Di Paola, S .: Sinks: Kasus Uji Dom XSS Wiki Project. http://code.google.com/p/   
domxsswiki / wiki / Sinks   
14. Boland, T., Black, PE: Juliet 1.1 C / C ++ dan Java test suite. J. Comput. **45** (10),   
89-90 (2012)   
15. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K .: Chucky: Mengekspos hilang   
cek di source code untuk penemuan kerentanan. Dalam: CCS ACM, pp 499-510 (2013).

https://www.gstatic.com/images/branding/googlelogo/1x/googlelogo_color_48x16dp.png

**Original English text:**

Symbolic execution has been proposed to do program path verification but it cannot resolve complex programs with enormous amount of path constraints [ 12 ].

http://www.google.com/images/zippy_plus_sm.gifContribute a better translation