CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **35**:1900–1917
Published online 5 June 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3532

SPECIAL ISSUE PAPER

CLORIFI: software vulnerability discovery using code clone verification[‡]

Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon and Heejo Lee[*,†]

Department of Computer Science and Engineering, Korea University, Seoul, Korea

SUMMARY

Software vulnerability has long been considered an important threat to the system safety. A vulnerability is often reproduced because of the frequent code reuse by programmers. Security patches are usually not propagated to all code clones; however, they could be leveraged to discover unknown vulnerabilities. Static code auditing approaches are frequently proposed to scan source codes for security flaws; unfortunately, these approaches generate too many false positives. While dynamic execution analysis methods can precisely report vulnerabilities, they are ineffective in path exploration, which limits them to scale to large programs. With the purpose of detecting vulnerability in a scalable way with more preciseness, in this paper, we propose a novel mechanism, called software vulnerability discovery using Code Clone Verification (CLORIFI), that scalably discovers vulnerabilities in real world programs using code clone verification. In the beginning, we use a fast and scalable syntax-based way to find code clones in program source codes based on released security patches. Subsequently, code clones are being verified using concolic testing to dramatically decrease the false positives. In addition, we mitigate the path explosion problem by backward sensitive data tracing in concolic execution. Experiments have been conducted with real-world open-source projects (recent Linux OS distributions and program packages). As a result, we found 7 real vulnerabilities out of 63 code clones from Ubuntu 14.04 LTS (Canonical, London, UK) and 10 vulnerabilities out of 40 code clones from CentOS 7.0 (The CentOS Project(community contributed)). Furthermore, we confirmed more code clone vulnerabilities in various versions of programs including Rsyslog (Open Source(Original author: Rainer Gerhards)), Apache (Apache Software Foundation, Forest Hill, Maryland, USA) and Firefox (Mozilla Corporation, Mountain View, California, USA). In order to evaluate the effectiveness of vulnerability verification in a systematic way, we also utilized Juliet Test Suite as measurement objects. The results show that CLORIFI achieves 98% accuracy with 0 false positives. Copyright © 2015 John Wiley & Sons, Ltd.

Received 19 December 2014; Revised 20 March 2015; Accepted 14 April 2015

KEY WORDS: software vulnerability; code clone; concolic testing; vulnerability verification

1. INTRODUCTION

Programmers often make code reuses when they develop software. These code reuses are considered to be code clones that refer to the same or similar code fragments in source code files. The behavior of code reuse usually causes the propagation of vulnerabilities when a piece of vulnerable code is reproduced. We call this kind of vulnerability as *code clone vulnerability*.

Security patches are released to fix vulnerabilities. However, the patch of a specific vulnerability often fails to propagate to code clones at other locations or programs, which, very possibly, present latent code clone vulnerability. Once a security patch is released, attackers could leverage patch

*Correspondence to: Heejo Lee, CCS Lab, Department of Computer Science and Engineering, Korea University, Seoul, Korea.
†E-mail: heejo@korea.ac.kr
‡This is an extended version of the ATIS'2014 paper [1].

information to dig out 0-day vulnerabilities and make great damage to the systems. Thus, there is an urgent need to detect them in an effective and efficient way.

For a long time, software testing has been actively studied to detect security vulnerabilities. Static code analysis [2–5] is proposed to discover vulnerabilities by analyzing source codes or binary objects. The large coverage of code and the access to the internal structures make this approach efficient to find potential warnings of vulnerabilities. However, static analysis approximate or even ignore runtime conditions, which makes them suffer from high false positives.

Besides from static analysis, research efforts have been spent on dynamic analyzing approaches as well. Dynamic analysis monitors program execution to discover security flaws [6–8]. These tools detect software vulnerabilities by monitoring the program run-time behavior and generating test cases for program inputs. Although dynamic analysis reduces false alarms, it requires the generation of actual bug-triggering test inputs, which makes it difficult to find critical security flaws in a reasonable time. What is more, high coverage of the huge input space is either too much expensive or impractical to achieve.

Symbolic execution [9] has been widely utilized to generate efficient testing inputs by solving paradigm program constraints of variables. Because of the impractical implementation on real-world programs, concolic testing (CONCrete + symbOLIC) is proposed to enhance the ability of symbolic execution. It involves concrete execution while doing symbolic execution so as to address unsolvable constraints [10–12]. Even though substantial efforts have been made to research about different searching strategies of concolic testing [13] to improve code coverage, program paths grow exponentially as the branches in programs increase. Because of this path explosion problem, single symbolic or concolic execution-based approach is either ineffective in path exploration or do not scale well to large programs.

In order to gain higher preciseness and better scalability in vulnerability discovery, we propose a mechanism, called CLORIFI, which takes advantages of both static and dynamic analysis to discover code clone vulnerability based on released security patches. Different from merely raising potential warnings of vulnerability in static analysis approaches, CLORIFI performs vulnerability verification using concolic testing to reduce false positives. What is more, backward data tracing has been proposed in our mechanism to mitigate the path explosion problem in concolic testing as well. First of all, we detect code clones in target-source code by doing syntax-based pattern matching in a scalable and efficient way. Second, we analyze the security sensitive data in code clones and perform backward input tracing to instrument the program source and prepare the testing object so that we can focus on program inputs that affect the potential vulnerable statement. Finally, code clones are automatically verified to confirm real vulnerabilities using concolic testing. This verification dramatically reduces false positives. In the evaluation, experiments were conducted with real world open source projects such as recent Linux OS distributions and different versions of Linux program packages. In the results, we found 7 real vulnerabilities out of 63 code clones from Ubuntu 14.04 LTS and 10 vulnerabilities out of 40 code clones from CentOS 7.0 . During the experiments, 260K source files from Ubuntu 14.04 and 520K files from CentOS 7.0 were processed within 7 and 9.2 hours, respectively. In one step further, we found more code clones and confirmed more vulnerabilities in different versions of program packages. Meanwhile, in order to evaluate the effectiveness of vulnerability verification in a systematic way, we also utilized Juliet Test Suite [14] as measurement objects. The results present that CLORIFI achieves 98% accuracy with 0 false positive and the average verification speed is 0.24 s.

Our contributions are summarized as follows:

- **Combination of static and dynamic analysis to reduce false positive**. We have developed a novel mechanism called CLORIFI, which combines the advantage of static and dynamic analysis to detect code clone vulnerability. Our mechanism suggests that the code clone vulnerability detection is scalable and with low false positives.
- **Backward sensitive data tracing to mitigate the path explosion problem**. The backward sensitive-data tracing enables our approach perform concolic testing to do verification in a way that mitigates the path explosion problem in conventional concolic execution approaches.

The remainder of this paper is displayed as follows. Section 2 introduces related works and a broad description of our work. We discuss our approach in detail in Section 3. Experimental results and evaluation are presented in Section 4, and we conclude the paper in Section 5.

2. RELATED WORK

Software vulnerability detection approaches mainly fall into three categories: *static analysis*, *dynamic analysis* and *symbolic (concolic) execution*.

2.1. Static analysis

Previous researchers have proposed different approaches for static source code auditing. Some of static analysis approaches focus on detecting code clones [15–17]. Deckard [16] and Deja vu [15] first parse the program to produce an abstract syntax tree(AST) to represent the source program and then use the vector as a fingerprint for ASTs. Similarity comparison is performed among fingerprinting vectors. These approaches require a very robust parser for the programming language, and they are not efficient and scalable enough in real large source-code pools according to Redebug [18]. Even though it can handle subtle code changes that may help them to find more code clones, this approach suffers from high false positive rate.

Redebug [18] tokenizes the source code into *n*-tokens and uses feature hash function to hash *n*-tokens. The code clone detection is performed by membership checking in bloom filter that stores the hash value of *n*-tokens. It is very practical and can scale very well in real-world usage in terms of code clone detection. However, because of a lack of automatic verification mechanism, most of un-patched code clones they reported are turned out not to be real vulnerabilities that gives them a super-high false positive rate in terms of vulnerability detection.

2.2. Dynamic analysis

Dynamic analysis checks program runtime-execution behaviors to detect security vulnerabilities. Purify [19] feeds the input data to a program and examines the execution information at runtime. A security flaw is reported when any abnormal behavior in the execution is detected. Hastings [20] proposed an approach to fuzz the program with large amount of input data considering the target program as a black box. Assumptions are defined and settled up into the program to monitor if any input data drives the program to a state that violates these assumptions. Even though dynamic analysis reduces false positives, exact concrete inputs are required to actually cause the security problems in the program. This places programmers a huge burden for testing.

2.3. Symbolic and concolic execution

Based on the shortness of the aforementioned discussion, we are looking into an automatic and efficient way to do vulnerability verification. Symbolic execution was proposed to do program testing and showed good performance in detecting some vulnerabilities [21]. Concolic testing [10, 12] was proposed later to improve symbolic execution in order to make it more practical in real world programs. KLEE [10] 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 [12] has shown a better performance than KLEE in branch coverage and the speed of test case generation. Nonetheless, the branch coverage rate of CREST-BV was still below 25% with baseline testing strategy and below 70% with the special designed testing strategy [12], which means, for some vulnerabilities, it is either impossible or too much time consuming to report them out.

The CREST [13] is a concolic test-generation tool for programs written in C. It has been widely used either as a basic concolic-testing engine or a benchmark program in previous works [12, 22, 23]. Seo *et al.* [22] introduced a context-guided searching strategy (CGS) in concolic testing that skips the selection of some branches with the same context information. CGS is also constructed based on CREST, and it improved the efficiency of concolic testing, given a certain goal of branch coverage and showed a relatively higher code coverage than conventional searching strategies such as depth-first searching (DFS) and control-flow graph-based searching (CFG) [13]. Li *et al.* [24]

presented a method that drives symbolic execution to less traversed paths. However, they used fixed size of subpath that will make them miss some critical paths. Hybrid concolic testing [25] combines random input testing and concolic testing. It switches to concolic testing when random execution reaches a deep state of program and lingers at a certain coverage plateau. These approaches suffer from path explosion problem when they are trying to generate inputs to cover every branch of the program. Moreover, when detecting software bugs or vulnerabilities, they usually take every normal statement(such as memory copy, buffer access, and arithmetic operations) as a potential bug. This makes the concolic testing very time-wasting and resource-wasting because of a huge input searching space, because only very small portion of the potential bugs turn out to be real security vulnerabilities in real-world programs.

However, CLORIFI focuses on the program input that affects the potential vulnerable spot using backward sensitive data tracing in order to drive the execution to reach to a specific vulnerable branch instead of covering all program branches. In our mechanism, we do vulnerability verification using concolic testing after a scalable process of code clone detection, which reduces false positives. We also propose backward data tracing in CLORIFI to assist concolic testing so as to mitigate the path explosion problem.

3. THE MECHANISM: CLORIFI

Discovery of vulnerabilities is crucial to maintain secure systems. Security patches are released to fix security flaws and vulnerabilities. However, not all the patches are well adopted and applied in all related programs. In a common case, released security patches are often not propagated to all vulnerable programs due to the heavy usage of the same piece of vulnerable code. To make things worse, attackers often find more critical vulnerabilities based on the information learned from released security patches. As security researchers, we had better move ahead of attackers to identify those vulnerabilities related to un-patched code clones. We call these vulnerabilities as *code clone vulnerability*. In this section, we explain CLORIFI: a mechanism that scalably discovers software vulnerability using code clone verification.

Before going into detail of CLORIFI, its general process is illustrated in Figure 1. First, we find code clones by doing static syntax-based pattern matching in a scalable and efficient way. Subsequently, we perform backward data tracing to prepare testing object. At last, we verify the code clones to report real code clone vulnerability using concolic testing in a way that mitigates the path explosion problem in conventional concolic testing domain. The automatic verification helps us dramatically reduce false alarms in terms of vulnerability detection.

3.1. Finding code clones

Code clones are described as follows: if a same piece of vulnerable code occurs in any other locations or programs, we call them as un-patched code clones. Figure 2 shows the concept and possible scenarios of code clones(e.g., CC@SP@S means code clone vulnerability at same program, at same location). In Figure 3, we could see that in some cases, after patch release, the vulnerability may not be patched until several versions later or the same vulnerability reoccurs in the later program versions. This, if leveraged by attackers, may cause serious damages to our systems.

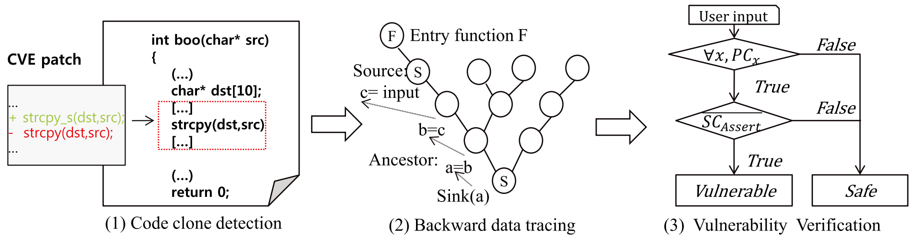

```
...
+ strcpy_s(dst,src);
- strcpy(dst,src);
...
```
```
int boo(char* src)
{
    (...)
    char* dst[10];
    [...]
    strcpy(dst,src)
    [...]

    (...)
    return 0;
```

Figure 1. General overview of CLORIFI.

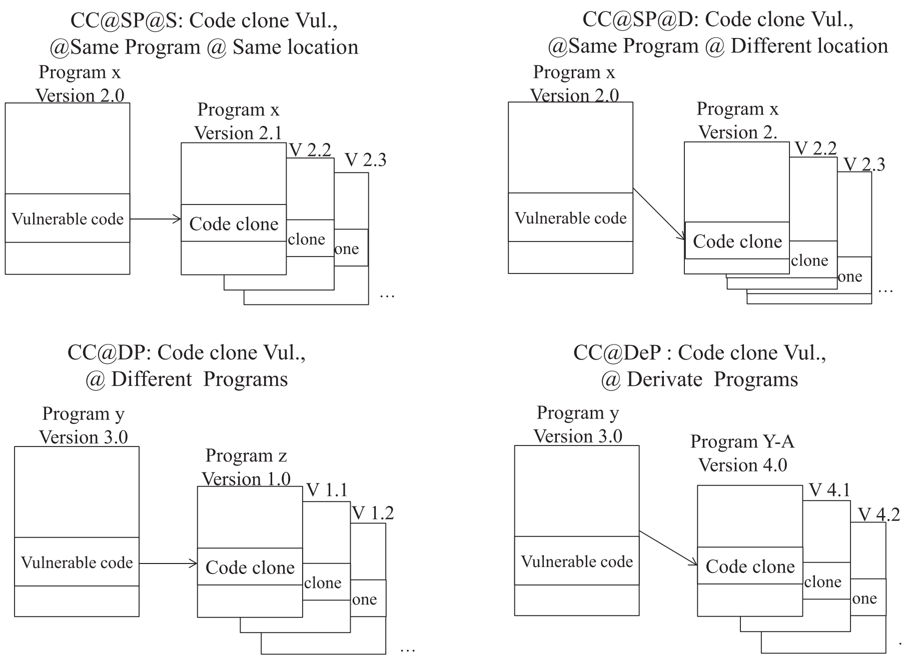


Figure 2. Possible scenarios of code clone vulnerability.

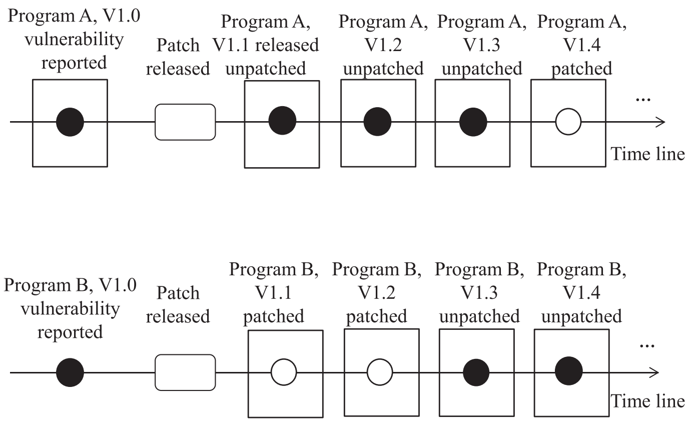

Program A, V1.0 vulnerability reported
Patch released
Program A, V1.1 released unpatched
Program A, V1.2 unpatched
Program A, V1.3 unpatched
Program A, V1.4 patched
Time line
Program B, V1.0 vulnerability reported
Patch released
Program B, V1.1 patched
Program B, V1.2 patched
Program B, V1.3 unpatched
Program B, V1.4 unpatched
Time line

Figure 3. Code clone vulnerability in the same program but different versions.

In order to find accurate code clones in an efficient and scalable way, we first make our detection engine scale well to large-code bases such as OS distributions. Second, we report accurate code clones with minimum false positives. By doing this way, we will find more precise code clones that will greatly help us to identify real code clone vulnerability later. The main steps of our code clone detection phase are described as follows.

(1) **Normalization of each file**. We do source code normalization by removing all non-ASCII character, redundant whitespaces, converting all characters to lower cases and braces.
(2) **Tokenization of each file**. After normalization, each file is tokenized by each line. We define each line as one 't' (token).
(3) ***N*-tokens**. We slide a window of n length over the tokenized file. Each n-tokens are considered a basic unit to compare. We define this basic unit as u. Figure 4 shows a four-token window sliding. Hence, a file $f(t_1, t_2, t_3, \ldots, t_l)$ is represented as $f(u_1, u_2, u_3, \ldots, u_x)$, where $x = l - n + 1$ (l is the number of lines in a certain file).

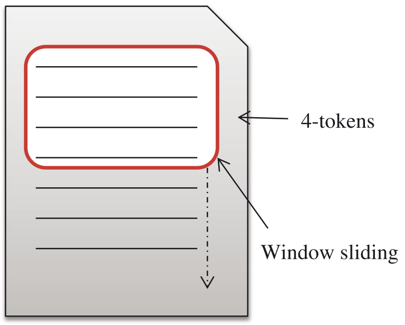


Figure 4. Window sliding of four tokens.

(4) **Checking definition for code clones**. We extract the original buggy code snippet from a security patch basically by removing lines prefixed by '+' and adding lines prefixed by '-'. Then, this original piece of buggy code is regarded as a single file called f_v. A code clone is reported when f_v is *contained* in any file f from target source code pool. Then, how can we define this *containment*? From the aforementioned steps, an n-token set for each file has been obtained, and we define this n-token set for each file as $S = \{u_1, u_2, u_3, \ldots, u_x\}$. The n-token set for the buggy code file (f_v) is defined as $S_v = \{u_1, u_2, u_3, \ldots, u_{x'}\}$. So we define the *containment* as f_v is *contained* in f if $S_v \subseteq S$.

(5) **Fast membership checking**. Because, in practice, there are tons of files that we need to deal with. So, to do membership checking in an extremely fast way is really necessary. Bloom filter [26, 27] is well known as fast membership checking that could be a very good choice to perform our task. Suppose there is a data set S, e.g., a set of n-tokens. A bloom filter represents set S as a vector of m bits initially all set to 0. To store data into the bloom filter (add an element x of S to the Bloom filter), We first apply k-independent hash functions with the value range of [1, m] on the n-tokens for files in source-code pool, in our case, $Hash(u_1), Hash(u_2), \ldots, Hash(u_x)$. For each hash $h(x) = i$, we set the i-th bit of the bit vector to 1. To check the membership in a bloom filter, we again apply k-independent hash functions on the target n-tokens data set. In our scenario, similarly, we apply k hash functions on n-tokens from f_v. Then, we check if all the corresponding bits are set to 1. If at least one of the hashed bits is 0, then we return a non-existence result.

3.2. Preparing testing source object

In order to reduce the input search space of the whole program, we propose backward sensitive-data tracing to make a testing source object. The preparing of source object is a preprocess for our concolic testing. It is performed by backward source-code analysis and program source instrumentation.

3.2.1. Backward sensitive-data tracing. Backward sensitive data tracing is used to perform an efficient static data-flow analysis backwardly to trace the sensitive data from the potential vulnerable statements back to the corresponding inputs. By doing so, we can focus on program inputs related to vulnerable statements only while leaving un-related inputs aside to reduce the whole input search space of the program. Before we explain how to perform backward sensitive data tracing in detail, important definitions of *Security Sinks, Sources, Sensitive data, Program Constraints and Security Constraints*, are presented as follows.

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 the succeeding paragraphs.

- Memory copy: The sensitive data is used as an argument to be copied in a destination buffer (e.g., *strcpy, memcpy*). When destination buffer cannot hold the sensitive data, serious security problems may occur like buffer overflow.

Table I. Security requirements for security-sensitive functions.

Security-critical func.	Security requirement
strcpy(dst,src)	dst.space > src.strlen
strncpy(dst,src,n)	(dst.space $\geqslant$ n) $\wedge$ (n $\geqslant$ 0)
strcat(dst,src)	dst.space > dst.strln + src.strlen
getcwd(buf,size)	(buf.space $\geqslant$ size) $\wedge$ (size $\geqslant$ 0)
fgets(buf,size,f)	(buf.space $\geqslant$ size) $\wedge$ (size $\geqslant$ 0)
scanf(format, ...)	# formats = # parameters-1
printf(format, ...)	# formats = # parameters-1

- Memory allocation: The sensitive data is used as an argument in memory allocation functions (e.g., *malloc, alloca*), and it usually causes insufficient memory allocation.
- Format string: The sensitive data is used improperly as argument in format functions (e.g., *printf, sprintf*). Attacker can take use of this vulnerability to take control of a system.
- Arithmetic operations: The arithmetic operations may cause integer overflow, underflow, or divided by zero problems.

Sources : *Sources* are starting points where un-trusted input data is taken by a program.

Sensitive data : Sensitive data are considered to be data depending on *Sources* that are used in the *security sinks*.

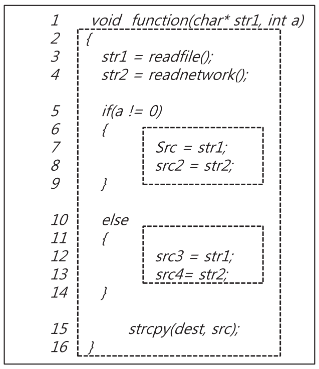
Program Constraints (PC) and Security Constraints (SC) : Program constraints are programming paradigm wherein relations between variables are stated in the form of constraints. Program constrains are generated by following a specific branch according to conditional statements in the program. Program inputs, which satisfy a set of program constraints, drive the program to execute a specific program path. Security constraints are clearly high-level security requirements. For example, the length of the string copied to a buffer must not exceed the capacity of the buffer. We need to define security requirements for statements like security-sensitive function parameters, memory access, and integer arithmetic. Table I shows our predefined security constraints for security-sensitive functions. When there are inputs that satisfy program constrains but violates security constrains($PC \wedge \overline{SC}$) at a certain point during the execution, the program is considered to be vulnerable.

Now, we discuss the detail process of backward sensitive-data tracing. First, *security sinks* and *sensitive data* are identified in the code clone. Then, we backwardly trace the *source* from the *sensitive data* to find the related input location. In order to perform an efficient backward tracing, we propose code structure graph (CSG) to store the information of source code and a recursive algorithm to do backward sensitive data tracing.

Code structure graph is a graph representing the structure of code blocks in the program source. CSG is constructed based on each function and the code blocks are identified by the pair of braces('{','}'). Figure 5(a) gives us basic code blocks from a sample code, and Figure 5(b) shows the corresponding code structure graph.

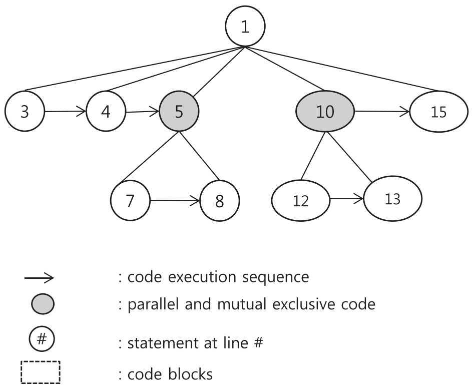
Code structure graph stores the code structure relationship and code execution sequence information among each statement and serves as an input for our recursive tracing algorithm that makes our tracing more convenient. Figure 6(a) shows the backward tracing scenario in the sample code and the algorithm is depicted in figure 6(b). In the algorithm, we take *V* (initial sensitive variable) and *G* (CSG) as inputs. Starting from the initial sensitive variable, the algorithm is designed to find the its *ancestor* by invoking *ancestor* tracing function (line 2) recursively. When the current *ancestor* becomes the *source*, the recursive algorithm exits and outputs the *source* statement. In our definition, a variable's *ancestor* is the code statement where the variable's value is being assigned. The *ancestor* statement of a certain variable v_0 could be one of the following four cases:

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



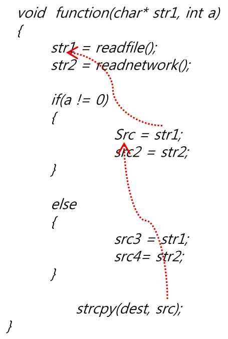
```
1   void  function(char* str1, int a)
2   {
3     str1 = readfile();
4     str2 = readnetwork();

5     if(a != 0)
6     {
7           Src = str1;
8           src2 = str2;
9     }

10    else
11    {
12          src3 = str1;
13          src4= str2;
14    }

15       strcpy(dest, src);
16  }
```

(a) Code blocks in a source code sample


1
3
4
5
10
15
7
8
12
13
: code execution sequence
: parallel and mutual exclusive code
#
: statement at line #
: code blocks

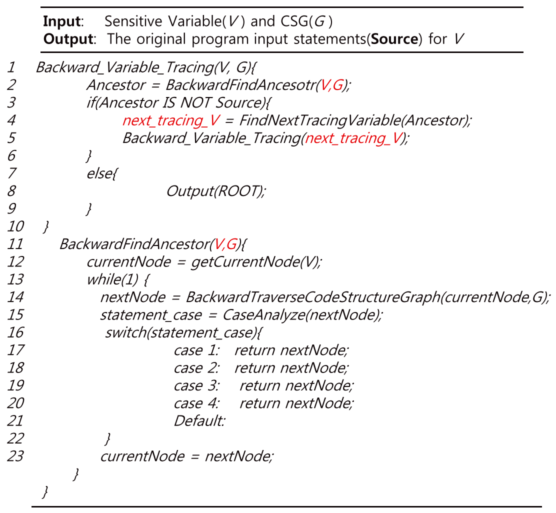
(b) Code structure graph extracted from the sample source code (a)

Figure 5. An example of code structure graph.



```
void  function(char* str1, int a)
{
     str1 = readfile();
     str2 = readnetwork();

     if(a != 0)
     {
             Src = str1;
             src2 = str2;
     }

     else
     {
             src3 = str1;
             src4= str2;
     }

        strcpy(dest, src);
}
```

(a) Backward tracing in a source code sample



Input: Sensitive Variable(V) and CSG(G)
Output: The original program input statements(**Source**) for V

```
1   Backward_Variable_Tracing(V, G){
2           Ancestor = BackwardFindAncesotr(V,G);
3           if(Ancestor IS NOT Source){
4                   next_tracing_V = FindNextTracingVariable(Ancestor);
5                   Backward_Variable_Tracing(next_tracing_V);
6           }
7           else{
8                         Output(ROOT);
9           }
10  }
11     BackwardFindAncestor(V,G){
12          currentNode = getCurrentNode(V);
13          while(1) {
14            nextNode = BackwardTraverseCodeStructureGraph(currentNode,G);
15            statement_case = CaseAnalyze(nextNode);
16             switch(statement_case){
17                        case 1:   return nextNode;
18                        case 2:   return nextNode;
19                        case 3:   return nextNode;
20                        case 4:   return nextNode;
21                        Default:
22             }
23            currentNode = nextNode;
        }
  }
```

(b) Recursive tracing algorithm

Figure 6. An example of code structure graph(CSG).

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

The algorithm tries to trace the *ancestor* statement by backward traversing the code structure graph (line 14) according the different cases earlier (line16). The algorithm finally helps us to find the *source*(the program original input statement) related to the sensitive data.

3.2.2. Program source instrumentation. To instrument the program source, we make assertions based on security requirements right before the *security sink* and replace the input statement with symbolic values. We can see this process from Figure 7.

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

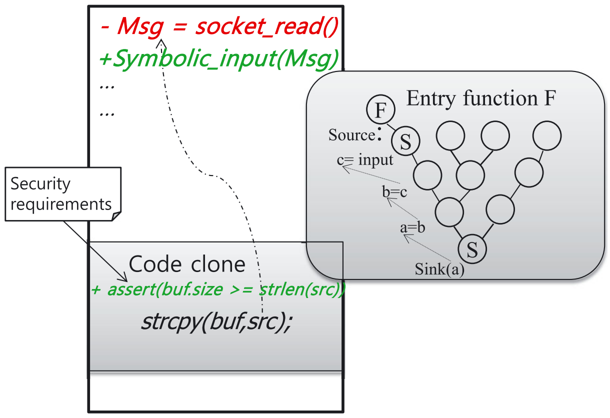


Figure 7. Instrumentation of program source.

3.3. *Code clone verification using concolic testing*

Symbolic execution and concolic execution have been widely used in software testing and some have shown good practical impact, such as KLEE [10], Concolic Unit Testing Engine (CUTE) [6] and Directed Automated Random Testing (DART) [28]. However, they suffer from path explosion problem that makes them cannot scale well to large real-world programs. H. Li *et al.* [29] has proposed variable backward slicing to analyze a program. This approach helps us to concentrate on those paths only related to sensitive sinks that dramatically reduce the number of paths to analyze. However, pure-static symbolic execution does not give us enough support on real-world programs. Driven by the aforementioned concerns, we apply concolic testing in our mechanism to verify the code clones. 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 Figure 1. The concepts of PC and SC are addressed in [29]. In our scenario, we focus on the paths related to code clones rather than the countless number of paths in the whole program, which helps us to mitigate the path explosion problem to a large extent. We also focus on the program inputs that are related to the sensitive data in the security sinks instead of the whole program input space. Our approach for concolic testing to verify code clones mainly follows a general concolic testing procedure [12]. However, the difference is that we focus on generating an input to execute the vulnerable branch instead of trying to generate inputs to traverse every possible paths of the program. Our approach for concolic testing is target branch-oriented rather than branch coverage-oriented. Hence, we are more time cost efficient when doing concolic testing. The detailed process is described as follows.

(1) *Declaration of symbolic variables*. Initially, a user must specify which variables should be handled as symbolic variables, based on which symbolic path formulas are constructed. This is performed by backward sensitive-data tracing in the second stage of CLORIFI.
(2) *Program instrumentation*. The target program source is instrumented with probes that record the symbolic branch conditions. For instance, a probe is inserted to record the branch condition. Then, the instrumented source program will be complied into an executable binary.
(3) *Concrete execution*. The instrumented binary is executed with given input values. For the first execution, initial input values are assigned randomly. From the second execution, input values are obtained from Step 6.
(4) *Obtain a symbolic path formula*. The symbolic execution part of the concolic execution collects symbolic path conditions over the symbolic input values at each branch point encountered for along the concrete execution path for a test case tc_i.
(5) *Generate a new symbolic path formula*. When a target program terminates, to obtain the next input values, a new symbolic path formula is generated by negating one path condition c_j and removing subsequent path conditions of current one (i.e., $c_1 \bigcap c_2 \bigcap ... \bigcap \overline{c_j}$).

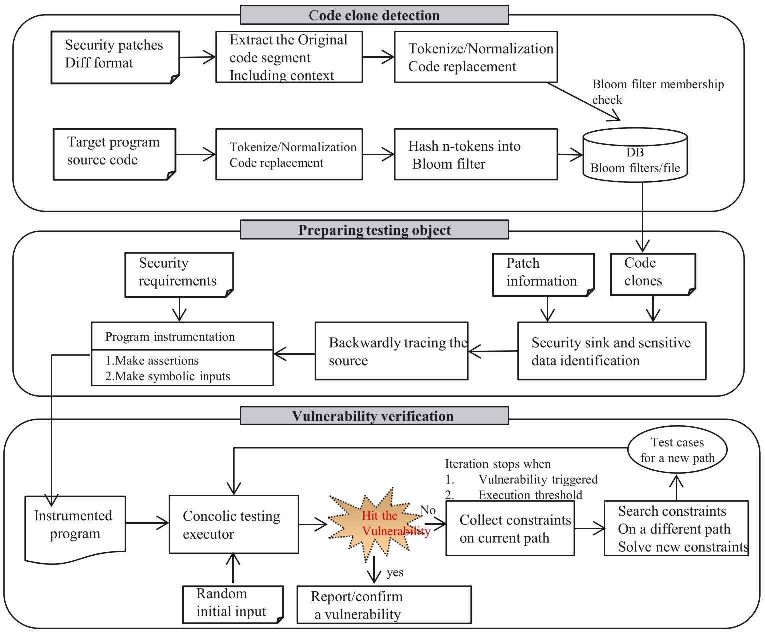

Code clone detection
Security patches Diff format
Extract the Original code segment Including context
Tokenize/Normalization Code replacement
Bloom filter membership check
Target program source code
Tokenize/Normalization Code replacement
Hash n-tokens into Bloom filter
DB Bloom filters/file
Preparing testing object
Security requirements
Patch information
Code clones
Program instrumentation
1.Make assertions
2.Make symbolic inputs
Backwardly tracing the source
Security sink and sensitive data identification
Vulnerability verification
Test cases for a new path
Iteration stops when
1. Vulnerability triggered
2. Execution threshold
Instrumented program
Concolic testing executor
Hit the Vulnerability
No
Collect constraints on current path
Search constraints On a different path Solve new constraints
yes
Random initial input
Report/confirm a vulnerability

Figure 8. The architecture of CLORIFI.

(6) *Select the next input values* tc_{i+1}. A constraint solver such as a Satisfiability Modulo Theory (SMT) solver [30] generates a model that satisfies a symbolic path formula. This model determines the next concrete input values to try (i.e., tc_{i+1}), and the concolic testing procedure iterates from Step 3 using these input values.

(7) *Iteration stop criteria*. When there is at least one test case which trigger the assertion at the code clone (violate the assertion), then this code clone is proved to be real code clone vulnerability. The iteration stops once the code clone is being successfully verified. The iteration stop criteria is:

- Execution stops when there is one triggering test case generated (vulnerable);
- Execution stops when a certain time threshold *t* has been reached; however, no triggering test case is generated(not vulnerable);

In Figure 8, we can see the detail architecture of CLORIFI, which consists of three phases: **Code clone detection**, **Preparation of testing source object** and **Vulnerability verification**. This mechanism is used to discover un-patched code clone vulnerabilities in real-world projects. We choose CREST-BV [15] as a basic concolic execution engine because of its good performance in test case generation speed.

4. EXPERIMENTAL RESULTS

To evaluate the effectiveness of CLORIFI, we applied real world open source projects as different target source pools including popular Linux OS distributions such us Ubuntu 14.04 (recent release) and CentOS 7.0 (latest release) and various versions of program packages such as Rsyslog, Apache, Firefox and etc. Test cases from Juliet Test Suite [14] are also selected as evaluation objects. In this section, experiment environment is firstly described, followed by the experimental results.

4.1. *Implementation*

Environment setup : We performed all experiments to discover code clone vulnerabilities on a desktop machine running Linux ubuntu 12.04 LTS (3.2 GHz Intel Core i7 CPU, 8 GB memory, 512 GB hard drive). We also setup some basic parameters for the code clone detection and concolic

Table II. Yearly distribution of collected CVE patches.

CVE patches	2010	2011	2012	2013	2014
Buffer overflow	2	1	5	8	7
Integer overflow	1	1	2	0	7
Buffer cased by IOS	0	0	3	2	2
Other	0	1	0	0	1
Total			43		

Table III. Detection results.

Target	CVE patches	Target src pool	# of files	# of CC	Execution time (CC part)	# of real vuln.	# of FP	# of FN (Veri. part)
SP1	CVE PP (2010–2014)	Ubuntu 14.04 OS distribution	259,346	63	24,812.5 s (7 h)	7	0	2 (3.2%)
SP2	CVE PP (2010–2014)	CentOS 7.0 OS distribution	520,549	40	33,146.5 s (9.2 h)	10	0	2 (5%)
SP3	CVE PP (2010–2014)	Httpd-2.2.23 to 2.4.6	5020	8	474.2 s (7.9 min)	8	0	0
SP4	CVE PP (2010–2014)	Rsyslog-5.8.13 to 8.2.1	1692	7	274.7 s (4.57 min)	7	0	0
SP5	CVE PP (2010–2014)	Firefox-23.0 to 31.0	156,620	6	8902.2 sec (148.4 min)	6	0	0
SP6	CVE PP (2010–2014)	Cpio-2.6 to 2.10	937	14	32.1 s (0.53 min)	5	0	0

Note: PP = patch pool; CC= code clone(s); vuln.= vulnerability; veri. part = verification part

verification. For the code clone detection, we set the length n over the tokenized file(n-tokens) to be 4, and we use three fast hash functions: CRC variant, PJW hash and BUZ hash [31]. In the implementation of concolic testing verification, we set the testing time-budget threshold t to be 5 min, which means a vulnerability is reported only if it has been successfully verified in t.

Dataset : For the security patches, we collected 135 security patches from 43 CVE [32] patch files (e.g., CVE-2010-0405.patch) related to Linux programs released from 2010 to 2014. We mainly collected patches related to buffer overflows and integer overflows because these are most common types of vulnerabilities. Table II shows the number of CVE patch files we collected on yearly base.

For the target testing programs, we collected the source of Linux Ubuntu 14.04 Operating System (OS) distribution to test our mechanism. In one step further, we collected various versions of linux packages to find more vulnerabilities from different program versions. Moreover, in order to prove the efficiency of vulnerability verification of CLORIFI, we collected 100 test cases from Juliet Test Suite. The Suite is created by US National Security Agency (NSA), and it has been widely used to test the effectiveness of vulnerability detection tools.

4.2. *Experimental results*

4.2.1. Detection results of different source pools. We have conducted our experiments with different target source pools (**SP1** to **SP6**).

SP1 and SP2. Linux Ubuntu 14.04 (recent version) OS distribution and CentOS7.0.1406 (latest version) OS distribution

We found over 63 code clones in Linux Ubuntu 14.04 and 40 code clones in Linux CentOS 7. Table III shows the detection results including the number of files processed, the execution time, false positives and false negatives. The processing time of code clone detection for Ubuntu and CentOS distributions is nearly 7 and 9.2 hours, respectively, which means this process can be conducted in daily base. Because the verification part (backward tracing assisted concolic testing) still involves intermediate manual processes such as program instrumentation and preparing testing object and

Table IV. Code clone vulnerabilities reported from SP1.

Program	CVE patch	Location of vulnerability	Verification time
Cmake-2.8.12.2	CVE-2010-0405.patch	/Utilities/cmbzip2/decompress.c:381	25.6 s
Firefox-28.0+build2	CVE-2010-0405.patch	/modules/libbz2/src/decompress.c:381	30.5 s
Thunderbird-24.4.0+build1	CVE-2010-0405.patch	/mozilla/modules/src/decompress.c:381	31.2 s
rsyslog-7.4.4	CVE-2011-3200.patch	/plugins/pmrfc3164sd/pmrfc3164sd.c:272	1.2 s
gegl-0.2.0	CVE-2012-4433.patch	/operations/external/ppm-load.c:87	9.8 s
linux-3.13(Linux kernel)	CVE-2014-2581.patch	/net/ipv4/ping.c:250	11.5 s
httpd-2.4.7(Apache)	CVE-2011-3368.patch	/server/protocol.c:625	20.5 s

Table V. Code clone vulnerabilities reported from SP2.

Program	CVE patch	Location of vulnerability	Verification time
Cmake-2.8.11	CVE-2010-0405.patch	/Utilities/cmbzip2/decompress.c:381	25.6 s
firefox-24.5.0esr.source	CVE-2010-0405.patch	/modules/libbz2/src/decompress.c:381	30.5 s
rsyslog-7.4.7	CVE-2011-3200.patch	/plugins/pmrfc3164sd/pmrfc3164sd.c:272	1.2 s
httpd-2.4.7(Apache)	CVE-2011-3368.patch	/server/protocol.c:625	20.5 s
ghostscript-9.07	CVE-2013-4276.patch	/lcms/samples/icctrans.c:651	25.6 s
glibc-2.17-2-gc4ccff1	CVE-2013-4458.patch	/sysdeps/posix/getaddrinfo.c:199	13.4 s
poppler-0.22.5	CVE-2013-4473.patch	/utils/pdfseparate.cc:69	15.3 s
openssl-1.0.1e	CVE-2014-0160.patch	/ssl/d1_both.c:1474	18.1 s
cpio-2.11	CVE-2014-9112.patch	/src/copyin.c:141	5.6 s
php5.5.9	CVE-2014-4049.patch	/ext/standard/dns.c:520	22.9 s

besides, the verification for each program is a completely independent process, so we measured the verification time for each program, respectively. The results are shown in Table IV and Table V. We get an *average* verification time of 18.6 s among seven programs from SP1 and an *average* verification time of 17.9 s among 10 programs from SP2. The fast verification benefits from our backward tracing assisted concolic testing that is further explained in section 4.2.2. From the results, though, the code clone detection part takes the majority overhead of overall execution process, because it is a pure batch processing procedure, we could use parallel and distributed computing frameworks such as MapReduce [33] to reduce the overhead to a great extent when adequate computing resource is available. This provides us a good insight for future research.

We reported 7 real-world vulnerabilities out of 63 code clones in Ubuntu 14.04 and 10 vulnerabilities out of 40 code clones from CentOS 7. Table IV and Table V show the results of the real vulnerabilities that we verified. As shown in Table III, CLORIFI reported zero false positives in all six source pools. In terms of false negatives, CLORIFI reported two False Negatives (FNs) in SP1 with the FN rate of 3.2% and two FNs in SP2 with the rate of 5%. For SP3 to SP6, no false negative was reported. We have looked into the false negative cases and found out that the limitation of constraint solver [30] in the concolic-testing engine caused these negatives. Yet, this topic is out of scope of this research. In the experiments, the false negatives were measured for code clone verification part only. The reasons why we did not measure false negatives for overall process of vulnerability detection are two fold. First, CLORIFI reports code clones based on the collected CVE patches. Because of resource limitation, collecting CVE patches for all kinds of vulnerabilities is not practical. So, we may miss vulnerabilities because of the limitation of patch collection. Second, for real-world source code base such as recent Ubuntu distributions, the ground truth of existences of real vulnerabilities is not available. So, measuring false negatives for vulnerability detection in general becomes infeasible. Nevertheless, CLORIFI's automatic verification states a huge improvement over other code clone-detection approaches [15, 16, 18]. To make it more convincing, we also measured the accuracy of CLORIFI on Juliet Test Suite [14], which is explained in section 4.3.

Table VI. Code clone vulnerabilities reported from SP3 (Apache).

Version	Release date	# LOC	# of reported code clones	# of vulnerability found/verified	# of false positives
Httpd-2.3.6	6/17/2010	209,369	1	1	0
Httpd-2.3.8	8/31/2010	210,564	1	1	0
Httpd-2.3.11-beta	3/7/2011	219,427	1	1	0
Httpd-2.3.15-beta	11/15/2011	226,497	0	0	0
Httpd-2.4.1	2/17/2012	223,050	1	1	0
Httpd-2.4.2	4/17/2012	223,265	1	1	0
Httpd-2.4.3	8/18/2012	223,921	1	1	0
Httpd-2.4.4	2/25/2013	226,000	1	1	0
Httpd-2.4.6	6/22/2013	233,330	1	1	0

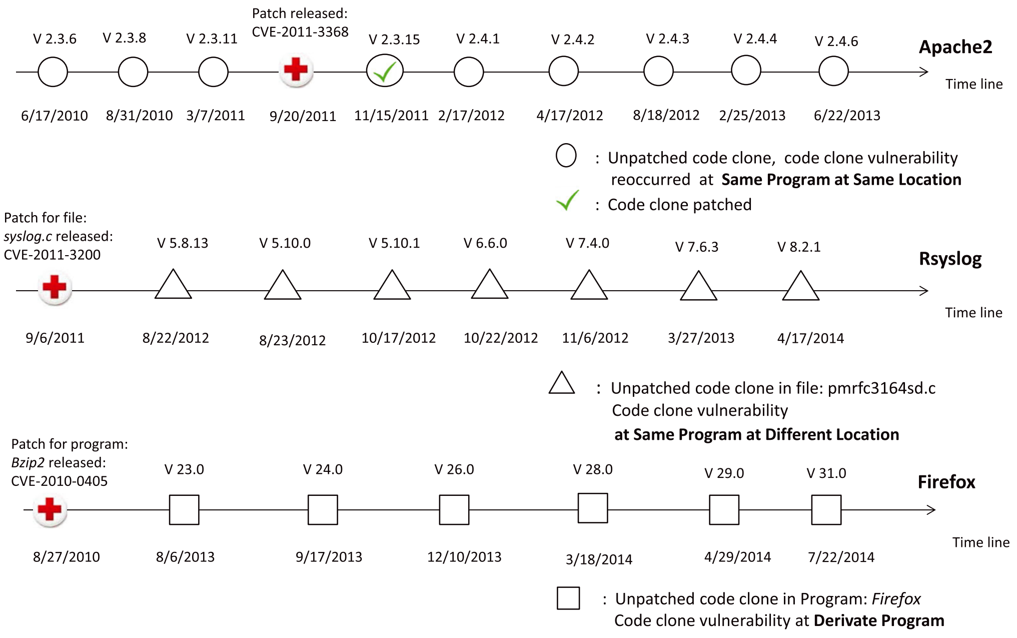


Figure 9. Code clone vulnerability results from different program versions.

SP3, SP4, SP5 and SP6. Different program versions(Apache, Rsyslog, Firefox, and Cpio). Based on the result of the previous experiment, we looked into different versions of the affected programs to see code clone vulnerability in different program versions. We collected different versions(released after the publication time of the security patch) of Rsyslog, Apache, Firefox, and Cpio. We used them as target source code pool 3 and 6, respectively. For the Apache case, we collected 11 different versions from 2.3.6 to 2.4.6. CLORIFI processed 5020 source files (3,642,170 code of lines) in nearly 7.9~min. As a result, we have found eight code clones and confirmed all of the eight code clones to be vulnerable. We can see the detail from Table III and Table VI.

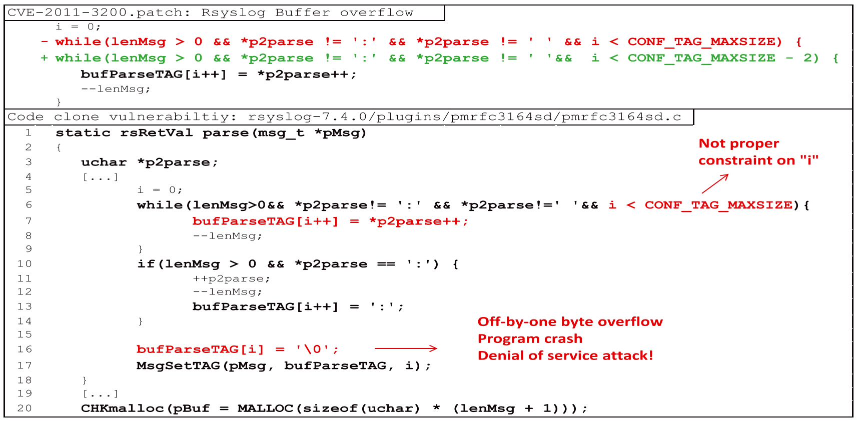
These code clones are detected from CVE-2011-3368.patch. From the result, we can see that after the release time of the CVE-2011-3368.patch, the Apache2 developers did not actually fix the vulnerability in the later release versions. For some reason, it was fixed in Httpd-2.3.15-beta, and then, the same vulnerability occurred again in the later release versions. This case corresponds to the code clone type–CC@SP@S as aforementioned in 3.1. Figure 9 shows the illustration.

Similarly, we also collected different versions of Rsyslog and reported 7 affected versions. Table III and Table VII show the results.

Our mechanism processed totally 1692 source files (656,708 code of lines) in nearly 4.57~min. These reported code clones are related to CVE-2011-3200.patch. After the release time of this security patch, the developing team fixed this vulnerability originally in the source file /syslogd.c.

Table VII. Code clone vulnerabilities reported with SP4 (Rsyslog).

Version	Release date	# LOC	# of reported code clones	# of vulnerability found/verified	# of false positives
Rsyslog-5.8.13	8/22/2012	78,937	1	1	0
Rsyslog-5.10.0	8/23/2012	78,259	1	1	0
Rsyslog-5.10.1	10/17/2012	77,811	1	1	0
Rsyslog-6.6.0	10/22/2012	92,448	1	1	0
Rsyslog-7.4.0	6/6/2012	105,324	1	1	0
Rsyslog-7.6.3	3/27/2013	111,218	1	1	0
Rsyslog-8.2.1	4/17/2014	112,711	1	1	0



```
CVE-2011-3200.patch: Rsyslog Buffer overflow
      i = 0;
    - while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' ' && i < CONF_TAG_MAXSIZE) {
    + while(lenMsg > 0 && *p2parse != ':' && *p2parse != ' '&&  i < CONF_TAG_MAXSIZE - 2) {
         bufParseTAG[i++] = *p2parse++;
         --lenMsg;
      }
Code clone vulnerabiltiy: rsyslog-7.4.0/plugins/pmrfc3164sd/pmrfc3164sd.c
   static rsRetVal parse(msg_t *pMsg)
   {
      uchar *p2parse;
      [...]
            i = 0;
            while(lenMsg>0&& *p2parse!= ':' && *p2parse!=' '&& i < CONF_TAG_MAXSIZE){
                  bufParseTAG[i++] = *p2parse++;
                  --lenMsg;
            }
            if(lenMsg > 0 && *p2parse == ':') {
                  ++p2parse;
                  --lenMsg;
                  bufParseTAG[i++] = ':';
            }

            bufParseTAG[i] = '\0';
            MsgSetTAG(pMsg, bufParseTAG, i);
      }
      [...]
      CHKmalloc(pBuf = MALLOC(sizeof(uchar) * (lenMsg + 1)));
```

Figure 10. Code clone vulnerability from Rsyslog.

Table VIII. Code clone vulnerabilities reported with SP5 (Firefox).

Version	Release date	# LOC	# of reported code clones	# of vulnerability found/verified	# of false positives
Firefox-23.0	8/6/2013	6,573,727	1	1	0
Firefox-24.0	9/17/2013	6,553,867	1	1	0
Firefox-26.0	12/10/2013	6,763,266	1	1	0
Firefox-28.0	3/18/2014	7,067,994	1	1	0
Firefox-29.0	4/29/2014	7,130,674	1	1	0
Firefox-31.0	7/22/2014	7,327,628	1	1	0

However, from the version Rsyslog-5.8.13, this code clone vulnerability re-occurred in another file */pmrfc3164sd.c* because of the careless code re-use by developers. This case corresponds to the code clone type–CC@SP@D(see 3.1) and Figure 9 shows the illustration. We can also see the code clone vulnerability in Rsyslog-7.4.0 in Figure 10.

For source pool 5 and 6, we collected different versions of Firefox and Cpio and reported 6 affected versions in Firefox and 5 versions in Cpio. Results are in Table VIII and Table IX.

The reported code clones from Firefox packages are related to CVE-2010-0405.patch. The vulnerability related to this patch was originally reported from the source code in bzip2. However, even after the release of the security patch, the Firefox packages still reuse the source code of bzip2 without being patched which introduces integer overflow vulnerability in Firefox. This case corresponds to the code clone type–CC@De@D (3.1). The illustration is presented in Figure 9.

The last source pool is Cpio, which is a program to manage archives of files. We have reported five different affected versions in this source pool, all of which are suffering from an integer overflow induced buffer overflow vulnerability. The attackers might make use of this vulnerability to crash the program or execute shellcode. Figure 11 shows the code clone vulnerability in Cpio-2.10 . The code clone vulnerability discovery illustration of different program versions is shown in Figure 9.

4.2.2. Comparison with conventional concolic testing. As we mentioned before, we apply backward sensitive data tracing to assist concolic testing for our verification. We have compared our approach with CREST [13]. Even though, recent works, for example, CGS [22], claim their efficiency improvement of concolic testing, they are targeting on the efficiency of covering overall program branches. However, we are aiming at generating an input that could trigger a certain vulnerability. That means, our goal is to generate an input that drives the execution to a specific branch (buggy branch). In this sense, those techniques including CGS have not been proved superior than other techniques. What is more, CREST has been widely used either as a basic concolic-testing

Table IX. Code clone vulnerabilities reported with SP6 (Cpio).

Version	Release date	# LOC	# of reported code clones	# of vulnerability found/verified	# of false positives
Cpio-2.6	12/20/2004	22,248	2	1	0
Cpio-2.7	10/21/2006	41,074	3	1	0
Cpio-2.8	6/8/2007	45,154	3	1	0
Cpio-2.9	6/28/2007	46,088	3	1	0
Cpio-2.10	6/20/2009	54,120	3	1	0

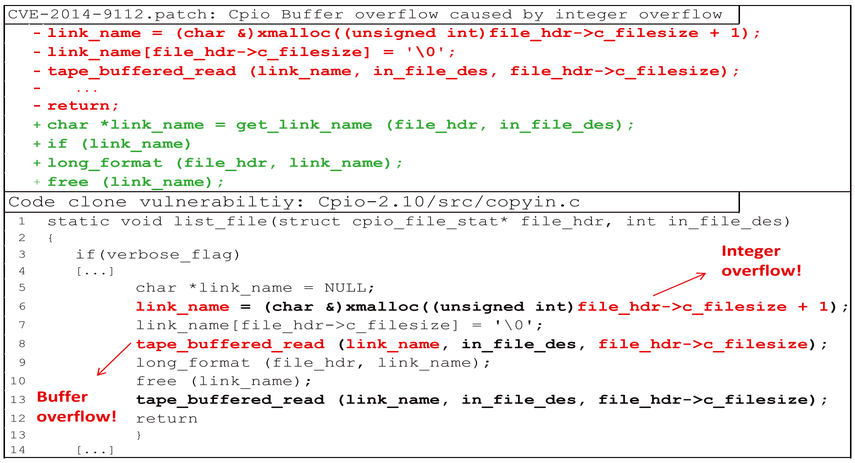
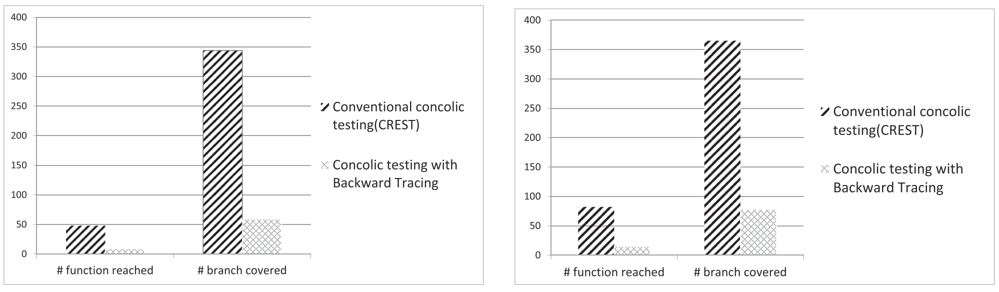

```
CVE-2014-9112.patch: Cpio Buffer overflow caused by integer overflow
  - link_name = (char &)xmalloc((unsigned int)file_hdr->c_filesize + 1);
  - link_name[file_hdr->c_filesize] = '\0';
  - tape_buffered_read (link_name, in_file_des, file_hdr->c_filesize);
  -    ...
  - return;
  + char *link_name = get_link_name (file_hdr, in_file_des);
  + if (link_name)
  + long_format (file_hdr, link_name);
  + free (link_name);
Code clone vulnerabiltiy: Cpio-2.10/src/copyin.c
  static void list_file(struct cpio_file_stat* file_hdr, int in_file_des)
  {
     if(verbose_flag)
     [...]
           char *link_name = NULL;
           link_name = (char &)xmalloc((unsigned int)file_hdr->c_filesize + 1);
           link_name[file_hdr->c_filesize] = '\0';
           tape_buffered_read (link_name, in_file_des, file_hdr->c_filesize);
           long_format (file_hdr, link_name);
           free (link_name);
           tape_buffered_read (link_name, in_file_des, file_hdr->c_filesize);
           return
           }
     [...]
```

Figure 11. Code clone vulnerability from Cpio.



(a) Rsyslog-7.4.0 (b) Cpio-2.10

Figure 12. The comparison with conventional concolic testing.

engine or a benchmark program in many previous works such as [12, 22] and [23]. So we chose CREST for our comparison experiments.

We have used two programs, Rsyslog-7.4.0 and Cpio-2.10, for testing targets. Both approaches have generated triggering inputs and successfully verified these vulnerabilities(e.g., CVE-2011-3200, Figure 6). However, Figure 8 shows a performance comparison in terms of *number of branches covered* and *number functions reached* when the triggering input has been generated.

As we can see, for the program Rsyslog-7.4.0 in Figure 12(a), in order to generate a triggering input for the vulnerability, CLORIFI has reduced the number of covered branches from 344 to 59 and has reduced the number of reached functions from 48 to 9. What is more, our approach only spent 1.2 s to trigger this vulnerability, while CREST took 24~min. In Figure 12(b), CLORIFI has made the number of covered branches and the number of reached functions 78 and 15, respectively, when the vulnerability from Cpio-2.10 being triggered; however, these two measurement values were as high as 365 and 78 when the conventional concolic testing approach is applied. This indicates that, with backward data tracing, we can dramatically reduce the number of paths to traverse and decrease the input searching space that mitigates the path explosion problem.

4.3. *Evaluations of vulnerability verification phase*

In order to prove the effectiveness of the vulnerability verification of CLORIFI, we collected 100 test cases from Juliet Test Suite. For every test case, there are 'good' functions and 'bad' functions that provide the ground truth for our evaluation. As shown in Table X, the 100 test cases consist of different vulnerable types, and there are totally 250 spots to verify (100 bad functions and 150 good functions). We can see the results from Table XI.

As we can see, CLORIFI generates no false positive and five false negatives. The precision is $\frac{TP}{TP+FP} = 100\%$ and the recall is $\frac{TP}{TP+FN} = 95\%$. It achieves the verification accuracy of $98\%(\frac{TP+TN}{TP+FP+TN+FN})$. We also measured the average verification time needed to verify each test case. The average verification time is 0.24 s. This proves that our mechanism has good verification accuracy, and it is time cost effective.

4.4. *Threats to validity*

Several threats to the validity of our experiments are discussed as follows.

- **The CVE patch files may not be representative and universal**. In our experiment, we have collected 135 security patches from 43 CVE patch files as our CVE patch pool. Most of them are overflow cases. This makes us might miss out some none overflow type of vulnerabilities, such as null pointer deference and format strings. What is more, some CVE patches make changes in the header files of the source code where we cannot get enough information to identify security thinks, and some vulnerabilities are caused by a number of patches in different locations in source code that also make us hard to verify them.

Table X. Distribution of test cases.

Vulnerability type	Number of test cases
Stack-based buffer overflow	25
Heap-based buffer overflow	25
Integer overflow	25
Format string	25

Table XI. Evaluation metrics of CLORIFI on Juliet Test Suite.

	True	False
Positive	95	0
Negative	150	5

- **The target programs for evaluation of concolic testing efficiency might not be representative**. In order to improve the efficiency of conventional concolic testing, we have proposed backward sensitive data tracing to assist the concolic testing for vulnerability verification. We have used Rsyslog-7.4.0 and Cpio-2.10 to perform our evaluation. Even though, these programs are very popular in current Linux operation systems, they might not be representatives of all other programs. Actually, because of the limitations of constraint solver [30] in concolic testing, some vulnerability cases cannot be verified when the constraints are not supported by the current concolic-testing engine. More precise and effective constraint solving techniques in concolic testing may help us yield different results.

5. CONCLUSION

In this paper, we develop a novel mechanism called CLORIFI, which combines the advantage of static and dynamic analysis to detect code clone vulnerability using code clone verification. The automatic verification of CLORIFI helps us to dramatically reduce the false positives when discovering vulnerabilities. What is more, by tracing the input from the sensitive data in code clones and preparing the testing source object, CLORIFI performs concolic testing to do verification in a way that mitigates the path explosion problem. We conducted several experiments with different target source pools of popular open source projects. The results show that CLORIFI can find real-world vulnerabilities with extremely low false positive within reasonable time.

However, there are several concerns as well. First of all, some CVE patches patch the vulnerable code in header files whose information is not enough to identify sensitive sinks. Second, several patches may contribute to a single vulnerability, which makes the verification impractical. Finally, because of the limitation of branch coverage of the concolic testing, we have false negatives in verification phase. In future research, we will study the classification of security patches and the searching strategies of concolic testing for more efficient vulnerability verification.

ACKNOWLEDGEMENTS

This research was supported by the MSIP(The Ministry of Science, ICT, and Future Planning), Korea and Microsoft Research, under IT/SW Creative research program supervised by the NIPA (National IT Industry Promotion Agency)(H0503-13-1038).

REFERENCES

1. Li H, Kwon H, Kwon J, Lee H. A scalable approach for vulnerability discovery based on security patches. *Proceedings of 5th International Conference on Applications and Techniques in Information Security*, Melbourne, Australia, 2014; 109–122.
2. Wheeler D. Flawfinder, 2011. (Available from: http://www.dwheeler.com/flawfinder) [Accessed on 27 April 2014].
3. Evans D. Splint. *Improving Security Using Extensible Lightweight Static Analysis*, 2002. (Available from: http://www.splint.org) [Accessed on 20 April 2014].
4. Yamaguchi F, Wressnegger C, Gascon H, Rieck K. Chucky: exposing missing checks in source code for vulnerability discovery. *Proceedings of ACM SIGSAC Conference on Computer & Communications Security*, Berlin, Germany, 2013; 499–510.
5. Viega J, Bloch JT, Kohno Y, McGraw G. ITS4: a static vulnerability scanner for C and C++ code. *Proceedings of the Annual Computer Security Applications Conference(ACSAC)*, New Orleans, Louisiana, USA, 2000; 257–267.
6. Sen K, Marinov D, Agha G. Cute: a concolic unit testing engine for C. *Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Lisbon, Portugal, 2005; 263–272.
7. Haugh E, Bishop M. Testing C programs for buffer overflow vulnerabilities. *Network and Distributed System Security Symposium*, San Diego, California, USA, 2003; 123–130.
8. Ghosh H, Connor TO, McGraw G. An automated approach for identifying potential vulnerabilities in software. *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, USA, 1998; 104–114.
9. Ma K, Phang K, Foster J, Hicks M. Directed symbolic execution. *Proceedings of 18th International Conference on Static Analysis*, Berlin, Heidelberg, 2011; 95–111.
10. Cadar C, Dunbar D, Engler D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, Vol. 8, 2008; 209–224.

11. Kim M, Kim Y, Choi Y. Concolic testing of the multisector read operation for flash storage platform software. *Journal of Formal Aspects of Computing* 2012; **24**(3):355–374.
12. Kim M, Kim Y, Jang Y. Industrial application of concolic testing on embedded software: case studies. *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, Montreal, Canada, 2012; 390–399.
13. Burnim J, Sen K. Heuristics for scalable dynamic test generation. *Proceedings of IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, 2008; 443–446.
14. Boland T, Black PE. Juliet 1.1 C/C++ and Java Test Suite. *Journal of Computer* 2012; **45**(10):88–90.
15. Gabel M, Yang J, Yu Y, Goldszmidt M, Su Z. Scalable and systematic detection of buggy inconsistencies in source code. *Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications*, Portland, OR, USA, 2011; 175–190.
16. Jiang L, Misherghi G, Su Z, Glondu S. Deckard: scalable and accurate tree-based detection of code clones. *Proceedings of International Conference on Software Engineering*, Minneapolis, Minnesota, USA, 2007; 96–105.
17. Uddin MS, Roy CK, Schneider KA, Hindle A. On the effectiveness of Simhash for detecting near-miss clones in large scale software systems. *Proceedings of the 18th Working Conference on Reverse Engineering*, Lero, Limerick, Ireland, 2011; 13–22.
18. Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions. *Proceedings of IEEE Symposium on Security and Privacy*, San Francisco, California, USA, 2012; 48–62.
19. Hastings R, Joyce B. Purify: fast detection of memory leaks and access errors. *Proceedings of the Winter USENIX Conference*, San Francisco, California, USA, 1992; 125–136.
20. Oehlert P. Violating assumptions with fuzzing. *IEEE Security and Privacy.* Oakland, California, USA, 2005; **3**(2): 58–62.
21. Zhang D, Liu D, Lei Y, Kung D, Csallner C, Wang W. Detecting vulnerabilities in C programs using trace-based testing. *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, Chicago, Illinois, USA, 2010; 241–250.
22. Seo H, Kim S. How we get there: A context-guided search strategy in concolic testing. *Proceeding (FSE 2014) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM: New York, NY, USA, 2014; 413–424.
23. Garg P, Ivancic F, Balakrishnan G, Maeda N, Gupta A. Feedback-directed unit test generation for C/C++ using concolic execution. *Proceedings of the 2013 International Conference on Software*, San Francisco, California, USA, 2013; 132–141.
24. Li Y, Su Z, Wang L, Li X. Steering symbolic execution to less traveled paths. *Proceedings of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, Indianapolis, Indiana, USA, 2013; 19–32.
25. Majumdar R, Sen K. Hybrid concolic testing. *Proceedings of ICSE International Conference on Software Engineering*, Minneapolis, Minnesota, USA, 2007; 416–426.
26. Broder A, Mitzenmacher M. Network applications of bloom filters: a survey. *Internet Mathematics* 2004; **1**(4): 485–509.
27. Shi Q, Petterson J, Dror G, Langford J, Smola A. Hash kernels for structured data. *Journal of Machine Learning Research* 2011; **10**:2615–2637.
28. Godefroid P, Klarlund N, Sen K. Dart: directed automated random testing. *Proceedings of ACM Sigplan Conference on Programming Language Design and Implementation*, Chicago, IL, USA, 2005; 213–223.
29. Li H, Kim T, Bat-Erdene M, Lee H. Software vulnerability detection using backward trace analysis and symbolic execution. *Proceedings of International Conference on Availability, Reliability and Security*, Regensburg, Germany, 2013; 446–454.
30. Moura d L, Bjorner N. Z3:An efficient SMT solver. *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Budapest, Hungary, 2008; 337–340.
31. 2000. (Available from: http://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html) [Accessed 8 August 2014].
32. Vulnerabilities C. Exposures cve. (Available from: http://cve.mitre.org.)
33. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communication of ACM* 2008; **51**(1):107–113.