# A Scalable Approach for Vulnerability Discovery Based on Security Patches

Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee*

Dept. of Computer Science and Engineering, Korea University
{hongzhe,chasm,signalnine,heejo}@korea.ac.kr

**Abstract.** Software vulnerability has long been considered an important threat to the system safety. A vulnerability often gets reproduced due to the frequent code reuse by programmers. Security patches are often not propagated to all code clones, however they could be leveraged to discover unknown vulnerabilities. Static auditing approaches are frequently proposed to scan code for security flaws, unfortunately, they often generate too many false positives. While dynamic execution analysis can precisely report vulnerabilities, they are in effective in path exploration which limits them to scale to large programs. In this paper, we propose a scalable approach to discover vulnerabilities in real world programs based on released security patches. We use a fast and scalable syntax-based way to find code clones and then, we verify the code clones using concolic testing to dramatically decrease the false positives. Besides, we mitigate the path explosion problem by backward data tracing in concolic execution. We conducted experiments with real world open source projects (Linux Ubuntu OS distributions and program packages) and we reported 7 real vulnerabilities out of 63 code clones found in Ubuntu 14.04 LTS. In one step further, we have confirmed more code clone vulnerabilities in various versions of programs including Apache and Rsyslog. Meanwhile, we also tested the effectiveness of vulnerability verification with test cases from Juliet Test Suite. The result showed that our verification method achieved 98% accuracy with 0 false positives.

## 1 Introduction

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

Security patches are released to fix vulnerabilities. However, a patch of certain vulnerability often fails to propagate to code clones at other locations which, very possibly, present latent code clone vulnerability. Once a security patch is released,

* 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).

L. Batten et al. (Eds.): ATIS 2014, CCIS 490, pp. 109–122, 2014.

attackers could leverage patch information to dig out 0-day vulnerabilities and make great damage to 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 researched to detect security vulnerabilities. Static code analysis [17] [6] [7] has been proposed to discover vulnerabilities by analyzing source code or binary. The large coverage of code and the access to the internal structures makes this approach very efficient to find potential warnings of vulnerabilities. However, they often approximate or even ignore runtime conditions, which makes them suffer from high false positives.

Dynamic analysis monitors program execution to discover security flaws [1] [2]. These tools detect software vulnerabilities by generating input test cases and monitoring its run-time behavior. Although dynamic analysis reduces false alarms, it requires the generation of actual bug triggering test inputs which often make us cannot find critical security flaws in a reasonable time. Moreover, the coverage of the huge inputs space is either too much time costly or just impractical to achieve. Symbolic execution [4] has been widely proposed to detect vulnerabilities. However, as branches in programs increase, program paths increase exponentially. These approaches are either ineffective in path exploration or do not scale well to large programs.

In order to gain both preciseness and scalability in vulnerability discovery, we propose an approach which takes advantages of both static and dynamic analysis to discover code clone vulnerability based on released security patches. We first 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. Finally, we verify code clones to report real vulnerabilities using concolic(CONCrete + symbOLIC) testing [8] [11] which dramatically reduces false positives. We have conducted our experiments with real world open source projects such as Ubuntu OS distributions. In the results, there are 7 real vulnerabilities reported out of 63 code clones found in Ubuntu 14.04 LTS(recent release by then) within 7 hours to process nearly 260K source files. In one step further, we have found more code clones and confirmed more vulnerabilities in different versions of program packages. Meanwhile, we have also tested our vulnerability verification phase of our mechanism with test cases from Juliet Test Suite. The result shows that our verification method achieves 98% accuracy with no false positive and the average verification speed is 0.24s.

Our contributions could be described like this:

–**Combination of static and dynamic analysis**. We have developed a novel mechanism 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 data tracing**. The backward 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.

## 2 Related work

Previous researchers have proposed different approaches for static source code auditing. Some of them focus on detecting code clones [9] [10]. Deckard [10] and Deja vu [9] 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 done 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 [3]. Even though it can handle subtle code changes which may help them to find more code clones, this approach may suffer from high false positive rate.

Redebug [3] 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 which 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, due to a lack of automatic verification mechanism, most of un-patched code clones they reported are turned out not to be real vulnerabilities which gives them a super high false positive rate in terms of vulnerability detection.

Based on the shortness 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 and has shown good performance in detecting some vulnerabilities [12]. Concolic testing [8] [11] was proposed later 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 [8] 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 [8] which means, for some vulnerabilities, it is either impossible or too much time consuming to report them out. The greatest challenge for these approaches is the scalability problem. They still could not handle the path explosion problem properly. Moreover, when detecting software bugs or vulnerabilities, they usually consider each normal statement(such as memory copy, buffer access, arithmetic operations and etc) as a potential bug. This makes the concolic testing very time and resource wasting due to a huge input searching space, since only very small portion of the potential bugs turn out to be real security vulnerabilities in real world programs.

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 to assist concolic testing so as to mitigate the path explosion problem.

## 3 Proposed Mechanism

Discovery of vulnerabilities in a program is a key process to the development and management of secure systems. Security patches are released to fix already

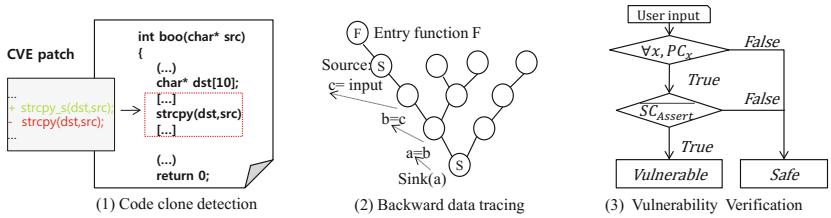


**Fig. 1.** General overview of our approach

found 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. In this paper, we call these vulnerabilities as code clone vulnerabilities.

Before we go into detail of our approach, the general process is illustrated in figure 1. We are first trying to find code clones by doing static syntax-based pattern matching in a scalable and efficient way and then 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 which helps us dramatically reduce false alarms in terms of vulnerability detection.

### 3.1 Finding Code Clones

Code clones could be described like this: if a same piece of vulnerable code occurs in any other locations or programs released. 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.

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

**Normalization of each file.** We do normalization by removing all non-ASCII character, redundant whitespaces, converting all characters to lower cases and braces.

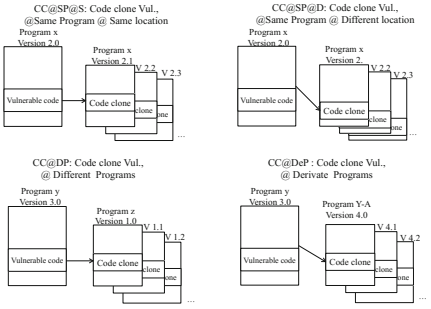


**Fig. 2.** Code clone vulnerability

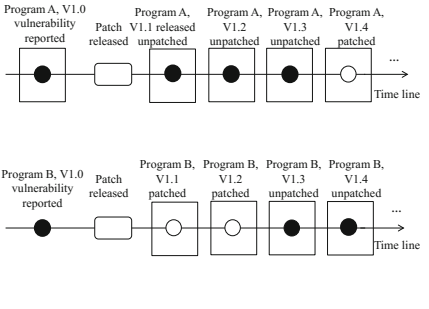


**Fig. 3.** Code clone vulnerability in the same program but different versions

**Tokenization of each file.** After normalization, each file is tokenized by each line. We define each line as one 't' (token).

**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 4-token window sliding.

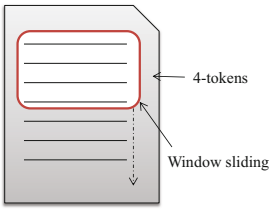


**Fig. 4.** Window sliding of 4-tokens

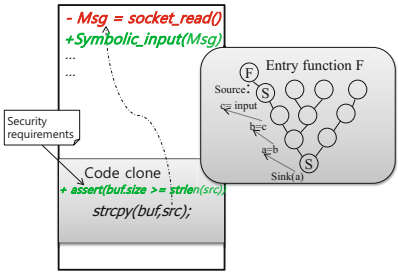


**Fig. 5.** Instrumentation of program source

Hence a file f($t_1$,$t_2$,$t_3$,...,$t_l$) could be represented as f($u_1$,$u_2$,$u_3$,...,$u_x$), where $x = l - n$ ($l$ is the number of lines in a certain file).

**Checking definition for code clones.** We extract original buggy code from a security patch basically by removing lines prefixed by '+' and adding lines prefixed by '-'. Then we regard this original piece of buggy code 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 above steps, we get a N-token set for each file and we define this n-token set for each file as $S = \{u_1,u_2,u_3,...,u_x\}$. We say $f_v$ is contained in $f$ when $S_v \subseteq S$.

**Fast membership checking.** Since, 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 [13] is well known as fast membership checking

which could be a very good choice to perform our task. Suppose there is a data set $S$, eg, 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, $Hash(u_1), Hash(u_2), ..., Hash(u_x)$. 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 testing source object is considered to be a preprocess for our concolic testing. It is done by backward source code analyzing and program source instrumentation. The stage mainly contains 2 steps: 1) make assertions 2)make symbolic inputs.

Assertions are to be made to set up security constraints before the potential vulnerable statements and symbolic input are used to generate different test cases for different execution paths. Before we expain how to actually do the instrumentation, we talk about some concepts and definition. In software, data flow can be thought as in water flow in aqueduct systems which starts from natural sources and ends to sinks [14].

**Security sinks:** Sinks are meant to be the points in the flow where data depending from sources is used in a potentially dangerous way. Several typical types of security sinks are shown below.

- Memory copy: The sensitive data is used as 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.
- Memory allocation: Memory allocation: The sensitive data is used as 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 the system.
- Arithmetic operations: The arithmetic operations may cause integer overflow, underflow or divided by zero problems.

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

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

**Security Constraints(SC):** Security constraints are clearly high-level security requirements. E.g., the length of the string copied to a buffer must not

**Table 1.** Security requirements for security-sensitive functions

| Security-critical func. | Security requirement |
|---|---|
| strcpy(dst,src) | dst.space > src.strlen |
| strncpy(dst,src,n) | (dst.space ≥ n) ∧ (n ≥ 0) |
| strcat(dst,src) | dst.space > dst.strln + src.strlen |
| printf(format, ...) | # formats = # parameters-1 |

exceed the capacity of the buffer. We need to define security requirements for statements like security-sensitive function parameters, memory access, integer arithmetic and etc(See Table 1).

**Backward sensitive data tracing:** We first identify *security sinks* and *sensitive data* in the code clone source. Then, we backwardly trace the *source* from the *sensitive data* to find the related input location. Afterwards, we instrument the program source to make assertions based on security requirements right before the security sink and replace the input statement with symbolic values. We could see this process from Figure 5.

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 which helps us to release the burden of our next stage.

### 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 [11], CUTE [1] and DART [5]. However, they suffer from path explosion problem which makes them cannot scale well to large real world programs. H.Li et al [15] has proposed variable backward slicing to analyze a program. This approach helps us to concentrate on those paths only related to sensitive sinks which 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 above concerns, we are trying to apply concolic testing in our code clone verification phase to help the verification of code clone vulnerabilities. The general principle of the verification is to find an input which satisfy all the program constraints(PCs) but violate the security constraints(SCs) as shown in figure 1. The concept of PC and SC could be find in H.Li et at [15]. In our scenario, we focus on the paths related to code clones rather than the countless number of paths in the whole program which could help us to mitigate the path explosion problem to a large extent. Our approach for concolic testing to verify code clones mainly follows a general concolic testing procedure [8](*Instrumentation,Concrete execution,Obtain a symbolic path formula,Select the next input values, Iterates back to execution*). However, the difference is that we are trying to generate 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

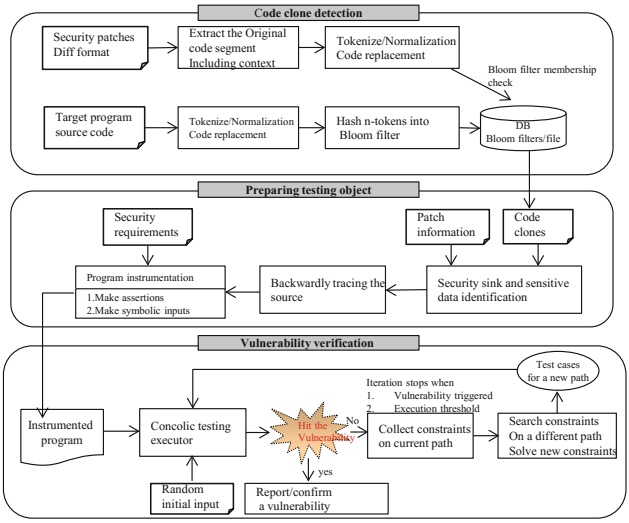


**Fig. 6.** The mechanism architecture

cost efficient when doing concolic testing. The detailed process is described in Figure 6.

In Figure 6, we can also see the detail mechanism architecture. Our mechanism consists of 3 phases: **Code clone detection**, **Preparation of testing source object** and **Vulnerability verification**. Our mechanism is used to discover un-patched code clone vulnerabilities in real world projects. We choose CREST-BV [9] as a basic concolic execution engine because of its good performance in test case generation speed. In the next part, we are going to talk about the implementation and experimental results.

## 4 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, 8GB memory, 512GB hard drive).

**Dataset:** For the security patches, we collected 106 security patches from 28 CVE [16] 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 2 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 OS distribution to test our mechanism. Based on the results, in one step further, we again collected various versions of linux packages in which code clones vulnerabilities have been found in the previous test such as Rsyslog and Apache

**Table 2.** Yearly distribution of collected CVE patches

| CVE patches | 2010 | 2011 | 2012 | 2013 | 2014 |
|---|---|---|---|---|---|
| Buffer overflow | 0 | 1 | 2 | 6 | 4 |
| Integer overflow | 1 | 0 | 2 | 0 | 7 |
| Buffer cased by IOS | 0 | 0 | 3 | 1 | 0 |
| Other | 0 | 1 | 0 | 0 | 0 |
| Total | 28 | | | | |

**Table 3.** Code clone detection results

| Target | CVE patches | Target src pool | # of files | # of reported code clones | Execution time |
|---|---|---|---|---|---|
| Src pool-1 | CVE patch pool (2010-2014, for C code) | Ubuntu 14.04 OS distribution | 259346 | 63 | 24812.5 sec (7 hours) |
| Src pool-2 | CVE patch pool (2010-2014, for C code) | Httpd-2.2.23 to 2.4.6 | 7820 | 14 | 738.6 sec (12.31 min) |
| Src pool-3 | CVE patch pool (2010-2014, for C code) | Rsyslog-5.8.13 to 8.2.1 | 1692 | 7 | 274.7 sec (4.57 min) |

trying to find more vulnerabilities in different program versions. What's more, in order to prove the efficiency of our vulnerability verification phase statistically, we collected 100 test cases from Juliet Test Suite. Juliet Test Suite is created by US National Security Agency's (NSA) Center for Assured Software which has been widely used to test the effectiveness of vulnerability detection tools.

### 4.2 Experimental Results

We have conducted our experiments with different target source pools.

**Target source pool-1.** Linux Ubuntu 14.04(latest version) OS distribution

We have found over 63 code clones in Linux Ubuntu 14.04(latest version) OS distribution. Table 3 shows the number of files processed and the execution time in detecting code clones

Our processing time is nearly 7 hours which means this experiment could be conducted in daily base. Among the 63 code clones, we have reported 7 real world vulnerabilities. Table 4 shows the detail information of the real vulnerabilities that we verified.

**Table 4.** Code clone vulnerabilities reported

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

In terms of vulnerability detection, our approach reported no false positives which is a huge improvement over other code clone detection approaches [3] [9] [10] without automatic verification.

**Target source pool-2 and 3.** Source packages of different program versions(Rsyslog and Apache).

Based on the result of the previous experiment, we are trying to look 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 and Apache and used them as target source code pool-2 and source code pool-3 respectively. For the Apache case, we collected 11 different versions from 2.2.23 to 2.4.6. Our mechanism processed totally 7820 source files (3642170 code of lines) in nearly 12.3 minutes. As a result, we have found 10 code clones and confirmed all of the 10 code clones to be actually vulnerable. We could see the detail from Table 3 and Table 5.

**Table 5.** Code clone vulnerabilities reported with source pool-2 (Apache)

| Version | # LOC | # of reported code clones | # of vulnerability found/verified | # of false positives |
|---|---|---|---|---|
| Httpd-2.2.23 | 350145 | 1 | 1 | 0 |
| Httpd-2.2.24 | 350256 | 1 | 1 | 0 |
| Httpd-2.3.6 | 209369 | 1 | 1 | 0 |
| Httpd-2.3.8 | 210564 | 1 | 1 | 0 |
| Httpd-2.3.11-beta | 219427 | 1 | 1 | 0 |
| Httpd-2.3.15-beta | 226497 | 0 | 0 | 0 |
| Httpd-2.4.1 | 223050 | 1 | 1 | 0 |
| Httpd-2.4.2 | 223265 | 1 | 1 | 0 |
| Httpd-2.4.3 | 223921 | 1 | 1 | 0 |
| Httpd-2.4.4 | 226000 | 1 | 1 | 0 |
| Httpd-2.4.6 | 233330 | 1 | 1 | 0 |

Actually, these code clones are detected from CVE-2011-3368.patch. From the result, we could see that after the release time of the CVE-2011-3368.patch, the Apache2 developers didn't 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 we have talked about in 3.1.

Similarly, we also collected different versions of Rsyslog and reported 7 affected versions. Table 3 and Table 6 showed the results.

Our mechanism processed totally 1692 source files(656708 code of lines) in nearly 4.57 minutes. 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. However, from the version Rsyslog-5.8.13, this code clone vulnerability re-occurred in another file /pmrfc3164sd.c due to the careless code re-use by developers. This case corresponds to the code clone

**Table 6.** Code clone vulnerabilities reported with source pool-3 (Rsyslog)

| Version | # LOC | # of reported code clones | # of vulnerability found/verified | # of false positives |
|---|---|---|---|---|
| Rsyslog-5.8.13 | 78937 | 1 | 1 | 0 |
| Rsyslog-5.10.0 | 78259 | 1 | 1 | 0 |
| Rsyslog-5.10.1 | 77811 | 1 | 1 | 0 |
| Rsyslog-6.6.0 | 92448 | 1 | 1 | 0 |
| Rsyslog-7.4.0 | 105324 | 1 | 1 | 0 |
| Rsyslog-7.6.3 | 111218 | 1 | 1 | 0 |
| Rsyslog-8.2.1 | 112711 | 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

```
1   static rsRetVal parse(msg_t *pMsg)
2   {
3       uchar *p2parse;
4       [...]
5               i = 0;
6               while(lenMsg>0&& *p2parse!= ':' && *p2parse!=' '&& i < CONF_TAG_MAXSIZE){
7                       bufParseTAG[i++] = *p2parse++;
8                       --lenMsg;
9               }
10              if(lenMsg > 0 && *p2parse == ':') {
11                      ++p2parse;
12                      --lenMsg;
13                      bufParseTAG[i++] = ':';
14              }
15
16              bufParseTAG[i] = '\0';
17              MsgSetTAG(pMsg, bufParseTAG, i);
18      }
19      [...]
20      CHKmalloc(pBuf = MALLOC(sizeof(uchar) * (lenMsg + 1)));
```

**Fig. 7.** Code clone vulnerability from Rsyslog.

type–CC@SP@D(see 3.1) and Figure 7 shows the code clone vulnerability in Rsyslog-7.4.0 .

**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 [18]. We have used Rsyslog-7.4.0 for our testing target. Both approaches have generated a triggering input and successfully verified CVE-2011-3200(see Figure 7) vulnerability in Rsyslog. However, Figure 8 has shown 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, in order to generate a triggering input for the vulnerability, our approach(concolic testing with backward tracing) has reduced the number of covered branches from 344 to 59 and has reduced the number of reached functions from 48 to 9. What's more, our approach only spent 1.2 secs to trigger

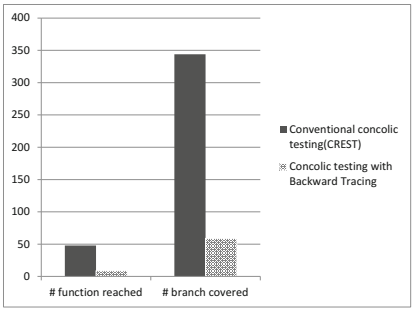


**Fig. 8.** The comparison with conventional concolic testing

this vulnerability while CREST took 24 mins. This indicates that, with backward data tracing, we can dramatically reduce the number of paths to traverse and decrease the input searching space which mitigates the path explosion problem.

### 4.3 Evaluations of Vulnerability Verification Phase

In order to prove the efficiency of our vulnerability verification phase, we collected 100 test cases from Juliet Test Suite. For every test case, there are "good" functions and "bad" functions which provides the ground truth for our evaluation. Our 100 test cases consist of different vulnerable types(see Table 7) and there are totally 250 spot to verify(100 bad functions and 150 good functions). We could see the results from Table 8.

**Table 7.** Distribution of test cases number

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

**Table 8.** Evaluation metrics of vulnerability verification on Juliet Test Suite test cases

| | True | False |
|---|---|---|
| Positive | 95 | 0 |
| Negative | 150 | 5 |

As we can see, our verification system generates no false positive and we got the verification accuracy of 98%. We also measured the average verification time needed to verify each test case, average verification time = 0.24s. This proves that our system has good verification accuracy and time cost effective.

## 5 conclusion

In this paper, we have developed a novel mechanism which combines the advantage of static and dynamic analysis to detect code clone vulnerability. Our

mechanism suggests a good performance for code clone vulnerability. What's more, by tracing the input from the sensitive data in code clones and preparing the testing source object, our approach performs concolic testing to do verification in a way that mitigates the path explosion problem. We conducted several experiments with different target source pools. The results showed our approach could find real world vulnerabilities with extremely low false positive rate within reasonable amount of time.

However, there are several concerns as well. First of all, some CVE patches patch in header files whose information is not enough to identify sensitive sinks. Sometimes, several patches contribute to one vulnerability case. This also makes us confused when we do verifications. Finally, due to the limitation of branch coverage of the concolic testing, we will have some false negatives in verification phase. In future research, we will look into classification of security patches and study about the searching strategy of concolic testing for higher branch coverage.

## References

1. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for C. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 263–272 (2005)
2. Haugh, E., Bishop, M.: Testing c programs for buffer overflow vulnerabilities. In: Network and Distributed System Security Symposium, pp. 123–130 (2003)
3. Jang, J., Agrawal, A., Brumley, D.: ReDeBug: finding unpatched code clones in entire os distributions. In: IEEE Symposium on Security and Privacy, pp. 48–62 (2012)
4. Ma, K.-K., Yit Phang, K., Foster, J.S., Hicks, M.: Directed symbolic execution. In: Yahav, E. (ed.) Static Analysis. LNCS, vol. 6887, pp. 95–111. Springer, Heidelberg (2011)
5. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Sigplan Conf. on Programming Language Design and Implementation (2005)
6. Wheeler, D.: Flawfinder (2011), `http://www.dwheeler.com/flawfinder`
7. Evans, D.: Splint, `http://www.splint.org`
8. Kim, M., Kim, Y., Jang, Y.: Industrial application of concolic testing on embedded software: Case studies. In: IEEE Int'l Conf. on Software Testing, Verification and Validation, pp. 390–399 (2012)
9. Gabel, M., Yang, J., Yu, Y., Goldszmidt, M., Su, Z.: Scalable and systematic detection of buggy inconsistencies in source code. In: ACM Int'l Conf. on Object Oriented Programming Systems Languages and Applications (2010)
10. Jiang, L., Misherghi, G., Su, Z., Glondu, S.: Deckard: Scalable and accurate tree-based detection of code clones. In: Int'l Conf. on Software Engineering, pp. 96–105 (2007)
11. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: USENIX Symp. 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 Int'l Conf. on Dependable Systems and Networks, pp. 241–250 (2010)

13. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet Mathematics 1(4), 485–509 (2004)
14. Sinks, http://code.google.com/p/domxsswiki/wiki/Sinks
15. Li, H., Kim, T., Bat-Erdene, M., Lee, H.: Software vulnerability detection using backward trace analysis and symbolic execution. In: Int'l Conf. on Availability, Reliability and Security, pp. 446–454 (2013)
16. Vulnerabilities, C.: Exposures cve., http://cve.mitre.org
17. Yamaguchi, F., Wressnegger, C., Gascon, H., Rieck, K.: Chucky: exposing missing checks in source code for vulnerability discovery. In: ACM SIGSAC Conference on Computer & Communications Security, pp. 499–510 (2013)
18. Burnim, J., Sen, K.: Heuristics for scalable dynamic test generation. In: IEEE/ACM Int'l Conf. on Automated Software Engineering, pp. 443–446 (2008)