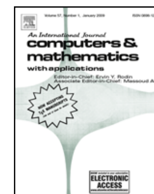
Computers and Mathematics with Applications 63 (2012) 469–480

Contents lists available at SciVerse ScienceDirect

Computers and Mathematics with Applications

journal homepage: www.elsevier.com/locate/camwa



Static program analysis assisted dynamic taint tracking for software vulnerability discovery

Ruoyu Zhang [a,*], Shiqiu Huang [a], Zhengwei Qi [a], Haibing Guan [b]

[a] *School of Software, Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China*

[b] *School of Electronic Information and Electrical Engineering, Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, China*

ARTICLE INFO

Keywords:
Taint analysis
Software vulnerability
Code coverage
Data flow analysis

ABSTRACT

The evolution of computer science has exposed us to the growing gravity of security problems and threats. Dynamic taint analysis is a prevalent approach to protect a program from malicious behaviors, but fails to provide any information about the code which is not executed. This paper describes a novel approach to overcome the limitation of traditional dynamic taint analysis by integrating static analysis into the system and presents framework SDCF to detect software vulnerabilities with high code coverage. Our experiments show that SDCF is not only able to provide efficient runtime protection by introducing an overhead of 4.16× based on the taint tracing technique, but is also capable of discovering latent software vulnerabilities which have not been exploited, and achieve code coverage of more than 90%.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

1.1. Motivation

In the last decade, software vulnerabilities have threatened computer security severely. Malicious users are able to gain access to confidential information inside the target program, even take control of it by taking advantage of design flaws. Take notorious buffer overflow as an instance, attackers can exploit this software vulnerability by manipulating the software input, and cause an overwrite in the stack in order to control the execution stream of the program [1].

Taint analysis is a prevalent approach to detect malicious behavior in recent years. Based on the concept that some data (such as the input from the user) is not trustworthy, taint analysis is proposed to keep track of the data which can be used to harm the software, and monitor suspicious actions. There exists several researches concerned with taint analysis and the details will be described in Section 2.

In current researches, the taint analysis technique is usually realized as a runtime method and referred to as dynamic taint analysis. Regardless of the implementation of the dynamic taint analysis, there is one limitation in current researches. Due to the fact that the dynamic taint analysis can only detect software vulnerabilities when the attack has been launched, it is impossible to locate the latent software weak spots, which is very desirable in many cases.

An example is presented in Fig. 1 to show the motivation of our approach. This vulnerability is exposed as *CVE-2007-6454*, and described with C++ code for clarity while our system is implemented to handle binary. *PeerCast* is an open source streaming media multicast tool. *HandshakeHTTP* is a procedure in *PeerCast* to handle http packages. The input package is controlled by users. The programmer attempts to copy the password and other information from the input package into two

* Corresponding author.
E-mail addresses: holmeszry@sjtu.edu.cn (R. Zhang), hsqfire@sjtu.edu.cn (S. Huang), qizhwei@sjtu.edu.cn (Z. Qi), hbguan@sjtu.edu.cn (H. Guan).

0898-1221/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.camwa.2011.08.001

```
    void Servent::handshakeHTTP(HTTP &http, bool isHTTP){
1       char *in = http.cmdLine;
2       if (http.isRequest("GET /")){
3           ...
4       }
5       ...
6       else if(http.isRequest("SOURCE")){
7           if (!isAllowed(ALLOW_BROADCAST))
8               throw HTTPException(HTTP_SC_UNAVAILABLE,503);
9           char *mount = NULL;
10          char *ps;
11          if (ps=strstr(in,"ICE/1.0"))
12          {
13              mount = in+7;
14              *ps = 0;
15          }else{
16              mount = in+strlen(in);
17              while (*--mount)
18                  if (*mount == '/')
19                  {
20                      mount[-1] = 0; // password preceeds
21                      break;
22                  }
23              strcpy(loginPassword,in+7);
24          }
25          if (mount)
26              strcpy(loginMount,mount);
27      }
28      ...
29  }
```

Fig. 1. A typical form of software vulnerability.

member variables of a *Servent* object by using library function *strcpy*. However, the procedure fails to verify whether the source string is longer than the destination string. It means there exists a vulnerability of heap overflow and the procedure allows remote attackers to cause a denial of service and possibly execute arbitrary code via a long source request. Line 23 and 26 are the vulnerable spots. However, the traditional dynamic taint analysis approaches cannot detect the potential threat when the code in these spots is not reached in execution.

In the field of dynamic taint tracking, there are also testing based techniques attempting to detect a potential security threat by improving the code coverage [2]. However, thorough static analysis is expensive and inaccurate [3]. High code coverage is difficult to achieve [4] till now, and the testing costs too much time.

1.2. Contribution

To address the problems described in the prior subsection, this paper proposes a novel approach, and realizes it as a framework. Moreover, we verify the practicality of the framework by building a vulnerability discovery tool on it.

Our contributions are summarized as follows:

- *Propose SDCF (Static and Dynamic Combined Framework) as a framework combining static and dynamic analysis.*
  This paper presents a novel approach composing the static and dynamic analysis to integrate their advantages, and provides SDCF as a framework for thorough analysis of the target program.
- *Implement a tool to detect latent software vulnerabilities.*
  We present and evaluate LSVD (Low-overhead Software Vulnerability Detector), an SDCF based tool to discover software vulnerabilities. LSVD cannot only detect software vulnerabilities being exploited at runtime, but also find the unexecuted code containing weak spots.

2. Related work

2.1. Dynamic taint analysis

Since most of the malicious users attack the software by manipulating the input, an intuitive approach to protect the software is monitoring the input from the user as tainted data [5–7]. Much attention has been drawn to suspicious data tracking with dynamic taint analysis. There are several binary instrument tools like DynamoRIO [8], Pin [9] and Valgrind [10] which can be used to facilitate taint tracking.

Newsome and Song [11] describe a dynamic-taint based approach to prevent overwrite attacks. Their approach taints any data read from a network socket which receives data from users. During execution, the approach monitors the binary program and guarantees that tainted data is not used as the destination of a control transfer instruction (such as jmp), a format string, or a system-call argument.

James Clause provides a generic dynamic taint analysis framework Dytan [5] and addresses several problems. Dytan is able to handle the data flow and control flow in its taint analysis. The system is also flexible and does not require any special support from the runtime system. LIFT [12] is also presented for the similar goal of providing a facility to keep an eye on the tainted data when the software is being executed.

Tools like IntScope [13] and Panorama [7] monitor the behavior of the target code by means of dynamic tainting. They are designed to detect integer overflow [14] and malware. Nguyen-Tuong and his colleagues [15] apply this approach in PHP-based web applications.

There are also researches applying the technique to generate test cases which are more well-targeted [2].

Generally, dynamic taint analysis supervises the behavior of the unsecured data at runtime to detect attacks aiming at the vulnerabilities of the software. However vulnerabilities cannot be detected by the technique until the target program is under attack.

2.2. Combining static and dynamic analysis

Since the static analysis is able to achieve high code coverage with low accuracy, and the dynamic analysis is just the opposite, there are already several researches attempting to integrate these two techniques in order to neutralize their drawbacks and maximize their advantages.

A direct way to compose the static and dynamic method is analyzing the target program in both ways and synthesizing the results. Ashish Aggarwal [16] presents a system on this principle. The system implements a static module to cover most of the code and testing based dynamic module to deal with the problems which can only be solved at runtime, such as pointer aliasing. The paper does not mention the time cost, yet it is difficult to achieve high efficiency by checking the code with both the static method and testing.

Recent research in this field shows great interest in directing test case generation with static analysis. Yannis Smaragdakis [17] has also made many contributions to static directed testing. He and his colleagues have proposed a series of tools [18] to generate the test cases with the assistance of static analysis. Saner [19] is a tool to detect vulnerabilities in the sanitization routines of web applications with a similar principle. Its efficiency evaluation shows that approximately only 50 lines of code can be analyzed per second on average.

Considering that dynamic analysis may incur great runtime overhead, Walter Chang and Calvin Lin [20] propose a solution invoking static analysis that identifies program locations where security policy violations might occur in order to reduce instrumented code. This solution requires source code to be recompiled which is not available for most of the vendor software. Different from SDCF, security violation is detected by their solution only when vulnerable code is executed.

Bouncer [21] applies dynamic analysis to static slicing for reducing false positives and summarizes library functions in order to reduce sliced instructions. Compared with SDCF, dynamic analysis is the secondary approach of static analysis.

BitBlaze [22] describes another form of the combination of the static and dynamic approach. The authors implement Vine, the static module of BitBlaze to translate the assembly instructions into intermediate language (IL) and gain important information such as control flow graphs (CFG). They implement their dynamic module TEMU for extracting OS-level semantics and user-defined dynamic taint analysis. BitBlaze integrates the two modules with Rudder which can leverage the information gathered by the previous modules to generate inputs traversing different execution paths.

3. System design

The presentation of the SDCF framework structure consists of 5 parts. The first part provides the overview of our general method and the architecture. Dynamic taint analysis and static analysis are described in the next two subsections followed by optimization detail. At last, the implementation of LSVD is briefly introduced.

3.1. Analysis architecture

Fig. 2 shows the system overview of the framework. All clients built on SDCF could define suspicious data as a taint source through Taint Source Interface, and track their propagation. An Illegal Behavior Definition Interface is provided for the client to define the dangerous behavior which the client may be interested in.

Binary Monitor is implemented on the dynamic instrumentation platform DynamoRIO [8] to monitor the target binary program dynamically. Dynamic Taint Analyzer is used to track the dynamic taint propagation during execution. At the same time, static analyzer analyzes the unexecuted part of the target program by the static approach. At last, the information about both executed and unexecuted parts of the program are provided to the client. The client on SDCF can acquire the information from the dynamic and static module, and this mechanism enables the client tools to apply analysis on the target program with high code coverage.

We apply an API filtering mechanism to filter uninterested API functions without analysis and improve the performance of the system. As most of the security attacks are aimed at the EXE module or third party libraries instead of other modules, the API Filter can be used to skip those API functions we are not interested in order to reduce the runtime overhead. However, the dynamic taint analyzer still works in these API functions to guarantee the accuracy of the taint tracking. The details of API filtering will be introduced in the following.

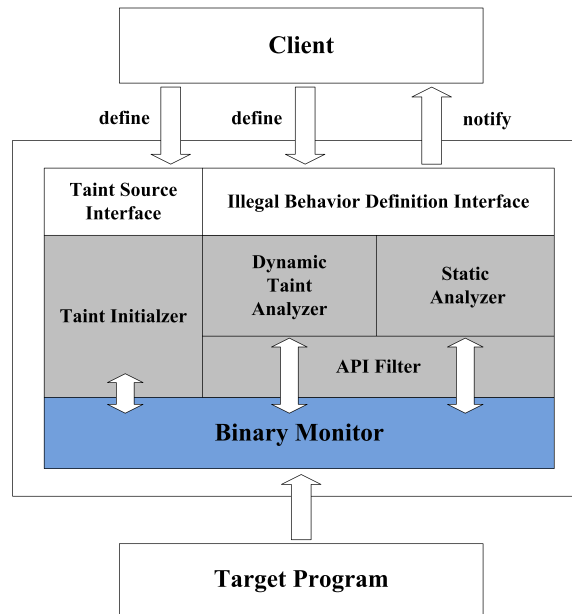


Fig. 2. Architecture overview.

3.2. Dynamic taint analysis

Dynamic taint analysis is an important part of our approach to analyze the program dynamically. Generally, this technique marks input data from unsafe channels as tainted, tracks the information flow and manages taint propagation as many similar works do [5]. In this way, the behavior of the target program can be analyzed and presented.

At the beginning, our approach locates the client-defined taint source in the memory and registers which are most likely referred to input data from some unsafe channels such as opening files and network packages. For instance, buffer overflow vulnerability detection tool LDCF is designed to locate the memory address of the taint source from opening files by tracking I/O-relevant APIs (such as *CreateFileW*, *CreateFileMapping* and *MapViewOfFile*) and mark the buffer that stores the data of taint source as tainted. For example, when a.txt document is opened, it is mapped to a block of memory with starting address 0x00b00000 and the length 4096 bytes. Taint Initializer gets the data of both 0x00b000000 and 4096, and marks the related memory area tainted.

After the taint source is located, information flow tracking begins to work. Information flow tracking is implemented on the instruction level. Up to now, we do not care about SSE/MMX instructions. Meanwhile ring0 instructions are also ignored because DynamoRIO cannot trace them in the kernel space. The remaining instructions can be grouped into two types according to their behaviors. One type of instruction can cause taint propagation such as the *mov* instruction and the push/pop instruction. Information flow tracking marks any data derived from tainted data as tainted when dealing with these type of instructions. The other type of instruction such as *cmp* and test instructions are considered not to affect taint propagation and are skipped in our framework.

For the first type of instruction, we only take care of the tainted or untainted states of source operands instead of their concrete values. Take the first *mov* instruction shown as follows for example, source operands are classified as general-purpose registers, memory locations and immediate values. Any tainted general-purpose registers and memory locations used in the source operand will make destination operands tainted. Otherwise, destination operands are marked untainted. However, one special situation should be taken into consideration. For the second *xor* instruction, no matter whether the source operand is tainted or not, the destination operand *eax* is set clean. In this case, *eax* is set untainted after the execution of this instruction.

```
mov    r32,    r/m32/imm32
xor    eax,    eax
```

In order to manage the taint propagation, memory and register models should be built to record tainted states of memory and registers. Because during the whole analysis process, most of memory addresses are untainted, and a chaining hash Table 3 is used to record memory tags (only tainted memory tags are recorded). Every node in the hash table represents one tainted memory byte in the target programs virtual address space. Normally there are 4096 slots in the hash table, and 125k nodes in a slot. The number of slots and nodes in a slot can be changeable. When there are 512 million slots and only one node in a slot, the hash table becomes the one-to-one mapping strategy. If any byte of memory is tainted, our approach

Table 1
Samples of APIs taint effects.

Function name	Module name	RetValueNum	RetValueLength	ParamNum	TaintType
GetModuleHandleA	Kernel32.dll	1	32	1	⟨1, ret⟩
SetLastError(NTDLL.RtlSetLastWin32Error)	Kernel32.dll	0	0	1	⟨1, null⟩
lstrlenW	Kernel32.dll	1	32	1	⟨1, ret⟩
CloseHandle	Kernel32.dll	1	32	1	⟨1, null⟩
RemoveDirectoryW	Kernel32.dll	1	32	1	⟨1, null⟩
UnmapViewOfFile	Kernel32.dll	1	32	1	⟨1, null⟩
CreateFileW	Kernel32.dll	1	32	7	⟨1, ret⟩
LoadStringW	User32.dll	1	32	4	⟨1, ret⟩, ⟨1, 2⟩, ⟨1, 3⟩
CharUpper	User32.dll	1	32	1	⟨1, ret⟩
_initterm	MSVCRT.dll	1	32	2	⟨1, ret⟩
wcsncpy	MSVCRT.dll	1	32	2	⟨2, 1⟩, ⟨2, ret⟩

Table 2
Coverage of SDCF in SPEC CINT2006.

Target program	Total BB	Executed BB	Complemented BB	Missed BB	Complement rate (%)	Coverage (%)
bzip2	24.7	12.7	8.5	3.5	70.8	85.8
gcc	16.8	10.6	6.1	0.1	98.4	99.4
mcf	27.3	12.9	9.4	5	65.3	81.7
gobmk	22.6	11.4	8.8	2.4	78.6	89.4
hmmer	20.7	10.8	6.5	3.4	65.7	83.6
sjeng	16.7	8.2	5.2	3.3	61.1	80.2
libquantum	26.2	13.8	8.3	4.1	66.9	84.3
h264ref	21.3	12.3	9	0	100	100
omnetpp	22.1	13.7	8.4	0	100	100
astar	17.6	10.8	6.2	0.6	91.2	96.6
Average	21.6	11.7	7.6	2.2	79.8	90.1

Table 3
Vulnerability detection result.

Target program	Attacks	Attacks detected	Normal test	Latent bugs detected	Source
BufferAttacker	11	11	41	41	Example program
TxtEdit	13	13	20	7	Example program
IrfanView 4.25	31	31	12	3	CVE-2010-1509
Foxit Reader 3.0 build 1120	22	22	23	17	CVE-2009-0836, CVE-2009-0837

handles it as follows: First, find which slot it belongs to by its memory address. Second, create a node in the slot, and record its memory address in the node. If a piece of tainted memory is cleaned, we delete the nodes in the table.

In Register Management, normally 1 bit is used to represent the state of a register (1 means tainted and 0 means not). However, there is a special relationship between registers: some registers are part of others. That means when the state of a register is changed, other registers may be affected. Let us take this short code for an example:

```
mov    eax,    ecx
mov    ax,     4
```

Assume that *ecx* is tainted. After the first instruction, *eax* is tainted, and *ax*, *ah*, *al* should all be tainted too. At the second instruction, an immediate is moved to *ax*, so *ax* is cleaned. Because ah and al are part of *ax*, at this time, they should all be untainted. However, *eax* is still tainted. For accuracy, that relationship must be taken into consideration.

3.3. Static analysis

The goal of unexecuted information supplement is to complement unexecuted code and information within the function of the target program statically during the process of dynamic taint analysis.

Fig. 3 presents the algorithm of this module to supplement unexecuted code. The static analyzer is working during the process of dynamic taint analysis. While the program is running, CG (Call Graph) is being built and for each function in the CG, the CFG (Control Flow Graph) of executed code will be constructed. At the same time, branch points like conditional *jmp* will be added to the branch point list of the function for the use of static completion analysis. After the function is visited by the dynamic taint analysis, all branch points will be traversed, and the structure of the target program will be constructed by applying the algorithm described in Fig. 3. For the sake that different parameters passed into the function may lead to different control flow and therefore different discovery of CFG, CFG is built each time a function is called. Considering that the static analysis is applied to intra-procedure as mentioned later, it is efficient to generate CFG on-the-fly and it is proved by our experiments. As shown in Fig. 4, SDCF also generates visualized CFG for the user for manual analysis.

Algorithm: *Static Completion*

```
Input: branch point list:
        P {p1,p2,p3...}
        CFG: B {b1,b2…}
                //basic block set
                E {e1,e2...}
                //edge set
procedure StaticAnalysis
n=1
for branch point: pn in P
  for each new b=GetNextBB();
  //get next unexecuted basic block
    if b not in B
      if last instruction is condition
branch
          add new p' into P
      end if
      add b into B;
      add new E into E;
    end if
n++;
```

Fig. 3. Algorithm of unexecuted information supplement. For each function in the Call Graph, the unexecuted information supplement will traverse the branch point list and complement unexecuted branch paths statically.

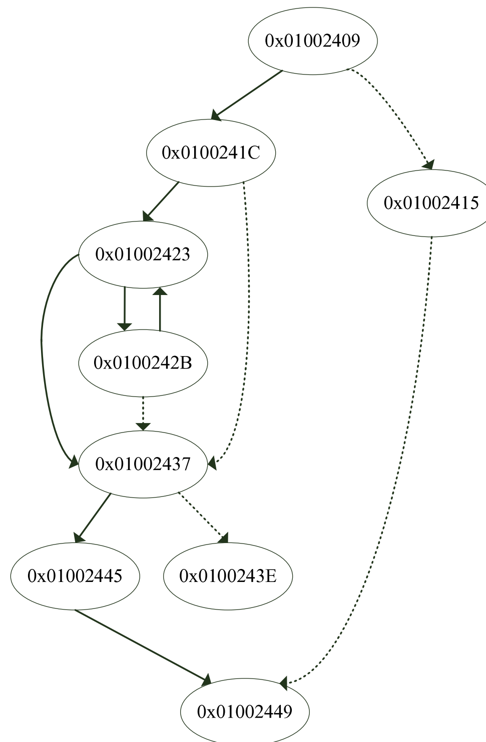


Fig. 4. CFG of a function generated by SDCF. The concrete path indicates the execution stream, and the dotted ones are supplemented by the static analyzer.

Since the structure of the program is acquired, the static analyzer is able to apply taint analysis on the unexecuted part of the code which cannot be reached by the dynamic taint analysis module. Similar to dynamic analysis, static analysis is context-sensitive and it analyzes the whole CFG by traversing each unexecuted path of it. As the experiments show in the first of the fourth section, this part of the code is usually not very large. Moreover, with the help of dynamic analysis, many of them can be skipped by API summaries as discussed in Section 3.4 because they are irrelevant to the tainted data. It is notable that the static analysis in SDCF is more accurate than most of the other static approaches because the runtime context information such as memory and register concrete values at the entry of the unexecuted code is achieved during dynamic analysis. However, this information also limits the scope of the static analysis, because they can be dissimilar in different instances. Although the static analysis is not sound enough due to the lack of accurate context information of the

$$\mathrm{TR_n} \neq \emptyset \wedge (< Dst, Src > \in \mathrm{TR_n}) \wedge \mathrm{Dst} \neq \emptyset \rightarrow \mathrm{TS}' = \mathrm{TS} \cup \mathrm{T_{Dst}}$$
$$\mathrm{TR_n} \neq \emptyset \wedge (< Dst, Src > \in \mathrm{TR_n}) \wedge \mathrm{Dst} = \emptyset \rightarrow \mathrm{TS}' = \mathrm{TS} - \mathrm{T_{Src}}$$
$$\mathrm{TR_n} = \emptyset \rightarrow \mathrm{TS}' = \mathrm{TS}$$

Fig. 5. The taint behavior policy of API filter. In these formulas, TR stands for the taint relation in the target function, which are represented as taint vectors ⟨Dst, Src⟩. TS is the set of the tainted data in the system scale, which records all the locations which are tainted, and should be monitored. T_{Dst} and T_{Src} are the taint marks of the corresponding variables.

unexecuted code and it may still produce both false negatives and false positives, it has greater advantage of scalability and performance.

We implement this part of the system focused on individual functions and ignore the effect of cross-functions. On one hand, tracking a program path within cross-functions statically will cause a lot of runtime overhead, which makes the program analysis framework unpractical. On the other hand, if it is desirable to analyze across the function, the cross-function analysis can be realized in the client tools by integrating the information about individual functions provided by SDCF.

3.4. Optimization

According to the common observation, a large part of the binary codes in software is loaded from the system library such as kernel32.dll, USER32.dll and ntdll.dll. The behavior of these modules is predictable, so it is not necessary to check every instruction in them. Walter Chang and Calvin Lin [20] bring forward an approach to eliminate unnecessary dynamic tracking with the help of inter-procedure static analysis. However their approach requires a powerful static analysis with inter-procedure pointer analysis that leverages semantic information. Bouncer [21] is a tool that reduces sliced instructions by summarizing library functions with symbolic execution while state explosion is a serious problem of symbolic execution. Different from these solutions, we propose a simple, intuitive and efficient solution to optimize the analysis technique by summarizing the taint effects of the API functions and skipping them.

In the first place, source code or documents of APIs are examined and their taint propagation information is gathered. Some API taint effects samples are shown in Table 1. Column RetValueNum and ParamNum represent the return value number and parameters number of each API. RetValueLength represents the length of the return value. TaintType shows the taint propagation of each API. For example, ⟨1, *ret* ⟩ means the tainted state propagates from parameter 1 to the return value and ⟨1, 2⟩ means tainted state propagates from parameter 1 to parameter 2. After taint effects of API functions are summarized, some principles are designed to provide the taint behavior policy of an API filter. (1) When a function can propagate the tainted data among its parameters and return a value, then the destination of the tainting should be marked as tainted and brought into the supervision of the system. (2) When the tainted data can be eliminated within the function, then the system will mark them as clean ones after this function has ended. (3) When a function does not do anything in the taint propagation, then the taint status of the program does not change, and this function will be considered as a taint irrelative one. These policies are described in Fig. 5.

With the principles mentioned above, taint irrelative API functions can be free from instrumentation of information flow tracking code. However, some functions cannot be simply skipped, such as *strcpy*, which has a tight relationship with some software vulnerabilities. Therefore, this kind of API function is still necessary to be analyzed in the information flow tracking.

3.5. LSVD

The vulnerability discovery tool LSVD is designed to detect the vulnerabilities regardless of whether it is executed or not. With dynamic taint analysis and static analysis in SDCF, important information of both executed and unexecuted code are reported to the client for analysis. LSVD makes use of the information to discover not only the possible weak points in the executed part of the program, but also the latent software vulnerabilities which have not been executed. It is realized with the conception that tainted data cannot be used as a parameter of the string handling APIs (such as the *printf* family) or source operand of repeat move instructions (such as *REP MOVS*) without checking its length. Because the length checking is difficult to identify, most of the traditional methods treat all these dangerous behaviors as potential threats [12].

Since LSVD is built as a client on top of SDCF, the taint source and illegal behavior are defined to detect vulnerabilities.

By the default setting, the inputs from the users are initialized as the source of the taint propagation. LSVD also takes data loaded from outside of the executable file as untrustworthy.

Although much work has been done to judge whether or not the security and privacy of the program has been jeopardized, there is still no mature theory to determine exactly what behavior will cause the trouble. The completeness and soundness of current taint sink policy in the state of the art cannot be guaranteed yet. Based on this fact, we conclude the illegal taint related behavior (which is also referred as taint sink) as follows.

- *Control transfers* whose destination address is determined by the tainted data are dangerous.
  The most intuitive approach to take direct control of the victim is by manipulating the target address of the control transfer instructions, such as *jmp*, *call* and *ret*. For example, malicious users usually cause buffer overflow to overwrite the system reserved stack, and modify the return address of the current function. By this means, the users are able to change the control flow of the program at will.

- *Repeat move instructions* are a potential threat to the target program when *ecx* is controlled by the outside data. *rep movs* is an instruction to move the data repeatedly, and the time of the move is indicated by *ecx*. If *ecx* has been controlled by the users' input, these instructions may devastate the execution of the program. By analyzing the contents in the target program, the users can overwrite the data which are not supposed to be accessed.
- *Several APIs* such as *strcpy* and *sprintf* are frequently exploited by malicious users.
  Our approach is implemented on binary code, and it is quite natural to define the illegal behavior at the instruction level. However, a common observation is that large proportions of existing vulnerabilities are caused because of the inappropriate use of the APIs, many of which include an incomplete check of the parameters. Take *strcpy* as an example, this function copies the data between locations indicated by the parameters, but when the destination is not big enough to store the whole content from the source, an overflow has been induced. The overflow can cause an overwrite to a location other than the one the destination parameter indicates. Except for unauthorized data access, this behavior can also result in manipulation of control flow.

3.6. Case study

This case study is performed on the weakness coming from *CVE-2007-6454* which is shown as our motivation in Fig. 1. The programmer firstly checks whether the input http package is a source request and then attempts to copy the password and other information from the input package into two member variables of a *Servent* object by using library function *strcpy*. The procedure allows remote attackers to cause a denial of service and possibly execute arbitrary code via a long source request. Line 23 and 26 in Fig. 1 are the vulnerable code.

Fig. 6 is the control-flow graph of the source code. In order to detect the vulnerability mentioned above, our approach will mark the memory area of the user-controlled http object as tainted. Assume that the input http package is constructed arbitrarily and it may not reach the vulnerable code. It means that the vulnerable code is not executed at runtime and not analyzed by SDCF dynamically. During the process of dynamic analysis, the call graph is constructed as well as the dynamic control-flow graph of the procedure *handshakeHTTP*. The condition statement in Line 6 of Fig. 1 takes the false value when executed. After the condition statement is analyzed, the remaining code along the unexecuted branch is constructed and analyzed by our approach. The whole control-flow graph of the binary code is shown in Fig. 7. SDCF tracks the taint propagation statically by using the taint analysis result before the condition statement. When the statements in Line 23 and 26 of the source code, or in the basic block starting with the address 0x0040158E and 0x004015AD, is reached, a report containing a detected heap overflow vulnerability is made by using the policy mentioned before.

4. Evaluation

Since SDCF is designed to discover the latent flaws in the target software, there are generally three concerns about it:

- *How much of the target program can be covered for analysis in the framework*?
- *How is the performance of the system*?
- *Is SDCF capable to detect software vulnerabilities, even when they are not actually executed*?

Therefore, the evaluation of our system is divided into three subsections to answer these questions respectively. We demonstrate the efficiency and the effectiveness with the experiments on SPEC CINT2006, a set of benchmarks widely used in performance evaluation.

4.1. Coverage

To evaluate the analysis coverage for target programs, SPEC CINT2006 benchmarks are used to evaluate SDCF in 6 aspects, including total BB (total basic block), executed BB (number of dynamically executed basic blocks), complemented BB (number of complemented basic blocks), missed BB (number of basic blocks which are not recognized in the static procedure), complement rate and coverage (coverage rate). Total BB, executed BB, complemented BB and missed BB are the average number of the corresponding BB in analyzed functions. The complement rate reflects how many basic blocks are complemented compared to all unexecuted basic blocks. Coverage depicts executed and complemented basic block percentage of all basic blocks in the target program.

Table 2 shows the result of coverage evaluation and it presents the average of the BBs of the functions in each benchmark. On average, the complement BB rate of SDCF is 79.8% and coverage reaches 90.1%. The reason why they are not 100% is that SDCF cannot handle the indirect control transfer which depends on the context.

4.2. Performance

Fig. 8 presents the normalized execution time (the ratio of our time to native execution time) of SDCF applied on the SPEC CINT2006 benchmarks. SDCF incurs 4.16 times overhead on average and the overhead without the optimization of API filtering is 6.51. Compared to the taint analysis tools such as LIFT (slows down target programs by 3.6 times), Dytan (50 times), Panorama (20 times), our system incurs relatively low runtime overhead which benefits from our optimization mechanism.

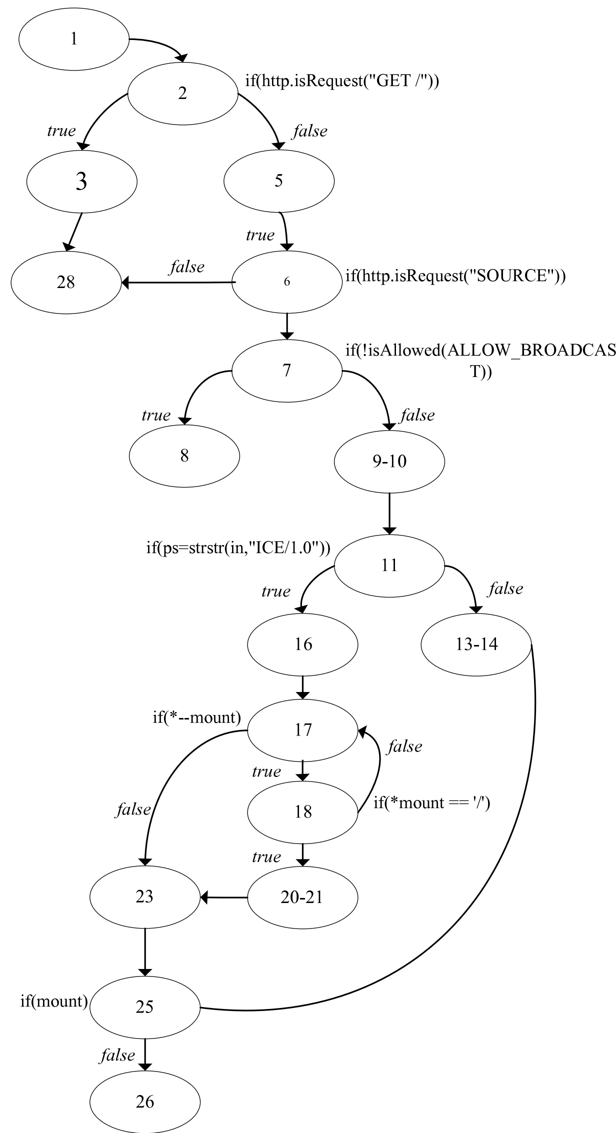


Fig. 6. The control-flow graph of the vulnerable procedure's source code.

4.3. Vulnerability detection

In this part of evaluation, we test the SDCF based software vulnerability detection tool *LSVD* on some programs to validate the practicability of our framework, and Table 3 provides the results.

IrfanView is a free graphic viewer for the *Windows* platform and *Foxit Reader* is a widely used document processor, in both of which buffer overflow vulnerabilities are reported [23]. *BufferAttacker* and *TxtEdit* are example programs containing typical buffer overflow weak spots in their code.

The "Attacks" and "Attack Detected" volume show the number of test cases which cause buffer overflow and whether they are detected. With our mechanism described at the end of Section 3, all these malicious behaviors are discovered. The "Normal test" and "Latent Bugs Detected" demonstrate the number of normal test cases which do not bring out buffer overflow and if the weak spots will still be found.

In SDCF, static analysis will only be applied based on the structure of functions, so it only acquires static information of the functions being executed, and skips the ones without being called in the execution. It is quite obvious that the latent bug detecting rate in IrfanView and TxtEdit is lower than the other two, because the functions which contain vulnerable code may be not even called in these programs.

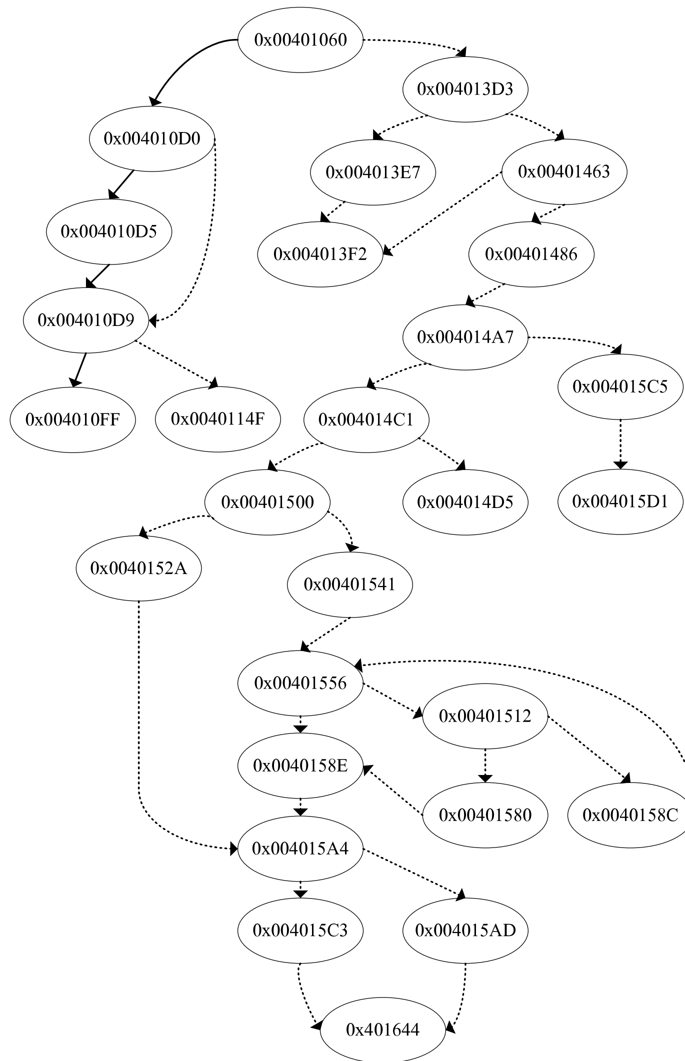


Fig. 7. The generated control-flow graph of the vulnerable procedure's binary code.

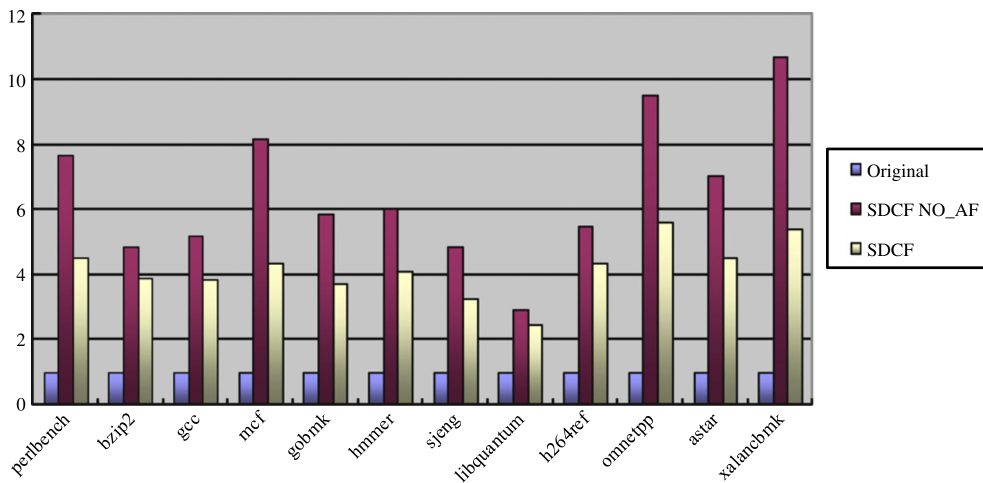


Fig. 8. The performance of SDCF. The measurement is normalized with "Original" execution of the benchmarks in the SPEC CINT2006. "SDCF" and "SDCF NO_AF" presents the execution time with and without API filtering mechanism respectively.

5. Discussion

In this section, we discuss strengths and limitations of our approach SDCF.

5.1. Strengths

The main strengths of SDCF are as follows:

- *Low overhead*.
  High runtime overhead is a prevalent limitation of many existing dynamic analysis tools. Benefited from DynamoRIO, the dynamic instrumentation framework and our optimization module, SDCF is able to achieve better performance. DynamoRIO significantly reduces the overhead of context switching between the original code and the instrumented code by basic block optimization. And we implement the API filter to decrease the time wasted on taint irrelative code, and make our system more practical.
- *High code coverage and precise analysis result*.
  Compared to traditional dynamic taint analysis approaches, SDCF is capable of analyzing with high code coverage. With the assistance provided by the static analysis engine, unexecuted paths of programs are brought in for analysis. Meanwhile, runtime information such as concrete value and branching address is made use of for a more precise analysis result than traditional static analysis tools.

5.2. Limitations

The main limitations of SDCF are as follows:

- *Coarse static taint analysis*.
  Dynamic taint analysis is sound, however, static taint analysis is unsound because the value of the memory object is unknown and indirect addressing results in undetermined memory address during the static taint analysis. It is difficult to define a precise static taint propagation model. According to our case studies, it is effective enough of the static taint analysis engine to detect software vulnerabilities with high performance. The reason is that our static analysis technique is the supplement of the dynamic taint analysis and it is applied to inner-procedure analysis.
- *Incomplete and unsound sink policy*.
  As mentioned in Section 3, our taint sink policy is incomplete and unsound, which may cause some false negatives and false positives. In the current stage, the illegal behavior definition interface is designed by SDCF for a new sink policy defined by the client application. In this way, our approach is very flexible in vulnerability detection. One of our future works is to improve the existing sink policy by summarizing common malicious behavior, and we are looking forward to a mature theory being established on this subject.

6. Conclusion

In order to discover the latent software vulnerabilities before they are exploited, we provide a novel approach to combine the static and dynamic analysis. The SDCF framework is presented to demonstrate our methodology and a buffer overflow discovery tool LSVD is built on it to validate the practicability of the framework. The result of the experiments on benchmarks shows that the system is both efficient and effective. In the future, program slicing will be applied to optimize the framework. Symbolic execution is also considered to be implemented to extend the utility of the framework.

Acknowledgments

This work is supported by *Key Lab of Information Network Security*, *Ministry of Public Security*, *National Natural Science Foundation of China* (Grant No. 60873209, 60970107, 61073151), the *Key Program for Basic Research of Shanghai* (Grant No. 09JC1407900, 09510701600, 10511500100, 10DZ1500200), *IBM SUR* Funding and *IBM Research-China JP* Funding.

References

[1] CWE. http://cwe.mitre.org/.
[2] V. Ganesh, T. Leek, M.C. Rinard, Taint-based directed whitebox fuzzing, in: International Conference on Software Engineering, ICSE, 2009.
[3] P. Zuliani, A. Platzer, E.M. Clarke, Bayesian statistical model checking with application to simulink/stateflow verification, in: International Conference on Hybrid Systems: Computation and Control, HSCC, 2010.
[4] P. Godefroid, M.Y. Levin, D.A. Molnar, Automated whitebox fuzz testing, in: Network and Distributed System Security Symposium, NDSS, 2008.
[5] J.A. Clause, W. Li, A. Orso, Dytan: a generic dynamic taint analysis framework, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, 2007.
[6] O. Tripp, M. Pistoia, S.J. Fink, M. Sridharan, O. Weisman, Taj: effective taint analysis of web applications, in: Programming Language Design and Implementation, 2009.
[7] H. Yin, D.X. Song, M. Egele, C. Kruegel, E. Kirda, Panorama: capturing system-wide information flow for malware detection and analysis, in: ACM Conference on Computer and Communications Security, CCS, 2007.

[8] D. Bruening, T. Garnett, S. Amarasinghe, An infrastructure for adaptive dynamic optimization, in: International Symposium on Code Generation and Optimization, CGO, 2003.
[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, in: PLDI, 2005.
[10] N. Nethercote, J. Seward, Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proc. of 2005 Programming Language Design and Implementation, PLDI, 2007.
[11] J. Newsome, D. Song, Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software, in: Network and Distributed System Security Symposium, NDSS, 2005.
[12] F. Qin, C. Wang, Z. Li, H.-S. Kim, Y. Zhou, Y. Wu, Lift: a low overhead practical information flow tracking system for detecting security attacks, in: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2006, pp. 135–148.
[13] T. Wang, T. Wei, Z. Lin, W. Zou, Intscope: automatically detecting integer overflow vulnerability in x86 binary using symbolic execution, in: Network and Distributed System Security Symposium, NDSS, 2009.
[14] D. Brumley, D.X. Song, T. cker Chiueh, R. Johnson, H. Lin, Rich: automatically protecting against integer-based vulnerabilities, in: Network and Distributed System Security Symposium, NDSS, 2007.
[15] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, D. Evans, Automatically hardening web applications using precise tainting, in: 20th IFIP International Information Security Conference, 2005.
[16] A. Aggarwal, P. Jalote, Integrating static and dynamic analysis for detecting vulnerabilities, in: 30th Annual International Computer Software and Applications Conference, 2006.
[17] Y. Smaragdakis, C. Csallner, Combining static and dynamic reasoning for bug detection, in: TAP, 2007.
[18] C. Csallner, Y. Smaragdakis, DSD-crasher: a hybrid analysis tool for bug finding, in: Proceedings of the 2007 International Symposium on Software Testing and Analysis, ISSTA, 2006.
[19] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, G. Vigna, Saner: composing static and dynamic analysis to validate sanitization in web applications, in: IEEE Symposium on Security and Privacy, 2008.
[20] W. Chang, B. Streiff, C. Lin, Efficient and extensible security enforcement using dynamic data flow analysis, in: Proceedings of the 15th ACM Conference on Computer and Communications Security, CCS'08, ACM, New York, NY, USA, 2008, pp. 39–50.
[21] M. Costa, M. Castro, L. Zhou, L. Zhang, M. Peinado, Bouncer: securing software by blocking bad input, in: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP'07, ACM, New York, NY, USA, 2007, pp. 117–130.
[22] D.X. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M.G. Kang, Z. Liang, J. Newsome, P. Poosankam, P. Saxena, Bitblaze: a new approach to computer security via binary analysis, in: International Conference on Information Systems Security, ICISS, 2008.
[23] CVE. http://cve.mitre.org/.