2014 Ninth International Conference on P2P, Parallel, Grid, Cloud and Internet Computing

Program Crash Analysis based on Taint analysis

Zhang Puhan[1],Wu Jianxiong[1],Wang Xin[1],Zehui Wu[2]
[1]China Information Technology Security Evaluation Center, Beijing,China
[2]State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, China
zhangph2008@gmail.com,wuzehui2010@foxmail.com

***Abstract*—Software exception analysis can not only improve software stability before putting into commercial, but also could optimize the priority of patch updates subsequently. We propose a more practical software exception analysis approach based on taint analysis, from the view that whether an exception of the software can be exploited by an attacker. It first identifies the type of exceptions, then do taint analysis on the trace that between the program entry point to exception point, and recording taint information of memory set and registers. It finally gives the result by integrating the above recording and the subsequent instructions analysis. We implement this approach to our exception analysis framework ExpTracer, and do the evaluation with some exploitable/un-exploitable exceptions which shows that our approach is more accurate in identifying exceptions compared with current tools.**

Keywords—Software engineering; crash analysis; taint analysis; exception classification

I. INTRODUCTION (*HEADING 1*)

Current software operators commonly used error-reporting mechanism for stability maintenance of the released software, as shown in Figure 1. Software developers receive such a report which is often a sample or a memory dump when the software crash and it cannot perform live trace debug. So it is very necessary how to quickly provide software crash-related information for software maintenance personnel. For example, software usually crash to an instruction, and we need to determine whether the crash is caused by internal logic error of the program or the external input data. If the cause is an external input, it is likely to be a serious crash, even an exploitable vulnerability. At this time we need to know which fields of the inputs are related to the data which cause this instruction an error, in order to further supply the judgment basis whether a crash can be used as vulnerability. Therefore, the ultimate goal of the software crash analysis is to determine whether the current software crash could be exploited by attackers. This paper uses data flow oriented analysis methods to directly analyze the binary program, and analyze whether the crash point can be controlled by an attacker to achieve crash threat classification to provide fix information for software maintenance personnel.

Current researches on the analysis of software crash determination mainly focus on buffer overflow and format string. Two representatives are "! Exploitable" plugin [1, 2] of Microsoft winDBG, and AEG (Automatic Exploit Generation) [3]. And the exploitable is used as Windbg plugin, when the program crash, using load MSECD.dll to load exploitable plugins, and then use "! Exploitable-v" commands to check the exploitability analysis results of current crash. Exploitable will divide the crash into exploitable, may be exploitable, may be not exploitable and the unknown to measure the degree of crash exploitability. This plugin is generated after Microsoft's security personnel analyzed ten million crashes on vista and found that many crashes have something in common. Although the path is not reachable which some crashes triggered, the root cause is the same, so the crashes which appear in one code area can be classified as a class. When it is implemented in reality, we classify the crashes by collecting the stack information of crash points and use implementation of the exceptions to Type outliers stack by collecting information, and use the primary hash and secondary hash to commutate stack frame information of crash point, and then according to the hash value classify crashes which are caused by the same defect to one category, and thus determine the exploitability of the crash point. But the plugin can only give an accurate crash judgment under windows, for other third-party software and some of the more complex crashes such as heap overflows, UAF (Use-After-Free), etc. often give false results.

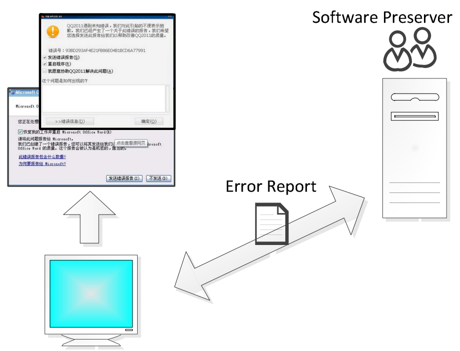

Fig. 1 Error reporting of software exceptions.

AEG is different from exploitable. It uses symbolic execution[4, 5] technology to get constraints of execution paths, and then use the constraint solver, in order to determine the exploitability of crashes. To be specific, first the target program source code is compiled into a binary program using the GCC compiler, and the binary program is to be tested by AEG. Then use the LLVM compiler to compile target program into byte code object, and the byte code is to be analyzed by AEG. Traverse the execution paths in source code level by symbolic execution technology. When crash occurs, AEG

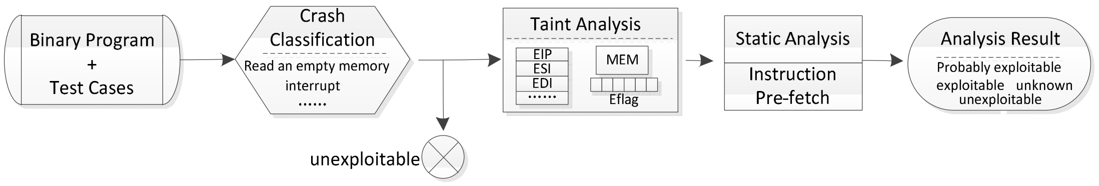

Fig. 2 Work flow of ExpTracer.

collects the path's constraint expression which reach the crash point, and solve specific input parameters by which the program can trigger vulnerability, then analyze the parameters to judge whether the crash is exploitable. However, because of the usage of symbolic execution technology, when there is a cycle occurs, it will lead to "the path explosion" phenomenon. The analysis efficiency will drop sharply, at the same time false positives rate raise up. And the tool can only determine the format string and buffer overflow crashes.

In addition, Dawn Song et al proposed crash exploitability judgment technology based on path signature[6].The technology extracts the signature of the path from program entrance to vulnerable point to obtain vulnerability type. When make crash exploitability judgment, first extract signature, and make a judgment by matching crash types. The method may lead to too many crash types, and the result depends on the match list of vulnerabilities, whose practicality is not enough.

In this paper we sum up the previous experience, and use a data flow guiding analysis technology based on taint analysis to judge software crash exploitability, according to the need of practical program analysis. As shown in Figure 2, the method identifies the crash type, screens and rejects crashes which are not exploitable, and then records taint propagation path and whether memory, register is tainted by analyzing the taint analysis path from program entrance to crash point. For instructions which change the control flow, we not only analyzes the taint situation of the current instruction, but also analyze whether the EIP register is tainted to give a more accurate analysis. At the same time the subsequent instructions are analyzed, to judge exploitability of current crash. We use binary instrumentation platform PIN to realize the research ideas -- the ExpTracer prototype system, and select opened loopholes to compare with Microsoft "! Exploitable". The results show that the method is more accurate in the identification of crashes, especially the writing cover type ones.

Contributions

1. We proposed a new dynamic taint analysis based on static optimization on DBI (Dynamic Binary Instrument) platform, by which we could effectively improve the efficiency of taint analysis.

2. We proposed a practical method to analysis instructions in the subsequent traces by instruction pre-fetch, which could highly improve the accuracy of crash determination.

3. We have built a crash determination framework named ExpTracer, and we have shown how this work could provide basic research foundation for vulnerability model study by specific case study.

Roadmap

Abnormal classification in the second part introduces ExpTracer the paper, in the third part introduces ExpTracer fine-grained stain was optimized by using static analysis algorithm, the fourth part introduced ExpTracer framework and function modules, the fifth part gives the results of the experiment, and analyze the results. The sixth part is the summary of the thesis, insufficient and future research work.

II. Crash Classification

Before determining crash availability, in order to reduce the scope for further determination and improve the efficiency of system execution, we need first to remove those crashes which are known not exploitable, then further classify crashes which are exploitable maybe. In this payer, we divide crashes into the following types by crash messages generated during system exception and types of instructions which lead to crashes.

A. NULL Point Deference

A null pointer is the pointer whose memory unit value is NULL, and crash occurs when using the pointer. Null pointers usually has two causes: first, because the release version is the wrong version of programs to initialize pointers, it often reports "fail to read data, memory address is NULL", but this problem does not appear in the debug version. These crashes can be resolved through compatibility check and static compilation. Two is in the internal procedures the logical processing incorrect, leading to zero the pointer and read error, which is a logical error. But the use of this two reason cannot change the program's control flow, therefore we determine "read empty memory" is not exploitable.

B. Direct Jump Instruction

For the situation that crashes come up when executing the jump instructions (such as JMP, CALL instruction), differently from using the approach of taint analysis such as TaintCheck[8], the system first determines whether the destination address is tainted. If the destination address is tainted, it needs to further determine whether the EIP register is tainted. Only when destination address and EIP register are both tainted, can the system determine the crash is exploitable. If the jump instruction is not tainted, the system will analyze whether subsequent instructions contain the ones which may change the control flow or not.

C. Memory/Register Modify Instruction

When crash instructions are memory / register modification instructions (such as ***mov***), it needs to determine whether the source operand is tainted and how many bytes are tainted. At the same time track where the tainted memory/register is

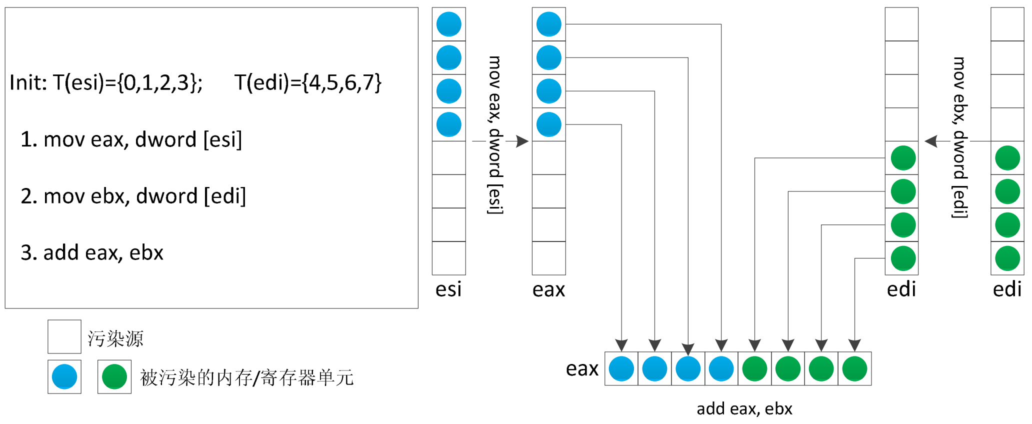

Fig.3 Work Flow of Taint Analysis

used, and whether information leak would occurs. If possible, crash is exploitable. Then analyze the subsequent instructions by the static analysis, and checking whether the tainted memory/register has affected the subsequent jump instructions. If there is, the control flow is likely to be hijacked, and the crash is also exploitable.

D. Interrupt Instruction

Usually, in order to protect control flow from being hijacked, the compiler will insert some interrupt instructions at compile time (0xCC). So when the process crashes, the crash point often appears "CC" command. The judgment on this type of crashes is difficult to realize automation, and usually these crashes are hard to use, because the control flow is relatively difficult to be changed. This type of crashes will be determined as "may be able to exploitable".

III. DATA FLOW DIRECT ANALYSIS METHOD

We adopt data flow analysis to improve the efficiency of data flow analysis. As a kind of fine-grained taint analysis based on static optimization, it can not only determine whether a memory unit/register is contaminated, but also be able to identify the offsets of taint sources that tainted the target memory or register, so as to identify the relationship between user input and software crash point. As shown in table 1, the first column represents the three order instructions. Among them, the variable T is used for taint variable, which means T (eax) ={0,1,2,3} induce that register eax is tainted by the first four bytes (offset 0,1,2,3) of taint source.

As shown in figure 3, the length of taint source is 8 bytes. Initially, the memory units referred by esi is tainted by the first four bytes of taint source, while the other four bytes taint memory units referred by edi. As you can see that after the first two mov instructions, eax is tainted by the first four bytes of taint source, ebx is tainted by the last four bytes. However, the instruction add will merge the information of eax and ebx both into eax, making the eax rely on the whole 8 bytes of taint source. The taint process record is shown in table 1.

But current fine-grained taint analysis methods still adopt the way that analysis instructions one by one, ignoring the taint transmission relationship between instructions, which increase a lot of extra overhead[9, 10, 11]. As shown in figure 3a, the sequences of instructions complete the assignment between memory operations. If we analysis this by instruction instrument one by one, the correspond taint spread relations is shown in figure 2b. T (eax) indicates that the eax register is pollution, "< -" represent the spread of taint. On the other hand, the final goal of the instruction sequence is spread the taint information of memory units referred by ebx to which referred by edi, which has nothing to do with eax. But in the actual taint analysis process, current methods use four taint spread relationships to represent the process, among these the second and forth of them are superfluous, the first and third can be combined.

Table 1 Taint Propagation Record

Instruction	T(eax)	T(ebx)	T(dword [esi])	T(dword [edi])
	{}	{}	{0,1,2,3 }	{4,5,6,7}
mov eax,[esi]	{0,1,2, 3}	{}	{0,1,2,3 }	{4,5,6,7}
mov ebx,[edi]	{0,1,2, 3}	{4,5,6, 7}	{0,1,2,3 }	{4,5,6,7}
add eax,ebx	{0,1,2, 3,4,5,6, 7}	{4,5,6, 7}	{0,1,2,3 }	{4,5,6,7}

On the basis of previous studies, we proposes a fine-grained taint analysis based on static optimization algorithm. We extract the semantic information of taint propagation through static analysis, deleting the instructions which has nothing to do with taint propagation, and merging the spread of the repeat order. Furthermore, according to the feedback of static analysis, the dynamic analysis will complete the specific taint analysis.

(1) Static Optimization for Dynamic Taint Analysis

A. Intermediate Representation

We design the intermediate representation language of taint propagation as follows.

In this IR, we have two operators ∪ and &, the former used to represent taint information combination, while the latter used to set the length of operator, for example 0x1&eax can get the last byte of eax, if the default length of operator is 4.

B. Non Taint Propagation Instruction elimination

As we know there are many non-taint propagation instructions during taint analysis, all these instructions could be eliminate by the following way. Firstly, we split our target into basic blocks, each block is constructed by many entrance point and only one exit point. Secondly, we set the input collection as ψ_I and the output collection as ψ_O. Lastly, we search and delete instructions that satisfy the express Inst$\notin\psi_O$.

```
1 mov eax, dword ptr ds:[ebx]
2 add eax, 0x10
3 mov dword ptr ds:[edi],eax
4 mov eax,ecx
…
```

（a）Instructions of assignment operator

```
1 T(eax)<- T([ebx])
2 T(eax)<- T(eax)
3 T([edi])<-T(eax)
4 T(eax)<-T(ecx)
…
```

（b）Corresponding taint propagation relationship

Fig.4 Taint Propagation Relationship before Optimization

```
reg :== variable name
mem : == '['mem-expr']'
const :== const value
var :== reg|mem|const
or-opr :== ∪
assign -opr :== &
mem-expr :== (reg|const)|{' * ' | '+' | '-' }(reg|const);
expr :== var | expr{var binary-opr}|'('expr')'
statement : (reg | mem ) assign-opr expr
```

Fig.5 The grammar of intermediate representation

C. Repeat Propagation of Taint Information

We set basic blocks as a sequence constituted by $state_1$, state2, state3…$state_n$. The taint express of each $state_i$ is represent by $oprand_i$<=exp_i. Based on this, we get the algorithm on repeat propagation of taint express, which is shown in algorithm 1.

With regard to assembly instructions in figure 6, after the static optimization, the taint propagation could eventually be simplified into the four transmission in figure 6c, and can be instrument before the fifth line, by which we could complete the whole fine-grained taint analysis process of seven instructions only by one instrument.

Using the above optimized taint analysis algorithm, we could effectively shorten the time of taint analysis and accurately identify the corresponding taint source, which provide detailed decision basis for crash exploitability determination.

```
Algorithm 1： RepeatPropTaint
Input: $block = state_1 state_2 ... state_n$
Output: $block' = state_1 state_2 ... state_k$ , $k <= n$
begin
1.  for i=1 to n do
2.    flags=false
3.    $state_i = (oprand_i \Leftarrow \exp_i)$ //get current taint express
      //if the left value of the express outside
4.    if $oprand_i \notin OUT(DAG)$
5.      for k=i+1 to n do    //traverse subsequent taint express
6.        $state_k = (oprand_k \Leftarrow \exp_k)$
          // $state_k$ and $state_i$ could be Repeat Propagat
7.        if( $oprand_i \in \exp_k$ )
8.          $state_k = (oprand_k \Leftarrow \exp_k')$
9.          flags=true  //set Flags of Repeat Propagation
10.       endif
11.     endfor
12.   endif
13.   if (flags)
      //if have repeat propagated, then delete it
14.     delete(block, $state_i$)
15.   endif
16. endfor
end
```

```
…
L1 mov eax, dword ptr ds:[ebx]
L2 add eax, 0x10
L2 mov dword ptr ds:[edi],eax
L3 mov eax,dword ptr [ebp+0x10]
L4 pop edi
L5 pop esi
L6 test al,al
L7 jnz 0x43002176
…
```

（a）Assembly instruction

```
…
L1 T(eax)<- T([ebx])
L2 T(eax)<-T(eax)
L2 T([edi])<-T(eax)
L3 T(eax)<-T([ebp+0x10])
L4 T(edi)<-T([esp])
L5 T(esi)<- T([esp])
L6 T(eax)<-T(eax)
L7
…
```

（b）Before Optimization

```
…
L1
L2
L2 T([edi])<-T([ebx])
L3 T(eax)<-T([ebp+0x10])
L4 T(edi)<-T([esp])
L5 T(esi)<- T([esp])
L6
L7
…
```

（c）After Optimization

Fig.6 The Optimization progress of Mov

(2) Crash Instruction Analysis

From crash classification we know that we could not give an accurate result to "memory/register modification instructions". So we need a further step to analysis, and we category this type into three classes as follows.

$$\boxed{\begin{array}{ll} mov\ \ reg8/16/32,\ byte/word/\text{dword}/\text{ptr}[mem] & (Type1) \\ mov\ \ byte/word/dword/ptr[mem],\ reg8/16/32 & (Type2) \\ mov\ \ reg8/16/32,\ reg8/16/32 & (Type3) \end{array}}$$

Fig.7 The combination type of Mov

For instruction ***mov***, it has three types because of non-directly copy of two memory units on x86 platform. However, when come to taint analysis, we only need to analyze Type1 and Type2. For Type1, $reg8/16/32 \Leftarrow byte/word/\text{dword}/\text{ptr}[mem]$, the particular memory unit will be copy to target register, so we need to record that if the memory unit is readable. And if it is true, then we need to analyze the length of the tainted memory, from which we could determine the exploitable of current crash. For example, if the whole 4 bytes are all tainted, then we could get the control of a memory space which is as large as 4G, inside which we could insert any dll making our exploit be success.

For Type2, $byte/word/dword/ptr[mem] \Leftarrow reg8/16/32$, the particular register will be copy to target memory units, so we need to record that if the memory units is writable. And if it is true, then we need to analyze the taint information of source operand, from which we could determine the exploitable of current crash.

IV. Crash Exploitable Determination Framework: ExpTracer

Crash exploitability determination is ultimately to determine whether EIP register can be controlled. The most direct way to determine whether EIP register is polluted is taint analysis techniques, and another way is to extract crash patterns through pattern matching method. In this paper, we complete the base portion ExpTracer, which classify the crash and then identify contaminate memory/register via taint analysis according to the different types of crash, based on which, we can determine whether the crash exploitable. Subsequently, under this framework we can extract patterns of different crash respectively, and perform more accurate crash determination through pattern matching to implement the extension of the ExpTracer framework.

A. Type Identification

For a binary program and a sample which can trigger crash point, we gather information of crash point by simulating execution and then triggering crash, and identify the type of crash by the error and command information of crash point system prompts when triggering crash. For "read empty memory" crash, because of its control flow cannot be exploited directly and we give "non-use" judgment result directly; while for "break" type of crash, because the program itself has a protection mechanism to control flow, given control flow is more difficult to hijack, we assume that it is "possible to use." For other types of crash, they need to be determined by further taint analysis.

B. Taint Analysis

In this paper, the taint analysis module contains two parts, static module and dynamic module. The static module is used to optimize the stain, and the dynamic module takes the results of static optimized result to conduct concrete implementation. Static Module will first converter the instruction to an intermediate representation based on which we complete the non-pollution instruction directive and the optimization of duplicate spread of pollution. The optimized result is returned to the dynamic taint analysis module. Dynamic taint analysis extracts taint source based on the sample, complete the instruction stub and record of taint propagation, construct and analyze real-time updates tainted record sheet, at last taint analysis results will be submitted to the crash determination module.

C. Instruction Pre-fetch

Typically instruction at the crash point can not change the control flow of the program, so it is difficult to achieve exploit. The exploit point of exploitable crash can be before crash points and can also be after the crash points. The former requires specific analysis by backtracking method, and the latter requires to analyze the instructions after the crash points. This paper only considers the case in which the exploit point is after the crash point, and reads the instructions after the crash point in a coarse-grained way, from which we extract the instructions which can change control flow of the program, such as call, jmp, etc. We perform static taint analysis on this type of Instructions to see the taint condition, and if the destination address is tainted, we think that it is possible to exploit. On the one hand because of the uncertainty whether the path up to that point, on the other whether the EIP instructions can be tainted.

D. Exploitable Determination

Exploitability determination module uses a different determination method on different types of instructions, according to the instruction type of crash points. For the jump instruction is mainly based on the results of the taint analysis, to see whether the destination address of the jmp instruction is tainted, and can be polluted by which bytes of taint sources. If they can be tainted, and EIP registers can be hijacked, then you the control flow of the program can be changed to achieve crash exploit. For memory / register modification instructions, we need to first determine whether the source operand can be tainted and if they can, then further trace taint condition of destination operand. If we detect taint spreads to the instruction which can change the program control flow, then it shows that the crash is also exploitable.

V. Evaluation

A. Experimental Environment

Table 2 Testing environment

Environment	Type	Configuration and Version
HardWare	Processor	Intel(R) Core(TM) i5-2300 CPU @ 2.80GHZ
	Memory	DDR3-4.00GB
Software	OS	Windows XP Professional SP0-SP3, Win2000 SP4 Virtual Machine
	DBI	Pin v2.11-43611

NO.	Vulnerability NO.	Crash Instruction	Taint Source (Byte)	"!exploitable"	ExpTracer
1	MS06-040	call ds:_imp_wcscat	128	exploitable	Probably exploitable
2	MS08-067	mov ecx, dword ptr ss:[ebp+8]	128	exploitable	exploitable
3	CVE-2011-2130	movzx eax, word ptr[eax+1ch]	64	Unknown	exploitable
4	CVE-2011-2595	mov ecx, dword ptr ds:[eax]	128	Unknown	exploitable
5	CVE-2013-2551	mov ecx, dword ptr[eax+14h]	256	Probably exploitable	Probably exploitable
6	CVE-2013-0753	movzx eax, word ptr[ecx+4ah]	128	Unknown	exploitable
7	CVE-2011-0609	call dword ptr[ebp+68h]	64	Unknown	exploitable
8	CNNVD-201310-129	mov ecx, dword ptr ss:[ebp+34h]	256	Probably exploitable	Unknown
9	CNNVD-201309-301	movzx eax, word ptr[ecx+34h]	128	Unknown	Unknown

Note: respectively from the Microsoft security center, CVE vulnerability database[12] [13] [14] and the national vulnerability database selection has publicly loophole, corresponding to the number of MS, CVE, CNNVD respectively. To "fly" abnormal crash and exploits the distance to use point address, abnormal points instructions for use of point; For abnormal is not available, the distance is up; Usually exploits point after the crash point, but also can appear before, after negative.

B. *Experimental Result*

From table2 we could get a direct point that ExpTracer could identify more crashes of third-part software than exploitable, especially for crashes that change control flow. And the table also show that UAF (Use-After-Free) vulnerabilities could not be determined accurately, because the POC (Prof Of Conscept) has many memory rewrite instructions, which leading a complicated logical result. So ExpTracer recognize this type as "probably exploitable". Figure 9 shows that "exploitable" take any unrecognized crash as "Unknow", while ExpTracer not, and for crash that occurred by stack overflow, our method has more accuracy. Figure 10 shows that the overhead of ExpTrace is more high than "exploitable", however, because we are offline analysis, and compared with manual analysis, the time consuming is acceptable.

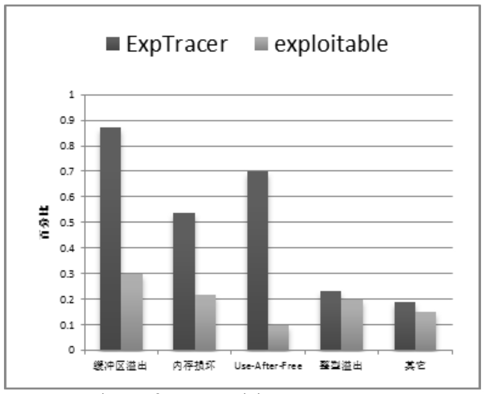

Fig.8 Comparing of Recognition Accuracy

From the exception type, divided into two aspects of data flow analysis research of binary anomaly judgment. Although able to identify the abnormal availability to identify the type of more, but it is still a coarse-grained anomaly judgment method,

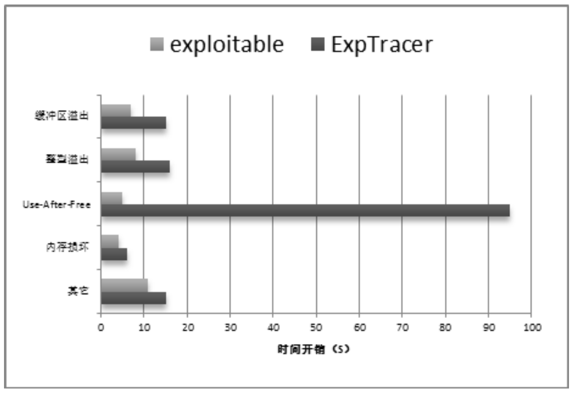

Fig.9 Comparing of Time Consuming

VI. Conclusion

Especially for abnormal for some logical relationship is relatively complex, still can't accurate judgment result is given. Especially to "fly" and abnormal points in the case of MOV instruction cannot path for effective analysis of the abnormal points later, just by the disassembly, instruction identification methods, such as not on subsequent path analysis, the flow of control will be the focus of future research.

Acknowledgment

This work is supported by the National High Technology Research and Development Program of China (863 Program) (2012AA012903,2012AA012902),the National Natural Science Foundation of China (61272493).

References

[1] http://blogs.technet.com.
[2] http://msecdbg.codeplex.com/.

[3] Avgerinos T, Cha S K, Hao B L T, et al. AEG: Automatic Exploit Generation[C]//NDSS. 2011, 11: 59-66.

[4] Jee K, Kemerlis V P, Keromytis A D, et al. ShadowReplica: efficient parallelization of dynamic data flow tracking[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013: 235-246.

[5] Schwartz E J, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)[C]//Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010: 317-331.

[6] Abadi M, Budiu M, Erlingsson Ú, et al. Control-flow integrity principles, implementations, and applications[J]. ACM Transactions on Information and System Security (TISSEC), 2009, 13(1): 4.

[7] Yamaguchi F, Wressnegger C, Gascon H, et al. Chucky: exposing missing checks in source code for vulnerability discovery[C]//Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013: 499-510..

[8] J. Newsome, D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software[C]//Proceedings of Network and Distributed System Security Symposium(NDSS). USA, 2005.

[9] Wartell R, Mohan V, Hamlen K W, et al. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code[C]//Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012: 157-168.

[10] Jee K, Portokalidis G, Kemerlis V P, et al. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware[J]. Proc. of the 19th NDSS, 2012.

[11] Zeng B, Tan G, Morrisett G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing[C]//Proceedings of the 18th ACM conference on Computer and communications security. ACM, 2011: 29-40.

[12] MS. www.technet.microsoft.com/zh-cn/security/default.

[13] CVE.www.cve.mitre.org

[14] CNNVD. www.cnnvd.org.cn