An Improved Method on Static Binary Analysis to Enhance the Context-Sensitive CFI

Qintao Shen, Lei Luo, Jun Ma, Jie Yu, Qingbo Wu, Yongqi Ma, Zhengji Liu

Abstract**—***Control Flow Integrity* (CFI) is one of the most promising technique to defend Code-Reuse Attacks (CRAs). Traditional CFI Systems and recent *Context-Sensitive* CFI use coarse control flow graphs (CFGs) to analyze whether the control flow hijack occurs, left vast space for attackers at indirect call-sites. Coarse CFGs make it difficult to decide which target to execute at indirect control-flow transfers, and weaken the existing CFI systems actually. It is an unsolved problem to extract CFGs precisely and perfectly from binaries now. In this paper, we present an algorithm to get a more precise CFG from binaries. Parameters are analyzed at indirect call-sites and functions firstly. By comparing counts of parameters prepared before call-sites and consumed by functions, targets of indirect calls are reduced. Then the control flow would be more constrained at indirect call-sites in runtime. Combined with CCFI, we implement our policy. Experimental results on some popular programs show that our approach is efficient. Further analysis show that it can mitigate COOP and other advanced attacks.

Keywords**—**Contex-sensitive, CFI, binary analysis, code reuse attack.

I. Introduction

CONTROL-FLOW INTEGRITY (CFI) [1] has been developed as one of the most promising techniques to stop code-reuse attacks (CRAs). Since all control-flow transfers in a program should act as designed in the original program, CFI provides a strong guarantee to protect running tasks. Unfortunately, CFI techniques are suffering from many difficulties when implemented actually accounting for troubles in static analysis at indirect transfers, especially without source codes. With the improving of code reuse techniques, even the existing fine-grained CFI systems could often be bypassed.

With a control flow graph (CFG) extracted from binaries, call relations are analyzed and every possible target is collected for those instructions leading to indirect control flow transfers, such as *ret* and indirect calls, etc. Most traditional CFI systems that work at binary-level adopt binary rewrite technique to insert validation codes into the original programs. However most of them only focus on the indirect transfers by validating that whether their targets is contained in the legitimate set collected by static analysis or not. Without the runtime control-flow context, the knowledge of execute path generated by branch instructions can not be attained. Mistakes may occur when deciding which transfer target is legitimate. For instance, if the function *D* is called by functions *A*, *B* and *C*. The return target of *D* will be one of all next instructions of call-sites in functions *A*, *B* and *C* and it is hard to determine. If we do not know exactly about the caller, we can not determine where *D* should return, which leave many gaps for attackers.

Context-Sensitive CFI (CCFI) [2], proposed recently, takes all transfers into consider by monitoring the branches of executable programs in runtime, which mitigates the problem. PathArmor achieved the goal of CCFI by using Last Branch Record (LBR) registers, the intel hardware feature, to capture the branches to analyze the runtime context.

CFI heavily depends on CFG. However, extracting a sound and accurate CFG at instruction-level from binaries is an unsolved problem. Therefore CFI systems have to enforce their policies with coarse CFGs. But coarse CFGs weakens the CFI systems to defend the advanced attacks. The existing CFI systems, including state-of-the-art traditional CFI [3]–[9] and CCFI, are encountering security problems.

Advanced attack techniques, e.g., *Counterfeit Object-Oriented Programming* (COOP) [10] and *Control-Flow Bending* (CFB) [11], have the ability to make full use of function existing in program binaries to enforce function-reuse attacks. The key factor that leads those exploitations effective is that coarse CFGs lose control of call targets at indirect call-sites. Once the control flow of a program has been hijacked, the attacker could jump to any function with an indirect call. More seriously, COOP has been proved Turing complete, which means that every Turing-computable problem can be resolved if exploited with such technique.

Dozens of methods and platforms [12]–[14] have been proposed to analyze the program binaries to extract their CFGs. They mainly could be divided into two categories, dynamic and static. The use of dynamic methods will face the problem that the resulting CFG may not be complete. At the same time, it may even suffer from errors because of lacking runtime conditions of stack and heap. In contrast, static methods, which will enumerate all possible paths to construct a complete CFG, will encounter the problem of path explosion. All methods mentioned above will consume considerable extra memory and computing resources.

In this paper, we try to mitigate the problem mentioned above by proposing a static analysis algorithm, which restricts the targets of the indirect calls in coarse CFG. Then we implement our CFI policies with PathArmor. We write a proof-of-concept program to help to analyze the security of our algorithm. Lastly, a bunch of typical programs are chosen to evaluate the effect of our algorithms.

The remainder of this paper is organized as follows. Section II covers advanced function reuse methods and some

Lei Luo is with the National University of Defense Technology, China (e-mail: l.luo@nudt.edu.cn).

Qintao Shen, Jun Ma, Jie Yu and Qingbo Wu are with the National University of Defense Technology, China.

Yongqi Ma and Zhengji Liu are with the Institute of Computer Application, China Academy of Engineering Physics, China

state-of-the-art CFI systems. Section III proposes our idea and algorithm. Section IV describes our implementation of the algorithm. Section V analyzes our experiment result on several typical programs and the security of our CCFI policy which combines our algorithms with PathArmor. Section VI concludes the paper.

II. Background

CRAs techniques have developed for a long time since the first exploitation proposed in 1997 [15], and bring great challenge to system security. The origin CFI was believed to have excellent security, but it and its' following CFI systems are all facing the problem that precise CFG is difficult to extract. The existing static analysis techniques on binaries consumes too much memory and computation resources, therefore simpler methods are adopted by CFI techniques.

A. Code Reuse Attacks

Code reuse attacks (CRAs) have developed for a long term since the first *ret2libc* technique published in 1997. *Ret2libc* [16] can find and take advantage of the function in the shared libraries and then obtain a shell by calling the *system()* function.

Return-Oriented Programming (ROP) [17], which can realize arbitrary computation without code injection over any sufficiently large program codebase, became popular after 2007. In ROP, code in shared libraries and also in program binaries can be reused. A series of short sections of code, ending with the low-level instruction *ret*, named gadgets, can be chained together into a coherent exploit. ROP is so strong and flexible that the automated attack methods and tools using ROP appeared quickly.

Jump-Oriented Programming (JOP) [18] was developed after ROP. Compared with ROP, JOP can take both gadgets ending with *ret* and gadgets which end with indirect *jmp*. Moreover, both ROP and JOP have been proved Turing complete, too.

COOP is a binary-level function reuse technology proposed recently to exploit C++ programs. Virtual functions and virtual tables are features of C++ language. At binary-level, virtual functions are all special function pointers recorded in virtual tables, which means that program will do indirect calls when using virtual functions. Based on the analysis of C++ programs, COOP finds that there exists many virtual functions with different effects on memory and registers (Program Counters, Parameter Registers. etc.). Object with virtual function table could be counterfeited by COOP to control where and when the selected virtual functions, which are called vfgadgets (virtual-function gadgets), should be loaded and executed.

CFB can achieve an attack vector even when an precise fine-grained CFI system is deployed. In the code space of program, what CFB do firstly is to find a path to *system()* in CFG. And then the control flow would be diverted down to this path, and the protection system would be bypassed successfully after executing *system()* lastly.

From the *ret2libc* to the advanced COOP and CFB, the reuse of existing binary code has become increasingly fine-grained and more and more carefully craft. The key point is that all those code reuse techniques using by attackers is based on the important fact that we do not know exactly how the control flow changes at indirect transfer points.

B. Control-Flow Integrity

CFI is a natural idea to protect programs against CRAs, which was believed to achieve excellent security. Although the origin CFI was put forward as early as 2005, it has not been widely applied in industry yet. Regardless of performance factors, the security of existing CFI systems is still questionable. Recent papers [10], [11], [19]–[22] show that for both coarse and fine grained CFI systems, there is the possibility of being bypassed.

Coarse-grained CFI systems [3], [5] do not require precise CFGs. And they only check that whether the targets of *ret* and indirect *call* are in the legitimate collection according to result of static analysis, when the control flow transfer is indirect. For example, in a coarse-grained CFI policy, the legal destination addresses of all *ret* instructions are any of the next instruction after a call, and all entry sites of functions in the binary are allowable when doing a indirect call instruction, etc. Such loose policies taken by coarse-grained CFI systems result in that carefully crafted gadgets can still be found and used to break the CFI.

Fine-grained CFI [4], [6], [7], [9] systems also suffer from approximate CFGs, especially in large-scale programs. In fine-grained CFI, the call relations between functions will be calculated firstly and then the functions are assigned with unique labels. Those labels will be checked before functions return to make sure that the return is correct. Because of the coarse CFG used in the CFI systems, attackers may succeed in mining a path from vulnerability code to a system call. Even a best formed Fine-grained CFI with the most restrictive policy can not stop all attacks.

Existing coarse and fine grained CFI systems are both far from desired effect. One of the most important reasons is that the context semantics are missing when program is running. PathArmor which realized a practical context-sensitive CFI mitigates the problem with the help of hardware (LBR). With a partly collection of control flow transfers, PathArmor will validate groups of execution paths, which occur during the runtime of program. By searching paths in CFG, PathArmor can judge whether the path is legitimate.

C. Binary Analysis

There are many famous platforms to analyze program binaries, such as IDAPro [23], BAP [13], angr [24], etc. One of the main problems to recover CFG completely and precisely from low-level binaries is indirect branch instructions.

Many programs try to address the problem through static analysis, in which filed symbolic execution is one of the popular methods. In symbolic execution, formulas and symbols need to be constructed based on binary code, and paths in CFG can be calculated with some algorithms such as enumeration of all possible transfers.

A precise CFG may be obtained by symbolic execution, only if the size of target program is not too big. Otherwise it would bring out another problem called *path explosion*. Plenty of memory and CPU resources would be needed even if a suitable size program is analyzed using this algorithm.

Since the performance cost is another problem in CFI systems, it is not a good choice to analyze binaries with static methods described above.

III. Methodology

The biggest difference between CCFI and other CFI systems is the way to validate the running state.

Traditional CFI systems always examine whether the next transfer target is in the expected legitimate collections at the indirect transfer points as isolated, ignoring the previous path the program executed.

CCFI makes it more certain that the path of program's executed control flow could be a real path defined in the CFG. All transfers gathered by hardware will be validated. CCFI is a more promising way to protect binaries against the advanced CRAs.

A. Weakness Analysis

CCFI provides stronger defensive ability than the origin CFI. However, it's practical implementation, PathArmor, does not care about the precise of CFG, either. The using of approximate CFGs weakens CCFI, and leaves gaps for attackers.

Just like other CFI systems, all possible entry sites of functions in binaries, with a very simple data-flow analysis, are taken as legitimate at the indirect call-sites with the policy of PathArmor. When validating the collected paths, PathArmor can not decide whether the targets of indirect calls are correct. So when an ”unknown” path is captured by PathArmor, it will be inserted into the CFG after that its' safety is confirmed.

As it is known, transfer targets at the indirect call-sites are undefined and ambiguity after compiled. In other words, any address can be called only if allowed by the CFI policy. It means that each entry address of functions can be called. Under a weak CFG, all we shall do to exploit binary is to find a path from vulnerability to system functions.

Origin CCFI has talked about the situation under attacking of COOP and CFB, the forward-edge invariants have been enhanced to raising the bar against COOP-like and CFB-like attacks. Actually, PathArmor depends on simple data-flow analysis to decide the target of transfers at indirect call-sites. It leads to gaps left for attackers as a result that many targets are taken and some impossible addresses are selected, either.

B. Parameter Counter Matching Based Indirect-Call Targets Reduction Algorithm

In most programs written by C/C++ language, there are many indirect calls as due to that C/C++ allow flexible function pointers, which make it very hard to analyze the real function to call at binary-level. As described above, the key to mitigate the problem is to get a more precise CFG to constraint transfers that occur at the forward-edges, which also means indirect call-sites. Existing techniques in binary analysis are always too heavy to use, especially in CFI systems that have already brought out high performance loss. To address the problem, we propose a new algorithm basing on the fact that the using of C/C++ functions follows the *Application Binary Interface* (ABI).

Algorithm 1 Forward analysis of functions

```
Input: Instructions Set of Function $F$
Output: Counter of parameters consumed by the function, $Counter$;
procedure FORWARD ANALYSIS($F$, $CFG$)
    $CFG, BBs$ ←F
    for all $block \in Set(EntryBlocks)$ do
        $registerStatus$ ←Set(rdi, rsi, rdx, rcx, r8, r9)
        $toVisit$ ←StackInitialize()
        $visited$ ←StackInitialize()
        $toVisit$.push($block$)
        while $toVisit.empty()$ not TRUE do:
            $targteBlock \leftarrow toVisit$.pop()
            if $targetBlock \in visited$ AND $targetBlock$ contains loops then
                Continue
            end if
            $insts$ ←GetInstructions($targetBlock$)
            $registerStatus$ ←UpdateStates($insts$)
            if $targetBlock$ contains Dynamic Site or $targetBlock$ is EixtBlock then
                Break
            end if
            $visited$.push($targetBlock$)
            $toVisit$.push($targetBlock.targets()$)
        end while
    end for
    $Counter$ ←Get($registerStatus$)
    return $Counter$
end procedure
```

The ABI specifies how parameters are passed when calling a function at binary-level. Six registers, *rdx, rcx, rdi, rsi, r8, r9*, are using as parameter registers in x86_64 Linux systems. Parameters which no more than six are loaded into those six registers in sequence, and used by the called functions in the same way.

It is difficult to get semantics at indirect call-sites, but functions are compiled and used in similar way. When calling a function, parameters should be loaded and used. So if we have the knowledge about the counts of parameters prepared at call-sites and consumed in functions, incorrect transfers can be limited. According to the thought, the *parameter counter matching based indirect-call targets reduction* (PCM) algorithm are proposed and designed.

C. Implementation

With the knowledge of ABI, we can get the call relations approximately by parameter analysis between call-sites and functions. We propose two algorithms to analyze the call-sites, functions and one algorithm to match them.

Algorithm 2 Backward analysis at callsites

```
Input: Instructions of Function Containing Callsites, $Function$, $Block$
Output: States of parameters prepared before the callsites, $registerState$
procedure BACKWARD ANALYSIS($F$, $CFG$, $block$)
    $registerStatus$ ←Set(rdi, rsi, rdx, rcx, r8, r9)
    $instSet$ ←GetInstructions($block$)
    for all $inst \in instSet$ do
        $readRegisters$ ←GetReadRegisters($inst$)
        UpdateState($registerStatus$)
        $writeRegisters$ ←GetWriteRegisters($inst$)
        UpdateState($registerStatus$)
    end for
    if $block$ is Entry OR $Counter$ equals 6 then
        return $Counter$
    end if
    $Edges$ ← GetIncomingEdges($block$)
    for all $edge \in Edges$ do
        if $edge$ is indirect then
            Continue
        end if
        $sourceBlock$ ←GetSourceBlock($Edge$)
        if $sourceBlock$ contains Loop then
            if $sourceBlock$ is NOT analyzed then
                SetLoopBlockAnalyzed($sourceBlock$)
                $prevState$ ←BackwardAnalysis($Function$, $CFG$, $sourceBlock$)
                merge($registerState$, $prevState$)
            end if
        end if
        if $sourceBlock$ is ExitBlock then
            Continue
        end if
        $prevState$ ←BackwardAnalysis($Function$,$CFG$, $sourceBlock$)
        merge($registerState$, $prevState$)
    end for
    return $registerState$
end procedure
```

The six registers are working with four states including that *Read*(R), *Write*(W), *Read-Before-Write*(RW), *Write-Before-Read*(WR). The first state of one register after a function entry can decide that whether it is used to pass a parameter because the parameters should be read from the register. Similarly, how the parameters are prepared can be learn from that the last state, R or W, of the six registers before a call-site, as the parameter value must be loaded into the registers. Basing on the idea, we achieves three processes to realize PCM.

The first forward-analysis algorithm is designed to calculate how many parameters are consumed in a function, and each instruction which would be executed in the function is analyzed as what done in algorithm 1. Parameter registers' states of instructions from the entry to the exit points in each function are calculated and merged lastly to analyze how many parameters the function consumed. Detailed as follows.

1. All basic blocks in a function are extracted into a CFG.

2. Parameter registers' states generated by instructions from the entry blocks to the exit blocks in the function are analyzed and collected with a depth-first traversal down to its next block following its' CFG.

Algorithm 3 Matching between callsites and functions

```
Input: Functions with counter of parameters consumed, $Functions$. Callsites Set with counter of parameters prepared, $Callsites$.
Output: Result of matching, $result$.
procedure MATCHING($Functions$, $Callsites$)
    for all $callsite \in Callsites$ do
        $callTargetSet \leftarrow InitialSet()$
        for all $func \in Functions$ do
            if $callsite$.paramCount==0 then
                if $func$.paramCount==0 then
                    $callTargetSet$.add($func$)
                end if
            end if
            if $callsite$.paramCount¿=$func$.paramCount AND $func$.paramCount¿0 then
                $callTargetSet$.add($func$)
            end if
        end for
    end for
    return $Result$
end procedure
```

3. Merge all states generated by the instructions.

If state of a register is R or RW, it should be considered as a parameter used by the function. If a register which locates on the right side in the order of {*rdi, rsi, rdx, rcx, r8, r9*} is used as parameter register but the one left of it is not used, the maximum number of parameters will be returned. For example, if a function takes r9 as parameter but null for r8, rcx or other registers, the counter of parameters should be 6 other than a fewer one, considering that sometimes parameters are passed to the called function but not used.

A backward-analysis recursion algorithm is implemented to compute the count of parameters prepared at call-sites, as shown in algorithm 2.

1. All basic blocks and indirect call-sites in a function are extracted into a CFG.

2. Parameter registers' states generated by instructions from the call-sites back to the entry of functions are analyzed and collected.

3. All states are merged to analyze the number of parameters prepared before call-sites.

Basic blocks are analyzed start at call-sites along the reverse direct of the blocks which are loaded when running. During the backward analysis, blocks with loops and dynamic transfers are specially handled for which loops and dynamic targets are hard to decide.

At last, based on the comparison between call-sites and functions, a simple match algorithm is implemented shown in algorithm 3. Due to the approximate result calculated with algorithm 1 and 2, we have to take a conservative approach to match the targets of call-sites. If a call-site prepares parameters is not less than that function could consume, the transfer is allowed exclude the function consumes 0 parameters. It is legal only if the count of parameter call-sites prepared and functions consumed are both 0.

IV. Result

In this section, we will evaluate our PCM algorithm.

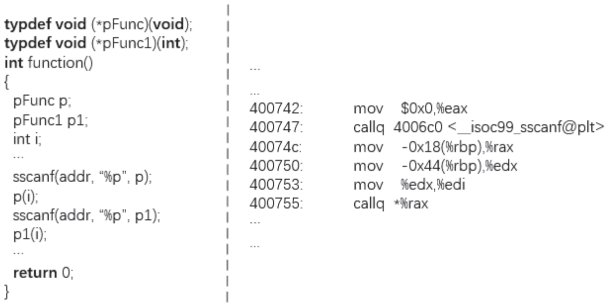
At first, we use a small toy code to analyze the security of PathArmor combined with our algorithms.

Then we choose some open-source C/C++ programs with different size and complexity described as Table I. We experiment our algorithms on UbuntuKylin14.04 x86_64 system with gcc version 4.8.4.

Programs were compiled in default way with their makefiles and the compile information were saved.

A. Security

In order to make sure that whether the security is enhanced by our algorithm, we make a demo program written in C, which the main function is to execute the code from the address accepted from the console.

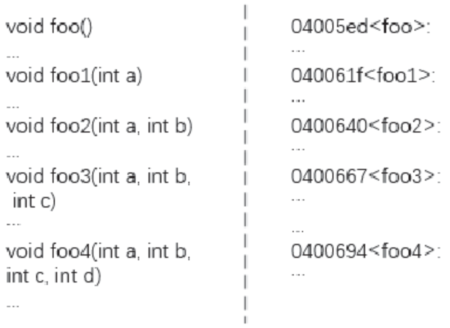


```
typdef void (*pFunc)(void);          |
typdef void (*pFunc1)(int);          |
int function()                       |   ...
{                                    |
  pFunc p;                           |   ...
  pFunc1 p1;                         |   400742:    mov   $0x0,%eax
  int i;                             |   400747:    callq  4006c0 <__isoc99_sscanf@plt>
  ...                                |   40074c:    mov   -0x18(%rbp),%rax
  sscanf(addr, "%p", p);             |   400750:    mov   -0x44(%rbp),%edx
  p(i);                              |   400753:    mov   %edx,%edi
  sscanf(addr, "%p", p1);            |   400755:    callq  *%rax
  p1(i);                             |   ...
  ...                                |   ...
  return 0;                          |
}                                    |
```

Fig. 1 Indirect callsite in our demo program

Fig. 1 shows that two function pointers with different styles are used in our demo program, and we have two indirect call-sites in binary. It is worth to noting that two pointers are assigned to the same value in the short section.

Fig. 2 describes some functions with different parameters needed and their addresses in binary.



```
void foo()                 |   04005ed<foo>:
...                        |   ...
void foo1(int a)           |   040061f<foo1>:
...                        |   ...
void foo2(int a, int b)    |   0400640<foo2>:
...                        |   ...
void foo3(int a, int b,    |   0400667<foo3>:
 int c)                    |   ...
...                        |   ...
void foo4(int a, int b,    |   0400694<foo4>:
int c, int d)              |   ...
...                        |
```

Fig. 2 Functions used in our demo program

At first we use the PathArmor to analyze the program and 5 addresses are taken totally including the entry of function *foo*, *foo1 main* and other two function with the prefix *__libc*. It is allowed to be called at any indirect call-sites if the addresses are taken by PathArmor. But in our demo program, the function pointer *p* is defined with no parameter and the other function pointer defined with one parameter. The function *foo1* can be loaded in *p* as well as *p1* and to execute.

In our algorithm, *p* is unable to execute with the address of *foo1* taken, for which the count of parameters prepared before *p* is 0 but the function *foo1* needs one parameter passed in. The indirect targets are constrained by our algorithm.

From our analysis on the binaries of popular programs shown in Table I, we find that function pointers are used few and cautious. It is hard to get all targets of indirect callsites precisely, all we can do is to use an approximate result conservatively. In our policy for CCFI, when calling indirectly, some occasional false positives are allowed and continuous error are stopped to guarantee the security.

B. PCM Algorithm

1) Functions Consumed Parameters: With the help of symbolic table generated when compiling, we analysis the result of our experiments. At first, we evaluate algorithm 1 at all functions for each program. Results are shown in Fig. 3.

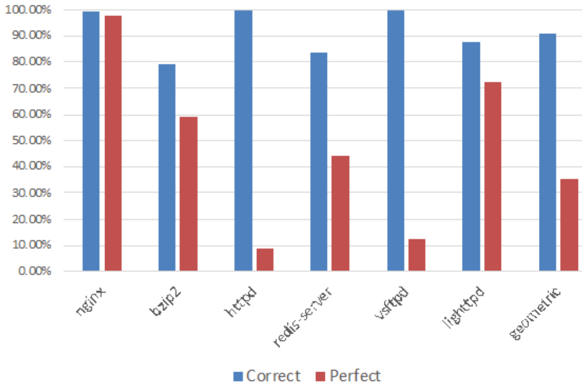


Fig. 3 Result on count of parameters functions consumed

In Fig. 3, it is considered as correct if the count of parameter of functions consumed calculated with algorithm 1 is equal to or greater than that of functions consumed actually. And perfect is refer to that parameters count are strictly equal.

Fig. 3 shows that the correct rate our algorithm obtained is good ranking with the worst which is little less than 80% (bzip2) to the best which is nearly 100% (nginx), with a geometric mean of 91.2%.

Some functions in C/C++ programs, like the *start_main* generated by *glibc* ending with jumps and without returns, are often mistaken by our algorithm. In fact in the program bzip2, a program for compressing and decompressing, the number of incorrect parameters are less as it is too small.

The perfect rate are different from each other from best greater than 95% to the worst less than 10%. But in this paper what we need is to constrained the targets at indirect call-sites, a precise result is expected but not necessary.

2) Call-Sites Prepared Parameters: In order to evaluate our algorithm 2, we test it on all direct call-sites for each program. The direct call-sites and functions are judged matched if the conditions which described in algorithm 3 are satisfied. The result are shown in Fig. 4.

As shown in Fig. 4, the rate of matched between direct call-sites and its' targets are all greater than 80% with a geometric mean 90.2%. It indicates that our PCM algorithm

TABLE I
PROGRAMS USED AS TESTCASES

Programs	version	functions number	direct callsites number	indirect callsites number	size
bzip2	1.0.6	81	468	22	190.6KB
vsftpd	3.0.2	602	2666	8	136.4KB
Lighttpd	1.4.33	314	2239	55	870.5KB
Apache Httpd	2.4.23	1209	5266	180	910.7KB
Nginx	0.8.54	1161	4895	281	2.3MB
redis-server	3.2.1	2557	5574	310	5.3MB

performance good when doing matching between callsites and its target functions and the same to the situation at indirect call-sites.

The number of indirect call-sites are far less than that of direct call-sites, which means it is a great probability that parameters matching by our algorithm would cover the real facts.

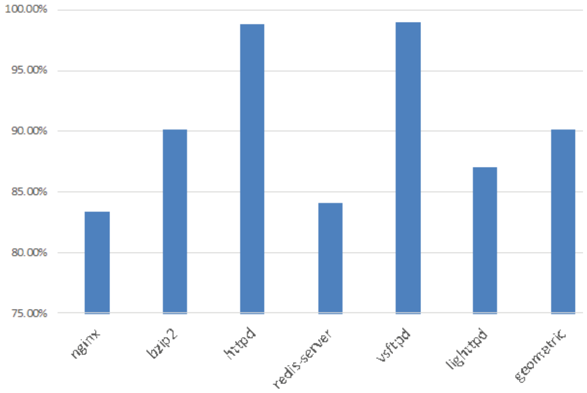


Fig. 4 Result on parameters matching at direct call-sites

3) Reduction in Indirect Call-Sites' Targets: Based on the result taken by PathArmor depending on simple data-flow analysis, we measure the effect of our algorithm. The result is shown in Fig. 5.

Comparing to the origin set of indirect call targets used by PathArmor, our algorithm achieves a good result. The set of indirect call-sites' targets is obviously smaller than PathArmor with a geometric mean 56.8%. And the smaller, the better.

V. CONCLUSION

In this paper, we present a PCM algorithm, a method to get a more precise CFG for CFI. PCM algorithm relies on binary-level static analysis to detect the call relations between indirect call-sites and functions.

To advanced CRAs like CFB and COOP, due to the use of function pointers, the reuse of functions in binaries are becoming popular. The targets of indirect call-sites can be constrained a lot to stop the function-reuse attacks combined with PathArmor. Gaps are still exists because of our conservative policy, but CCFI is further enhanced with our algorithm.

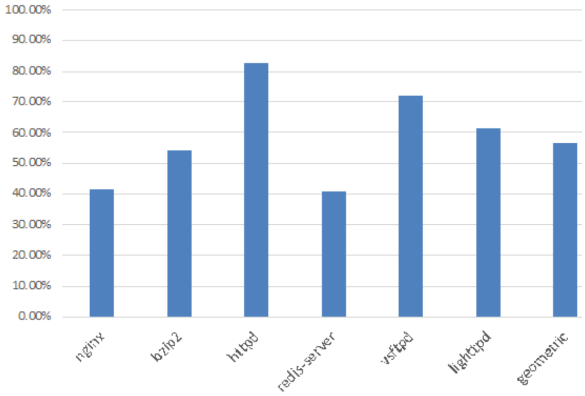


Fig. 5 Result on constraint analysis: the indirect call targets comparing to PathArmor

ACKNOWLEDGMENT

We thank the open-source program of PathArmor and its contributors. This work was supported by the National Natural Science Foundation of China (61303191,6130319,61402504).

REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[2] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 927–940.

[3] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, p. 4, 2009.

[5] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 559–573.

[6] Z. Wang and X. Jiang, "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 380–395.

[7] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[8] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 2011, pp. 353–362.

[9] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[10] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[11] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[12] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2009, pp. 214–228.

[13] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[14] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[15] S. Designer, "Getting around non-executable stack (and fix)," 1997.

[16] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[17] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[18] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

[19] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[20] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[21] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[22] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[23] I. P. Disassembler, "Debugger," 2010.

[24] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.