ROPOB: Obfuscating Binary Code via Return Oriented Programming

Dongliang Mu[1(✉)], Jia Guo[1], Wenbiao Ding[1], Zhilong Wang[1], Bing Mao[1], and Lei Shi[2]

[1] State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China
mudongliangabcd@163.com, njuguojia@163.com, wbdingzx@163.com, njuwangzhilong@163.com, maobing@nju.edu.cn

[2] Zhengzhou University, Henan, China
shilei@zzu.edu.cn

Abstract. Software reverse engineering has been widely employed for software reuse, serving malicious purposes, such as software plagiarism and malware camouflage. To raise the bar for adversaries to perform reverse engineering, plenty of work has been proposed to introduce obfuscation into the to-be-protected software. However, existing obfuscation methods are either inefficient or hard to be deployed. In this paper, we propose an obfuscation scheme for binaries based on *Return Oriented Programming* (ROP), which aims to serve as an efficient and deployable anti-reverse-engineering approach. Our basic idea is to transform direct control flow to indirect control flow. The strength of our scheme derives from the fact that static analysis is typically insufficient to pinpoint target address of indirect control flow. We implement a tool, ROPOB, to achieve obfuscation in Commercial-off-the-Shelf (COTS) binaries, and test ROPOB with programs in SPEC2006. The results show that ROPOB can successfully transform all identified direct control flow, without causing execution errors. The overhead is acceptable: the average performance overhead is less than 10% when obfuscation coverage is over 90%.

Keywords: Obfuscation · Return-oriented programming
Reverse engineering

1 Introduction

Along with the booming development of software market, illegal reuses of software with malicious purposes, such as *software plagiarism* and *malware camouflage*, bring a lot of negative influence. Software plagiarism happens when the adversaries develop and release software with components "stolen" from programs owned or licensed under others' names. Malware camouflage refers to cases where the adversaries repackage released software to embed malicious payloads, and then publish the resulted in "malware" with the name of the original

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 721–737, 2018.
https://doi.org/10.1007/978-3-319-78813-5_38

software. These intentional torts are causing billions of dollars worth of damage to the software market every year [1].

Commercial softwares always appeal to adversaries for malicious reuse. As most of them are released in the form of binaries, adversaries can analyse the binaries to extract their working logic to reuse them. The analysing process is commonly termed as *software reverse engineering*. A major line of effort on preventing software from being reverse engineered is to introduce *obfuscation* into software. Basically, obfuscation deliberately transforms readable codes into obfuscated codes that are difficult for humans or tools to understand, aiming to conceal the original logics of the software.

Plenty of techniques have been proposed to achieve software obfuscation in different phases of reverse engineering. Linn and Debray propose to obfuscate executable code to disrupt static disassembly [2], which is often the first step of binary reverse engineering. Igor et al. propose to keep control flow under cover by signal handlers [3]. However, leveraging signal mechanisms to handle control flow introduces significant overhead (typically higher than 21%). and it is not thread-safe. Chen et al. leverage the characteristic - information tracking support of Itanium processor, to obfuscate control flow with exception handling [4]. This mechanism is more efficient but can only be deployed when the required processors are available.

Before we introduce our approach, we first briefly explain the concept of ROP. ROP is a type of advanced code-reuse attack proposed by Hovav Shacham in [5]. A ROP attack hijacks the control flow to a sequence of code pieces (or "gadgets") that end with a return instruction. The ROP attack will pre-set the return address for the return instruction in each gadget on the stack, to make sure these gadgets are executed sequentially.

RopSteg [6] is proposed for code protection that attempts to hide selected instruction sequence by executing their "unintended matches" located elsewhere. And instruction snippet that they can hide is much smaller than the whole program.

In this paper, we propose a new ROP-based approach to perform software obfuscation. The core idea of our approach is to take advantage of ROP to obfuscate control flow in basic block granularity as follows. First, we disassemble a to-be-protected ELF file and divide executable code into basic blocks. Then, do some instrumentation on basic blocks to convert them into gadgets. We transform all identified direct control flow and hide them by ret instruction. Finally, add all those gadgets and designed payload into original file and leverage binary rewrite to produce obfuscated file. Note that the designed payload will be used for control flow transfers and will be stored in a newly added payload section. As we use ROP payload and gadgets to complete control flow transfer, static reverse engineering methods can not find the real control flow, even though they can disassemble software correctly. And it is a lightweight method to do obfuscation with ROP. As ROP works only in user space, does not involve signal handler or other kernel space, the whole process of control flow transfer is quicker than signal methods theoretically. And ROP method can be thread-safe.

Our contributions are as follows:

- We propose a novel ROP based approach to achieve control flow obfuscation. Our experiment proves that this method is effective and practical against static reverse engineering analysis.
- Our obfuscation approach is efficient and widely deployable.
- We develop a tool ROPOB to implement our approach. Experiment results show that ROPOB can correctly transform all identified direct control flow. The average overhead introduced by our obfuscation is less than 10% when obfuscation coverage is above 90%.

The remainder of this paper is organized as follows. In Sect. 2, we explain the overview of our approach. Section 3 details our design of ROPOB. Then we present the evaluation of our approach in Sect. 4. Section 5 summarizes related work and Sect. 6 discusses some issues. Section 7 concludes this paper with future work.

2 Overview

Our goal is to convert ELF (Executable and Linkable Format) files to ROP-obfuscated ones, whose control flow information has been concealed, so that static de-obfuscation methods will fail to construct the control flow graph (CFG). The obfuscated files are semantically equal to the original ones. In this section we will give an overview of our method.

We consider a model, in which the defender develops a commercial software, prepares to obfuscate and release its binary version, and the adversary aims to reverse engineer the binary for malicious reuse. The following assumptions should be satisfied in this model:

- The un-obfuscated binary file is in ELF file format (with or without symbol information);
- The obfuscated binary file is supposed to run on unmodified Linux systems;
- The adversary only employs static reverse engineering tools, such as IDA Pro [7], to analyse the obfuscated binary file.

Our approach takes a to-be-protected ELF file as input and outputs the obfuscated version. The workflow of our approach is shown as Fig. 1(a), which consists of four major steps:

- Disassemble the text section of an ELF file and divide executable code into basic blocks;
- Do some instrumentation on basic blocks to convert them into gadgets, which are ended with *ret* instruction;
- Write all those gadgets and designed payload into an assembler file and assemble it into a new ELF file. Note that the designed payload is a list of start address of gadgets. Its function is to guide the execution of all the gadgets.
- Copy text and payload section of new ELF file into original ELF file to produce our obfuscated file;

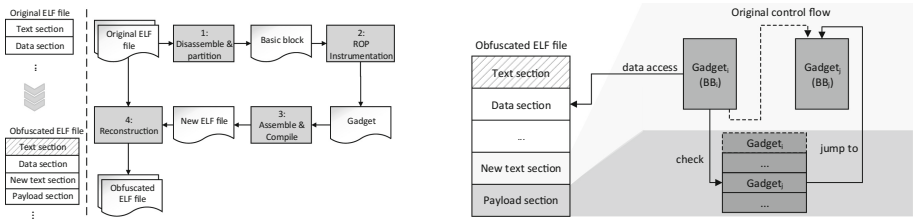


Fig. 1. Workflow of our approach

When the above steps are finished, we wipe out the original text section from the resulting ELF file. Otherwise the adversary can still recover the control flow information from this section. Note that all sections copied to the obfuscated ELF, including the new text section and the payload section, do not overlap with any previous sections. Notably, we maintain the data section to be intact, which will be directly reused by the obfuscated code section. Therefore, we essentially maintain the data integrity.

In summary, all our work is in user space and does not involve kernel space. Furthermore, our method can invalidate all static de-obfuscation techniques, because there is no control flow information in static analysis. Meanwhile, it can increase the difficulty for dynamic de-obfuscation methods. Because there is no function call in our obfuscated files, it is hard to extract high level semantics, even when attackers find an execution path dynamically.

The workflow of our approach is straightforward. However, there are multiple challenges to be tackled in the workflow, which are summarized as follows:

- Basic blocks can't be partitioned thoroughly. Therefore, some indirect control flow may jump into the body of basic block (or gadget), rather than the entrance. That will fail payload entrance check (used for gadget location when ROP runs).
- Indirect control flow can't be analysed statically. It is necessary to make indirect control flow jump to destination correctly. We design a control flow map table to solve this problem.
- We must keep data access correct, with control flow obfuscated. We design a reconstruction framework to reuse the whole data sections of original programs.

Figure 1(b) depicts an instance of our design. The dotted line in this figure represents there is a control flow path from BB_i to BB_j in the original file (BB means basic block). After basic blocks (BB) are transformed to gadgets ($Gadget$), we maintain the control flow path from BB_i to BB_j through designed payload. Before $Gadget_i$ executes ret, we push the address of $Gadget_j$ onto the stack. Therefore, we can direct the control flow to $Gadget_j$.

3 Design and Implementation

We design our method as a tool, called ROPOB, which takes an ordinary ELF file as input and generates an ROP-obfuscated ELF file with the same semantic as output. This tool has a basic reconstruction framework, which supports instrumentation works on the input ELF file. Apart from this framework, there are other challenges needed to be resolved in ROPOB, such as basic block partition and control flow integrity. We will present all technical details in this section.

3.1 Reconstruction Framework

Our goal is to transform an original ELF file to a ROP-obfuscated one, keeping the semantics equal. We design a framework to reconstruct an ELF file, and to support any assembly-level instrumentation, including our ROP-obfuscation work. This framework mainly analyzes the assembly code, which is obtained from disassembling the original ELF file, and then recompiles it into a new ELF file. Our reconstruction framework is shown in Fig. 2.

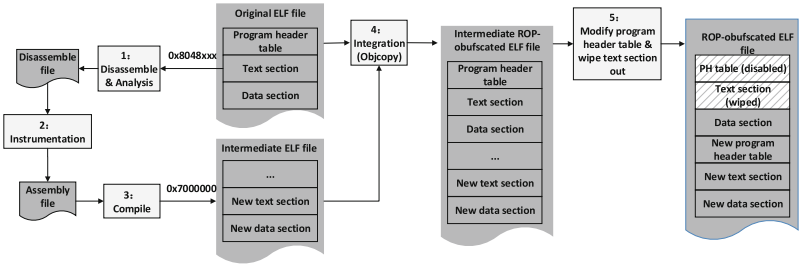


Fig. 2. Reconstruction framework

Generally speaking, the text section of original ELF file starts from address 0x8048xxx. In our reconstruction framework, we extract the text section from the original ELF file, disassemble the text section and divide the text section into basic blocks. Meanwhile the control flow information between basic blocks are collected. Thus, we can apply any assembly-level instrumentation work on those basic blocks, such as our ROP obfuscation instrumentation. Then we use the rewritten basic blocks and previously collected control flow information to write an assembly file. After recompiling the assembly file, we can get an intermediate ELF file, whose text section is lowered down to address 0x7000000. The reason for changing base of text section to 0x7000000 is that we will copy the text section into the original ELF file as new text section. As we do some instrumentation works, the scale of text section in intermediate file is larger than that of the original one. So if we set the base address of text section in intermediate file the same as that of the original one, some data sections of the original file will be destroyed, which is not expected. We integrate new text section and some new

data sections into original file by *objcopy* tool after recompilation in order to reuse all data sections from the original file. The last step of our framework is to modify the program header table to include new text section and data sections, whose addresses are all above 0x7000000. For some security reasons, we need to wipe out the original text section in case of control flow information leak. By means of this framework, we can maintain the same semantics between the original file and the ROP-obfuscated file.

3.2 Basic Block Partition

It is common sense that control flow information exists in relationship between basic blocks. A basic block is a straight-line code sequence, with no branches in except to the entry and no branches out except at the exit. The rules we use to divide basic blocks are as follows:

- A control flow related instructions, like a *jmp/jcc/call/ret* instruction, indicates an exit of a basic block. The target operand of a direct *jmp/jcc/call* instruction is an entrance of a basic block.
- We ignore the target operand of indirect *jmp/jcc/call* instructions.
- The next instruction of a *jmp/jcc/call/ret* instruction is an entrance of a basic block.

We do not deal with the target operand of indirect *jmp/jcc/call* instructions (control flow related instruction, CFRI for short), because the target operand is unknown in static method. Although it is possible to explore some information through some data sections, such as finding a jump table in rodata section, it is hard to locate the boundary of a jump table. However, some basic blocks cannot divided correctly, for example, an entrance of a jump table is not found. To deal with this problem, we design a control flow mapping table, which will be discussed in next subsection.

3.3 Control Flow Mapping Table

The control flow information of direct CFRI is obvious. However, we can't work out control flow information of indirect CFRI, whose target operand is often determined by some data sections during the runtime. As we have mentioned, we reuse all data sections from original file, and the target addresses computed by original indirect CFRIs are the same as those calculated by indirect CFRIs in new text section of ROP-obfuscated file. Under the circumstances, if we make no change to the target address calculated by indirect CFRI in new text section, control flow will be guided into wiped text section which will crash the program. So, we will redirect such addresses and design a control flow mapping table to solve this problem. Figure 3 illustrates the principle of control flow mapping table.

There is a basic block (BB_1) of bzip2 presented in Fig. 3. After instrumentation, we get $Gadget_1$. 0x804cac4 and 0x7007dff are entrance addresses. The

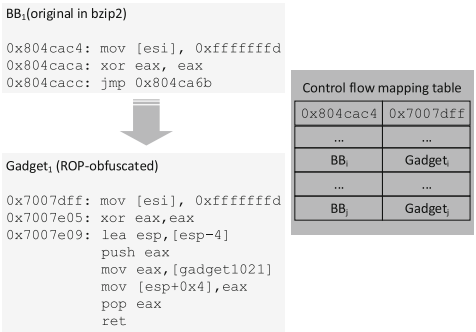


Fig. 3. Control flow mapping table

control flow mapping table is in the right side of Fig. 3. If the target of an indirect CFRI is entrance of BB_1, we will calculate the target address first and check control flow mapping table to locate the correct target address 0x7007dff in $Gadget_1$.

It is tricky to solve the problem presented in Sect. 3.2 by means of control flow mapping table in Fig. 3. Just thinking that if our basic block partition misses a basic block, whose entrance is 0x804caca in BB_1 and an indirect CFRI jumps to 0x804caca, we can't find any table entry to match 0x804caca. But the offset between 0x804caca and 0x804cac4 equals that between 0x7007e05 and 0x7007dff, with the acknowledgement that we only apply instrumentation at the end of basic block, rather than in the middle. Therefore we can locate 0x7007e05 correctly.

3.4 ROP Instrumentation

Our goal is to use ROP technique to hide control flow information, so that static method can't analyze the control flow information. Control flow from one basic block to another is completed by *ret* instruction and ROP payload. The payload is a list of entries of all generated gadgets, converted from original basic blocks. We design different instrumentation policy for different control flow cases.

- **Case 1:** For each basic block ended with non-CFRI (maybe *mov* or *add* instruction), we push the start address of next gadget onto the stack. Note that the next gadget is transformed from the basic block next to it and its address is stored in the payload;
- **Case 2:** For each basic block ended with direct CFRI, there is only one target address of *call*/*jmp*. If the direct CFRI is *call*/*jmp*, we push the start address of next gadget onto the stack like Case 1. But the next gadget is transformed from the basic block starting from the target address. Figure 3 shows a simple example about direct *jmp* (Note that [*gadget*1021] stores the start address of next gadget). The difference between *call* and *jmp* is that for *call* instruction, return address is pushed onto the stack at first. If the direct

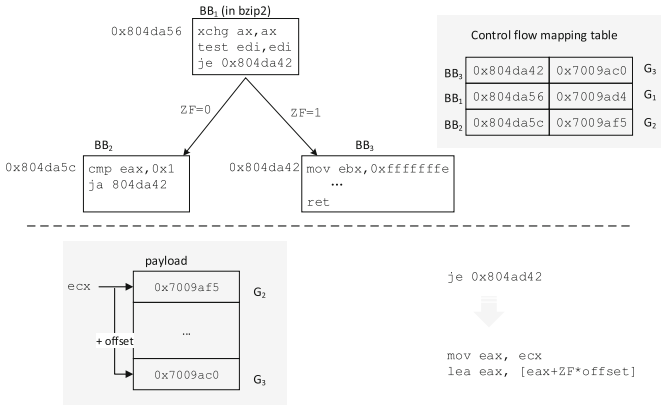


Fig. 4. Jcc transformation

CFRI is *jcc* with two target addresses, we provide two paths at the end of gadgets and deal with each path like direct *jmp* instructions, or we can use characteristic of ROP to transform two paths into one unified form. We will discuss this transformation later in this subsection;

- **Case 3:** For each basic block ended with indirect CFRI, we take advantages of control flow mapping table to relocate the target address. Then the remaining work is like Case 2;
- **Case 4:** For each basic block ended with *ret* instruction, nothing is to be done. The return address has been pushed in the stack previously;

Figure 4 tells how we transform jcc instruction into a unified form. There is a branch from BB_1 to two destinations: BB_2 and BB_3 in program bzip2. If condition is met at instruction 'je 0x804da42' ($ZF = 1$), control flow goes to BB_3. Otherwise, control flow goes to BB_2. Accordingly, there are gadgets, G_2 and G_3, in payload. The difference of start address between G_2 and G_3 in payload is offset, a parameter shown in Fig. 4. Then we can get the condition flag and use it to compute the proper target address. For example, we use register eax to store address of G_2 in payload, and then we calculate the target address with the help of flag and offset. The flag is the conditional judgment bit in $eflags$. We ensure that if condition is satisfied, eax points to G_3 in payload. Otherwise, register eax points to G_2 in payload.

3.5 Special Case of Data Access

In ROPOB, we reuse all data sections from original file. The major data accesses are absolute addressing, with addresses in data sections directly. However, there are special cases - access data with relative addressing. Since we have dropped the new text section to a low address space - 0x7000000, it is a mistake to access data relative to instructions in new text section. This problem is addressed in

Oxymoron [8]. Here we hold the same viewpoint as Oxymoron, where the authors believe this is not a general case and can be located statically.

4 Evaluation

We tested our method on fourteen programs in SPEC2006, and succeed to obfuscate those programs and evaluate our method in three aspects, including control flow concealing, program size and program execution speed. Our experiments are performed on CentOS 6.6 x86, with 2G memory and kernel version - 2.6.32.

4.1 Control Flow Concealing

There is no standard for obfuscation strength. But in the aspect of concealing control flow, we work out two metrics to measure obfuscation degree of our method. They are CFG-level stealth and instruction-level stealth. We measure those two metrics on all the fourteen programs in SPEC2006.

CFG-Level Stealth. We choose CFG fragmentation to measure it. Our method hides paths between basic blocks, so an original big CFG is cut into small pieces. We use the ratio of independent CFG (a function is an independent CFG) to measure the degree of fragmentation (DF).

$$DF = ObCFG/OrCFG$$

$ObCFG$ represents the number of independent CFG in obfuscated program. $OrCFG$ is the number of independent CFG in original program. The bigger DF is, the more difficult can the reverse engineering analysis dig out control flow information statically. The result is shown in Table 1. The average DF is 22.79 (from 8.32 to 63.66).

Instruction-Level Stealth. Direct CFRIs are major leakage points of control flow information. The direct CFRIs between basic blocks must be replaced to hide control flow information. We check whether direct CFRIs exist in original and obfuscated programs and analyze those existing cases. The columns jmp_{dec}, $call_{dec}$, jcc_{dec} in Table 2 represent the decrease degree of CFRIs, which is calculated by the following formula.

$$DecCFRI = \frac{OrC(CFRI) - ObC(CFRI)}{OrC(CFRI)}$$

$OrC(CFRI)$ represents the number of direct CFRIs in original program and $ObC(CFRI)$ is the count of direct CFRIs in obfuscated program.

It is obvious to find that there are almost no direct jmp instructions in our obfuscated programs. While direct call and jcc instructions are still there, these cases don't leak any control flow information. Because direct call instructions in

Table 1. CFG fragmentation

Programs	Original	Obfuscated	DF
astar	97	1367	14.09
bzip2	81	1942	23.98
gobmk	2535	32453	12.80
h264ref	531	14166	26.68
hmmer	501	11451	22.86
lbm	28	233	8.32
libquantum	108	1509	13.97
mcf	33	451	13.67
milc	244	3962	16.24
namd	105	6684	63.66
perlbench	1723	53973	31.33
sjeng	143	4904	34.29
soplex	900	15575	17.31
sphinx3	335	6670	19.91

obfuscated programs all call the same one function, which is used for indirect control flow redirection. And direct jcc instructions inherit from original programs but the targets of jcc are inside basic blocks in obfuscated programs rather than outside basic blocks. So control flow information remains under cover through our method. The average $DecCFRI$ of jmp is 96.85% and that of call is 94.80%. The $DecCFRI$ of jcc is negative. Because including inherited jcc instructions, there are other jcc cases in our inserted functions. If we do not take inherited jcc instructions into account to measure $DecCFRI$, the $DecCFRI$ of jcc will be modified as

$$DecCFRI = \frac{2 * OrC(jcc) - ObC(jcc)}{OrC(jcc)}.$$

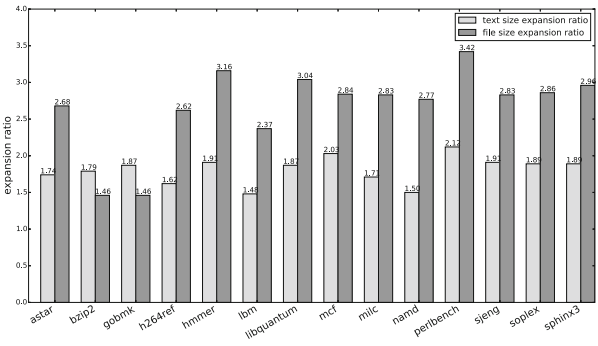


Fig. 5. Size expansion ratio

Table 2. CFRI Decrease. jmp_{or}, $call_{or}$, jcc_{or} represents the number of $CFRI$ in original file; jmp_{ob}, $call_{ob}$, jcc_{ob} represents the number of $CFRI$ in obfuscated file.

Programs	Original			Obfuscated			*DecCFRI*			
	jmp_{or}	$call_{or}$	jcc_{or}	jmp_{ob}	$call_{ob}$	jcc_{ob}	jmp_{dec}	$call_{dec}$	jcc_{dec}	jcc_real
astar	174	400	587	11	9	595	93.71%	97.74%	−1.36%	98.64%
bzip2	356	338	1058	8	30	1065	97.73%	91.10%	−0.67%	99.33%
gobmk	4475	9112	13076	17	66	13085	99.63%	99.27%	−0.07%	99.93%
h264ref	2575	2729	7346	43	373	7354	98.33%	86.32%	−0.11%	99.89%
hmmer	1697	3542	5075	47	41	5082	97.26%	98.85%	−0.13%	99.87%
lbm	26	73	83	4	8	90	84.64%	89.06%	−8.42%	91.58%
libquantum	252	452	592	3	8	599	98.84%	98.25%	−1.16%	98.84%
mcf	64	89	230	3	8	238	95.34%	90.98%	−3.47%	96.53%
milc	528	1543	1367	19	17	1374	96.38%	98.90%	−0.52%	99.48%
namd	1110	1129	4206	14	23	4213	98.72%	98.00%	−0.16%	99.84%
perlbench	10167	13869	25777	25	221	25789	99.75%	98.40%	−0.05%	99.95%
sjeng	940	1102	2504	8	24	2511	99.16%	97.85%	−0.28%	99.72%
soplex	2881	4210	6011	31	719	6018	98.92%	82.92%	−0.12%	99.88%
sphinx3	850	2502	2629	21	12	2637	97.50%	99.52%	−0.31%	99.79%

The real $DecCFRI$ of jcc instructions is listed in the last column of Table 2. The average $DecCFRI$ of jcc instructions is 98.80%.

4.2 Size Measurement

The scales of programs expand in different degrees after being obfuscated by our method. We measure size of text section and size of ELF file in original and obfuscated program respectively. Figure 5 describes the expansion ratio of text section and file size. The mean expansion ratio of text section is 1.81 (from 1.48 to 2.12), and the mean expansion ratio of file size is 2.66 (from 1.46 to 3.42). There are several factors leading to size expansion. Our instrumentation work increases the size of text section. Additionally, we integrate some data sections, such as payload and mapping table, into our obfuscated program, which certainly increases the file size.

4.3 Overhead

As we translate direct CFRIs into indirect ones, memory accessing time will increase and CPU pipe-line will be affected and slow down. Theoretically, our method will increase overhead of programs. We apply our method to those fourteen programs with 100% obfuscation coverage. The overhead is unacceptable and is shown in Fig. 6. There are twelve programs' overhead beyond 200%, with the highest of 1194.74% (libquantum). The average overhead in Fig. 6 is 524.87% (from 15.8% to 1194.74%). To cut down overhead, we adopt an optimization policy of decreasing the obfuscation coverage.

Under our optimization policy, we gain acceptable (overhead,coverage) pairs. We test our obfuscated programs under four coverage standards (95%, 90%, 85%,

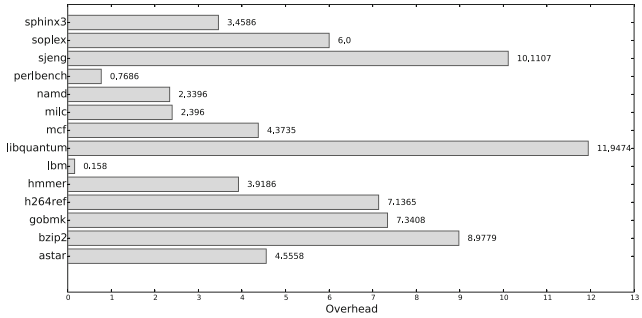
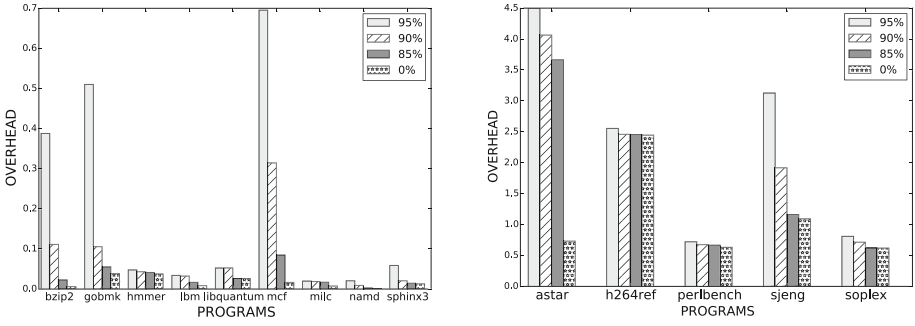


Fig. 6. Overhead with 100% obfuscation coverage

0%). Our optimization policy works on nine programs, depicted in Fig. 7(a). The mean overhead of 95% obfuscation coverage is 20.31% (2.0%–69.54%). That is a great progress, comparing to 100% obfuscation coverage in Fig. 6. When obfuscation coverage is decreased to 90%, the mean overhead is 7.86% (0.91%–31.45%). And for each program, the overhead is cut down to 10% or less, except mcf, whose overhead is 31.45%. As obfuscation coverage comes to 85%, all the nine programs' overhead are below 8.49%, and some programs' overhead approximate to 0%.

Other five programs' overhead is not shown in Fig. 7(a), as their overhead is still very high (mean overhead is up to 110.41%, in arrange from 61.9% to 244.61%) even when obfuscation coverage is 0%. These five programs' overhead are shown in Fig. 7(b). We analyze these five programs deeply, and find that they execute indirect CFRIs frequently. Our method utilizes a function to redirect indirect control flow during the runtime. That is time-consuming. No matter how low our obfuscation coverage is, the overhead is still high.



(a) Nine optimized programs' overhead

(b) Other five programs' overhead

Fig. 7. Overhead with four kinds of obfuscation coverage

5 Related Works

ROP, proposed by Hovav Shacham to enhance return-into-libc attack, is a code-reuse technique [5]. Its execution unit is gadget, a piece of instruction snippet ended with *ret* instruction. ROP uses payload on stack and *ret* instruction to organise its control flow. This code-reuse technique is developed by researchers in many ways. Jiang find gadgets ended with *jmp* instructions can also be used for code-reuse attacks [9]. Q is an automatic method to construct ROP payload to bypass ASLR defense [10]. Printable ROP, whose payload is all printable ASCII bytes, is another branch of ROP attacks [11]. Although defenses against ROP vary too much, such as ASLR , new attack methods can still utilize ROP to launch attacks, such as JIT ROP [12], side channel ROP [13,14]. Not only on traditional PC platform, but also on mobile devices, ROP is an effective way to attack [15–17].

Binary obfuscation focuses on fighting against reverse engineering analysis. Cohen is the first to present binary obfuscation. He changes the layout of instructions to prevent disassembling [18]. Later Igor finds another way to fool disassembler by inserting junk bytes to replace useless instructions [3]. This way is based on an assumption of disassembly algorithms, which treats CFRIs and their targets as hints of instructions' beginning. Nevertheless, Igor does more than that. They utilize signal handling mechanism to conceal control flow information. Their method is effective for obfuscation. However, overhead of their method is unacceptable even their obfuscation coverage is 90%.

Control flow obfuscation aims to protect programs' semantics from being analysed. One way is to hide control flow information like Igor and Chen [4]. Chen takes advantages of characters of Itanium processors, which support information flow tracking. Their method resolves the problem of high overhead but it is not in common use on x86. Another way is to make CFG complicated, so that reverse engineering can't reconstruct high-level program structure. Xin et al. attaches many useless or semantics-equal paths to CFG [19] and fakes a different CFG. Thus some analysis methods based on birthmark or pattern matching fail to dig out real semantics of programs. Control flow flattening also changes the whole CFG of a program [20]. They obfuscate C++ source code to a large loop, and use switch statements to judge which case to be executed in each iteration. Yet their work is based on source code.

ROPSTEG [6], also leverages ROP to perform binary obfuscation, but is different. First, ROPSTEG aims to make use of *unintended instructions* to hide sensitive instructions, while our approach takes advantage of ROP payload and gadgets to hide direct control flow in the form of indirect control flow. And chaining payloads by gadget and *ret* instruction is the core idea of ROP, other than unintended instructions. Consequently, major control flow information can still be recovered from binary obfuscated by ROPSTEG, which, however, is completely hidden by our approach. Second, applicability and obfuscation strengthen of ROPSTEG are restricted by certain properties of the to-be-protected binary, such as the available unintended instructions. To the contrary, our approach has no requirements on the binaries.

Virtualization based methods are also effective for obfuscation. [21–25] mentioned, it is possible to use emulated instruction sets to rewrite programs. Programs' representation is neither IA32 instructions nor ARM instructions and becomes difficult to analyze.

On the opposite side, software reverse engineering is to analyse programs and dig out useful information [26–30]. It can be classified into two kinds, static method and dynamic method. Static reverse analysis often start with disassembling and translate binary code into high level programs. IDA and Hex-Ray are practical business de-compiler tools [31]. Phoenix [32] is another state-of-art decompiler, which uses semantics-preserving structural analysis, and it can reconstruct high-level control flow structure. But all those static methods do not work on control flow obfuscation programs. Dynamic method can find some control flow paths through execution. TOP reconstructs control flow structures dynamically [33]. However, path coverage is the main limitation of dynamic method, since execution can't find out all paths in CFG.

6 Discussion and Limitation

Since gadgets itself is helpful for ROP attacks, it is dangerous to transform basic blocks of original programs into gadgets. We prevent reuse of generated gadgets in two aspects. On one hand, our instrumentation design is unfriendly to ROP attacks, because our generated gadgets all do the same thing, reading data from memory addresses. On the other hand, we can implement load-time basic block level ASLR just like binary stirring [34], as our generated gadgets are independent from each other. Here, we discuss the potential limitations of ROPOB:

- **Dynamic analysis.** As discussed in Sect. 1, ROPOB does not hide control flow information from dynamic analysis as the operand of indirect jmp instruction will be shown when executing. We have an idea to defend from dynamic analysis and show it in Future Work.
- **Payload hiding.** As shown in Sect. 3, ROPOB puts payload into one data section named ".payload". As this section is in the binary, it may raise suspicion in static method.
- **Compatible with ROP Defense.** Since ROPOB makes use of ROP to obfuscate control flow information, our work should be compatible with ROP defense schemes. Although ROPOB does not make use of "unintended instructions" in ROPSTEG [6], there are also CFI security policies which ROPOB violates.

7 Conclusion and Future Work

In this paper, we design and implement ROPOB, a ROP-based binary obfuscation scheme that obfuscates the control flow of programs by chaining basic

blocks as ROP gadgets. We show that ROPOB can protect programs against static analysis, effectively and practically.

Here we show our work in future in the aspect of reuse or replace gadgets. Control flow graphs of functions in original program are independent. If we can reuse gadgets in our obfuscated programs, or replace the gadgets with gadgets in the libraries just like ROP attacks, the independence of CFGs will be broken, and code of each function will interweave together. That will make function extraction difficult. Thus, we need to analyse functionality of each gadget and automatically construct ROP payload to replace functionality-equal gadgets [10].

Acknowledgments. We thank the anonymous reviewers for their helpful feedback. This work was supported by Chinese National Natural Science Foundation 61272078.

References

1. Alliance, B.S.: Global software privacy. http://globalstudy.bsa.org/2010/index.html
2. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: 10th ACM Conference on Computer and Communications Security, pp. 290–299. ACM, October 2003
3. Debray, S.K., Popov, I.V., Andrews, G.R.: Binary obfuscation using signals. In: USENIX Security. USENIX, August 2007
4. Chen, H., et al.: Control flow obfuscation with information flow tracking. In: Proceedings of 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 391–400. ACM (2009)
5. Shacham, H.: 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, pp. 552–561. ACM (2007)
6. Lu, K., Xiong, S., Gao, D.: Ropsteg: program steganography with return oriented programming. In: Proceedings of 4th ACM Conference on Data and Application Security and Privacy (CODASPY), pp. 265–272. ACM (2014)
7. The IDA Pro disassembler and debugger. http://www.hex-rays.com/idapro/
8. Backes, M., Nürnberger, S.: Oxymoron: making fine-grained memory randomization practical by allowing code sharing. In: Proceedings of 23rd Usenix Security Symposium, pp. 433–447 (2014)
9. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of 6th ACM Symposium on Information, Computer and Communications Security, pp. 30–40. ACM (2011)
10. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: exploit hardening made easy. In: USENIX Security Symposium (2011)
11. Lu, K., Zou, D., Wen, W., Gao, D.: Packed, printable, and polymorphic return-oriented programming. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 101–120. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_6
12. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.-R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 574–588. IEEE (2013)

13. Seibert, J., Okkhravi, H., Söderström, E.: Information leaks without memory disclosures: remote side channel attacks on diversified code. In: Proceedings of 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 54–65. ACM (2014)
14. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D., Boneh, D.: Hacking blind. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 227–242. IEEE (2014)
15. Wang, T., Lu, K., Lu, L., Chung, S., Lee, W.: Jekyll on iOS: when benign apps become evil. In: Usenix Security, vol. 13 (2013)
16. Davi, L., Dmitrienko, A., Sadeghi, A.-R., Winandy, M.: Privilege escalation attacks on android. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 346–360. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18178-8_30
17. Lee, B., Lu, L., Wang, T., Kim, T., Lee, W.: From Zygote to Morula: fortifying weakened ASLR on android. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 424–439. IEEE (2014)
18. Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. **12**(6), 565–584 (1993)
19. Xin, Z., Chen, H., Wang, X., Liu, P., Zhu, S., Mao, B., Xie, L.: Replacement attacks on behavior based software birthmark. In: Lai, X., Zhou, J., Li, H. (eds.) ISC 2011. LNCS, vol. 7001, pp. 1–16. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24861-0_1
20. László, T., Kiss, Á.: Obfuscating C++ programs via control flow flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **30**, 3–19 (2009)
21. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 94–109. IEEE (2009)
22. Wang, C.: A security architecture for survivability mechanisms. Ph.D. dissertation, University of Virginia (2001)
23. Oreans Technologies: Code virtualizer: total obfuscation against reverse engineering. http://www.oreans.com/codevirtualizer.php
24. Oreans Technologies: Themida: advanced windows software protection system. http://www.oreans.com/themida.php
25. VMProtect-Software: VMProtect - new-generation software protection. http://www.vmprotecct.ru/
26. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy, pp. 32–46. IEEE (2005)
27. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Polymorphic worm detection using structural information of executables. In: Valdes, A., Zamboni, D. (eds.) RAID 2005. LNCS, vol. 3858, pp. 207–226. Springer, Heidelberg (2006). https://doi.org/10.1007/11663812_11
28. Lakhotia, A., Kumar, E.U., Venable, M.: A method for detecting obfuscated calls in malicious binaries. IEEE Trans. Softw. Eng. **31**(11), 955–968 (2005)
29. Singh, P.K., Moinuddin, M., Lakhotia, A.: Using static analysis and verification for analyzing virus and worm programs. In: Proceedings of 2nd European Conference on Information Warfare and Security, pp. 281–292 (2003)
30. Xu, Z., Miller, B.P., Reps, T.: Safety checking of machine code. In: ACM SIGPLAN Notices, vol. 35, pp. 70–82. ACM (2000)
31. Guilfanov, I.: Decompilers and beyond. Black Hat USA (2008)

32. Schwartz, E.J., Lee, J., Woo, M., Brumley, D.: Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In: Proceedings of USENIX Security Symposium, p. 16 (2013)
33. Zeng, J., Fu, Y., Miller, K.A., Lin, Z., Zhang, X., Xu, D.: Obfuscation resilient binary code reuse through trace-oriented programming. In: Proceedings of 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 487–498. ACM (2013)
34. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: Proceedings of 2012 ACM Conference on Computer and Communications Security, pp. 157–168. ACM (2012)