LETTER *Special Section on Cryptography and Information Security*

Zero-Sum Defender: Fast and Space-Efficient Defense against Return-Oriented Programming Attacks*

Jeehong KIM[†], **Inhyeok KIM**[†], **Changwoo MIN**[††], ***Nonmembers*, *and*** **Young Ik EOM**[†a)], ***Member***

SUMMARY Recently, return-oriented programming (ROP) attacks have been rapidly increasing. In this letter, we introduce a fast and space-efficient defense technique, called *zero-sum defender*, that can respond against general ROP attacks. Our technique generates additional codes, at compile time, just before `return` instructions to check whether the execution has been abused by ROP attacks. We achieve very low runtime overhead with very small increase in file size. In our experimental results, performance overhead is 1.7%, and file size overhead is 4.5%.

key words: *return-oriented programming, malware defense, software security*

1. Introduction

Return-Oriented Programming (ROP) [1] subverts the normal execution flow of a program by chaining together existing code sequences(*gadgets*), each of which ends with `return` instruction. ROP can perform arbitrary computation based on gadgets, i.e., it is *Turing-complete*. Since ROP attacks can occur with any code injection and code modification, defending ROP attacks is very difficult. To this end, ROP attacks are becoming pervasive in various architectures and systems.

To isolate ROP attacks, various defense techniques have been proposed, which can be classified into three categories. First, *dynamic instrumentation approaches* analyze the code at runtime to determine whether the gadgets have been injected. While they can support arbitrary programs without source code, they have large runtime overhead [2]. Second, *randomization approaches* scatter the virtual address mapping of every process [3] and every instruction [4] at load time. Although they provide a high degree of entropy, their applicability is limited due to high processing overhead. Moreover, since they have limitations on randomizing dynamic libraries, some unrandomized codes can be abused for attacks. Finally, *compiler-based approaches* put impediment codes in the binary at compile time. Their runtime overhead is very low by simply replacing the return addresses with the index of return address table [5] or encrypting the return addresses with pushing a random cookie [6].

However, existing approaches require to modify all assembly routines (x86) manually [5] or impose too much file size overhead (about 30%) [6].

In summary, the previous work has many limitations such as high performance overhead, high file size overhead, and restricted defense in scope.

In this letter, we propose a novel ROP defense technique, *zero-sum defender*, which does not have the previously mentioned limitations. In order to defend ROP attacks, our approach counteracts the fundamental behavior of ROP attacks — *return without call* — by testing whether or not the number of calls and returns is zero-sum.

2. Zero-Sum Defender

Our zero-sum defender is a compiler-based defense scheme against ROP attacks. It has very low runtime overhead and file size overhead without limiting the types of defendable gadgets. In this section, we will discuss the design and implementation of our scheme.

2.1 Design

The organizational unit of an ROP attack is the gadget. Each gadget is a small part of the existing code that ends with `return` instruction, `ret` in x86 architecture. An ROP attack manipulates return addresses in a call stack and hijacks the program control flow from one gadget to another gadget [1]. An attacker can synthesize any malicious behavior by chaining a series of gadgets. The fundamental property of gadget execution is that the control flow starts in the middle of a function without `call` instruction and ends with a `return` instruction.

Figure 1 illustrates the overview of our zero-sum defender, which exploits the fundamental property of gadget execution. In normal operations, the control flow starts from the beginning of a function and returns to the caller at the end of the function. Therefore, the number of calls should be same as that of returns. When a vulnerable program is offended by ROP attacks, the number of calls may not be equal to that of returns since gadget execution starts in the middle of the function without `call` instruction. As shown in Fig. 1, we track whether the number of calls matches with that of returns by increasing a control variable, `call_level`, by 1 at a function entry and decreasing it by 1 at a function exit. In our zero-sum defender, the simple test to de-

Manuscript received March 29, 2013.
Manuscript revised July 19, 2013.

[†]The authors are with the College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea.
[††]The author is with Samsung Electronics, Suwon, Korea.

[*]This research was supported by Basic Science Research Program through the NRF of Korea funded by the MEST (2010-0022570).

a) E-mail: yieom@ece.skku.ac.kr

DOI: 10.1587/transfun.E97.A.303

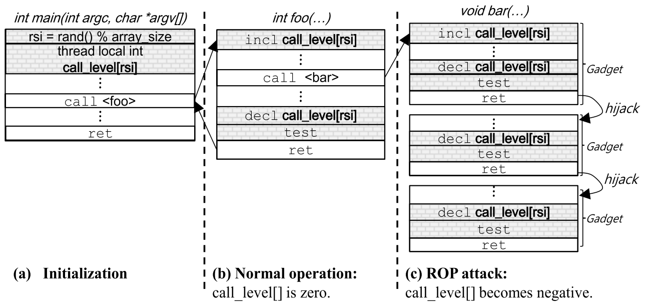

int main(int argc, char *argv[])
rsi = rand() % array_size
thread local int call_level[rsi]
call <foo>
ret
int foo(...)
incl call_level[rsi]
call <bar>
decl call_level[rsi]
test
ret
void bar(...)
incl call_level[rsi]
decl call_level[rsi]
test
ret
Gadget
hijack
(a) Initialization
(b) Normal operation: call_level[] is zero.
(c) ROP attack: call_level[] becomes negative.

Fig. 1 Overview of zero-sum defender in x86 architecture.

termine whether or not the `call_level` is negative can be used for detecting ROP attacks. Listing 1 and Listing 2 show an example of how our approach actually generates additional ROP detection codes for a function, `foo()`, in x86 architecture. In Listing 2, the shaded texts are newly inserted codes for detecting ROP attacks. In our example, we forcibly terminate a process when an ROP attack is detected at `jne` instruction. Therefore, the zero-sum defender completely blocks the abused return execution in a gadget sequence. Moreover, since our approach add only five instructions for each function, its performance overhead and file size overhead are very low.

Listing 1: Original `foo()`

```
sub  $0xc, %esp
movl $0x0, 0x8(%esp)
call 80483c0 <bar>
xor  %eax, %eax
add  $0xc, %esp
ret
```

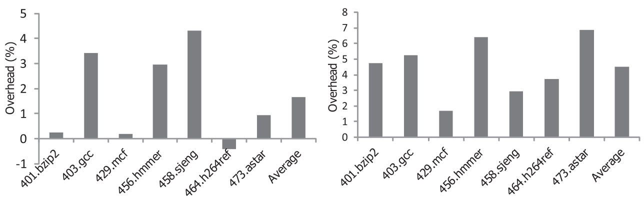
Listing 2: Zero-sum-defended `foo()`

```
sub  $0xc, %esp
incl 0x804a01c
movl $0x0, 0x8(%esp)
call 80483e0 <bar>
xor  %eax, %eax
add  $0xc, %esp
decl 0x804a01c
xor  %al, %al
test %al, %al
jne  8048450 <main+0x30>
ret
```

2.2 Implementation

Even though the key idea behind the zero-sum defender is simple, several implementation issues arise for wider coverage and security. First, when multiple threads execute a function, updating the control variable, `call_level`, could be interfered by the concurrent execution, and there is a possibility of making an incorrect decision. To safely support multi-threaded applications, we represent `call_level` as a thread-local variable. Next, the control variable could also be dealt by malicious codes. We adopt a randomization technique to increase entropy and avoid such attacks. Now, the control variable is represented as a thread-local *array* and we use only one randomly selected slot in the array. As illustrated in Fig. 1, the randomized slot index, `rsi`, is calculated once at the beginning of the `main` function. The degree of entropy is controlled by changing the array size.

In this section, we described the design and implementation of our zero-sum defender. Our scheme can impede mutated ROP attacks such as JOP [7] in a similar way. We omitted the details on how to deal with the mutated ROP attacks due to space limitation.



(a) Performance overhead. (b) File size overhead.

Fig. 2 Normalized performance and file size overhead.

3. Evaluation

We implemented zero-sum defender as a transformation pass of LLVM 3.2. We ran SPEC CPU INT 2006, which is a CPU-intensive performance benchmark suite, on a 3.4 GHz Intel Core i7 machine with Linux kernel 3.2. We measured the performance overhead, how much binary file size is increased, and whether zero-sum defender prevents the real-world ROP exploit. A baseline for comparison is a run without the zero-sum defender. The performance overhead of the dynamic instrumentation approach [2] is 217% on average. The file size overhead of the randomization approach [4] and compiler based approach [6] is 73% and 30%, respectively, on average. Figure 2(a) shows that the zero-sum defender has only 1.67% performance overhead on average. Also, Fig. 2(b) shows that the size of binary file is increased by 4.50% on average. Our experimental results show that the performance overhead and file size overhead of zero-sum defender are significantly lower than the state-of-the-art works. To verify whether zero-sum defender prevents real-world ROP exploits, we tested ROPEME [8], which exploits a buffer overflow vulnerability. ROPEME first searches gadgets in vulnerable program, and thereafter overwrites `ret` in the gadget to procedure linkage table (PLT) of `strcpy()`, transfers bytes of the address in `system()` into global offset table (GOT) of `printf()`, and executes it. The shell code obtains the root privilege. Our zero-sum defender successfully obstructs the abused return execution when printf() is called.

4. Conclusion

Recently, ROP attacks have become more prevalent. In this letter, we proposed a fast and space efficient zero-sum defender against general ROP attack. Compared to prior works, the zero-sum defender significantly reduces the performance overhead and file size overhead with invincible security. Our experiments show that the overhead of our approach is negligible: performance overhead is 1.67% and file size overhead is 4.50%.

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," Proc. 14th CCS, pp.552–561, 2007.
[2] L. Davi, A.R. Sadeghi, and M. Winandy, "Ropdefender: A detection

tool to defend against return-oriented programming attacks," Proc. 6th ASIACCS, pp.40–51, 2011.

[3] S. Bhatkar, R. Sekar, and D.C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," Proc. 14th USENIX Security Symposium, pp.271–286, 2005.

[4] R. Wartell, V. Mohan, K.W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," Proc. 19th CCS, pp.157–168, 2012.

[5] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram, "Defeating return-oriented rootkits with return-less kernels," Proc. 5th EuroSys, pp.195–208, 2010.

[6] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-free: Defeating return-oriented programming through gadget-less binaries," Proc. 26th ACSAC, pp.49–58, 2010.

[7] T. Bletsch, X. Jiang, V.W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," Proc. 6th ASIACCS, pp.30–40, 2011.

[8] L. Le, "Payload already inside: Datafire-use for rop exploits," Black Hat, 2010.