JOP-alarm: Detecting Jump-oriented Programming-based Anomalies in Applications

Fan Yao, Jie Chen, Guru Venkataramani
Department of Electrical and Computer Engineering,
The George Washington University, Washington, DC, USA
{*albertyao, jiec, guruv*}*@gwu.edu*

***Abstract*—Code Reuse-based Attacks (popularly known as CRA) are becoming increasingly notorious because of their ability to reuse existing code, and evade the guarding mechanisms in place to prevent code injection-based attacks. Among the recent code reuse-based exploits, Jump Oriented Programming (JOP) captures short sequences of existing code ending in indirect jumps or calls (known as gadgets), and utilizes them to cause harmful, unintended program behavior. In this work, we propose a novel, easily implementable algorithm, called JOP-alarm, that computes a score value to assess the potential for JOP attack, and detects possibly harmful program behavior. We demonstrate the effectiveness of our algorithm using published JOP code, and test the false positive alarm rate using several unmodified SPEC2006 benchmarks.**

***Keywords*—Jump-oriented programming, Code reuse attack, Detection algorithm**

I. Introduction

As software systems become increasingly complex, they become more vulnerable to malicious users and security exploits. Among the many different classes of vulnerabilities, code-based exploits utilize techniques based on code reuse to achieve their goal of harming program execution. In code reuse exploits, the adversary typically assembles short segments of program code called *gadgets* that end in return, indirect jump or indirect call instructions. These control transfer instructions redirect the program flow to instructions that are adversary-desired targets (gadgets crafted by the malicious user using the existing program code), ultimately leading them to do arbitrary computation and produce unintended consequences during program execution. Although code reuse based attacks represent a narrow class of security attacks, their detection/prevention can be tricky (unless we fully understand their typical attack behavior), and their consequences post manifestation can be devastating.

A. *Existing code reuse attacks and detections*

One typical code reuse based-attack is Return-oriented programming (ROP). ROP was first demonstrated by Shacham et al [4] where the program control flow is directed through a series of gadgets, carefully chosen from *GNU libc* library code, each ending with a *ret* instruction. Preventative techniques include runtime monitor for integrity checking and compiler-based solution to eliminate all unaligned free-branch instructions inside a binary executable.

This material is based upon work supported by the National Science Foundation under CAREER Award CCF-1149557.

Jump-oriented programming (JOP) are a new class of code reuse attacks that utilize indirect *jmp* and indirect call instructions along with a dispatcher gadget that governs the control flow among various jump-oriented targets [1]. The dispatcher gadget internally maintains a dispatch table that specifies the control flow among the functional gadgets. The adversary constructs the dispatcher and functional gadgets using the wealth of code available in the system libraries. While ROP relies on stack pointer *esp* for program counter and *ret* instructions for control transfer, JOP can use any register that points into the dispatch table as its program counter and control flow is driven by the specially identified dispatcher gadget. Due to the sophisticated nature of JOP, defense mechanisms against ROP typically do not prevent JOP attacks.

B. *JOP-alarm detection scheme*

In this work, we propose a novel, simple, and elegant scoring-based algorithm, which we call *JOP-alarm*, to detect JOP-based code reuse vulnerabilities. We study the most common characteristics of JOP such as length of functional gadgets, indirect jump (or call) distances, and tune our algorithm to adjust its scoring based on such details. Our solution does not maintain any state information for individual instructions or data, and can be implemented relatively easily in software with small amounts of code instrumentation or via simple hardware logic. To reduce the rate of false positives, we identify certain commonly observed circumstances where standard compilers emit indirect jumps or calls that may potentially be misinterpreted as JOP, and incorporate such cases into scoring methodology to detect JOP.

II. Understanding Jump-oriented Programming

A. *JOP Example*

An overview diagram of a classic JOP attack code with gadgets and control flow orchestration between them is shown in Figure 1. We show an example JOP code in Figure 2 based on the JOP code published by Bletsch et al [1]. The intent of this code is to invoke a system call that results in unauthorized launch of an interactive shell using gadgets ending with indirect jump and indirect call instructions.

B. *JOP characteristics*

To successfully launch a JOP attack, a rigorous gadget discovery algorithm is required that must chain together pieces of existing code to overwrite some of the architectural registers.

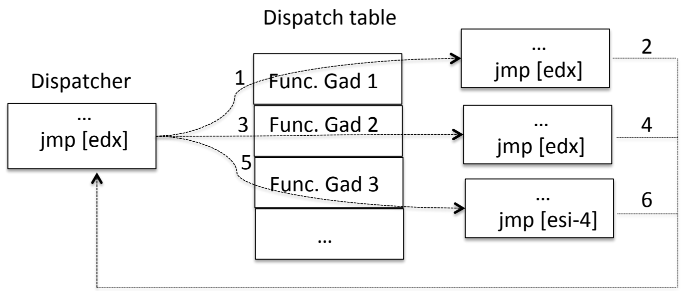


Fig. 1. JOP Architecture with control flow sequences (shown above arrows).

```
popa; //initializer gadget
jmp [ebx-0x3e];
add ebp, edi; //dispatcher gadget
jmp [ebp-0x39];
popa; //func. gadget 1
fdivr st(1), st;
jmp [edx];
inc eax; //func. gadget 2
fdivr st(1), st;
jmp [edx];
mov [ebx-0x17bc0000], ah; //func. gadget 3
stc;
jmp [edx];
inc ebx; //func. gadget 4
fdivr st(1), st;
jmp [edx];
popa; //func. gadget 5
cmc;
jmp dword [ecx];
xchg ecx, eax; //func. gadget 6
fdiv st, st(3);
jmp [esi-0xf];
mov eax, [esi+0xc]; //func. gadget 7
mov [esp], eax;
call [esi+0x4];

//At this point, eax is set to
//execve, ebx points to "/bin/bash"
//argv(ecx) and envp(edx) are NULL
sysenter; //func. gadget 8
```

Fig. 2. Example JOP x86 code snippet that launches unauthorized shell.

Also, since JOP relies on indirect jump and call instructions, the gadget discovery algorithm has to carefully choose code segments that end in either indirect jump or indirect call instructions.

Gadget length is one of the important factors in successfully launching JOP. Kayaalp et al. [2] conducted studies on average gadget lengths that can extracted from standard C library (libc), and found that a vast majority (≈85%) of gadgets have 5 or less instructions. This is because lengthier gadgets have potential side-effects such as having a set of instructions that cannot be used for meaningful JOP computation, possibility of unintended change of register values that may ruin a successful JOP attack and so on.

Another factor to consider is the target distance of indirect jumps and indirect calls. Note that it is relatively hard to find gadgets (that are suitable for use) within short distances (or short range of addresses). In fact, we measured the values of target distance in the successful JOP code written by Bletsch et al [1]. Our results showed that the average distance between functional gadgets from libc code was 227,466 bytes with the maximum being 1,096,593 bytes and the minimum being 67,010 bytes. Since the gadget discovery algorithms need to find reasonable gadgets from libc to launch a JOP attack, the minimum and average target distances for indirect jumps and indirect calls can also be valuable metrics to capture possible JOP attack during program execution.

III. Threat Model and Assumptions

Our solution aims to *detect* JOP-based attack, and *does not prevent JOP*. We assume that the underlying system supports W⊕X model that prohibits writes to executable memory, and hence assume that code injection based attacks are not possible. We assume that the attacker is able to perform a stack or a heap overflow to overwrite the target address in the jump buffer (setjmp.h of libc code) and transfer control to initializer gadget.

We assume that the adversary operates in the user mode and not in privileged mode. Although the detection of JOP had no bearing on whether the system is in user or privileged mode, we note that a sufficiently privileged adversary might be able to overrule known JOP defenses and render JOP detection and defense mechanisms useless.

Kayaalp et al. [2] describe stealth-JOP, that uses lengthier delay gadgets to evade detection. Realistically, such gadgets are harder to find and integrate into the JOP attack due to higher likelihood of side effects. Our evaluation includes modeling of state of the art delay gadgets, and our algorithm correctly detects JOP attack in this instance.

IV. JOP-Alarm: Detecting the presence of Jump-Oriented Programming

Based on the discussion of commonly observed characteristics of JOP in Section II-B, we present our JOP-alarm, a detection algorithm to assess the possibility of JOP. We note that our algorithm is completely tune-able and can be adjusted for more conservative settings based on user preferences.

$$score_{inc} = f_{ind\_jmp/ind\_call}(jumptarget\_dist, step\_up\_value) \quad (1)$$

$$score_{dec} = f_{otherinst}(step\_down\_value) \quad (2)$$

Our algorithm (shown in Figure 3) uses a scoring-based approach to detect JOP. The score is incremented by a step_up_value every time we encounter an indirect call or indirect jump instruction as shown in Equation 1. This function takes distance of the jump target as its input to reduce the false positive rate that are likely to result from indirect jumps emitted by compilers in certain common cases). For all other instructions, we decrement the score by a constant step_down_value as shown in Equation 2. This is done to avoid saturating the score value through repeated additions of step_up_value. Intuitively, this decrement operation serves to slowly forget the effect of indirect jumps and calls, especially when there are long sequences of instructions between two

```
At the beginning of the program:
Initialize score_value to 0

//dist_threshold -- set by the user based
//empirically measured minimum
//jump target distance of indirect jumps
//and calls in JOP gadgets

//jop_threshold -- empirically estimated
//score value that may likely indicate
//JOP attack

for every indirect jump or indirect call
instruction:
if (jumptarget_dist >= dist_threshold) {
  score += step_up_value;
}
else { //jumptarget_dist < dist_threshold
  //not a potential gadget; do nothing
}
if (score >= jop_threshold) {
  //potential JOP attack
  RAISE_alarm();
}

for every other instruction:
if (score > 0) {
  score -= step_down_value;
}
```

Fig. 3. JOP-Alarm Algorithm

indirect jumps (that are unlikely to be JOP gadgets). On the other hand, if there are few instructions in between two indirect jumps or calls (e.g., JOP gadgets), the number of decrement operations will be fewer in comparison to the increment operations. This will result in a *large score indicating the presence of frequent indirect jumps/calls interspersed with a short number of other instructions (potential JOP gadgets)*. These increment/decrement operations are necessary to build the hysteresis information on code execution involving a mix of indirect jumps/calls and other instructions.

Our algorithm maintains a *score* variable throughout the program execution that is initialized to zero at the start of the program. There are two thresholds that are empirically determined, and used by our algorithm namely,

- *dist_threshold*, that is set by the user, based on the empirically measured minimum value of the target distance in indirect jumps and calls in JOP gadgets. We *conservatively* set the dist_threshold as 4,196 bytes, a value actually much lower than the observed minimum target distance (See Section II-B). We note that we could set this threshold even lower for a more conservative JOP assessment.
- *jop_threshold*, that is empirically estimated score value to indicate potential JOP attack. Through careful analysis of existing JOP attacks [1], [2], we assume 20 to be the step_up_value[1] and a step_down_value of 1, the score value at the end of fifth gadget would be at least 120. Therefore, we set 120 to be the value of jop_threshold.

[1]Realistic JOP gadgets in libc do not have more than 10 instructions [2], and hence 20 would be a sufficiently conservative step_up_value needed to remember the occurrence of indirect jump and indirect call instructions.

Standard compiler emits indirect jumps and calls to reach long distance targets in programs. However, there are two special circumstances where the compiler may use indirect jumps despite short target distance. This includes (1) switch..case.. statements, and (2) virtual function calls.

Fortunately, in the case of switch..case.. statements, we find that the number of statements within each *case* statement is small enough that our preset dist_threshold value of 4,196 bytes is sufficient to ignore these benevolent indirect jumps without treating them as JOP gadgets.

In case of virtual function calls, however, we find that a vast majority of cases in SPEC2006 applications [5] do not trigger any false alarms since their jumptarget_dist value is observed to be less than 4,196. To minimize false postive alarms, we could set the dist_threshold value to a higher value such as 8,192 or 16,384.

Our JOP-alarm algorithm can be implemented either completely in software or using simple hardware at the commit stage of the processor pipeline.

V. Evaluation

A. Setup

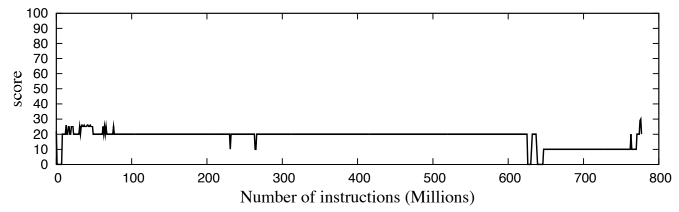
We used marssx86 [3], a tool for cycle accurate full system simulation of the x86-64bit archiecture. Our simulated machine configuration has a single, out-of-order core configuration with 128 KB L1 and 2 MB L2 cache. We run marssx86 on CentOS 6.4 with 3.0 GHZ Intel Xeon CPU. We evaluate our JOP-alarm algorithm using a number of SPEC2006 benchmarks [5] with reference input sets. The threshold, step_up and step_down values used in our experiments are shown in section IV.

B. Score values in unmodified SPEC 2006 benchmarks

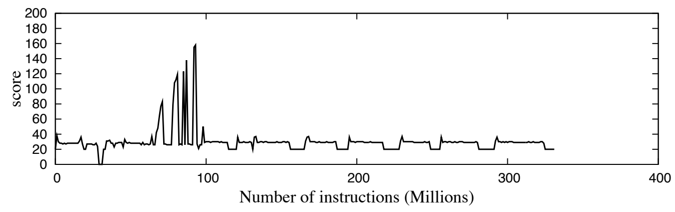
We first conduct experiments to understand the effect of our scoring approach in regular, unmodified SPEC2006 applications [5] that have indirect jumps including switch..case..statements and virtual function calls. We show some of our results over a wide variety of benchmarks in Figure 4. The x-axis in each benchmark shows the number of instructions simulated (in millions) and the y-axis shows the highest score observed within windows of million instructions, as calculated by our JOP-alarm algorithm in Figure 3. We observe that most benchmarks do not trigger any false positives except xalancbmk which has a short virtual function (with 7 x86 instructions) called frequently.

C. JOP code without and with delay gadgets

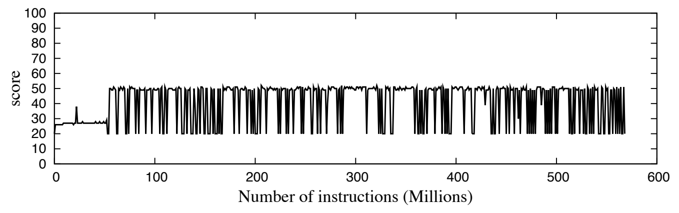
We perform experiments to check the functionality of our algorithm on the code published by Bletsch et al [1]. The code under test has a total of 10 JOP gadgets and is very similar to the JOP example presented in Figure 2. The values of score calculated by our JOP-alarm algorithm is shown in Figure 5(a), where it reaches a maximum of around 460 at about 800,000 instruction mark during program execution. This maximum value of score is well above our jop_threshold of



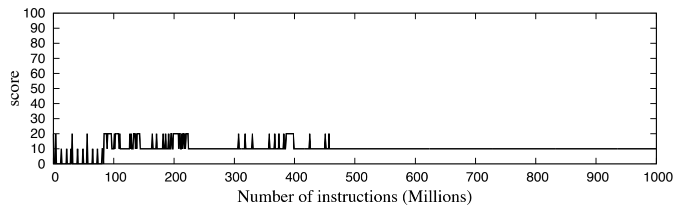
(a) gcc



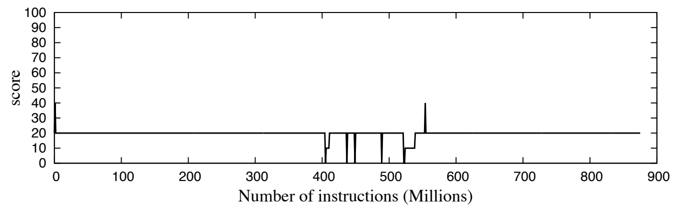
(b) xalancbmk



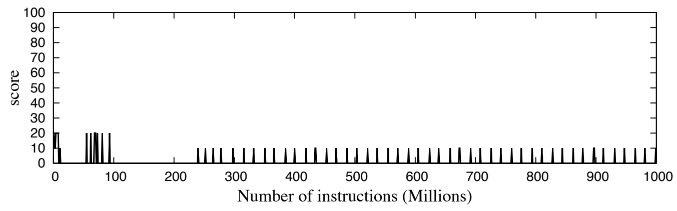
(c) omnetpp



(d) sjeng

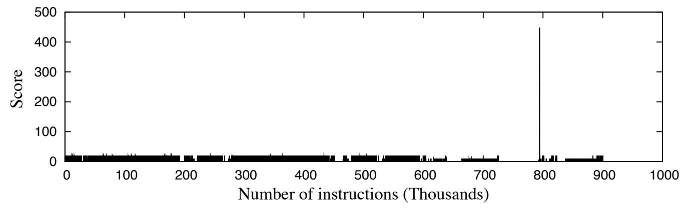


(e) soplex

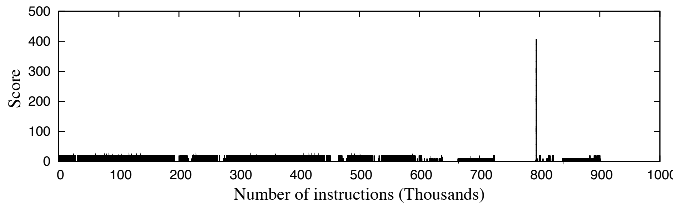


(f) milc

Fig. 4. Highest JOP-alarm score within consecutive windows of one million instructions across Spec2006 benchmarks.



(a) Without delay gadget



(b) With delay gadgets

Fig. 5. JOP-alarm score on codes without and with delay gadgets. In case of delay gadgets, we assume a medium length gadget with 15 instructions for every third JOP gadget. The rest of the JOP gadgets have a maximum of 4 instructions.

120, and hence we are able to reliably detect the presence of JOP attack.

Kayaalp et al. [2] introduced the notion of delay gadgets, where certain medium length gadgets are intentionally inserted between the regular JOP gadgets to evade detection. Figure 5(b) shows the results of our experiments where 3 out of 10 gadgets are assumed to be delay gadgets. Even with such medium length gadgets, the score value still reaches to a high score of around 400 due to indirect jumps at the end of dispatcher and functional gadgets resulting in reliable discovery of JOP attack.

VI. Conclusions and Future Work

In this paper, we proposed JOP-alarm, a novel and elegant algorithm to detect Jump-oriented programming based code reuse attacks. The algorithm understands the JOP characteristics and utilizes this information to accurately assess the potential for JOP during program execution. We adopt a scoring based approach to detect JOP and demonstrate the effectiveness of our algorithm on published JOP code.

As future work, we plan to extend our framework to other types of code reuse attacks, and develop algorithms by understanding the common characteristics of such exploits.

References

[1] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 2011, pp. 30–40.

[2] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *Proceedings of the 2013 IEEE conference on High Performance Computer Architecture*, 2013.

[3] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011 (DAC'11)*, 2011.

[4] 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*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[5] Standard Performance Evaluation Corporation, "Spec 2006 benchmark suite," *http://www.spec.org*, 2006.