# Jump-Oriented Programming: A New Class of Code-Reuse Attack

Tyler Bletsch, Xuxian Jiang, Vince W. Freeh
Department of Computer Science
North Carolina State University
{tkbletsc, xuxian_jiang, vwfreeh}@ncsu.edu

Zhenkai Liang
School of Computing
National University of Singapore
liangzk@comp.nus.edu.sg

## ABSTRACT

Return-oriented programming is an effective code-reuse attack in which short code sequences ending in a `ret` instruction are found within existing binaries and executed in arbitrary order by taking control of the stack. This allows for Turing-complete behavior in the target program without the need for injecting attack code, thus significantly negating current code injection defense efforts (e.g., W⊕X). On the other hand, its inherent characteristics, such as the reliance on the stack and the consecutive execution of return-oriented gadgets, have prompted a variety of defenses to detect or prevent it from happening.

In this paper, we introduce a new class of code-reuse attack, called *jump-oriented programming*. This new attack eliminates the reliance on the stack and `ret` instructions (including `ret`-like instructions such as `pop`+`jmp`) seen in return-oriented programming without sacrificing expressive power. This attack still builds and chains *functional gadgets*, each performing certain primitive operations, except these gadgets end in an indirect branch rather than `ret`. Without the convenience of using `ret` to unify them, the attack relies on a *dispatcher gadget* to dispatch and execute the functional gadgets. We have successfully identified the availability of these jump-oriented gadgets in the GNU libc library. Our experience with an example shellcode attack demonstrates the practicality and effectiveness of this technique.

## 1. INTRODUCTION

Network servers are under constant threat by attackers who use maliciously crafted packets to exploit software bugs and gain unauthorized control. In spite of significant research addressing the underlying causes of software vulnerabilities, such attacks remain one of the largest problems in the security field. An arms race has developed between increasingly sophisticated attacks and their corresponding defenses.

One of the earliest forms of software exploit is the *code injection attack*, wherein the malicious message includes machine code, and a buffer overflow or other technique is used to redirect control flow to the attacker-supplied code. However, with the advent of CPUs and operating systems that support the W⊕X guarantee [3], this threat has been mitigated in many contexts. In particular, W⊕X enforces the property that "a given memory page will never be both writable and executable at the same time." The basic premise behind it is that if a page cannot be written to and later executed from, code injection becomes impossible.

Unfortunately, attackers have developed innovative ways to defeat W⊕X. For example, one possible way is to launch a *code-reuse attack*, wherein existing code is re-purposed to a malicious end. The simplest and most common form of this is the *return-into-libc* technique [33]. In this scenario, the adversary uses a buffer overflow to overwrite part of the stack with return addresses and parameters for a list of functions within libc (the core C library that is dynamically linked to all applications in UNIX-like environments). This allows the attacker to execute an arbitrary sequence of libc functions, with a common example being a call to `system("/bin/sh")` to launch a shell.

While return-into-libc is powerful, it does not allow arbitrary computation within the context of the exploited application. For this, the attacker may turn to *return-oriented programming* (ROP) [36]. As before, ROP overwrites the stack with return addresses and arguments. However, the addresses supplied now point to arbitrary points within the existing code base, with the only requirement being that these snippets of code, or *gadgets*, end in a `ret` instruction to transfer the control to the next gadget. Return-oriented programming has been shown to be Turing complete on a variety of platforms and codebases [10, 15, 21, 32, 30], and automated techniques have made development of such attacks a straightforward process [10, 23, 30]. The real-world danger of this technique was shown when Checkoway et al. used it to violate the integrity of a commonly deployed electronic voting machine [15].

Since the advent of return-oriented programming, a number of defenses have been proposed to either detect or prevent ROP-based attacks. For example, DynIMA [18] detects the consecutive execution of small instruction sequences each ending with a `ret` and suspects them as gadgets in a ROP attack. DROP [16] observes that a ROP execution continuously pops return addresses that always point to the same specific memory space, and considers this as a ROP-inherent feature to be useful for its detection. The return-less approach [31] goes a step further by eliminating all `ret` instructions in a program, thereby removing the existence of return-oriented gadgets and precluding the possibility of a ROP-based attack.


*ASIACCS '11,* March 22–24, 2011, Hong Kong, China.

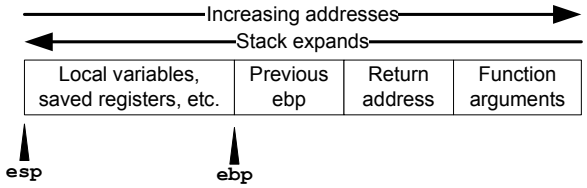



**Figure 1: Simplified layout of an x86 stack frame.**

In this paper, we present an alternative attack paradigm called *jump-oriented programming* (JOP). In a JOP-based attack, the attacker abandons all reliance on the stack for control flow and `ret` for gadget discovery and chaining, instead using nothing more than a sequence of indirect jump instructions. Because almost all known techniques to defend against ROP depend on its reliance on the stack or `ret`, none of them are capable of detecting or defending against this new approach. The one exception are systems that enforce full control-flow integrity (e.g. [4]); unfortunately, such systems are not widely deployed, likely due to concerns over their complexity and negative performance impact.

Similar to ROP, the building blocks of JOP are still short code sequences called *gadgets*. However, instead of ending with a `ret`, each such gadget ends with an indirect `jmp`[1]. Some of these `jmp` instructions are intentionally emitted by the compiler. Others are not intended but present due to the density of x86 instructions and the feasibility of unaligned execution. However, unlike ROP, where a `ret` gadget can naturally return back the control based on the content of the stack, a `jmp` gadget is performing an uni-directional control-flow transfer to its target, making it difficult to regain control back to chain the execution of the next jump-oriented gadget.

We note that a code-reuse attack based on indirect `jmp`s was put forth as a theoretical possibility as early as 2003 [35]. However, there always remained an open problem of how the attacker would maintain control of the program's execution. With no common control mechanism like `ret` to unify them, it was not clear how to chain gadgets together with uni-directional `jmp`s.

Our solution to this problem is the proposition of a new class of gadget, the *dispatcher gadget*. Such a gadget is intended to govern control flow among various jump-oriented gadgets. More specifically, if we consider other gadgets as *functional gadgets* that perform primitive operations, this dispatcher gadget is specifically selected to determine which functional gadget is going to be invoked next. Naturally, the dispatcher gadget can maintain an internal dispatch table that explicitly specifies the control flow of the functional gadgets. Also, it ensures that the ending `jmp` instruction in the functional gadget will always transfer the control back to the dispatcher gadget. By doing so, jump-oriented computation becomes feasible.

In order to achieve the same Turing-complete expressive power of ROP, we also aim to identify various jump-oriented gadgets for memory load/store, arithmetic calculations, binary operations, conditional branching, and system calls. To do that, we propose an algorithm to discover and collect jump-oriented gadgets, organize them into different categories, and save them in a central gadget catalog.

[1]Independent of our work, a concurrent approach by Checkoway *et al.* [14] proposes to replace `ret` in ROP with a `pop+jmp`. However, the x86-based approach still relies on the stack to govern control flow among gadgets and the `pop+jmp` sequence is rare – as detailed in Section 6.

In summary, this paper makes the following contributions:

1. We expand the taxonomy of code-reuse attacks on the x86 to include a new class of attack: *jump-oriented programming*. When compared to existing return-oriented programming, our attack has the benefit in *not* relying on the stack for control flow. Instead, we introduce the notion of a *dispatcher gadget* to take the role of executing functional gadgets.

2. We present a heuristic-based algorithm to effectively discover a variety of jump-oriented gadgets on the x86, including the critical dispatcher gadget. Our results indicate that all of these gadgets are abundantly available in GNU libc that is dynamically linked to almost all UNIX applications.

3. We demonstrate the efficacy of this technique with a jump-oriented shellcode attack based on the gadgets discovered by our algorithm.

The rest of the paper is organized as follows: Section 2 provides a background of the relevant aspects of the x86 architecture and the existing ROP methodology. Next, Section 3 explains the design of the new jump-oriented programming attack, then Section 4 presents an implementation on an x86 Linux system, including a concrete example attack. Section 5 examines the limitations of our approach and explores ways for improvement. Finally, Section 6 covers the related work and Section 7 concludes this paper.

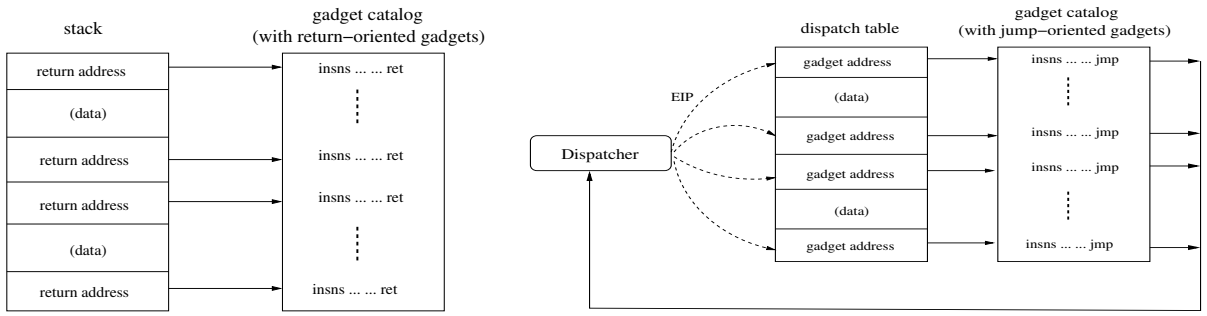
## 2. BACKGROUND

To understand the contributions of this paper, it will be necessary to briefly summarize the techniques behind return-oriented programming. Since our system is developed on the 32-bit x86 architecture[2], our discussion primarily focuses on that platform.

As illustrated in Figure 1, the x86 stack is managed by two dedicated CPU registers: the `esp` "stack pointer" register, which points to the top of the stack, and the `ebp` "base pointer" register, which points to the bottom of the current stack frame. Because the stack grows downward, i.e., grows in the direction of decreasing addresses, `esp` $\leq$ `ebp`. Each stack frame stores each function call's parameters, return address, previous stack frame pointer, and automatic (local) variables, if any. The stack content or pointers can be manipulated directly via the two stack registers, or implicitly through a variety of CPU opcodes, such as `push` and `pop`. The instruction set includes opcodes for making function calls (`call`) and returning from them (`ret`)[3]. The `call` instruction pushes the address of the next instruction (the return address) onto the stack. Conversely, the `ret` instruction pops the stack into `eip`, resuming execution directly after the `call`.

[2]The x86 assembly language used in this paper is written in Intel syntax. This means that destination operands appear first, so `add eax,ebx` indicates $eax \leftarrow eax + ebx$. Memory dereference is indicated by brackets, e.g., `[eax]`. Also, the x86 platform allows dereference operations to encode fairly complex expressions within a single instruction, e.g., `[eax+ebx*4+0x1234]`.

[3]There are actually multiple flavors of `call` and `ret` to support inter-segment control transfers ("far" calls) and automatic stack unwinding. For this discussion, these distinctions have little relevance, so we speak about `call` and `ret` in generic terms.




(a) The ROP model (b) The JOP model

**Figure 2: Return-oriented programming (ROP) vs. jump-oriented programming (JOP)**

An attacker can exploit a buffer overflow vulnerability or other flaw to overwrite part of the stack, such as replacing the current frame's return address with a supplied value. In the traditional return-into-libc approach, this new value is a pointer to a function in libc chosen by the attacker. After the victim program uses the new value and enters the function, the memory cells next to the overwritten return address are interpreted as parameters by the function, allowing the execution of an arbitrary function with attacker-specified parameters. By chaining these malicious stack frames together, a sequence of functions can be executed. While this is undoubtedly a very powerful ability, it does not allow the attacker to perform arbitrary computation. For that, the attack needs to launch another process (e.g., via `exec()`) or alter memory permissions to make a traditional code injection attack possible (e.g., via `mprotect()`).

Because these operations may lead to detection or interception, the stealthy attacker may instead turn to return-oriented programming, which allows arbitrary computation within the context of the vulnerable application. Return-oriented programming is driven by the insight that return addresses on the stack can point *anywhere*, not just to the beginning of functions like in a classic return-into-libc attack. Therefore, it can direct control flow through a series of small snippets of existing code, each ending in `ret`. These small snippets of code are called *gadgets*, and in a large enough codebase (such as libc), there is a massive selection of gadgets to choose from. On the x86 platform, the selection is made even larger because instructions are of variable length, so the CPU will interpret the raw bytes of an instruction differently if decoding is started from a different offset.

Based on this, the return-oriented program is simply a sequence of gadget addresses and data values laid out in the vulnerable program's memory. In a traditional attack, it is overflowed into the stack, though the buffer can be loaded elsewhere if the attacker can redirect the stack pointer `esp` to the new location. The gadget addresses can be thought of as opcodes in a new return-oriented machine, and the stack pointer `esp` is its program counter. Under this definition, just as a basic block of traditional code is one that does not explicitly permute the program counter, a "basic block" of return-oriented code is one that does not explicitly permute the stack pointer `esp`. Conversely, conditional branches and loops can be created by changing the value of `esp` based on logic. The combination of arithmetic, logic, and conditional branching yields a Turing complete return-oriented machine. A set of gadgets that satisfies these requirements was first discovered on the x86 [36] and later expanded to many other platforms [10, 15, 21, 32, 30]. In addition, such attacks can also make arbitrary system calls, as this is simply a matter of calling the appropriate library routine, or even accessing the kernel system call interface directly. Because of this, a return-oriented attack is equivalent in expressive power to a successful code injection.

A number of researchers have attempted to address the problem of return-oriented programming. Each of the proposed defense systems identifies a specific trait exhibited by return-oriented attacks and develops a detection or prevention measure around it. Some enforce the LIFO stack invariant [19, 22], some detect excessive execution of the `ret` instruction [16, 18], and one went so far as to eliminate every instance of the `ret` opcode from the kernel image [31]. What these techniques have in common is that they all assume that the attack must use the stack to govern control flow. This paper introduces *jump-oriented programming* as a new alternative that has no reliance on the stack, and is therefore immune to such defenses.

**Threat model** In this work, we assume the adversary can put a payload (e.g., the dispatch table – Section 3) into memory and gain control of a number of registers, especially the instruction pointer `eip` to divert the program execution. The assumption is reasonable, as several common vulnerabilities such as buffer overruns, heap overflows, and format string bugs exist that fulfill this requirement. We also assume the presence of a significant codebase in which to find gadgets. As with ROP, we find that this can be fulfilled solely with the content of libc, which is dynamically linked to all processes in UNIX-like environments. On the defensive side, the vulnerable program is protected by a strict enforcement of code integrity (e.g., W⊕X) that defeats the traditional code injection attack.

# 3. DESIGN

Figure 2 compares return-oriented programming (ROP) and our proposed jump-oriented programming (JOP). As in ROP, a jump-oriented program consists of a set of gadget addresses and data values loaded into memory, with the gadget addresses being analogous to opcodes within a new jump-oriented machine. In ROP, this data is stored in the stack, so the stack pointer `esp` serves as the "program counter" in a return-oriented program. JOP is not limited to using `esp` to reference its gadget addresses, and control flow is not driven by the `ret` instruction. Instead, JOP uses a dispatch table to hold gadget addresses and data. The "program counter" is any register that points into the dispatch table. Control flow is driven by a special *dispatcher gadget* that executes the sequence of gadgets. At each invocation, the dispatcher advances the virtual program counter, and launches the associated gadget.

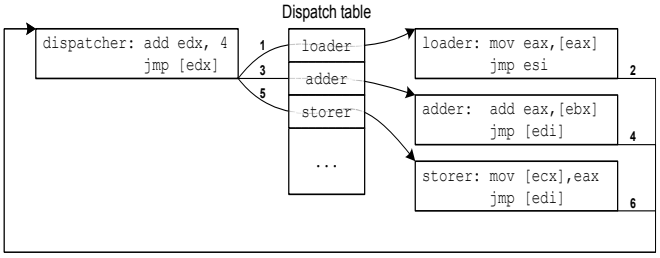



**Figure 3: Control flow in an example jump-oriented program, with the order of jumps indicated by the numbers 1..6. Here, `edx` is used as *pc*, which the dispatcher advances by simply adding 4 to get to the next word in a contiguous gadget address table (so $f(pc) = pc+4$). The functional gadgets shown will (1) dereference `eax`, (2) add the value at address `ebx` to `eax`, and (3) store the result at the address `ecx`. The registers `esi` and `edi` are used to return control to the dispatcher – `esi` does so directly, whereas `edi` goes through a layer of indirection.**

An example control flow of a JOP program is shown in Figure 3. In this example, we essentially add two memory values (pointed to by `eax` and `ebx`, respectively) and store the sum into another memory location pointed to by `ecx`, i.e., [`ecx`] ← [`eax`] + [`ebx`].

The main goal of this work is to demonstrate the feasibility of jump-oriented programming. We show that its expressive power is comparable to that of return-oriented programming. However, by not relying on the stack for control flow, JOP can potentially use any memory range, including even non-contiguous memory, to hold the dispatch table.

Below, we further elaborate on the dispatcher gadget (Section 3.1) as well as the functional gadgets (Section 3.2) whose primitive operations comprise the actual computation. After that, we discuss how to discover these gadgets from the commonly available codebase (Section 3.3). Finally, we explore possible ways to bootstrap a jump-oriented program (Section 3.4).

## 3.1 The Dispatcher Gadget

The dispatcher gadget plays a critical role in the JOP technique. It essentially maintains a virtual program counter, or *pc*, and executes the JOP program by advancing it through one gadget after another. Specifically, each *pc* value specifies an entry in the dispatch table, which points to a particular jump-oriented functional gadget. Once invoked, each functional gadget will perform a basic operation, such as arithmetic calculation, branching, or the invocation of a particular system call.

We consider any jump-oriented gadget that carries out the following algorithm as a dispatcher candidate.

$pc \leftarrow f(pc)$;
$goto\ *pc$;

Here, *pc* can be a memory address or register that represents a pointer into our jump-oriented program. It is *not* the CPU's instruction pointer—it refers to a pointer in the gadget table supplied by the attacker. The function $f(pc)$ is any operation that changes the program counter *pc* in a predictable and evolving way. In some cases, it may be simply expressed via pure arithmetic (e.g., $f(pc) = pc + 4$ as shown in Figure 3). In other cases, it could be a memory dereference operation (e.g., $f(pc) = *(pc-18)$) or any other expression that can be predicted by the attacker beforehand. Each time the dispatcher gadget is invoked, the *pc* will be advanced accordingly. Then the dispatcher dereferences it and jumps to the resulting address.[4]

Given the wide definition of what constitutes a dispatcher, we had little trouble in finding several viable candidates within libc. The way the dispatcher gadget advances the *pc* affects the organization of the dispatch table. Specifically, the dispatch table can be a simple array if *pc* is repeatedly advanced by a constant value (e.g., $f(pc) = pc+4$) or a linked list if memory is dereferenced (e.g., $f(pc) = *(pc-18)$). The example attack in Section 4 uses an array to organize the dispatch table.

This new programming model expands the basic code-reuse attack used in ROP. Specifically, if we consider the stack used in a ROP-based program as its dispatch table and `esp` as its *pc*, the `ret` instruction at the end of each return-oriented gadget acts as a dispatcher that advances the *pc* by 4 each time a gadget is completed, i.e., $f(pc) = pc + 4$. However, all ROP-based attacks still rely on the stack, which is no longer necessary in a JOP-based attack.

## 3.2 Functional Gadgets

The dispatcher gadget itself does not perform any actual work on its own—it exists solely to launch other gadgets, which we call *functional gadgets*. To maintain control of the execution, all functional gadgets executed by the dispatcher must conclude by jumping back to it, so that the next gadget can be launched.

More formally, a functional gadget is defined as a number of useful instructions ending in a sequence that will load the instruction pointer with the result of a known expression. This expression may be a register (`jmp edx`), a register dereference (`jmp [edx]`), or a complex dereference expression (`jmp [edx+esi*4-1]`). The only requirement is that by the time the branch is executed, it must evaluate to the address of the dispatcher, or to another gadget that leads to the dispatcher. However, the attack does not rely on specific operands for each of these branches: functional gadgets may change the CPU state in order to make available a different set of gadgets for the next operation. For example, one gadget may end in `jmp [edx]`, then a second may use the `edx` register for a computation before loading `esi` with the dispatcher address and terminating with `jmp esi`. Furthermore, the functional gadget may have an effect on *pc*, which makes it possible to implement conditional branching within the jump-oriented program, including the introduction of loops. The most obvious opcode to use for the branch is an indirect jump (`jmp`), but one interesting thing to note is that because there is no reliance on the stack, we can also use sequences that end in a `call`, because the side effect of pushing the return address to the stack is irrelevant.

There are a few different kinds of functional gadgets needed to obtain the same expressive power of ROP, which we briefly review below. Examples will be presented in Section 4.

**Loading data** In the return-oriented approach, there is an obvious place to place data: in the stack itself. This allows ubiquitous `pop` instructions to load registers. In JOP, however, one may load data values in a variety of ways –

[4]On the x86, it is possible to add a constant to a register and dereference the result within one instruction; such instructions can be used in dispatchers without difficulty, as the constant is known beforehand.

any gadget that loads from and advances a pointer will do. On the x86, there are a variety of string loading and loop sequences that do this. Further, even though JOP does not rely on the stack for control flow, there is no reason the stack cannot be co-opted to serve as a data loading mechanism as in ROP, as the existing defense techniques focus on protecting stack-based control flow, not simple data access. In our implementation, the stack pointer `esp` is redirected and the stack is used for this purpose.

**Memory access** To access memory, load and store gadgets are required. These gadgets take a memory address and reads or writes a byte or word at that location.

**Arithmetic and logic** Once operands (or pointers to operands) are loaded into CPU registers, ALU operations can be applied by finding gadgets with the appropriate opcodes (`add`, `sub`, `and`, `or`, etc.).

**Branching** Unconditional branching can be achieved by modifying the register or memory location used for *pc*. Conditional branching is performed by adjusting *pc* based on the result of a previous computation. This may be achieved several ways, including adding a calculated value to *pc*, using a short conditional branch within a gadget to change *pc* based on logic, or even using the x86's special *conditional move* instruction to update *pc* (`cmov`).

**System calls** While the above gadgets are sufficient to make JOP Turing complete (i.e., capable of arbitrary computations), system calls are needed to carry out most practical tasks. There are a few different ways to make a system call. First, it is possible to call legitimate functions by setting up the stack with appropriate parameters and a return address of a gadget that will restore the appropriate CPU state and execute the dispatcher. However, because it may be possible for existing defenses against ROP to detect this, a more prudent approach is to make system calls directly. The methodology for doing this varies by CPU and operating system. On the x86-based Linux, one may execute `int 0x80` to raise an interrupt, jump to a kernel-provided routine called `__kernel_vsyscall` to execute a `sysenter` instruction, or even execute a `sysenter` instruction directly.

## 3.3 Gadget Discovery

The naïve method to locate gadgets within the target binary is to simply disassemble it and search for indirect jump or call instructions. However, instructions on the x86 platform are of variable length, so decoding the same memory from one offset versus another can yield a very different set of operations. This means that every x86 binary contains a number of *unintended* code sequences that can be accessed by jumping to an offset not on an original instruction boundary. Given this, an algorithm for locating gadgets ending in `ret` was given by Shacham in the context of ROP [36].

We adopt a similar approach in our gadget discovery process. The algorithm works by scanning the executable region of the binary for the valid starting byte(s) of an indirect branch instruction. On the x86, this consists of the byte `0xff` followed by a second byte with a specific range of values.[5] Such sequences can be located by a linear search. From there, it is a simple matter to step backwards byte by byte and decode each possible gadget terminating in the indirect jump. This approach is defined formally in Algorithm 1.

As shown in the algorithm, the $FindGadget(C)$ procedure uses a string search to find indirect jumps in a codebase $C$,

[5]For full details on the precise encoding of indirect `jmp` and `call` instructions, see [24].

**Algorithm 1**

**procedure** $IsViableGadget(G)$
1: $V \leftarrow$ {Registers and writable memory addresses}
2: $J \leftarrow$ (Last instruction of $G$)
3: **if** ($J$ is not an indirect jump) $\vee$ ($J.operand \notin V$) **then**
4: **return false**
5: **end if**
6: $A \leftarrow$ {Addresses of each instruction in $G$}
7: **for all** instructions $I \in G$, such that $I \neq J$ **do**
8: **if** ($I$ is illegal) $\vee$ (($I$ is a branch) $\wedge \neg$(($I$ is a conditional jump) $\wedge$ ($I.operand \in A$))) **then**
9: **return false**
10: **end if**
11: **end for**
12: **return true**

**procedure** $FindGadgets(C)$
1: **for** each address $p$ that is an indirect branch in $C$ **do**
2: $len \leftarrow$ (Length of the branch at $C[p]$)
3: **for** $\delta = 1$ to $\delta_{max}$ **do**
4: $G \leftarrow disassemble(C[p - \delta : p + len])$
5: **if** $IsViableGadget(G) \wedge Heuristic(G)$ **then**
6: **print** $G$
7: **end if**
8: **end for**
9: **end for**

then walks backwards by up to $\delta_{max}$ bytes and disassembles each resulting code region. The value of $\delta_{max}$ is the maximum size of a gadget, in bytes. Its selection depends on the average length of instructions on the given architecture and the maximum number of instructions per gadget to consider. Our experience is that, as observed in ROP [36], useful gadgets need not be longer than 5 instructions.

There are several criteria by which a potential gadget can be eliminated at this stage; these are detected by the procedure $IsViableGadget(G)$. First, because the algorithm walks backward one byte at a time, it is possible that the sequence that was originally an indirect jump is no longer interpreted as such. If this is the case, the gadget is eliminated. Second, the target of an indirect jump can be a register value (e.g., `esi`), the address pointed to by a register (`[esi]`), or the address pointed to by a memory dereference (`[0x7474505b]`). In the latter case, if the address given is not likely to be valid, writable location at runtime, then the gadget is eliminated. Third, if any part of the gadget does not encode a legal x86 instruction, the gadget is eliminated. Finally, the gadget itself may contain a conditional branch separate from the indirect branch at the end. If the target of this branch lies outside of the gadget bounds, the gadget is eliminated. Further, if the target of the branch does not align with the instructions identified in the gadget, it is eliminated.

This yields the set of potentially useful gadgets in the codebase, and on a large codebase such as libc, that will mean tens of thousands of candidate gadgets. The set is narrowed down further by $Heuristic(G)$, which filters gadgets based on their viability for a particular purpose. While there has been much work on completely automating the gadget search in ROP [10, 23, 30], the JOP gadget search adds additional complexity. Because each gadget must end with a jump back to the dispatcher, care must be taken to ensure that the register used for this purpose is set properly before it is needed. This introduces two requirements when locating and chaining jump-oriented gadgets. (1) The gadget must not destroy its own jump target. The target may be

modified, however, if this modification can be compensated for by a previous gadget. For example, if a gadget increments `edx` as a side-effect before ending in `jmp [edx]`, then the value of `edx` when the gadget starts should be one less than the intended value. (2) Because gadgets are chained together, the side-effects of an earlier gadget must not disturb the jump targets of subsequent ones. For example, if a register is used for a calculation in gadget $A$ and used as a jump target in gadget $B$, then an intervening gadget must set this register to the dispatcher address before gadget $B$ can be used.

Because of this added complexity, the search for gadgets in this work requires additional heuristics, represented in the algorithm as $Heuristic(G)$. We describe the most interesting of these heuristics below.

To locate potential dispatcher gadgets within the codebase, we developed the *dispatcher heuristic*. This algorithm works by filtering all the potential gadgets located by the search algorithm down to a small set from which the attack designer can choose. For each gadget, we begin by getting the jump target in the gadget's last instruction, then examining the first instruction in the gadget sequence based on three conditions.

First, the instruction must have the jump target as its destination operand. If the gadget is not modifying the jump target, then it cannot be a dispatcher.

Second, we filter the gadgets based on opcode. Because of the wide variety of x86 opcodes which could possibly advance *pc*, it is more expedient to filter opcodes via a blacklist rather than a whitelist. Therefore, we throw out opcodes that are unable to permute the target by at least the word size.

Third, operations that completely overwrite the destination operand (e.g., `mov`) must be *self-referential*, i.e., the destination operand is also present within the source operands. For example, the "load effective address" opcode (`lea`) can perform calculations based on one or more registers. The instruction `lea edx,[eax+ebx]` is unlikely to be useful within a dispatcher, as it overwrites `edx` with the calculation `eax+ebx` – it does not advance `edx` by a predictable value. Conversely, the instruction `lea edx,[edx+esi]` advances `edx` by the value stored in `esi`, and is therefore a dispatcher candidate. The self-referential requirement is not strictly necessary, as there could be a multi-register scheme that could act as a dispatcher, but enforcing the requirement simplifies the search considerably by eliminating a vast number of false positives.

Once the gadgets have been filtered by these three conditions, we examine each candidate and choose one that uses the least commonly used registers. This is because the register or registers used by the dispatcher will be unavailable for computation, meaning that functional gadgets that rely on those registers will unusable. Therefore, to make available the greatest number of functional gadgets, we select the the dispatcher that uses the least common registers.

There are a number of heuristics available to locate different kinds of functional gadgets. In the case of conditional branch gadgets, the conditional branch operation can be separated into two steps: (1) update a general purpose register based on a comparison, and (2) use this result to permute *pc*. Because step 2 is a simple arithmetic operation, we instead focus on finding gadgets that implement step 1.

The result of a comparison are stored in CPU's comparator flags register (`EFLAGS` on the x86), and the most common way to leverage these flags is with a conditional jump instruction. For example, on the x86, the `je` instruction will "jump if equal", i.e. if the "zero flag" `ZF` is set. To find gadgets that leverage such instructions, the heuristic need only locate those gadgets whose first instruction is a conditional jump to another instruction later in the same gadget. Such a gadget will conditionally jump over some part of the gadget body, and can potentially be used to capture the result of a comparison in a general purpose register, where it can later be added to *pc*.

In addition to using conditional jumps, some CPUs, such as modern iterations of the x86, support the "conditional move" (`cmov`) and "set byte on condition" (`set`) instructions. We can search for a gadget that uses these instructions to conditionally alter a register.

Finally, there are also instructions that implicitly access the comparator flags, such as `adc` ("add with carry"). This instruction works like a normal `add`, except that the destination operand will be incremented one further if the "carry flag" is set. Because the carry flag represents the result of an unsigned integer comparison whenever the `cmp` instruction is used, instructions like `adc` behave like conditional move instructions, and can therefore be used to update general purpose registers with the comparison result.

The heuristics for finding arithmetic, logic, and memory access gadgets are much simpler, by comparison. We need only restrict the opcode to the desired operation (`add`, `mov`, `and`, etc.) and ensure that any destination operands do not conflict with the jump target.

## 3.4 Launching the Attack

The vulnerabilities that can lead to a jump-oriented attack are similar to those of return-oriented programming. The key difference, however, is that while ROP requires control over the instruction pointer `eip` and stack pointer `esp`, JOP requires `eip` plus whatever set of memory locations or registers are used to run the dispatcher gadget. In practice, this can be achieved by first directing control flow through a special *initializer gadget*. Specifically, the initializer gadget fills the relevant registers either by arithmetic and logic or by loading values from memory. Once this is done, the initializer jumps to the dispatcher, and the jump-oriented program can begin. The initializer gadget can take many forms, depending on the mix of registers that need to be filled. One simple case is a gadget that executes the `popa` instruction, which loads every general-purpose register from the stack. The initializer is not strictly necessary in all cases: if the attacker can take over control flow at a time when registers happen to be set at useful values, the dispatcher can be run directly from there.

The precise vulnerabilities that can lead to a return-oriented attack have been discussed in depth previously [36, 10, 15, 21, 32, 30, 14]. Due to space constraints, we omit the details here and merely summarize that the attacker can conceivably launch a jump-oriented attack by overwriting the stack, a function pointer, or a setjmp buffer. As the first two are already well-known, we explain the setjmp buffer below.

**A setjmp buffer** The C99 standard specifies the `setjmp()` and `longjmp()` functions as a means to achieve non-local gotos [25]. This functionality is often used for complex error handlers and in user mode threading libraries, such as certain versions of pthreads [26]. The programmer allocates a `jmp_buf` structure and calls `setjmp()` with a pointer to this structure at the point in the program where control flow

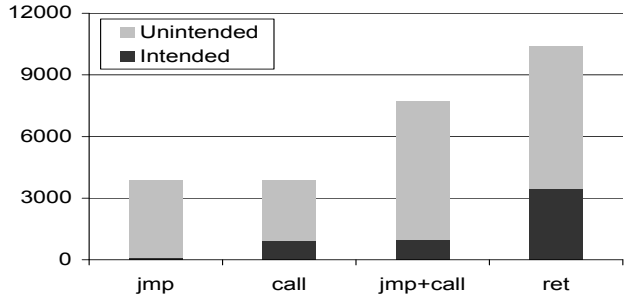



**Figure 4: The frequency of indirect `jmp` and `call` instructions, both intended and unintended, vs. `ret` instructions in libc.**

will eventually return. The `setjmp()` function will store the current CPU state in the `jmp_buf` object, including the instruction pointer `eip` and some (but not all) general-purpose registers. The function returns 0 at this time.

Later, the programmer can call `longjmp()` with the `jmp_buf` object in order to return control flow back to the point when `setjmp()` was originally called, bypassing all stack semantics. This function will restore the saved registers and jump to the saved value of `eip`. At this time, it will be as if `setjmp()` returns a second time, now with a non-zero return value. If the attacker can overwrite this buffer and a `longjmp()` is subsequently called, then control flow can be redirected to an initializer gadget to begin the jump-oriented program. Because of the straightforward nature of this technique, it is employed in our example attack (Section 4.4).

# 4. IMPLEMENTATION

To demonstrate the efficacy of the JOP technique, we developed a jump-oriented attack on a modern Linux system. Specifically, the attack is developed under Debian Linux 5.0.4 on the 32-bit x86 platform, with all gadgets being gleaned from the GNU libc library. Debian ships multiple versions of libc for different CPU and virtualization environments. Our target library was `/lib/i686/cmov/libc-2.7.so`,[6] the version for CPUs supporting the conditional move (`cmov`) instruction. In the following, we first examine the overall availability of gadgets within libc, and then cover the selection of the dispatcher and other functional gadgets. After that, we present a full jump-oriented example attack.

## 4.1 Availability of Gadgets

Jump-oriented programming requires gadgets that end in indirect branches instead of the `ret` instruction. These branches may be `jmp` instructions, or, because we are not concerned with using the stack for control flow, `call` instructions. Recall that the x86's variable instruction size allows for multiple interpretations of the same code, leading to a set of *intended* instructions generated by the compiler, plus an alternative set of *unintended* instructions found by reinterpreting the code from a different offset. To examine the relative availability of gadgets in JOP versus ROP, we show in Figure 4 the comparison between the number of `ret` instructions and the number of indirect `jmp` and `call` instructions.

If we were constrained to use only intended `jmp` and `call` gadgets, it is unlikely that there would be enough gadgets in libc alone to sustain a Turing-complete attack code, as there are only a few hundred such instructions present. However, when unintended instruction sequences are taken into account, a far greater selection of gadgets becomes available. This is due in large part to a specific aspect of the x86 instruction set: that the first opcode byte for an indirect jump is 0xff. Because the x86 uses two's complement signed integers, small negative values contain one or more 0xff bytes. Therefore, in addition to the 0xff bytes provided within opcodes, there is a large selection of 0xff bytes within immediate operands stored in the code stream. In fact, 0xff is the second most prevalent byte in the executable region of libc, with 0x00 being the first. This means that, probabilistically, indirect calls and jumps are far more prevalent than would otherwise be the case. Thanks to this, we have a large number of candidate jump gadgets to choose from.

To search for gadgets, we apply the algorithm given in Section 3.3. In doing so, we must select a value for $\delta_{max}$, the largest gadget size to consider, in bytes. A conservative value would be the average gadget length (5) multiplied by the average instruction's length (3.5), i.e. $\lceil 5 \cdot 3.5 \rceil = 18$. However, the only side-effect of making $\delta_{max}$ too large is including gadgets that may be of limited usefulness due to their length, so we err on the side of inclusiveness and set $\delta_{max} = 32$ bytes. Later, the gadget list may be sorted by number of instructions per gadget in order to focus on shorter and therefore more likely choices.

When the gadget search algorithm is applied to the executable regions of libc, 31,136 potential gadgets are found. The following two sections describe how these candidates are filtered by heuristics and manual analysis in order to locate the dispatcher and functional gadgets to mount our attack.

## 4.2 The Dispatcher Gadget

Using the heuristics described in Section 3.3, the complete set of potential gadgets was reduced to 35 candidates. Because there are so many choices, we can eliminate sequences longer than two instructions (the minimum length of any useful gadget) and still have 14 candidates to choose from. Through manual analysis, we find that 12 of these are viable. These choices use either arithmetic or dereferencing to advance *pc*, and rely on various registers to operate. Because the registers used by the dispatcher are unavailable for use by functional gadgets, choosing a dispatcher that uses the least common registers will make available the broadest range of functional gadgets. With this in mind, we selected the following dispatcher gadget in our example shellcode:

```
add   ebp, edi
jmp   [ebp-0x39]
```

This gadget uses the stack base pointer `ebp` as the jump target *pc*, adding to it the value stored in `edi`. We find that, as far as functional gadgets are concerned, neither of these registers play a prominent role in code generated by the compiler. Also, the constant offset -0x39 applied to the `jmp` instruction is of little consequence, as this can be statically compensated for when setting `ebp` to begin with. Because it is straightforward, predictable, and uses only two little-needed registers, we selected this dispatcher gadget to drive the shellcode example employed in Section 4.4.

## 4.3 Other Gadgets

Once the dispatcher is in place, one of the first functional gadgets needed is a means to load operands. In ROP, this is achieved by placing data on the stack, intermixed with return addresses that point to gadgets. This way, gadgets can use `pop` instructions to access data. There is no reason

[6]File size: 1413540 bytes,
MD5 checksum: e4e7e3c6b4f1be983e00c0daafc3aaf3.

why this approach cannot be applied in JOP, as anti-ROP defense techniques focus on abuses of the stack as a means for controlling the flow of execution, not data. In our implementation, part of the attack includes moving the stack pointer `esp` to part of the malicious buffer. Data can then be loaded directly from the buffer by `pop` instructions. This forms the basis for our *load data* gadget. A heuristic can be applied to locate such gadgets; the only requirements are that (a) the candidate's first instruction must be a `pop` to a general purpose register other than those used by our chosen dispatcher (`ebp` and `edi`), and (b) the indirect jump at the end must not use this register for its destination. This heuristic yields 60 possibilities within libc, so we filter the result further to only include gadgets with three instructions or fewer; this gives 22 possibilities. Manual analysis of this list yields 14 load data gadgets which can be used to load any of the general purpose registers not involved in the dispatcher. There is no need to filter further – because these gadgets have different side-effects and indirect jump targets, each of them may be useful at different times, depending on the registers in use for a calculation within the jump-oriented program.

If all registers need to be loaded at once, a gadget using the `popa` instruction can be executed. This instruction loads all general purpose registers from the stack at once. This forms the basis of the *initializer gadget*, which is used to prepare the CPU state when the attack begins.

Similar to the search for the load data gadgets, basic arithmetic and logic gadgets can be found with simple heuristics. Due to space constraints, suffice it to say there is a plentiful selection of gadgets implementing these operations. Restricting the length of a gadget to three instructions, we find 221 choices for the `add` gadget, 129 choices for `sub`, 112 for `or`, 1191 for `xor`, etc.

Achieving arbitrary access to memory is achieved by similar means. The most straightforward memory gadgets use the `mov` instruction to copy data between registers and memory. A heuristic to find memory write gadgets simply needs to find instructions of the form `mov [`*dest*`],` *src*, while the memory read gadget is of the form `mov` *dest*`, [`*src*`]`. As with most x86 instructions, the memory address in the `mov` may be offset by a constant, but this can be compensated for when designing the attack. Based on the above observations, a search of libc finds 150 possible load gadgets and 33 possible write gadgets based on `mov`. This does not include the large variety of x86 instructions that perform load and store operations implicitly, such as the string manipulation instructions `lod` and `sto`.

To locate conditional branch gadgets, we applied the heuristics described in Section 3.3. By far the most common means of moving the result of a comparison into a general purpose register is via the `adc` and `sbb` instructions, which work like `add` and `sub`, except incrementing/decrementing one further if the CPU "carry flag" is set. Because this flag represents the result of an unsigned integer comparison, gadgets featuring these instructions can be used to perform conditional branches. There are 1664 such gadgets found in libc, 333 of which consist of only two instructions. These gadgets can update any of the general purpose registers. To complete the conditional jump, we need only apply the plain arithmetic gadgets found previously to add some multiple of the updated register to *pc*.

To perform system calls, there are a number of different approaches the attacker can take. Of course, the attacker could arrange to call a regular library routine such as `system()`. However, because this would involve constructing an artificial stack frame, this approach runs the risk of being detected by existing anti-ROP defenses. Instead, the attacker can directly request a system call through the kernel's usual interface. On x86-based Linux, this can be done by executing a `sysenter` instruction to access the "fast system call" functionality.

To use this mechanism, the caller will (1) set `eax` to the system call number, (2) set the registers `ebx`, `ecx`, and `edx` to the call's parameters, and (3) execute the `sysenter` instruction. Ordinarily, the caller will also push `ecx`, `edx`, `ebp`, and the address of the next instruction onto the stack, but this bookkeeping is optional for the jump-oriented attacker. Instead, we can take advantage of the fact that the return address is specified on the stack by pointing it back to the dispatcher. This means that the `sysenter` gadget needs not end in an indirect jump. Note that this return address is *not* the same as a normal function return address – the kernel interface allows for this value to be set by the user. This is because all system calls have the same exit point in userspace: a small snippet of kernel-provided code which jumps back to the stored address.

Given this, the only challenge to making a system call is populating the correct registers. This becomes increasingly difficult as the number of parameters increases. For calls with three parameters such as `execve()`, it is necessary to simultaneously set `eax`, `ebx`, `ecx`, and `edx`. This is somewhat tricky, as there is no `popa` gadget that jumps based on a register other than the ones needed for the system call, and the selection of gadgets becomes limited as general purpose registers become occupied with specific values. Nevertheless, it is possible to make arbitrary system calls using only material from libc by chaining together multiple gadgets. For example, the following sequence of gadgets will load `eax`, `ebx`, `ecx`, and `edx` from attacker-supplied memory, then make a system call. This gadget sequence was used in constructing the shellcode for the example attack presented below.

```
popa                ; Load all registers
cmc                 ; No practical effect
jmp far dword [ecx] ; Back to dispatcher via ecx

xchg ecx, eax       ; Exchange ecx and eax
fdiv st, st(3)      ; No practical effect
jmp [esi-0xf]       ; Back to dispatcher via esi

mov eax, [esi+0xc]  ; Set eax
mov [esp], eax      ; No practical effect
call [esi+0x4]      ; Back to dispatcher via esi

sysenter            ; Perform system call
```

## 4.4 Example attack

Because of its simplicity, we use a vulnerable test program similar to the one given by Checkoway and Shacham [14]. The source code to this program is given in Figure 5. In essence, this program copies the first command line argument `argv[1]` into a 256 byte buffer on the heap. Because the program does not limit the amount of data copied, this program is vulnerable to the `setjmp` exploit described in Section 3.4. The attacker can overflow the buffer and, when the `longjmp` function is called on line 17, take control of the registers `ebx`, `esi`, `edi`, `ebp`, `esp`, and the instruction pointer `eip`. This specific application is merely an example: any exploit which delivers control of the instruction pointer and other registers can potentially be used to start a jump-

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <setjmp.h>
 5
 6 struct foo {
 7     char buffer[256];
 8     jmp_buf jb;
 9 };
10
11 int main(int argc, char** argv, char** envp) {
12     struct foo *f = malloc(sizeof(*f));
13     if (setjmp(f->jb)) {
14         return 0;
15     }
16     strcpy(f->buffer, argv[1]);
17     longjmp(f->jb, 1);
18 }
```

**Figure 5: The example vulnerable program**

oriented attack.

We use this program as a platform to launch a jump-oriented shellcode program which will ultimately use the `execve` system call to launch an interactive shell. Specifically, our example attack was constructed in NASM [1], which, despite being an assembler, was only used to specify raw data fields. The macros and arithmetic features of NASM allow the expression of the exploit code in a straightforward way. The attack source code is given in our technical report [9].

When assembled by NASM, this script will produce a binary exploit file, which is then provided to the vulnerable program as a command line argument:

```
$ ./vulnerable "`cat exploit.bin`"
```

This launches the jump-oriented program and ultimately yields an interactive shell prompt without a single `ret` instruction.

# 5. DISCUSSION

In this section, we examine possible limitations and discuss further refinements in the jump-oriented programming technique. First, while we have found that JOP is capable of arbitrary computation in theory, constructing the attack code manually is a complex task, moreso even than in ROP. The main reason is an added layer of interdependency in JOP gadgets. Specifically, because of the reliance on certain registers to serve as the "state" for the jump-oriented system (e.g., the pointer to the dispatch table and the callback to the dispatcher after each gadget execution), there are complex restrictions on the sequence of gadgets that can be assembled. Oftentimes, the attack designer will need to introduce gadgets whose sole purpose is to make the next gadget work (e.g., by setting a jump target register). This naturally complicates the development of automated techniques to facilitate the jump-oriented programming.

Second, while the idea of jump-oriented programming is applicable in theory to architectures with fixed-length instructions (SPARC, ARM, etc.), it may be the case that a much larger codebase is required to realize full Turing-complete operation. This is because two features of the x86 conspire to make gadgets based on `jmp` and `call` especially plentiful: (1) variable length instructions allow multiple interpretations of the code stream, and (2) indirect branch instructions begin with the especially common 0xff byte. A thorough analysis of the feasibility and efficiency of applying JOP to alternative platforms (e.g., MIPS), including the portability of dispatcher gadgets and the associated dispatch table (Section 3), is an important question which we leave to future work.

Third, if we examine the nature of the two different programming models, i.e., ROP and JOP, the basis of the vulnerability is not the returns or the indirect jumps, but rather the promiscuous behavior of allowing entry to any address in an executable program or library. To defend against them, there is a need to enforce control-flow integrity. From another perspective, it may be tempting to assume that this attack might be trivially defeated by identifying anomalies, such as dispatcher-like behavior or high-frequency indirect jumps. Unfortunately, this is not the case. Such defenses can be easily evaded by arranging for the execution of long-running functions and changing dispatchers periodically. Next, we examine related work and discuss a number of orthogonal defenses which could be used to impede or prevent either return- or jump-oriented programming.

# 6. RELATED WORK

**Anti-ROP defenses** Recently, a number of defense systems have been proposed to detect or prevent return-oriented programming attacks. For example, based on a separate shadow stack (similar to [17, 22]), ROPdefender proposes a binary rewriting approach to ensure the validity of each return target, thus blocking the execution of return-oriented gadgets [19]. DROP [16] and DynIMA [18] detect a ROP-based attack by monitoring the execution of short instruction sequences each ending with a `ret`. The return-less approach [31] recognizes the need of `ret` for the gadget construction and chaining and develops a compiler-based approach to remove the presence of the `ret` opcode. In contrast, jump-oriented programming is made immune from these defenses by avoiding the reliance on `ret` or the return stack to launch a JOP-based attack. In this respect, JOP reflects the trend of the ongoing security arms-race. Most recently, Onarlioglu et. al introduced G-Free, a compiler designed to produce gadget-less binaries [34]. This is achieved by removing all *unintended* control flow transfers, then protecting the intended ones through pointer encyption and stack cookies. Because indirect jump and call instructions are protected in this scheme, it would prevent them from being misused and essentially de-generalize existing ROP attacks to the traditional return-into-libc attacks.

Independent of our work, a concurrent approach by Checkoway *et al.* [14] proposes to replace `ret` in ROP with a `pop+jmp` on x86, which arguably is a step further from the original ROP model. However, `pop+jmp` sequences are rare, necessitating the "bring your own `pop+jmp`" paradigm, where the sequence must be found in a particularly large code base. In fact, our analysis of the text section of the default libc in Debian Linux 5.0.4 on the 32-bit x86 platform does not yield a single `pop+jmp` sequence. Also, the use of such sequence still imposes the need of relying on the stack to govern control flow among gadgets. In comparison, our JOP model has no such restrictions, and therefore threatens a much broader set of applications and environments. An ARM implementation was also presented in this work which relied on an "update-load-branch" gadget to maintain control flow. This similarity lends evidence to the theory that jump-oriented attacks could be a cross-platform threat, not limited to x86.

From another perspective, other orthogonal defense schemes (e.g., randomization) have been proposed to defend against code injection attacks. In particular, address-space layout randomization (ASLR) [2, 7, 8] randomizes the memory layout of a running program, making it difficult to determine the addresses in libc and other legitimate code on which

return-into-libc or ROP/JOP-based attacks rely. However, there are de-randomization attacks to bypass ASLR [20, 33, 37] or limit its effectiveness. Instruction-set randomization (ISR) [6, 27] instead randomizes the instruction set for each running process so that instructions in the injected attack code fail to execute correctly even though the attacks may have successfully hijacked the control flow. However, it is not effective to return-into-libc and ROP/JOP-based attacks.

**Memory safety** In the past, many defense mechanisms have also been proposed to better enforce enhanced memory safety. For example, CFI [4] and program shepherding [28] are designed to protect the control-flow integrity property of a running program. DFI [12] and others [5, 13] build on the control-flow integrity property and further extend it for other types of memory safety (e.g., data-flow integrity). Note that if control-flow integrity is strictly enforced, both ROP and JOP will be blocked from hijacking the control flow in the first place. However, precise CFI enforcement requires complex code analysis, which can be difficult to obtain especially for programs with a large codebase, including libc or modern OS kernels. Further, CFI has not seen wide deployment, likely due to concerns over performance, especially in the case of real-time enforcement.

**Other code re-uses** Most recently, researchers found interesting applications of re-using certain code snippets from malicious code to better understand them. For example, Caballero *et al.* proposed BCR [11], a tool that aims to extract a function from a (malware) binary so that it can be re-used later. Kolbitsch *et al.* developed Inspector [29] to re-use existing code in a binary and transform it into a stand-alone gadget that can be later used to (re)execute specific malware functionality. In comparison, ROP and JOP re-use legitimate code of a vulnerable program to construct arbitrary computation without injecting code.

# 7. CONCLUSION

In this paper, we have presented a new class of code-reuse attack, *jump-oriented programming*. This attack eliminates the reliance on the stack and `ret`s from return-oriented programming but without scarifying its expressive power. In particular, under this attack, we can build and chain normal *functional gadgets* with each performing certain primitive operations. However, due to the lack of `ret` to chain them, this attack relies on a *dispatcher gadget* to dispatch and execute next functional gadget. We have successfully developed an example shellcode attack based on jump-oriented programming, and the abundance of `jmp` gadgets in GNU libc indicates the practicality and effectiveness of this attack.

# 8. REFERENCES


[1] NASM. http://www.nasm.us/.

[2] PaX ASLR Documentation. http://pax.grsecurity.net/docs/aslr.txt.

[3] W^X. http://en.wikipedia.org/wiki/W^X.

[4] M. Abadi, M. Budiu, Úlfar Erilingsson, and J. Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *12th ACM CCS*, October 2005.

[5] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *28th IEEE Symposium on Security and Privacy*, May 2008.

[6] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *10th ACM CCS*, 2003.

[7] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. *12th USENIX Security*, 2003.

[8] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *14th USENIX Security*, 2005.

[9] T. Bletsch, X. Jiang, and V. Freeh. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *CSC Technical Report TR-2010-8, NC State University*, 2010.

[10] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *15th ACM CCS*, pages 27–38, New York, NY, USA, 2008. ACM.

[11] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *17th ISOC NDSS*, 2010.

[12] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *7th USENIX OSDI*, 2006.

[13] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black. Fast Byte-Granularity Software Fault Isolation. In *22nd ACM SOSP*, October 2009.

[14] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th ACM CCS*, 2010.

[15] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, , and H. Shacham. Can DREs provide long-lasting security? The case of return-oriented programming and the AVC Advantage. In *EVT/WOTE 2009, USENIX*, 2009.

[16] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In *5th ACM ICISS*, 2009.

[17] T. Chiueh and F. Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *21st IEEE ICDCS*, April 2001.

[18] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In *2009 ACM STC*.

[19] L. Davi, A.-R. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. Technical Report HGI-TR-2010-001, Horst Görtz Institute for IT Security, March 2010.

[20] T. Durden. Bypassing PaX ASLR Protection. *Phrack Magazine, Volume 11, Issue 0x59, File 9 of 18*, 2002.

[21] A. Francillon and C. Castelluccia. Code Injection Attacks on Harvard-Architecture Devices. In *15th ACM CCS*, 2008.

[22] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *10th USENIX Security Symposium*, 2001.

[23] R. Hund, T. Holz, and F. C. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In *19th USENIX Security Symposium*, Aug. 2009.

[24] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 2. Mar. 2010.

[25] ISO. The ansi c standard (c99). Technical Report WG14 N1124, ISO/IEC, 1999.

[26] R. Johnson. Open source posix threads for win32 faq. http://sourceware.org/pthreads-win32/faq.html.

[27] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *10th ACM CCS*, 2003.

[28] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*, August 2002.

[29] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries. In *30th IEEE Symposium on Security and Privacy*, May 2010.

[30] T. Kornau. *Return oriented programming for the ARM architecture*. Master's thesis, Ruhr-Universität Bochum, January 2010.

[31] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. In *5th ACM SIGOPS EuroSys Conference*, Apr. 2010.

[32] F. F. Lidner. Developments in Cisco IOS Forensics. In *CONference 2.0*, Nov. 2009.

[33] Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine, Volume 11, Issue 0x58*, 2001.

[34] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *26th ACSAC*, 2010.

[35] PaX Team. What the future holds for PaX. http://pax.grsecurity.net/docs/pax-future.txt, 2003.

[36] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *14th ACM CCS*, 2007.

[37] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address Space Randomization. *11th ACM CCS*, 2004.