# Return-Oriented Programming without Returns

Stephen Checkoway[†], Lucas Davi[‡], Alexandra Dmitrienko[‡], Ahmad-Reza Sadeghi[‡], Hovav Shacham[†], Marcel Winandy[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

[‡]System Security Lab
Ruhr-Universität Bochum
Bochum, Germany

## ABSTRACT

We show that on both the x86 and ARM architectures it is possible to mount return-oriented programming attacks without using return instructions. Our attacks instead make use of certain instruction sequences that behave like a return, which occur with sufficient frequency in large libraries on (x86) Linux and (ARM) Android to allow creation of Turing-complete gadget sets.

Because they do not make use of return instructions, our new attacks have negative implications for several recently proposed classes of defense against return-oriented programming: those that detect the too-frequent use of returns in the instruction stream; those that detect violations of the last-in, first-out invariant normally maintained for the return-address stack; and those that modify compilers to produce code that avoids the return instruction.

## Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

## General Terms

Security, Algorithms

## 1. INTRODUCTION

This paper is about the feasibility of certain defenses against return-oriented programming. In the last year, several natural defenses have been proposed that target properties of return-oriented attacks and are intended to be simpler and have lower overhead than a comprehensive defense such as Control-Flow Integrity (CFI) [1, 14].[1] In this paper, we show that these narrowly tailored defenses are incomplete by devising a new variant of return-oriented programming that evades them. Our results call into doubt the usefulness of these ad-hoc defenses.

### *Return-Oriented Programming.*

Return-oriented programming allows an attacker to exploit memory errors in a program without injecting new code into the program's address space. In a return-oriented attack, the attacker arranges for short sequences of instructions in the target program to be executed, one sequence after another. Through a choice of these sequences and their arrangement, the attacker can induce arbitrary (Turing-complete) behavior in the target program. Traditionally, the instruction sequences are chosen so that each ends in a "return" instruction, which, if the attacker has control of the stack, allows control to flow from one sequence to the next — and gives return-oriented programming its name.

The organizational unit of return-oriented programming is the *gadget*, an arrangement of instruction sequence addresses and data that, when run, induces some well-defined behavior, such as computing an exclusive-or or performing a conditional branch. Return-oriented exploits begin by devising a Turing-complete set of gadgets, from which any desired attack functionality is then synthesized.[2]

Return-oriented programming was introduced by Shacham in 2007 [41] for the x86 architecture. It was subsequently extended to the SPARC [3], Atmel AVR [15], PowerPC [26], Z80 [4], and ARM [23] processors. While the original return-oriented attack was largely manual, later work showed that each stage of the attack can be automated [3, 38, 20, 23]. Return-oriented programming has proved useful for compromising Harvard-architecture platforms, such as Sequoia's AVC Advantage voting machine [4] and Apple's iPhone [22, 30], on which traditional code injection is not a possibility.

### *Defenses Against Return-Oriented Programming.*

The instruction stream executed during a return-oriented attack as described above is different from the instruction stream executed by legitimate programs in at least two ways: first, it contains many return instructions, just a few instructions apart; second, it unwinds the stack with return instructions for which there were no corresponding "call" instructions. These differences have been seized

[1]Note that even the closest to a generally available CFI instantiation for ARM, Google's NaCl [40], would require substantial additional development to support full CFI for general-purpose ARM code.

[2]The crucial feature of return-oriented programming is *Turing completeness without code injection*. A great deal of work prior to 2007 showed how to leverage control of the stack to invoke and chain libc functions [29, 31] and short instruction sequences such as pops followed by returns [33, 24] and even to produce unconditional loops [36, 37]. On most platforms, one can use these techniques to mark some memory region both writable and executable, then inject and execute arbitrary native machine code from that memory region as a second stage; the machine code is Turing complete, of course, so the first stage need not be Turing complete. (McDonald proposed essentially this approach in 1999 to bypass Solaris's nonexecutable stack [29].) Exploits of this sort are *not* a contribution of this paper, nor of Shacham's 2007 paper [41]. Indeed, setting aside Turing completeness, the observation that code reuse attacks might be feasible using chaining instructions other than "return" was made by the PaX team in 2003 [34].


*CCS'10,* October 4–8, 2010, Chicago, Illinois, USA.

upon by researchers as the basis for mechanisms to detect and defeat return-oriented attacks:

- The first difference suggests a defense that looks for instruction streams with frequent returns. Davi, Sadeghi, and Winandy [12] and Chen et al. [6] both use dynamic binary instrumentation frameworks (Pin [28] and Valgrind [32], respectively) to instrument program code. With both systems, three consecutive sequences of five or fewer instructions ending in a return trigger an alarm.
- The second difference suggests a defense that looks for violations of the last-in, first-out stack invariant usually maintained in benign programs by the call and return instructions. These solutions can be categorized in *compiler*-based solutions like Stack Shield [45] and RAD [7]; *instrumentation*-based solutions securing function prologues and epilogues [35, 18]; those using just-in-time compilation, e.g., TRUSS [42] and ROPdefender [13]; and, finally, *hardware-facilitated* solutions [16, 17]. All these proposals guarantee the integrity of return addresses, which is violated in conventional return-oriented programming attacks.
- More generally, if a body of code doesn't contain return instructions, then traditional return-oriented programming is impossible. Li et al. [25] propose a compiler for the x86 that avoids issuing `0xc3` bytes that can be used as unintended return instructions and that replaces intended call and return instructions with an indirect call mechanism that pushes a "return index" onto the stack instead of a return address to avoid the return in the function epilogue.

While several of these defenses build on binary instrumentation platforms and inherit the performance degradation that binary instrumentation entails, the properties they verify are amenable to hardware implementation at greatly reduced overhead. What we show in this paper is that these defenses would not be worthwhile even if implemented in hardware. Resources would instead be better spent deploying a comprehensive solution, such as CFI [1, 14].

### *Our Contribution.*

We show that, on both the x86 and the ARM, it is possible to perform return-oriented programming *without using return instructions*. We show that instruction sequences exist that behave like returns, and that these can be used instead of returns to chain useful sequences together to yield Turing-complete functionality.

Return instructions have two properties that make them useful for return-oriented programming: (1) they transfer control of execution by means of an indirect jump; and (2) they update some processor state, so a subsequent return will not transfer control to the same location. In the case of actual return instructions, the location to which control is transferred is the address at the top of the stack, and the updated shared state is the stack pointer. On both the x86 and the ARM, instruction sequences exist that have these same two properties, but that do not include a return instruction. These sequences update a piece of global state (e.g., the stack pointer), load the address of the next instruction sequence to execute based on this updated state, and branch to the address loaded.

Unlike return instructions, which are plentiful, our update-load-branch instruction sequences occur too infrequently for us to expect to obtain a gadget set where each instruction sequence used ends in the two or three instructions that make up the update-load-branch operation. To overcome this, we reuse a single such update-load-branch instruction sequence as a *trampoline*. Each instruction sequence we use in composing our gadget set ends in an indirect jump to the trampoline, which redirects the execution to the next sequence of instructions. We discuss these techniques in Section 2.

In Section 3 we describe a Turing-complete gadget set for the x86 that we have created based on the libc and certain large libraries distributed with Debian GNU/Linux 5.0.4 ("Lenny"). In Section 4 we similarly describe a Turing-complete gadget set on Google's Android 2.0 ("Eclair"). The x86 is the dominant architecture for desktop and server computing; within the last few years ARM has achieved similar dominance in mobile computing. Thus our findings have implications for both major architectures in use today.

In Section 5, we consider how an attacker undertakes what Dino Dai Zovi calls the *stack pivot* [9] — taking control of the stack pointer to execute return-oriented code — without using return instructions. And, for completeness, we show in Section 6 complete return-oriented exploits without return instructions against sample target programs for both architectures.

### *Negative Implications for Defenses.*

Our attack has negative implications for defenses against return-oriented programming that look for return instructions in order to recognize a return-oriented instruction stream. Defenses of the first kind considered above, which detect the use of several return instructions in close succession, will not detect attacks structured like the ones we introduce in this paper since these attacks make use of either one return or none at all. When it is possible to initiate an attack without a return, the LIFO invariant of the return-address stack is not violated, so defenses of the second sort will also not detect the attacks. Defenses of the third kind, which rewrite binaries to eliminate return instructions, are likewise irrelevant, since return instructions, whether intended or unintended, are never used.

Because our attack does not violate the LIFO invariant of the return-address stack, it is not clear that defenses that maintain a shadow return-address stack can be salvaged. Maintaining a shadow copy of jump targets would not be useful, because no simple invariant governs these targets in benign programs.[3]

On the other hand, it may be possible to patch defenses of the first kind to look not just for several returns in quick succession but also for several indirect jumps in quick succession. This would detect attacks structured as ours are. Doing so without being able, provably, to detect that every kind of return-like instruction sequence that a return-oriented program might use risks engaging in a classic cat-and-mouse game in which attackers switch to new return-like sequences to evade the upgraded defenses. Prior to our results in this paper, it appeared that return-oriented programming unavoidably relied on return instructions, making these instructions attractive targets for detection and defense. Now, however, it appears that a different property must be found by which to detect return-oriented attacks. Instead of such a cat-and-mouse game, it would be better to deploy a comprehensive defense such as CFI.

### *Full Versions.*

This paper represents the merging of two papers available separately as technical reports [5, 11]. Some details are omitted in this version due to space constraints.

## 2. ROP WITHOUT RETURNS

In this section we describe how return-like instruction sequences can substitute for rets, allowing return-oriented programming without use of return instructions.

[3]We further observe that shadow return-address stacks are difficult to keep synchronized in the presence of longjmp calls, thunks, and other unusual forms of control transfer; a defense that relies on the correctness of the shadow return-address stack may be brittle.

*Assumptions and Adversary Model.*

We define a strong adversary model. For our attack we assume the availability of standard protection mechanisms against code injection and return address corruption attacks.

1. We assume that the target platform may enforce the $W \oplus X$ security model.
2. We assume that the target platform may use countermeasures to defend against/detect conventional return-oriented programming attacks, as described in Section 1.
3. We assume that the target platform provides an application with some bug allowing us to divert a target program's control flow without using any return (i.e., function epilogue) instruction. We want to avoid the use of any return instruction, so that our attack circumvents return address checkers.

*The High-Level Idea.*

A ret instruction has the following effects: (1) it retrieves the four-byte value at the top of the stack, and sets the instruction pointer (eip) to that value, so that the instructions beginning at that address execute; and (2) it increases the value of the stack pointer (esp) by four, so that the top of the stack is now the word above the word assigned to eip. In return-oriented programming, the location of each instruction sequence is written to the stack; when a sequence has executed, reaching the ret that ends it, that ret causes the next instruction sequence to be executed.

One way to view this arrangement of the stack, suggested by Roemer et al. [39], is that in return-oriented programming the stack pointer takes the place of the instruction pointer in ordinary programming; that each gadget on the stack is an instruction for a custom-built virtual machine; and that the ret at the end of each instruction sequence acts like a typewriter carriage return to advance the processor to the next instruction — something the processor does automatically for ordinary programs.

Our insight in this paper is that many other instruction sequences have ret-like properties, and that such instruction sequences make possible return-oriented programming without returns. Our replacement for the ret instruction is an *update-load-branch* sequence, so named because it first updates the global state that acts as the return-oriented program's instruction pointer, then uses the updated state to load from memory the address of the next instruction sequence to execute, and finally branches to the loaded address.

On the x86, we recommend the use of return-like instruction sequences of the form "pop $x$; jmp $*x$", where $x$ is any general-purpose register. (One can also use other kinds of sequences, including ones where the register updated is not the stack pointer; see Section 3.1.) On the ARM, we recommend the use of "update-load-branch" return-like instruction sequences such as "adds r6,#4; ldr r5, [r6,#124]; blx r5"; where a general-purpose register (in this example, r6) is the updated state.

*Reusing an Update-Load-Branch Sequence.*

It turns out that update-load-branch instruction sequences are more rare than the ret instructions they take the place of. (Return instructions are just one instruction, not two or more, and are used in almost every function.) Instead of trying to build a Turing-complete gadget set where every instruction sequence ends in update-load-branch, we look for a single update-load-branch sequence in the target program, then reuse this sequence as a *trampoline*. To reuse the trampoline, we select instruction sequences ending in an indirect jump instruction whose target is the trampoline.[4] The trampoline updates the program's global state and transfers control to the next instruction sequence. There are sufficiently many instruction sequences ending in indirect jumps that a Turing-complete gadget set can be constructed this way.

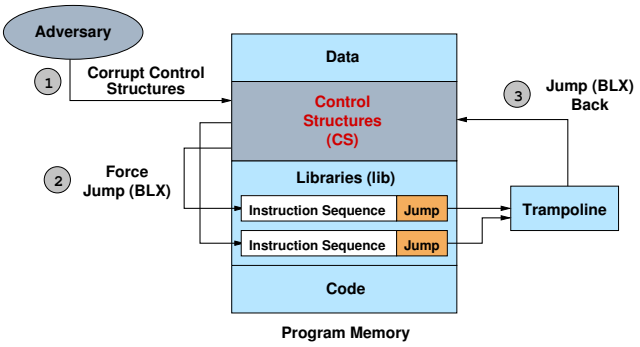


**Figure 1: Return-oriented programming without returns**

If, on the x86, we reserve a register $y$ to store the trampoline's address then sequences ending in an indirect jump through $y$ will behave as though they themselves ended in update-load-branch instructions, allowing the sequences to be chained together in the return-oriented style. The principle is the same for ARM, where we use blx (Branch-Link-Exchange) as our indirect jump.

The principle of a jump-based attack is depicted in Figure 1. It shows an abstract view of a program's memory. The adversary cannot inject own malicious code due to enabled $W \oplus X$ protection (see Assumption 1). However, an adversary is still able to use existing code of the target program and its libraries. Therefore, the adversary corrupts the control structure (CS) section so that program execution transfers to a specific piece of code in a linked library (lib). Usually control structures (such as return and jump addresses) are located on the stack or on the heap. The instruction sequence of the linked library is executed until an indirect jump instruction has been reached which redirects the execution to the next sequence of instructions by using our trampoline. The trampoline is also part of the linked libraries and is responsible for redirecting execution to the next instruction sequence.

# 3. INSTANTIATION ON INTEL X86

## 3.1 Update-Load-Branch on the x86

Consider the sequence "pop %eax; jmp $*$%eax", which updates the stack pointer as the global program state, and has the side effect of overwriting the eax register. This is a usable update-load-branch, and one example of a large class of such sequences.

More generally, we can, first, substitute another general-purpose register (esp excepted, for obvious reasons) for eax. Second, we can use doubly indirect jumps instead of singly indirect jumps, by storing at the stack word popped into eax not the address of the next sequence to run but the address of another location in memory that holds that sequence address. A return-oriented exploit that uses such doubly indirect jumps can be organized to include a *sequence catalog* of useful instruction sequence addresses, something like the Global Offset Table used in dynamic linking. Third, we can use doubly indirect jumps with an immediate offset (either 8-bit or 32-bit), by adding to or subtracting from the sequence address to cancel out the effect of the immediate offset. Fourth, we can observe that the x86 provides two kinds of doubly indirect jumps:

[4] Shacham observed [41, Section 5.1] that if ebx contains the address of a ret instruction then any instruction sequence ending in jmp $*$%ebx behaves just as if it had ended in ret; here we are replacing ret by an update-load-branch sequence. For the use of related techniques in the context of code injection, see [10, 27, 8].

near jumps, which take a 32-bit address in the current segment; and far jumps, which take a 32-bit address together with a 16-bit segment selector. Far jumps allow for sophisticated privilege domain regimes with restricted cross-domain calls (as, e.g., in the x86 NaCl sandbox [48]). We, however, need only the following fact: An appropriate choice of segment selector (on our Debian system, `0x0073`) leaves the code segment unchanged; a far jump to an address with this segment selector behaves exactly like a near jump to the same address.[5] Because the segment selector follows the address in memory, we can follow each address in the sequence catalog with the appropriate segment selector and thereafter use far and near doubly indirect jumps interchangeably. (This introduces zero bytes into the catalog; if this is a problem for a particular exploit, the zero bytes can be patched in at runtime; see Section 6.1.)

We use sequences of the sort described above in constructing our gadgets. We refer to all of them using the shorthand pop *x*; jmp ∗*x*, where *x* is any general purpose register. Pop-jump sequences occur with some frequency as unintended instructions because of accidental features of the x86 ISA; see our tech report [5].

There are yet more possibilities for update-load-branch instruction sequences on the x86. One could imagine a sequence based on call ∗*x*, which would *decrease* esp each time it is used. Or a different register than esp could be used, as, e.g., in add `0x4`, %eax; jmp ∗(%eax); this is similar to the update-load-branch sequence we use in Section 4 for our ARM instantiation. Or, using SIB addressing, a combination of registers could be used, with the index register scaled by 4 and incremented after each dereference. Or a memory location could serve as the mutable state instead of a register. The point here is that many possible types of instruction sequence exhibit the necessary behavior and are potentially suitable for return-oriented programming. A defense that detects some but not all of these types of instruction sequences would be of limited value, as attackers may be able to switch to a different return-like sequence and thereby evade detection.

## 3.2 Gadget Set

To demonstrate that Turing-complete return-oriented computation without returns is feasible in real programs, we design a set of *gadgets* each of which performs a discrete computation and can be reasoned about independently by virtue of little or no state maintained between gadgets. We build these gadgets by examining the C standard library found in Debian GNU/Linux 5.0.4 ("Lenny"), GNU libc 2.7, which is 1294572 bytes.[6] As we will see below, by itself, Debian's libc is almost sufficient. We need a single instruction sequence to exist in the either target program or in a library loaded by the target program. We find this additional instruction sequence in two large libraries: Mozilla's libxul (11857460 bytes), distributed with Firefox and Thunderbird; and the PHP language's libphp5 (5450680 bytes). These libraries are used in Web browsers and Web servers, respectively — common targets for exploitation.

(Could compilers be modified to avoid emitting pop-jump sequences? We note, first, that these instructions need be intended instructions placed in the binary by the compiler; second, that we do not require the pop immediately to precede the jump, making the compiler's job harder; and, third, that other instruction sequences than pop-jump could be used. Modifying compilers is a complicated project. We believe that the effort would be better spent deploying a comprehensive solution.)

As described in Section 2, rather than using sequences of instructions that end in pop *x*; jmp ∗*x*, we use sequences of instructions that end in jmp ∗*y* where *y* is a pointer to a pop *x*; jmp ∗*x* sequence. It is exactly this pop *x*; jmp ∗*x* that we do not find in libc[7] and so must exist in the target program or one of its libraries. We call this (facetiously) the *bring your own pop-jump* (BYOPJ) paradigm.

Because libc is loaded into every Linux executable, we gain confidence by using it as the corpus for our instruction sequences (except the pop-jump) that return-oriented programming without returns is likely possible in any large Linux program that an attacker might target. We stress that using most instruction sequences from libc but a pop-jump from libxul is not how a real attacker would go about mounting an attack. Libxul is larger and has more convenient instruction sequences than libc does; a Turing-complete gadget set could be constructed more easily from libxul alone than from libc with a libxul pop-jump. However, any program that did not link against libxul would require an entirely different gadget set. Unlike creating a new gadget set, testing that a program contains a suitable pop-jump is simple and easily automated.

Most of the useful instruction sequences end with either a near (resp. far) indirect jump to the address stored in the near (resp. far) pointer in memory at an address stored in register edx. That is, many instruction sequences end with jmp ∗(%edx) or ljmp ∗(%edx).

Each gadget could be made fully independent from the others, but since register edx is so useful for chaining instruction sequences, we ensure that at the end of each gadget, it holds the address of the sequence catalog entry for the pop *x*; jmp ∗*x*. In most cases, this required no additional work. The function call gadget is the only one which required the fix up.

Following Checkoway et al. [4], we design a *three-address code* memory-memory gadget set: our gadgets are of the form $x \leftarrow y \,\mathrm{op}\, z$, where $x$, $y$, and $z$ are literal locations in memory that hold the operands and destination. We use register edx to chain our instruction sequences; for the pop *x*; jmp ∗*x* sequence in our BYOPJ paradigm, we use register ebx. This means that we cannot store any state in register ebx, but we need not worry about changing its contents during the course of an instruction sequence since it will be overwritten during the pop %ebx. This leaves us with five registers, eax, ecx, ebp, esi, and edi, to do with as we please.

### *Instruction Sequences.*

We used 34 distinct instruction sequences ending with jmp ∗*x* to construct 19 general purpose gadgets: load immediate, move, load, store, add, add immediate, subtract, negate, and, and immediate, or, or immediate, xor, xor immediate, complement, branch unconditional, branch conditional, set less than, and function call. The

[5] A 16-bit segment selector consists of a 13-bit index, a 1-bit table indicator, and a 2-bit requested privilege level. The index specifies a 64-bit segment descriptor in either the global descriptor table or the local descriptor table as specified by the table indicator. Each segment descriptor contains a number of bit-fields including the segment base address, segment limit and privilege level. Since Linux uses a flat address space, most of the segment descriptors used in user programs specify a base address of zero and a limit of 4 GB [21]. The selector `0x0073` corresponds to an index of 14 in the global descriptor table with a requested privilege level of ring 3.

[6] There are actually two distinct libcs on our test system: `/lib/libc-2.7.so` and `/lib/i686/cmov/libc-2.7.so`. The gadgets described in this section and the example exploit in Section 6.1 are constructed from the former. However, the latter library is loaded at runtime instead on some machines, apparently those that support the conditional-move instructions cmov*cc* (introduced with the Intel Pentium Pro). We have verified that this libc also provides instruction sequences sufficient for constructing a Turing-complete gadget set without returns, which gives additional evidence for our thesis in this paper.

[7] In the second libc described in footnote 6, there is a single pop %edx; jmp ∗(%edx) sequence but as we show below, edx is too useful to use for this purpose. Other minor differences exist between the two libraries but we do not dwell on them further.

majority of the instruction sequences contain four or fewer instructions. The sequences were chosen by hand out of a collection of potential instruction sequences in libc discovered by the algorithm given by Shacham [41].

Loading data from the stack into a register can be accomplished by means of a pop *x*; jmp ∗*y* instruction sequence:

| | | |
|---|---|---|
| pop %eax; | sub %dh, %bl; | jmp ∗(%edx) |
| pop %ecx; | cmp %dh, %dh; | jmp ∗(%edx) |
| pop %ebp; | or $0xF3, %al; | jmp ∗(%edx) |
| pop %esi; | or $0xF3, %al; | jmp ∗(%edx) |
| pop %edi; | cmp %bl, %dl; | jmp ∗(%edx) |
| pop %esp; | or %edi, %esi; | jmp ∗(%eax) |
| popad; | cld; | ljmp ∗(%edx) |

The first five can be used to load any of the registers we wish to use as long as we load register eax after registers ebp and esi. The sixth allows for a simple jump by changing the stack pointer, see below. Instruction popad pops all seven general purpose registers off of the stack (it does not pop register esp, but it require 4 bytes that are ignored for a total of 32 bytes popped off of the stack). Without a pop %edx; jmp ∗*x* instruction in the target binary or its libraries, popad is the only way to load register edx. This is only an issue for our function call gadget described below.

The gadgets need to be able to move data between memory and registers as well as between multiple registers. Moving a word from memory into a register is accomplished by means of a mov *n*(*x*), *y* instruction where *n* is some immediate offset. The analogous instruction mov *x*, *n*(*y*) allows for the reverse operation. Movement between registers is less straight-forward because while such an x86 instruction exists, we find none in sequences ending in jmp ∗*x*. Instead, the contents of two registers can be exchanged with the xchg instruction, or by arranging for the destination register to be `0x00000000` or `0xffffffff`, the source register can be ored or anded with the destination, effecting the move.

One difficulty we will frequently encounter is the need to use a register for holding data in one instruction sequence and for being the *x* in the jmp ∗*x* in another sequence within a single gadget. Handling this requires careful structuring of the gadget to ensure that the register has been loaded with the address of the pointer to the pop *x*; jmp ∗*x* sequence before it is needed.

By now, the gadget-construction procedure is well-described in the literature [41, 3, 15, 20, 19, 26]. As such, we only briefly describe each of our standard gadgets and focus more on the gadgets that require extra finesse.

### *Data Movement.*

The first thing we wish to do is to *load immediate* values into memory at a fixed address. This is easily accomplished by loading esi with the immediate value and eax with the fixed address plus `0xb`. This takes two pops. Then we use mov %esi, `-0xb`(%eax) to write the immediate value to memory.

Since we want a collection of memory-memory gadgets, we need to load a word from one (constant) location in memory and store it into another (constant) location in memory. This is accomplished by loading the source address into eax, loading the destination address into ebp, loading from eax into edi, and finally storing edi into memory at the address in ebp. This is the *move* gadget.

A simple modification to the move gadget yields the *load* gadget. Rather than storing the word in memory at the source address into the destination address, that word is used as a pointer to another word in memory which is loaded into another register and then stored at the destination address. In pseudo code, the operation is the following.

```
eax ← source
edi ← (eax)
esi ← (edi)
eax ← destination
(eax) ← esi
```

A *store* gadget is similar except that the address where the source value is to be stored is itself stored at a fixed location. That is, the store gadget performs the operation $(A) \leftarrow B$ where $A$ is the word in memory at the destination address and $B$ is the word in memory at the source address. In fact, we can perform the operation $(A+n) \leftarrow B$ where $n$ is a literal value. This allows for easy constant array indexing into an array that is not at a fixed location in memory, where $A$ is the array base and $n$ is the offset.

### *Arithmetic Operations.*

The *add*, *add immediate*, and *subtract* gadgets are straight forward. They work by loading the source operands into registers, performing the appropriate operation, and then storing the result back to memory. The x86 ISA allows one of the operands to be a location in memory which would obviate the need to load one of the operands. This could potentially simplify the gadgets.

The *negate* gadget loads the word from the source address, takes the two's complement of the word, and stores it back to memory. The neg instruction, which performs the two's complement of a register does not appear near a jmp ∗*x* instruction; we instead load esi with zero using xor %esi, %esi, then use the sequence subl `-0x7D`(%ebp,%ecx), %esi; jmp ∗(%ecx) to subtract the value from zero. The subl instruction performs the operation esi ← esi − (ebp + ecx − `0x7D`).[8] Since our jmp ∗*x* uses ecx, we have to load it with the address of a pointer to the pop *x*; jmp ∗*x* sequence. This means that ebp must have the value of the source address plus `0x7D` minus the address of the pointer to pop *x*; jmp ∗*x*.

### *Logical Operations.*

The *and*, *and immediate*, *or*, and *or immediate* gadgets are constructed in an analogous manner to the add gadget. Namely, the operands are loaded into registers, the operation is performed, and the result is stored back to memory. The only tricky part is the movement of data between registers as described above.

The *xor* and *xor immediate* gadgets are similar, but instead of xoring the value of two registers and then storing the results back to memory, they write the first source word to the destination and then xor that location with the second source word.

The *complement* gadget computes the one's complement of the source value and stores it into the destination address. Similar to the situation with the negate gadget, there is an x86 instruction not which performs the one's complement, but it does not appear in the useful instructions sequences in libc. Instead, we proceed exactly as for the negate gadget except instead of loading esi with zero, we load it with `0xffffffff` $= -1$. This works because $-1-x = \neg x$.

### *Branching.*

In a normal program, a branch can either be to an absolute address or to an address relative to the current instruction. In return-oriented programming, a branch is performed by changing the stack pointer rather than the instruction pointer. An absolute branch can be effected by popping a value off the stack into esp. Alternatively, a negative offset from the end of the gadget can be popped into edi which is then subtracted from the stack pointer using the sequence sub %edi, %esp; ljmp ∗(%eax) This allows stack-pointer-relative branching. This is the basis for our *branch unconditional* gadget.

[8] The parentheses denote dereference, not grouping.

In order to have Turing-complete behavior, we must have a way to perform a conditional branch. The x86 has a number of conditional branch operations; however, these are unsuitable for our purpose since they affect the instruction pointer rather than the stack pointer. Instead, we need a way to change the stack pointer conditioned on the word stored in memory at a known address. If the word is zero, then we do not change the stack pointer. If the word is `0xffffffff`, then we subtract an offset from the stack pointer as in the unconditional case. The way we do this is by loading the word into a register and anding with the offset. The result is subtracted from the stack pointer. The implementation is a straight-forward combination of the and gadget and the branch unconditional gadget and is our *branch conditional* gadget.

In any gadget set, the most difficult gadget to construct is the one that compares two values and performs an operation based on the relative magnitude of the values. Taking a cue from the MIPS architecture, we implement a *set less than* gadget that sets the word at the destination address equal to `0xffffffff` if the first source word is less than the second source word.

The implementation of the set less than gadget is given in Figure 2. The string compare instruction cmpsl compares the two words pointed to by %ds:%esi and %es:%edi and sets the carry flag if the latter is greater than the former. As a side effect, it increments or decrements registers esi and edi based on the direction flag; however, this is of no concern since we are only comparing a single word. The sbb instruction subtracts esi plus the value of the carry flag from esi. In essence, if the first source value is less than the second source value, then the carry flag will be set and esi is set to `0xffffffff`, otherwise, the carry flag will not be set and so esi will be set to zero, exactly as required for the branch conditional gadget. The one thing we have to be careful of is register cl cannot be zero otherwise a divide by zero exception will occur.

With the set less than and logical gadgets, a conditional branch based on comparing any two values for any of the six relations $<$, $\leq$, $=$, $\neq$, $\geq$, and $>$ can be formed. At this point our set of gadgets is Turing-complete.

### *Function Calls.*

Now that we have a Turing-complete set of gadgets, we extend their functionality by adding a gadget to perform function calls. This gives us two new abilities: we can call normal return-oriented instruction sequences — i.e., those ending in return — or we can call legitimate functions. Since we use an actual call instruction, any return-oriented programming defense relying on the LIFO nature of the call stack will be thwarted since this invariant is maintained. Any defense relying on the frequency of return instructions will be thwarted as long as the number of other instructions executed between these calls is sufficiently high.

Since calling legitimate functions is the more complicated of the two operations, we focus on it here. Calling a sequence ending in return is roughly the same except for moving the stack pointer and handling the return value.

Before a function call is made, the stack pointer must be moved to a new location to keep from overwriting our previous gadgets on the stack. If $n$ is the address where the stack pointer should be when the function begins to execute — i.e., the location where the return address will be stored — then the $k$ arguments should be stored at addresses $n+4, n+8, \ldots, n+4k$. This can be done using the load immediate or move gadgets. The *function call* gadget is then used to perform the computation $A \leftarrow fun(arg_1, arg_2, \ldots, arg_k)$ with the stack pointer set to $n$.

Since the Linux *application binary interface* (ABI) for x86 specifies that registers eax, ecx, and edx are caller-saved, we must take care that return-oriented code is not confused if a called function overwrites these registers. One particularly tricky point is that since edx is caller-saved, once we return from the call we need to restore it to the address of the pointer to the pop *x*; jmp ∗*x*. We *cannot* do this using only the instruction sequences in libc if we care about the return value which is in eax. Continuing our BYOPJ paradigm, if the target program has either a pop %edx; jmp ∗(%edx) or a pop %edx; jmp ∗(%esi), then we can restore edx without overwriting the return value in eax. Mozilla's libxul has such a sequence. Without such a sequence, the function call gadget has to be tailored for each application rather than being generic.

The implementation of the function call gadget is given in Figure 3. Some parts of the implementation are rather subtle. The first thing it does is to load registers esi, ebp, and eax. Register esi is loaded with the address of the sequence catalog entry for the call-jump sequence, ebp is loaded with the actual address of the leave-jump sequence, and eax is loaded with the literal value $n$ (plus the offset for our store sequence). Next, the address of the sequence catalog entry for the call-jump is stored at address $n$. Register esi is then loaded with `0x38` and the value of the stack pointer is added to it. At this point, esi holds the address we will set the stack pointer to after the the function call returns.

Now that we know the location on the stack we wish to return to after our function call, we need to move it into ebp. The easiest way to do this is to store it to memory (where we will eventually store the function's return value), load it back from memory into edi, then exchange it with ebp. After the exchange, edi holds the address of the leave-jump sequence and ebp holds the value we will set the stack pointer to after the function call. Next, we load esi with the address of the sequence catalog entry for pop *x*; jmp ∗*x*; load ecx with the address where the pointer to the function is stored (plus an offset); and load eax with the value $n$. Registers esp and eax are exchanged causing the stack pointer to be set to $n$.

Recall that the first thing the function call gadget did was to store the address of the catalog entry for the call-jump sequence to $n$. At this point, the indirect call of the function *fun* happens. After *fun* returns, we cannot rely on the values in registers ecx or edx while eax holds the return value. However, edi holds the address of the leave-jump sequence, thus the jmp ∗%edi instruction causes a leave instruction to be executed which sets the stack pointer to ebp — which is still holding the address we placed into it with the first xchg instruction — and then pops the value off of the top of the stack into ebp. This causes the address of the sequence catalog entry for pop *x*; jmp ∗*x* (plus an offset) to be loaded into ebp causing the subsequent jmp ∗`-0x7d`(%ebp) instruction to chain the next instruction sequence.

We now have two choices for the implementation. In the absence of a pop %edx; jmp ∗(%edx) sequence, we use a popad; jmp ∗(%edx), losing the return value. In this case, the function call gadget is complete. If we do have a pop %edx; jmp ∗(%edx) sequence, we execute that, and then store the return value in eax into memory. The latter form of the gadget is shown in Figure 3.

## 4. INSTANTIATION ON ARM

In this section we introduce our attack method and gadget set for ARM. We also provide some background information on ARM's RISC architecture.

### 4.1 ARM/THUMB Instruction Set

ARM is a 32-bit processor and features 16 general-purpose registers r0 to r15 as depicted in Table 1. All these registers can be accessed/changed directly. In contrast to the Intel x86 architecture, even the program counter pc can be accessed directly. Additionally,

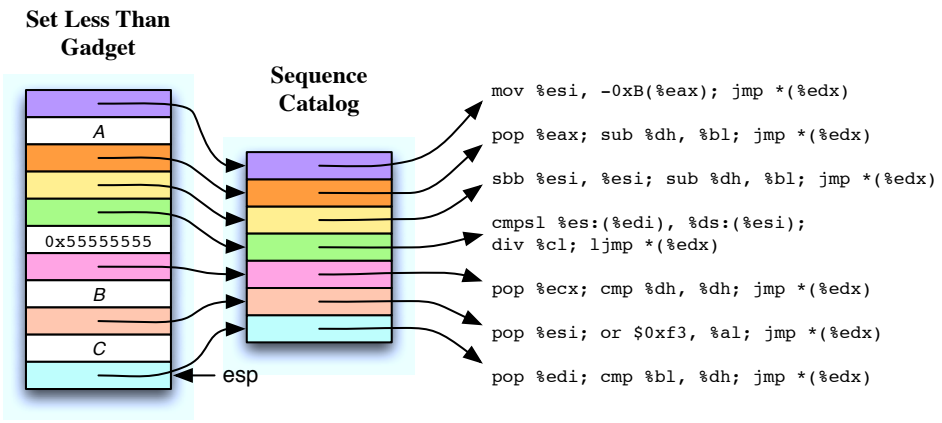


**Figure 2: Set less than gadget. If the word at address *B* is less than the word at address *C*, set the word at address *A* to `0xffffffff`; otherwise set it to `0x00000000`. The gadget begins executing with the stack pointer (`esp`) pointing to the bottom-most (smallest address) cell of the gadget. As execution proceeds, the stack pointer moves to higher cells (higher addresses). Each cell is either a pointer to an entry in the sequence catalog — which is itself a pointer to the instruction sequence that is actually executed — or data. After the final instruction sequence in the gadget has executed, the stack pointer points to the next gadget to be executed.**

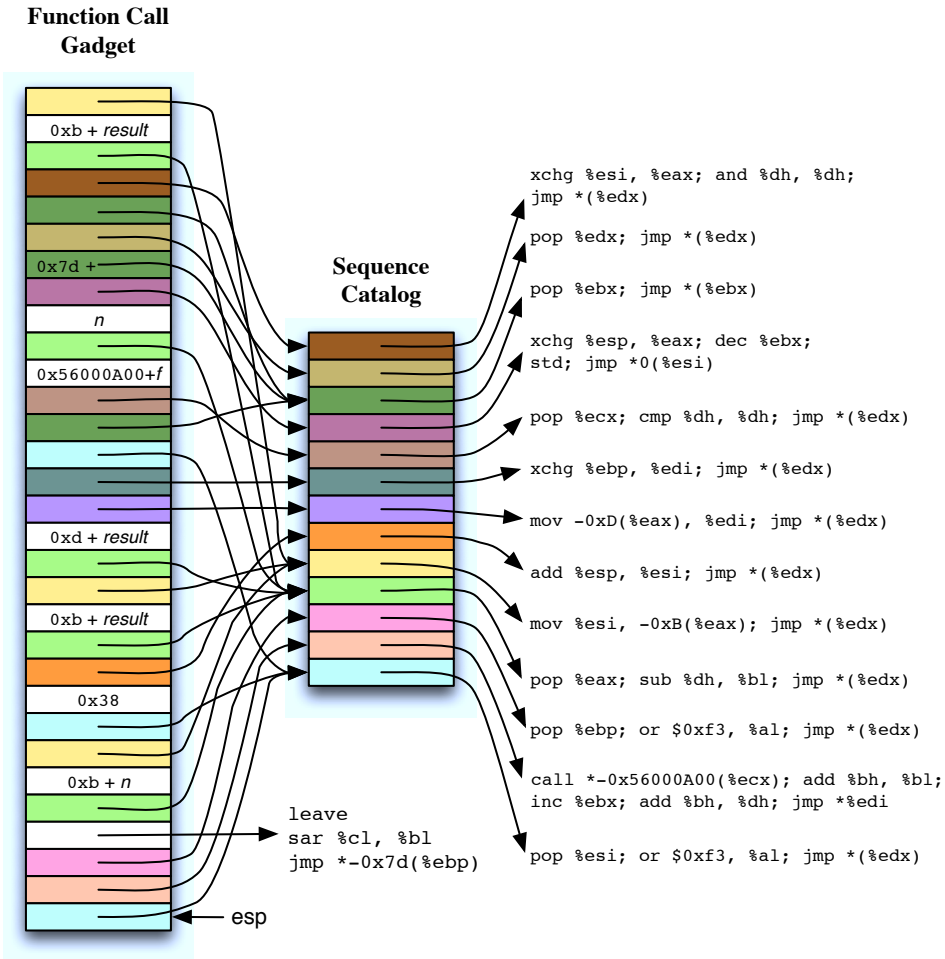


**Figure 3: Function call gadget. This convoluted gadget makes the function call $result \leftarrow f(arg_1, arg_2, \ldots, arg_k)$ where the arguments have already been placed at $n+4, n+8, \ldots, n+4k$. The return value is stored into memory at address *result*.**

ARM processors feature a current program status register (cpsr), which holds the current state of the system. It contains condition flags, interrupt enable flags, and the current mode.

Although ARM has a 32-bit RISC architecture, it also provides a 16-bit instruction set, called THUMB. The THUMB instruction set is a subset of the ARM instruction set and is in particular suit-

**Table 1: ARM Registers**

| Register | Purpose |
|---|---|
| r0–r3 | Function arguments; function results |
| r4–r11 | Register variables (callee saved) |
| r12 | Scratch register |
| r13 (sp) | Stack Pointer |
| r14 (lr) | Link Register (subroutine return address) |
| r15 (pc) | Program Counter |
| cpsr | Control Program Status Register |

able for embedded systems which often suffer from greater memory restrictions as PCs. In particular, the libraries libc and libwebcore which we use as the code base for our attack on ARM contain mainly THUMB instructions.

*Function Calls.*

According to the ARM Architecture Procedure Call Standard (AAPCS) [2], function calls are to be performed either through a bl or through a blx instruction. Both instructions perform a branch with link operation — that is, the program counter pc is loaded with the address of the subroutine and the link register lr is loaded with the return address. The blx instruction additionally allows switching between the ARM and THUMB instruction sets. Of the two instructions, only blx can perform an indirect jump through a register. Note that, in practice, not all function calls follow the AAPCS calling convention: Instead of transferring the return address to lr, the ARM C compiler may push the return address onto the stack and perform a direct branch to the function.

A function return is effected by writing the return address to the program counter pc. For this, the ARM architecture provides no dedicated return instruction. Instead, any instruction that is able to write to the program counter can be used as return instruction. For instance, one common return instruction is the bx lr instruction that branches to the address stored in the link register lr. It is also possible to use the ldm (load multiple) instruction to load the return address from the stack.

## 4.2 Attack Method Design

Our attack uses the indirect call form of the blx (Branch-Link-Exchange) instruction — that is, we use blx *r* where *r* is a general purpose register — to cause control to flow from one instruction sequence to another. As described above, the blx instruction is usually used for indirect function calls, potentially exchanging instruction sets.

The blx instruction is not a part of a function epilogue. Hence, an attack based on blx instructions cannot be detected by return address checkers. Moreover, in contrast to Intel's x86 call instruction, the blx instruction does not impact values on the stack (or generally on the memory), which makes the blx instruction very suitable for an attack. However, since the program counter pc can be accessed as a general purpose register, any instruction which uses the program counter pc as a destination register could also be used for the attack. We selected blx instructions because most of the instruction sequences we identified in our code base end with blx.

For extraction of a Turing-complete gadget set we manually inspected the system libraries libc and libwebcore of an Android 2.0 ("Eclair") platform for instruction sequences ending in blx instructions. Android's libc version is very compact, hence, we included Android's Web Browser library libwebcore to enlarge the code base. Both of the libraries (by default) are linked into the memory space of an application to fixed addresses. Note that the ARM gadget compiler proposed in [23] automatically extracts instruction sequences ending in a so-called "free branch" (such as blx). However, the gadgets in [23] are chained together by function epilogues which allow return address checkers to detect the attack. As we show in this paper our attack technique is solely based on blx instructions and requires no function epilogues. The instantiation of this type of attack is more involved (see also [11] for more details).

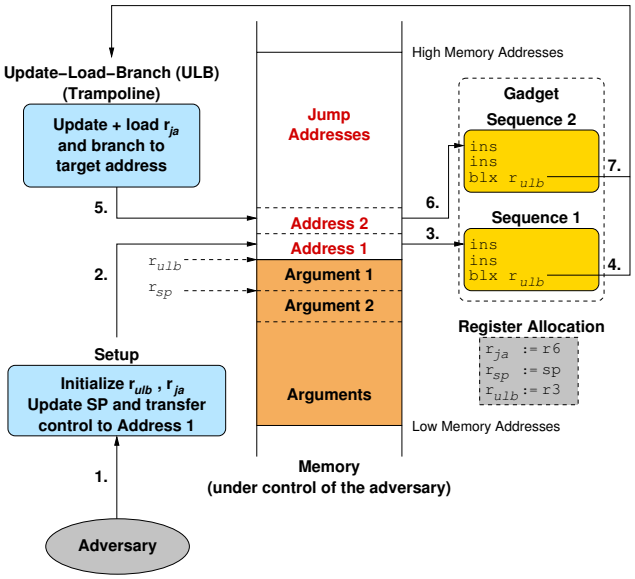


**Figure 4: The Attack Method/Architecture**

*Memory Layout.*

Figure 4 depicts the memory layout and the attack steps. The memory area under control of the adversary consists of jump addresses and arguments which are clearly separated from each other. Each jump address points to a specific instruction sequence whereas each sequence ends with a blx instruction in order to allow chaining of multiple sequences. We misuse the stack pointer sp as a pointer to arguments and need a second register (denoted with $r_{ja}$) as a pointer to jump addresses. We use the stack pointer because many sequences we identified in our code base contain load/store operations where sp is used as base register. However, the adversary is not forced to control the stack pointer. Instead any register ($r_{sp}$) can be used as pointer to arguments and data. The order of jump addresses and arguments highly depends on the appropriate instruction sequences found on a platform. For instance, if the instruction sequence which updates $r_{ja}$ adds a positive constant then jump addresses have to go from lower to higher memory addresses. In Figure 4 jump addresses go from lower to higher memory addresses and arguments are ordered vice versa. Of course, if jump addresses are not separated from arguments then one register could be saved. This is the approach we take on the x86 (see Section 3), where it is convenient to use pop instructions to load arguments while updating the stack pointer. On ARM, unfortunately, the libraries we examined load arguments without updating the stack pointer. That is the reason why we use $r_{ja}$ as pointer to jump addresses which is updated after each instruction sequence.

*Attack Steps.*

First, the adversary injects jump addresses and arguments to the stack or the heap (see Section 6 for a concrete example). Similar to x86 (see Section 3.2), our attack method on ARM consists mainly of three parts: (1) setup, (2) update-load-branch (ULB) sequence,

and (3) gadgets which consist of several instruction sequences. By subverting the control-flow, the adversary is able to initialize several registers. We refer to this process as a *setup* (step 1 in Figure 4).

The setup initializes three registers: $r_{ja}$, $r_{ulb}$, and $r_{sp}$. Registers $r_{ja}$ and $r_{sp}$ are used as a pointer to jump addresses and arguments. Register $r_{ulb}$ is loaded with the address of our ULB sequence (see below). Finally, the last action of our setup phase is to redirect execution to sequence 1 (steps 2 and 3). After sequence 1 completes its task, the blx instruction (located at the end of the sequence) redirects execution to our ULB sequence (step 4). The ULB sequence acts as trampoline by *updating* register $r_{ja}$, *loading* the address of sequence 2, and *branching* to sequence 2 (steps 5 and 6).

## 4.3 Gadget Set

The crucial part of our attack is to build a Turing-complete gadget set allowing an adversary to generate arbitrary program behavior. Generally, gadgets consist of several instruction sequences, whereas for our purposes the instruction sequences on ARM have to end in a blx instruction to redirect execution to our ULB sequence. We could construct all these gadgets using the sequences in our code base, namely the libraries libwebcore and libc of an Android 2.0 device.

### *Details of Setup and ULB Sequence.*

First, we describe the details of our setup and the ULB sequence. Since, our concrete attack directly initializes register r4 through r15 by exploiting a setjmp vulnerability on the heap, we assume for the moment that the adversary can directly initialize these registers. We will describe in Section 6 in more details how this can be achieved.

In Section 4.2 we introduced the registers $r_{ja}$, $r_{ulb}$, and $r_{sp}$ as the basis for our attack. The allocation of these registers fundamentally depends on the identified instruction sequences in our code base and involves technical challenges because these registers must be preserved during the execution of the gadget chain. For our code base we made the following allocation (as depicted in Figure 4): $r_{ja} = r6$, $r_{ulb} = r3$, and $r_{sp} = sp$. Further, we use following sequences for the setup and the ULB sequences:

```
ldr  r3,[sp,#0]; blx r3 // Setup
adds r6,#4; ldr r5,[r6,#124]; blx r5 // ULB
```

We use r3 for $r_{ulb}$ because most of the sequences in our code base end with a blx r3 instruction. Our setup sequence initializes r3 by loading the address of the ULB sequence from the stack through a ldr (load register) instruction into r3. Note, since our adversary is able to directly initialize r4–r15 by the setjmp vulnerability, we require no additional setup sequences for $r_{ja}$ and $r_{sp}$.

The ULB sequence acts as connector for all executed instruction sequences by (1) updating $r_{ja}$ after each sequence and (2) transferring control to the subsequent instruction sequence. Since registers r0–r3 are often used as destination registers before a blx instruction, we decided to use r6 as $r_{ja}$ register. The ULB sequence first increases register r6 by 4 bytes (Update), then loads the next jump address (by an offset of 124 bytes to r6) in r5 (Load), and finally branches to the loaded address (Branch). However, this sequence does not directly use $r_{ja}$ as branch destination register, rather it uses for this r5. Thus, we must take into account that the content of r5 is overwritten after each ULB sequence.

One technical problem we have to address is that most of our sequences use the pre-indexed addressing mode, which means that sp does not change its value after it is used as base register in a load operation. It would be desirable to directly load sp, but unfortunately, we have no such load operation in the sequences of our code base. Hence, we use the following sequence to update sp:

```
sub sp,#12; adds r0,r4,#0; blx r3
```

This sequence decreases the value of the stack pointer by 12 bytes and as a side-effect overwrites the value of register r0 with the content stored in r4. To preserve register r0, its value could first be stored to memory or moved to a free register.

### *Data Movement.*

Data movement gadgets are needed for loading and storing values from and to memory. Due to the RISC architecture of ARM processors, load and store operations are only permitted through dedicated instructions. The ARM instruction set offers for this the ldr and str instructions.[9] A register can be loaded through the ldr instruction. Storing a register to memory is performed through the str instruction. For instance, the following sequence loads a word from the stack (with zero bytes offset) into r1:

```
ldr r1,[sp,#0]; blx r3
```

To load an immediate value into a register the following sequence could be used, which loads NULL into r2:

```
movs r2,#0; blx r3
```

For a store operation we need at least two registers, one holding the word to be stored and one holding the target address. This can be achieved by load gadgets as described above. Finally, the store operation is performed through the str instruction.

### *Arithmetic Operations.*

Arithmetic operation gadgets include gadgets for addition, subtraction, multiplication and division. The ADD gadget can be realized with the arithmetic addition instruction adds as follows:

```
adds r0,r0,r2; blx r3
```

This sequence adds the contents of register r0 and r2 and stores the result in register r0.

Our SUB gadget is based on the arithmetic subtraction instruction subs as depicted in Figure 5. This gadget subtracts r0 from r4. Sequences 1 and 2 load the first operand into r4 through r0, whereas the conditional branch in sequence 2 will be never taken, because r3 holds the address of the ULB sequence (which does not equal NULL). Afterwards, sequence 3 loads r0 with the second operand. The fourth sequence loads the address where the result will be stored into register r2. Finally, the last sequence performs the subtraction and stores the result at memory position [sp, #32] and in register r1.

The remaining MUL and DIV gadget can be realized by invoking the ADD and SUB gadget in a loop.

### *Logical Operations.*

The design of logical gadgets is very similar to the design of arithmetic gadgets. First, they load the source registers. Then, they perform the desired operation. For example, we instantiate the AND gadget as depicted in Figure 6.

First, sequences 1 and 2 load the first operand into register r7. Next, sequence 3 loads the second operand into register r1, and finally, sequence 4 performs the and operation on register r1 and r7, the result is stored into register r7.

One important logical gadget to mention is the NOT gadget that computes the two's complement of a specific value. We realize the NOT gadget (based on the ideas presented in Section 3.2) by subtracting the source register from $-1$. The AND and NOT gadgets

[9] In addition to these two instructions, ARM provides the ldm and stm instructions for a multiple load and store operation.

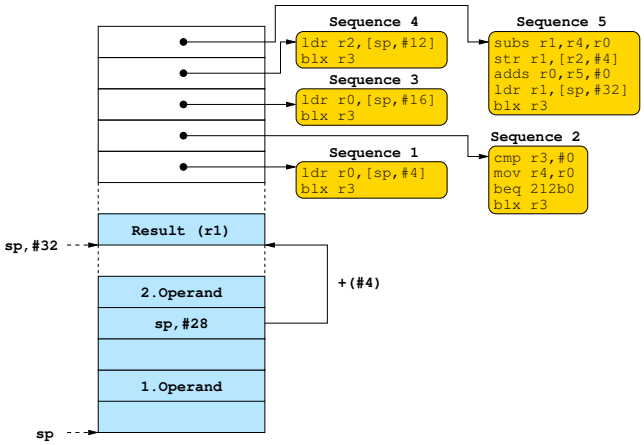


**Figure 5: The Subtract Gadget**

can be combined to form a NAND gadget. All other logical operations (such as or, xor) can be emulated through our NAND gadget. Similarly, the negate gadget can be simulated through a SUB gadget by subtracting the source register from zero.

### *Branching.*

Branching implicates changing the $r_{ja}$ (r6) register rather than the instruction pointer. The unconditional branching gadget can be realized by adding an offset to register $r_{ja}$, or by directly loading $r_{ja}$ with a new value.

Our conditional branching gadget is realized by the same approach we used for Intel x86 (see Section 3.2): We compare two values and depending on the result, $r_{ja}$ is either changed by an unconditional branch gadget or remains as it is. To realize this gadget, we need a compare operation. This can be simulated through a SUB gadget updating the carry flag in the cpsr register. The updated carry bit is then added to the constant `0xffffffff`, hence the result will be either `0x0` or `0xffffffff`. Finally, the result must be anded with the desired branch offset. The result of this last operation will be either `0x0` (Carry Bit = 1) or the offset (Carry Bit = 0), which is finally added to $r_{ja}$.

### *System and Function Calls.*

System calls are highly important for runtime attacks. Basically, they are needed to invoke special services of the operating system (e.g., opening a file or executing a new program). For instance, conventional code injection attacks use the execve system call to execute a program such as */bin/sh*. System calls are also often implemented as functions in libc. Thus, a program only needs to invoke the appropriate system call function. A common alternative to this scheme consists of passing arguments in registers and in storing the system call number in a register (e.g., on ARM r7, and on Intel eax). The system call is then invoked through a software interrupt (e.g., on ARM svc `0x0` (Supervisor Call), and on Intel sysenter).

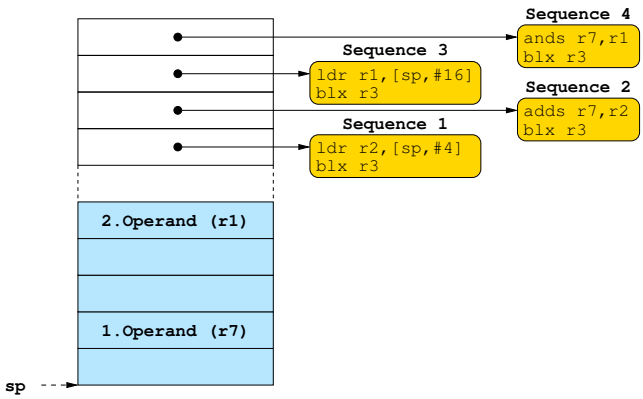


**Figure 6: AND Gadget**

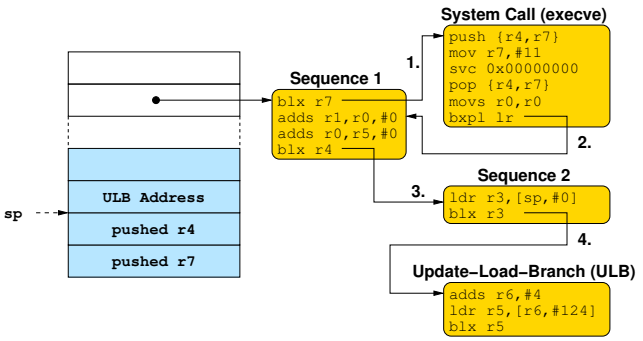


**Figure 7: System Call Gadget**

The libc version of Android OS implements system calls by transferring the system call number to r7. Therefore all system call functions only differ in the movs r7, #SYS_NR instruction. We have inspected the appropriate libraries and could not identify a svc `0x0` instruction. Hence, we can only invoke a system call by calling the appropriate function in libc. Thus, our system call gadget is, in actuality, a general purpose function call gadget.

Our system/function call gadget is depicted in Figure 7. We have to take into account that the blx instruction loads the return address into the link register lr. Since the bxpl lr[10] (located at the end of the execve function) redirects execution back to the value stored in the link register, we have to ensure that lr points at that time to a valid instruction sequence. However, when the blx instruction is invoked, lr will be automatically loaded with the address of [pc,#2] (for Thumb compiled code). Hence, we use an instruction sequence with two blx instructions (sequence 1). The arguments for the system call must be initialized by load gadgets (not depicted in Figure 7). Usually, registers r0 through r3 hold arguments for a system call. If a system call expects an argument in r3, then our $r_{ulb}$ will be overwritten. Thus, we must temporarily change the $r_{ulb}$ to a different register if r3 is used as argument.

First, sequence 1 invokes the system call function with the address of the system call function stored in r7 (step 1). After the system call returns, the bxpl lr instruction redirects execution back to sequence 1 (step 2). Afterwards, sequence 1 performs two data movement instructions[11] and then redirects execution to sequence 2 (step 3). This sequence re-initializes our $r_{ulb}$ register r3 with the address of the ULB sequence. Finally, sequence 2 redirects execution to the ULB sequence which loads the next jump address (step 4).

As can be seen in Figure 7, the system call function pushes two values onto the stack. Since we separated arguments from jump addresses, the two push instructions will only overwrite arguments and not jump addresses. However, one can backup the affected arguments through a store gadget before invoking the system call.

[10]The condition flag pl means that the branch will only be executed if the N flag in the cpsr register is not set. The N flag will be set if r0 holds a negative value. This will only be the case if an error occurred during the system call.

[11]Note that a function returning a 64 bit value (e.g., long long or double), will load the return value in r0 and r1. However, as a side-effect our gadget overwrites the value of r1 after the function retuns. To allow function calls with a return value greater than 32 bits we have to use a sequence that preserves r1.

# 5. GETTING STARTED

Return-oriented programming is an alternative to code injection when an attacker has diverted a target program's control flow by taking advantage of a memory error such as a buffer overflow. How the initial control flow diversion is accomplished, then, is orthogonal to the question of return-oriented programming.[12]

All the same, some of the traditional means of diverting control flow require the target program to execute a return instruction, which means they risk detection by the defenses our new return-oriented programming are designed to evade.

In some cases, a different approach will allow attackers to avoid this initial return. In this section, we discuss four classes of memory errors from the perspective of the returnless return-oriented programming paradigm and consider for each the prospects for an attacker to take control without using a return instruction. Recall that, in order for a return-oriented exploit to be successful, the attacker must gain control of both the instruction pointer and the stack pointer — or, more generally, the state used for the update-load-branch instruction sequence. In addition, the return-oriented program must be some place in memory.

### *Stack Buffer Overflow.*

The traditional means of exploiting a stack buffer overflow is to overwrite the saved instruction pointer in some function's stack frame. When that function returns, control will flow not to the instruction after the call that invoked the function but rather to any location of the attacker's choosing. In a return-oriented attack, this will be the first instruction sequence in the first gadget laid out on the stack; conveniently, the stack pointer will point to the next word on the stack, which is also under attacker control. By this point, however, the LIFO invariant of the return-address stack has been violated. (A single return instruction would not, of course, be caught by defenses that look for several returns in close succession.)

To take advantage of a stack buffer overflow without a return, an attacker must overwrite stack frames without modifying any saved instruction pointers. Instead, she should change pointer data such as function pointers in a function frame above the one that contains the overflowed buffer. Once the function containing the buffer has returned (to the function that legitimately called it), the memory around the stack pointer will be attacker-controlled; when the pointer she modified is used, an instruction sequence such as popad; jmp $*y$ or ldm sp!, {r0-r11}; blx r3 as its target will give her control of the registers and begin running return-oriented code.

### *Setjmp Buffer Overwrite.*

The setjmp and longjmp functions allow for nonlocal gotos. A program allocates space for a jmp_buf structure, which includes an array of words long enough to hold the callee-saved registers. When setjmp is called, it stores the values of those registers into the jmp_buf. The instruction pointer stored into the buffer is the normal subroutine return address — the saved instruction pointer pushed onto the stack by the call instruction on x86 or the lr on the ARM — and the stored stack pointer is the value the stack pointer had before the call to setjmp. When setjmp returns, it returns zero.

When, later, longjmp is called, it restores the general-purpose registers to their previous values, sets the return-value register — eax or r0 — to longjmp's second argument, sets the stack pointer, and finally does an indirect jump to the saved instruction pointer. In essence, setjmp returns two times while longjmp never returns.

If an attacker is able to write the exploit program to some location in memory and overwrite two words of a jmp_buf — the stack and instruction pointers — that is subsequently the first argument to a longjmp call, then the attacker can arrange for his return-oriented exploit to run. This method of transferring control to a return-oriented program is so convenient that it was employed for testing the gadgets described in Sections 3 and 4. See Section 6 for an example of this method.

### *C++ Vtable Pointer Overwrite.*

If the attacker overwrites an object instance of a class with virtual functions on the heap, then there is (in the general case) no hope of controlling memory around the stack pointer. However, the attacker will control the memory around the object itself, as well as around the object's vtable, since in overwriting the object she can cause the vtable pointer to point at some memory under her control, such as a packet buffer on the heap. Depending on the code that the compiler generates for virtual method invocation, then, at the time that an instruction sequence is invoked, one or more registers will point to the object, the vtable, or both. The attacker must leverage these pointers (1) to change the stack pointer to memory she controls, and (2) to cause a second instruction sequence to execute after the first.

Being able to leverage a vtable pointer overwrite to take control in a generic way (i.e., one that depends only on the compiler version and flags, not on the program being attacked) is an open problem. The alternative is to generate an exploit that is specific to the program attacked, the way that, for example, alphanumeric shellcodes must be written differently depending on what register or memory location they can consult to find the shellcode's location [47].

### *Function Pointer Overwrite.*

With a function pointer overwrite on the heap, as with a vtable pointer overwrite, the challenge for the attacker is twofold. The first code sequence she causes to execute must both relocate the stack to memory she controls and arrange for a second instruction sequence to execute in turn. It is likely that this is impossible generically without using the return instruction, and a specific exploit must be crafted for each target program.

# 6. CONCRETE ATTACKS

## 6.1 Linux Intel x86

We construct a complete, working shellcode using a return-oriented program without returns and which contains no zero bytes making it usable with a strcpy vulnerability. Once control flow has transferred to the shellcode, it sets up the arguments for a call to the syscall function.

```
syscall( SYS_execve, "/bin/sh",
         argv, evnp )
```

The target program, given in Listing 1, allocates enough memory on the heap to hold a 160 byte character array and a jmp_buf. Then, setjmp is called to initialize the jmp_buf and the target program's first argument is copied to the character array. Finally, longjmp causes control flow back to the point of the setjmp's return and the program exits. The target program is compiled and linked with Mozilla's libxul to provide the two instruction sequences pop %ebx; jmp $*$(%ebx) and pop %edx; jmp $*$(%edx) as described in Section 3. This is a toy program; we include it not because we are interested in exploiting such programs but because it lets us gauge a baseline for the size of a complete return-oriented exploit.

[12] Also orthogonal are defenses against buffer overflows such as stack cookies or generally against reliable exploitation such as address-space randomization. Such defenses, like the ones we consider in this paper, and unlike CFI, are ad-hoc. They defeat certain exploits but can be bypassed in some cases. See, e.g., [43, 46].

**Listing 1: Target program for our example exploit.**

```
struct foo {
    char buffer[160];
    jmp_buf jb;
};

int main( int argc, char **argv ) {
    struct foo *f = malloc( sizeof *f );
    if( setjmp(f->jb) )
        return 0;
    strcpy( f->buffer, argv[1] );
    longjmp( f->jb, 1 );
}
```

The shellcode "egg" we wrote (see [5, Listing 2]) consists of four parts: (1) the return-oriented program; (2) data used by the program; (3) the instruction sequence catalog; and (4) data overwriting the jmp_buf. The program consists of a sequence of pointers to the sequence catalog and values to load into registers. The jmp_buf pointers are overwritten to point the stack pointer at the beginning of the program and the instruction pointer at the sequence pop %edx; jmp *(%edx) in libxul. The program xors esi with itself to clear it and uses this register to write zero words in the data section as needed; it then restores important nonzero data that was overwritten; and finally, it calls the syscall function, with arguments from the exploit's data section.

The pop %edx; jmp *(%edx) sequence can be replaced with popad; cld; ljmp *(%edx) from libc. This requires the use of a far pointer which contains `00` as its final byte. A strcpy vulnerability allows writing a single terminating zero byte. Thus, our shellcode egg can contain exactly one far pointer at the very end.

When the target program is run with the exploit egg as its first argument, the result is a new shell.

## 6.2 Google Android ARM

In the following we provide background information on Google Android and show details of our attack mounted on a device emulator hosting Android 2.0 ("Eclair").

### *Background on Google Android.*

Android is an open source operating system for mobile devices which includes a customized Linux kernel, middleware framework and core applications. It is used in modern Google smartphones such as Motorola Droid and a number of devices from the HTC manufacturer (HTC Droid Eris, HTC Imagio, HTC Hero, etc.).

The Android platform is based on a Linux kernel, which provides low-level services to the rest of the system such as networking, storage, memory and processing. A middleware layer consists of native C/C++ libraries, an optimized Java virtual machine called Dalvik Virtual Machine (DVM), and core libraries written in Java. The DVM executes binaries of applications from upper layers.

Android applications are written in Java, but can also access C/C++ libraries via the Java Native Interface (JNI). Application developers may use JNI to incorporate C/C++ libraries into their applications. Moreover, many C libraries are mapped by default to fixed memory addresses in the program memory space. This provides a large C/C++ code base that we exploit for our attack.

### *Attack Instantiation.*

Similar to our attack on Intel x86, we aim to launch Android's terminal application. The terminal application is part of the DevTool application, which is included by default in the Android emulator image. However, we were able to launch a similar attack on a real device. In particular, we succeeded to run the attack on Dev Phone 2 with the latest Android version available for this device, Android 1.6 ("Donut"). However, the Android image flashed on to the real device differs from the image in the emulator in that it has no DevTool application installed by default. Thus, the attack on a real device would require the additional assumption that a terminal application such as DevTool or AndroidTerm[13] is installed on the device.

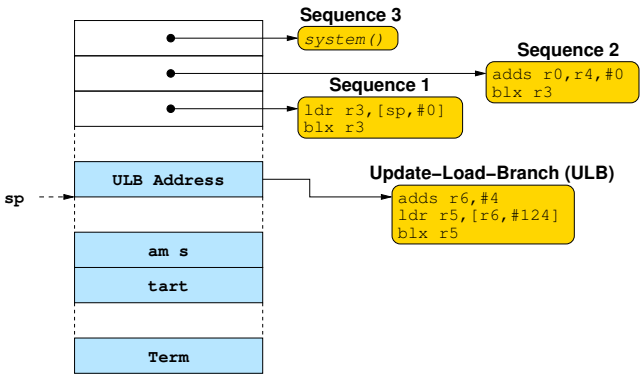


**Figure 8: The Gadget Chain of the Attack on Android**

We incorporated our target program, given in Listing 1, as native code to a standard Java application in Android by using the JNI. Due to the inclusion of C/C++ libraries, the security guarantees provided by the Java programming language do not hold any longer.[14] Instead of using the strcpy vulnerability, on Android, we use the fgets function, because it allows us to read zero bytes, which we need for initializing sp, because our target program always allocates the `foo` structure at a memory position starting with `0x00`. The fgets function reads the specified number of bytes from a file into the `buffer` member of the `foo` structure without checking the bounds of the buffer.

In order to mount our attack against the target program, our gadget chain invokes the system libc function as follows:

```
am start -a android.intent.action.MAIN
-c android.intent.category.TEST
-n com.android.term/.Term
```

This command invokes the Activity Manager application which in turn starts a terminal from the DevTool application.

All used gadgets used in our attack on Android are shown in Figure 8. To invoke the system function, we (1) initialize register r6 and sp by means of the setjmp heap overflow; (2) load r3 with the address of our ULB sequence (sequence 1); (3) load the address of the interpreter command in r0 (sequence 2); (4) finally invoke the libc system function (sequence 3). The corresponding exploit payload is included in the tech report [11].

# 7. CONCLUSION

We have shown that on the x86 and ARM it is possible to mount a return-oriented programming attack without using any return instructions. In the new attack, certain return-like instruction sequences take the place of the return instruction. These instruction sequences are rare but a single one used as a trampoline suffices to chain together other instruction sequences that each end in an indirect jump, which makes it possible to construct a Turing-complete

[13] `http://code.google.com/p/androidterm/`

[14] In particular, Tan and Croft [44] identified various vulnerabilities in native code of the JDK (Java Development Kit).

gadget set without return instructions given large Linux (x86) or Android (ARM) platform libraries.

Because it does not make use of return instructions, our new attack has negative implications for recently proposed classes of defense against return-oriented programming that detect too-frequent use of returns in the instruction stream, that detect violations of the LIFO invariant normally maintained for the return-address stack, or that rewrite binaries to avoid use of the return instruction.

The major open problem suggested by our work is whether it is possible to find some property that *all* return-oriented attacks provably must share, but that is more specific (and therefore more efficiently checked) than CFI, which would rule out all control-flow attacks. The use of return instructions to chain sequences appeared to be such a property, but we have shown that it is not. Such a property could be used as part of a defense against return-oriented programming, assuming that it can be efficiently tested. In the absence of such a narrowly tailored property, it is not clear that effective defenses against return-oriented programming can be deployed at lower overhead than full CFI.

# 8. ACKNOWLEDGMENTS

We thank Thorsten Holz, Tim Kornau, Benny Pinkas, Stefan Savage and Geoff Voelker for helpful discussions. This material is based upon work supported by the National Science Foundation under Grant No. 0831532. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The second author was supported by EU FP7 project CACE and the third author by the Erasmus Mundus External Co-operation Window Programme of the European Union.

# 9. REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In V. Atluri, C. Meadows, and A. Juels, editors, *Proceedings of CCS 2005*, pages 340–53. ACM Press, Nov. 2005.

[2] ARM Limited. Procedure call standard for the ARM architecture. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf`, 2009.

[3] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 27–38. ACM Press, Oct. 2008.

[4] 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 D. Jefferson, J. L. Hall, and T. Moran, editors, *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, Aug. 2009.

[5] S. Checkoway and H. Shacham. Escape from return-oriented programming: Return-oriented programming without returns (on the x86). Technical Report CS2010-0954, UC San Diego, Feb. 2010.

[6] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. DROP: Detecting return-oriented programming malicious code. In A. Prakash and I. Sengupta, editors, *Proceedings of ICISS 2009*, volume 5905 of *LNCS*, pages 163–77. Springer-Verlag, Dec. 2009.

[7] T. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In P. Dasgupta and W. Zhao, editors, *Proceedings of ICDCS 2001*, pages 409–17. IEEE Computer Society, Apr. 2001.

[8] J. R. Crandall, S. F. Wu, and F. T. Chong. Experiences using Minos as a tool for capturing and analyzing novel worms for unknown vulnerabilities. In K. Julisch and C. Krügel, editors, *Proceedings of DIMVA 2005*, volume 3548 of *LNCS*, pages 32–50. Springer-Verlag, July 2005.

[9] D. Dai Zovi. Practical return-oriented programming. SOURCE Boston 2010, Apr. 2010. Presentation. Slides: `http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf`.

[10] dark spyrit. Win32 buffer overflows (location, exploitation and prevention). *Phrack Magazine*, 55(15), Sept. 1999. `http://www.phrack.org/archives/55/p55_0x0f_Win32%20Buffer%20Overflows..._by_dark%20spyrit.txt`.

[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-oriented programming without returns on ARM. Technical Report HGI-TR-2010-002, Ruhr-University Bochum, July 2010. Online: `http://www.trust.rub.de/home/_publications/DaDmSaWi2010/`.

[12] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In N. Asokan, C. Nita-Rotaru, and J.-P. Seifert, editors, *Proceedings of STC 2009*, pages 49–54. ACM Press, Nov. 2009.

[13] 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, Ruhr-University Bochum, Mar. 2010. Online: `http://www.trust.rub.de/home/_publications/LuSaWi10/`.

[14] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: Software guards for system address spaces. In B. Bershad and J. Mogul, editors, *Proceedings of OSDI 2006*, pages 75–88. USENIX, Nov. 2006.

[15] A. Francillon and C. Castelluccia. Code injection attacks on Harvard-architecture devices. In P. Syverson and S. Jha, editors, *Proceedings of CCS 2008*, pages 15–26. ACM Press, Oct. 2008.

[16] A. Francillon, D. Perito, and C. Castelluccia. Defending embedded systems against control flow attacks. In S. Lachmund and C. Schaefer, editors, *Proceedings of SecuCode 2009*, pages 19–26. ACM Press, Nov. 2009.

[17] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In D. Wallach, editor, *Proceedings of USENIX Security 2001*, pages 55–66. USENIX, Aug. 2001.

[18] S. Gupta, P. Pratap, H. Saran, and S. Arun-Kumar. Dynamic code instrumentation to detect and recover from return address corruption. In N. Gupta and A. Podgurski, editors, *Proceedings of WODA 2006*, pages 65–72. ACM Press, May 2006.

[19] R. Hund. Listing of gadgets constructed on ten evaluation machines. Online: `http://pi1.informatik.uni-mannheim.de/filepool/projects/return-oriented-rootkit/measurements-ro.tgz`, May 2009.

[20] R. Hund, T. Holz, and F. Freiling. Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms. In

F. Monrose, editor, *Proceedings of USENIX Security 2009*, pages 383–98. USENIX, Aug. 2009.

[21] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*, 2001.

[22] V. Iozzo and C. Miller. Fun and games with Mac OS X and iPhone payloads. Black Hat Europe 2009, Apr. 2009. Presentation. Slides: http://www.blackhat.com/presentations/bh-europe-09/Miller_Iozzo/BlackHat-Europe-2009-Miller-Iozzo-OSX-IPhone-Payloads-whitepaper.pdf.

[23] T. Kornau. Return oriented programming for the ARM architecture. Master's thesis, Ruhr-Universität Bochum, Jan. 2010. Online: http://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf.

[24] S. Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, Sept. 2005. http://www.suse.de/~krahmer/no-nx.pdf.

[25] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with "return-less" kernels. In G. Muller, editor, *Proceedings of EuroSys 2010*, pages 195–208. ACM Press, Apr. 2010.

[26] F. Lidner. Developments in Cisco IOS forensics. CONFidence 2.0, Nov. 2009. Presentation. Slides: http://www.recurity-labs.com/content/pub/FX_Router_Exploitation.pdf.

[27] D. Litchfield. Defeating the stack based buffer overflow prevention mechanism of Microsoft Windows 2003 Server, Sept. 2003. Online: http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf.

[28] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In V. Sarkar and M. W. Hall, editors, *Proceedings of PLDI 2005*, pages 190–200. ACM Press, June 2005.

[29] J. McDonald. Defeating Solaris/SPARC non-executable stack protection. Bugtraq, Mar. 1999. Online: http://seclists.org/bugtraq/1999/Mar/4.

[30] R. Naraine. Pwn2Own 2010: iPhone hacked, SMS database hijacked. Online: http://blogs.zdnet.com/security/?p=5836, Mar. 2010.

[31] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 58(4), Dec. 2001. http://www.phrack.org/archives/58/p58_0x04_Advanced%20return-into-lib(c)%20exploits%20(PaX%20case%20study)_by_nergal.txt.

[32] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In J. Ferrante and K. S. McKinley, editors, *Proceedings of PLDI 2007*, pages 89–100. ACM Press, June 2007.

[33] T. Newsham. Re: Smashing the stack: prevention? Bugtraq, Apr. 1997. Online: http://seclists.org/bugtraq/1997/Apr/129.

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

[35] M. Prasad and T. Chiueh. A binary rewriting defense against stack based overflow attacks. In B. Noble, editor, *Proceedings of USENIX Technical 2003*, pages 211–24. USENIX, June 2003.

[36] G. Richarte. Re: Future of buffer overflows? Bugtraq, Oct. 2000. Online: http://seclists.org/bugtraq/2000/Nov/32 and http://seclists.org/bugtraq/2000/Nov/26.

[37] G. Richarte. Insecure programming by example: Esoteric #2. Online: http://community.corest.com/~gera/InsecureProgramming/e2.html, July 2001.

[38] R. Roemer. Finding the bad in good code: Automated return-oriented programming exploit discovery. Master's thesis, UC San Diego, Mar. 2009. Online: https://cseweb.ucsd.edu/~rroemer/doc/thesis.pdf.

[39] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. Manuscript, 2009. Online: https://cseweb.ucsd.edu/~hovav/papers/rbss09.html.

[40] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In I. Goldberg, editor, *Proceedings of USENIX Security 2010*, pages 1–11. USENIX, Aug. 2010.

[41] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In S. De Capitani di Vimercati and P. Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, Oct. 2007.

[42] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702, 2008.

[43] A. Sotirov and M. Dowd. Bypassing browser memory protections in Windows Vista. Online: http://www.phreedom.org/research/bypassing-browser-memory-protections/, Aug. 2008. Presented at Black Hat 2008.

[44] G. Tan and J. Croft. An empirical security study of the native code in the JDK. In P. Van Oorschot, editor, *Proceedings of USENIX Security 2008*, pages 365–77. USENIX, July 2008.

[45] Vendicator. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield.

[46] P. Vreugdenhil. Pwn2Own 2010 Windows 7 Internet Explorer 8 exploit. Online: vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf, Mar. 2010.

[47] B.-J. S. Wever. ALPHA2: Zero tolerance, Unicode-proof uppercase alphanumeric shellcode encoding. Online: http://skypher.com/wiki/index.php/ALPHA2, 2004.

[48] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In A. Myers and D. Evans, editors, *Proceedings of IEEE Security and Privacy ("Oakland") 2009*, pages 79–93. IEEE Computer Society, May 2009.