On the Expressiveness of Return-into-libc Attacks

Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang,
Vincent Freeh, and Peng Ning

Department of Computer Science
North Carolina State University, Raleigh, NC, USA
`{mqtran,mnetheri,tkbletsc,pning}@ncsu.edu`, `{jiang,vin}@csc.ncsu.edu`

Abstract. Return-into-libc (RILC) is one of the most common forms of code-reuse attacks. In this attack, an intruder uses a buffer overflow or other exploit to redirect control flow through existing (libc) functions within the legitimate program. While dangerous, it is generally considered limited in its expressive power since it only allows the attacker to execute straight-line code. In other words, RILC attacks are believed to be incapable of arbitrary computation—they are not Turing complete. Consequently, to address this limitation, researchers have developed other code-reuse techniques, such as return-oriented programming (ROP). In this paper, we make the counterargument and demonstrate that the original RILC technique is indeed Turing complete. Specifically, we present a generalized RILC attack called *Turing complete RILC* (TC-RILC) that allows for arbitrary computations. We demonstrate that TC-RILC satisfies formal requirements of Turing-completeness. In addition, because it depends on the well-defined semantics of libc functions, we also show that a TC-RILC attack can be portable between different versions (or even different families) of operating systems and naturally has negative implications for some existing anti-ROP defenses. The development of TC-RILC on both Linux and Windows platforms demonstrates the expressiveness and practicality of the generalized RILC attack.

Keywords: Return-into-libc, return-oriented programming, Turing-complete

1 Introduction

Computer systems are under constant threat by hackers who attempt to seize unauthorized control for malicious ends. One popular method of attack is *code injection*, in which the attacker injects machine code into the target application's memory, then exploits a software bug to divert control flow to the injected code. Recently, code injection has been largely mitigated with the proposition and deployment of the W⊕X scheme, wherein hardware and OS features are employed to guarantee that writable memory pages cannot be executed.

Because of this, attackers have turned to code-reuse attacks, in which legitimate code is reused for malicious purposes. The simplest and most common form of code-reuse attack is *return-into-libc* (RILC) [1]. In RILC, the attacker arranges for the stack pointer to point to a series of malicious stack frames injected into the program's memory. When the program returns from the current function, control flow is redirected to the entry point of another function chosen by the attacker. The stack frame also contains necessary function arguments, so that the function is executed with attacker-supplied parameters. Moreover, such calls can be chained, allowing the attacker to execute a sequence of arbitrary function calls [1]. This capability is most commonly used to execute `mprotect()` to disable W$\oplus$X protection or `system()` to launch another program.

Though the RILC technique is indeed powerful, it is widely believed that a RILC attack is capable of only linearly chaining multiple functions, but *not* arbitrary computations—i.e., it is not Turing complete [1–6]. For example, in the seminal ROP paper by Shacham et al. [2], it is explained that "in a return-into-libc attack, the attacker can call one libc function after another, but this still allows him to execute only straight-line code, as opposed to the branching and other arbitrary behavior available to him with code injection"[2]. This common belief motivated researchers to develop a new code-reuse attack, i.e., *return-oriented programming* (ROP), in which a similar stack exploit is used to weave together small snippets of code called gadgets.[1] Given a sufficiently large codebase (such as the ubiquitous libc), ROP has been shown to be Turing complete. Since the introduction on x86, there has been a flurry of research that apply ROP to other platforms (including SPARC and ARM) and build ROP-based malware to subvert kernel integrity, bypass software-based attestation schemes, compromise electronic voting machines, and more [7], [8], [9], [10], [11].

In this paper, we investigate the expressiveness of traditional RILC attacks and make the counterargument that they are in fact Turing complete and therefore equal in expressive power to ROP. Specifically, based on the previous capability of calling one function after another (exhibited in traditional RILC attacks), our extension uniquely combines existing libc functions to construct arbitrary computations. We call this variant of RILC *Turing-complete return-into-libc* (TC-RILC). This result directly challenges the notion that the traditional RILC attack is limited in expressive power.

In addition, because TC-RILC relies on the intended semantics of the functions being used, we also show that it inherits one inherent advantage from traditional RILC attacks over ROP: it is relatively straightforward to port attacks between different versions (or even different families) of operating systems. For example, the adversary can retarget their RILC-based Linux attack code to any other UNIX-style operating system (or a Microsoft Windows attack code to any other version of the Windows from Windows 95 to Windows 7). Specifically, if an attack can be constructed from widely available functions (e.g., POSIX standard functions that are common on virtually all Linux, UNIX, and Win-

[1] There is some dispute over the precise definitions of *return-into-libc* attacks versus *return-oriented programming*. For clarity, in this paper we adopt the view that the two are separate techniques which, though they use similar means, differ wildly in construction.

dows environments), such attack code can be nearly universal.[2] Our experience indicates that the portability directly comes from traditional RILC attacks and the implementation-specific data needed are usually the actual function entry points and certain data structures. This is a stark contrast to ROP, wherein one needs to implement a scanner to find all the gadgets again. In other words, moving a ROP attack to a different version of the same OS or a different OS family requires re-identifying a complete new set of gadgets. Further, even though our focus in this paper is to correct the record, and not about presenting TC-RILC as an invincible threat to negate existing defenses, we note that because TC-RILC attacks do not have certain peculiarities specific to ROP, our technique naturally has negative implications for some anti-code-reuse defenses [4],[12],[13] that target ROP.

Recognizing the evolving nature of arms-race between code injection attacks and defenses, we believe it is important to fully understand the limits and capabilities of these attack techniques. By clarifying the expressiveness of RILC attacks with this paper, we hope to rectify the previous misconception of its capability and further spur research into better defenses.

To summarize, the contributions of this paper are as follows:

– First, we show that traditional RILC attacks can be Turing complete, disproving the commonly held misconception that such attacks are inherently linear and therefore less expressive than ROP.
– Second, we show that TC-RILC largely depends on the well-defined semantics of libc functions instead of the low-level machine code snippets used by ROP. As these well-defined semantics are consistently maintained and compatible among different versions or even different families of OSs, a TC-RILC attack can be ported more easily across OS variants and families.
– Third, we demonstrate the practicality of this technique by implementing two example exploits: a universal Turing machine simulator and an implementation of the selection sort algorithm. Together, these examples demonstrate the expressiveness and practicality of the technique.

2 Traditional View of RILC Attacks (on x86)

Our work aims to demonstrate the expressive power of the traditional return-into-libc (RILC) attack; thus, we adopt the same threat model and assumptions as prior literature dealing with this technique. Specifically, the traditional RILC attack requires that an attacker be able to place a payload into memory (i.e., onto the stack) and hijack the `esp` register (which essentially becomes the defacto program counter in RILC). Such assumptions are made possible by the commonality of vulnerabilities such as buffer overruns and format string bugs. In addition, the attack depends on the presence of functionality useful to the attacker being present in the existing codebase. RILC, as the name suggests,

[2] There is a caveat on the Windows platform as it is mostly POSIX-compliant, but not fully POSIX-compliant. This distinction is explored in greater depth in Section 3.

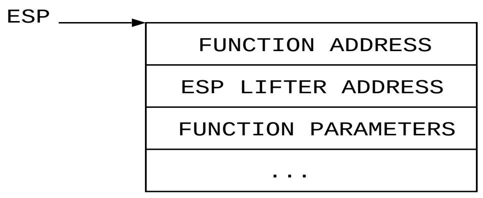


Fig. 1. Format of a malicious RILC stack frame. The `esp` lifter address corresponds to the function's return address, allowing sequential execution of functions.

leverages the vast catalog of functions present in the C standard library to fulfill this requirement, as libc is dynamically linked to all processes in UNIX-like environments.[3] Further, our threat model specifies that the vulnerable programs are protected via enforcement of code integrity (i.e., the ubiquitous W⊕X policy), negating the possibility of a direct code injection.

As mentioned above, executing a RILC attack requires the ability to overwrite the stack with arbitrary content via a buffer overflow, format string bug, or similar vulnerability. The content written to the stack is composed of valid (in regards to platform-specific calling conventions) but malicious function call frames that are specially crafted by the attacker in order to achieve an intended purpose. Once the stack has been populated with malicious content, the frame pointer (`esp`) must be redirected such that the next frame accessed is the first frame crafted by the attacker. There exist several methods by which this redirection can be achieved and the method often differs from one exploit to the next. The example exploits presented in this work leverage a `pop esp ; ret` sequence that exists as part of the function epilogue in the main method of a vulnerable application; thus, stack pointer redirection is as simple as injecting the address of the first malicious frame into the correct stack position.

As powerful as individual libc functions are, they are also highly specific; thus, using a single libc function limits an attacker to only the most basic of exploits. However, there are techniques available to chain multiple libc functions[1], [14], including one called *`esp` lifting* [1]. This method operates by using small instruction sequences to *glue* multiple functions (i.e., stack frames) together. In particular, these instruction sequences are composed of some number of `pop` instructions followed by a `ret`, which are rather common as they are used to implement standard C function epilogues. By inserting the memory location of such a sequence into the current stack frame's return address, an attacker can advance the stack pointer to the location of the next stack frame, thereby chaining multiple functions together. This method was proposed in 2001 [1] as an "advanced" RILC attack (at that time), which is being re-assessed in this paper for its expressiveness.

The format of a malicious stack frame is shown in Figure 1. The first item in the stack, located at the top of the frame, is the address of the function

[3] Note that Windows environments also support a variant of this attack through the Visual C++ Runtime (`msvcrt.dll`) and Windows core libraries (e.g., `user32.dll`), which are linked to most Windows applications.

to be executed. This is immediately followed by the address of an *`esp` lifting* instruction sequence, which acts as the return address of said function. In this way, the stack pointer can be immediately advanced to the next frame upon return from the previously called function. The final entries in the frame are the parameters to be passed to the function. Such a layout complies perfectly with the C standard for function frames while still allowing the attacker to maintain control of the exploit's execution.

The operation of ROP is in some ways similar to that of RILC. Most apparent is the use of the stack for program control. In addition, both paradigms utilize the concept of found code segments ("gadgets", in ROP parlance) in order to perform arbitrary computations; however, the length and location of these segments differ greatly between the two. Specifically, ROP typically utilizes small segments (often only a few instructions long) located arbitrarily in memory. These segments can be either code intentionally emitted by the compiler or, because instructions on the x86 are of variable length, unintended code sequences found by jumping to an offset that does not lie on an instruction boundary. On the other hand, RILC identifies segments solely by their intended definitions, namely as pre-defined functions. The `esp` lifter could be an additional requirement, but one that is trivially satisfied, due to the nature of the C calling convention. It is important to clarify that while there may be some similarities between the `esp` lifter and ROP gadgets, the former was published as *part of (advanced) RILC attacks* six years before ROP was even proposed, and was in use even earlier. Also, the former serves only the basic purpose of gluing multiple functions, *not* any particular functionality such as arithmetic or logic pursued in the latter. In this paper, our re-assessment of RILC's expressiveness only utilizes the ingredients behind traditional RILC attacks, which include various legitimate libc functions and the `esp` lifter.

RILC has been long noted in the past as being capable of executing only straight-line code, while ROP is capable of conditionally altering program flow (e.g., [2, 4]). As a result, RILC is generally considered as being incapable of fulfilling the requirements for Turing-completeness – a classification that severely limits its expressive power and capabilities. This work attempts to correct this misconception by providing proof of and methods for achieving Turing-completeness by utilizing commonly-available POSIX functions. By doing so, we can better understand the limits and capabilities of RILC and its comparison with ROP.

3 Turing-Complete RILC

In the traditional view, RILC is limited in its expressive power to perform arbitrary operations. For instance, in a pure RILC attack, parameter data of a function is static and needs to be pre-stored in stack before its execution. However, its return value is typically kept in `eax`, which makes it challenging to carry over the result of one libc function to another. Most importantly, during the execution of a RILC attack, stack frames are unwound in linear order, which makes it challenging to support conditional branching. Note that conditional branching is an essential operation for a system to be Turing complete.

In the development of TC-RILC, we have found a solution to the above challenges. Specifically, our solution is based on the observation that many functions have side-effects which may modify memory (including the stack) or system state. For the ease of presentation, we identify the functions whose side-effects are the result of useful computations and simply call these functions *widgets* (analogous to ROP's gadgets). To demonstrate the Turing completeness of RILC, we define a variety of essential classes of widgets that are needed to perform arbitrary computation, and show that such widgets are available in commonly deployed code (e.g., libc).[4] It is important to stress that widgets are literally entire functions, and that they are being exploited for their intended side-effects.

As our attack refines the traditional RILC, the structure of launching a TC-RILC is basically the same as in traditional RILC. That is, the injected buffer is comprised of malicious stack frames containing function entry points and parameters. However, one key difference is the specific functions that have been chosen and misused in a unique way that makes it possible to support arbitrary operations. Specifically, we find that widgets are available in commonly deployed code and can be efficiently misused to solve the two problems listed above. First, to achieve persistent data across function calls, we observe that widgets can be found that use pointers to read or write to locations within the attacker's stack. Therefore, these functions can "forward copy" the result of one widget into a future widget's input parameters (see Section 3.2). Additionally, functions whose inputs come via pointers or another method of indirection (e.g., environment variables) can also be used to side-step this problem. Second, to achieve conditional branching, we find a class of widgets capable of conditionally altering the program counter in RILC or the stack pointer (see Section 3.3).

We point out that other intended effects may not be useful (or harmful) to us. Among the intended effects, the returned value of a function – if any – is typically stored in a register (e.g. `eax`) and is discarded by design (our widgets cannot take registers as input). The other intended side effects in the form of memory changes may be needed for TC-RILC (e.g., `memcpy()`) or irrelevant as far as the effects do not change the memory content used in TC-RILC.

In the following, we categorize these widgets by their functional purpose. When presenting each widget category, we also report example functions found in libc, as specified in the POSIX standard.

3.1 Arithmetic and Logic

In this category, we consider any function as a candidate arithmetic and logic widget if the result of an arithmetic or logic operation is made available as a side-effect, i.e., written to memory as opposed to a register. In libc, the `wordexp()` function (specified in POSIX.1-2001 [15]) achieves this in a straightforward way. In essence, this function performs the expansion of expressions used by UNIX shells such as `bash`, and arithmetic is a natural component of that. It turns out that this functionality serves a number of purposes, including integer addition,

[4] These widgets are identified using manual analysis. See Section 5 for further discussion.

subtraction, multiplication, and division.[5] Of course, shell expansion is based on human-readable strings rather than binary arithmetic. Therefore, to leverage this functionality, we need to combine the string/integer conversion functions `itoa()` and `atoi()` (as well as the standard string-manipulation functions) to build input strings for `wordexp()`. This rather unorthodox approach allows us to perform arithmetic solely with side-effects, a requirement in constructing the TC-RILC attack. In addition to `wordexp()`, we can also make use of other pointer-driven arithmetic and logic functions, such as `sigandset()` and `sigorset()`, which flip numbered bits in an in-memory data structure.

In our development, we found that some of these functions (e.g., `wordexp()`) are not included in Windows environments. It turns out that the Microsoft Visual C++ Runtime only supports a subset of POSIX, which may probably explain why Windows is mostly POSIX-compliant, but *not* fully POSIX-compliant. Nevertheless, it still does not prevent us from locating other alternative functions in core Windows libraries. These core Windows libraries are loaded into almost all running Windows processes. For example, if we just consider one core Windows library – *user32.dll*, a quick examination of one co-author's Windows XP (SP3) desktop machine indicates that there are 74 running processes and 71 of them load this particular library in memory.[6] In addition, we have manually verified its presence in a variety of 32-bit Windows OSs we can install, ranging from Windows 95 all the way up to and including Windows 7. In our prototype, we simply choose from the functions or APIs defined in *user32.dll*[7] and use them to provide the arithmetic/logic operations as needed.

In particular, Windows provides a suite of functions to manipulate geometric shapes mathematically. For example, to perform the arithmetic addition or subtraction operation, we make use of the *OffsetRect()* function [16]. The intended use of this function is to move a specified rectangle (in the Cartesian coordinate system of the screen) by a certain offset along the X and Y axes. By making this function call, we can effectively cast an arbitrary memory area as four consecutive integers representing the top, left, bottom, and right coordinates of a rectangle data structure and then modify it by providing the corresponding offset. In other words, the intended operation of this function is exploited to perform addition or subtraction. For simplicity in building our exploits, multiplication is achieved by leveraging addition operations. The loop operation requires branching support, which will be discussed later in this section. Multiplication and division can also be achieved with a function like `ScaleViewportExtEx()` from `gdi32.dll`, but this involves a measure of added complexity, as certain Windows-specific objects must be prepared first.

[5] The POSIX standard actually calls for a full compliment of logical and bitwise operations as well, but this does not appear to be implemented in our version of libc. However, this limitation does not hinder the TC-RILC technique.

[6] The three which do not load *user32.dll* are the pseudo-processes *System* and *System Idle Process*, as well as the session manager *smss.exe*.

[7] There are other core libraries in Windows (e.g., *kernel32.dll*, *ntdll.dll*, *gdi32.dll*, and *shell32.dll*) that are loaded in almost every running process and can also be potentially (mis)used for the same purpose.

3.2 Memory Accesses

Arbitrary access to memory in a RILC attack is as simple as employing any function which performs a memory copy. These functions can be used to move data into and out of the RILC stack area. For this, libc provides us with a myriad of choices: `memcpy()`, `strcpy()`, etc. These functions are especially important in the context of TC-RILC, as they form the key to preserving data between calls. Additionally, one can make use of more esoteric data storage mechanisms, such as environment variables, which are automatically expanded by some functions, including `wordexp()`. When a widget executes, the only results useful to the attacker are side-effects. In order for the side-effect to be used as an input to a later widget, an intervening memory access widget copies this result into a future stack frame (or into a location referenced by a pointer in a future stack frame). The end result is a data model where variables in the TC-RILC program do not occupy a single place in memory, but rather are copied into places (or carried over) just in time for their next use.

3.3 Branching

Branching, especially conditional branching, is the practice of altering the flow of execution. In our context of launching a TC-RILC attack, this does *not* mean simply altering the CPU's instruction pointer `eip`. Rather, one must alter the stack pointer `esp`, which serves as the de-facto virtual program counter. This is a crucial ingredient to Turing complete computation, and has been long thought to be impossible in a RILC attack. Our solution to this problem has two steps.

First, to perform an unconditional branch, we identify any widget which explicitly alters the stack pointer. The C89 and POSIX standards define such a function: `longjmp()`. The intended use of `longjmp()` is to support non-local gotos [17], and is commonly used in threading libraries and error handlers. For the attacker, however, `longjmp()` represents a convenient means to alter much of the CPU state in a single call, including the stack and base pointers (`esp` and `ebp`). This allows for unconditional branching within the RILC attack.

Next, to make this branch conditional, a pointer to the `longjmp()` function can be provided as a parameter to another function which will execute the pointer conditionally. A convenient choice for this role is the `lfind()` function (defined in the POSIX standard [15] and supported in Windows). This function is intended to help with linear searches through an array, and has the form:

```
lfind(void*key,void*base,size_t*nmemb,size_t size,int(*compar)(void*,void*))
```

Normally, this function would walk through the array starting at `base`, calling `compar()` with the given key and each iterated element. In TC-RILC, we instead set `compar` to `longjmp` and `key` to the address of an attacker-supplied `jmp_buf` structure (which includes values for a number of registers, including `esp` and `eip`). The `nmemb` parameter is the conditional variable: `longjmp()` is called if and only if this is non-zero. If it is called, execution of `lfind()` ends and both `eip` and `esp` are rewritten with new attacker-supplied values. In addition to `lfind()`, we have also identified that `lsearch()` can be used for the same purpose. From

Table 1. A subset of POSIX-compliant widgets used in TC-RILC

Category	**Widgets**	**POSIX?**
Branching	`lfind()`+`longjmp()`, `lsearch()`+`longjmp()`	Yes
Arithmetic/Logic	`wordexp()`, `sigandset()`, `sigorset()`	Yes[8]
Memory access	`memcpy()`, `strcpy()`, `sprintf()`, `sscanf()`, etc.	Yes
System calls	Usual functions: `open()`, `close()`, `read()`, `write()`, etc.	Yes

this building block, it is relatively straightforward to implement regular control flow primitives like `if()` and `for()`. Note some widgets may destroy the content of the stack frame below the current `esp`. To guard against this situation, special care is taken to backup the whole content of the stack frame that contains the rest of the TC-RILC program before entering the loop. At the beginning of each loop iteration, the content of the stack frame is restored. This functionality successfully preserves the content and allows the creation of arbitrary control flow branching, which makes TC-RILC possible.

3.4 System Calls

While not strictly necessary for Turing complete computation, almost all useful attacks will need to make use of system calls. This is straightforward in a RILC attack, as library functions can be employed just as they would in a user program. For example, for file input/output, the attack can simply make use of the `open()`, `close()`, `read()`, and `write()` functions as normal.

We stress that unlike the machine-code-based gadget scan used in ROP, the discovery of widgets in TC-RILC is much more straightforward. Because the attack depends only on the intended side-effects of existing functions, the attacker needs only consult the code's documentation to locate the necessary functions. To maximize compatibility of the attack to multiple platforms, we primarily use functions from the well-documented and widely-deployed POSIX standard [15]. For the Windows port, as it is not fully POSIX-compliant, we first attempt to use supported POSIX functions. Only when they are not supported, will we fall back to standard Windows APIs provided in core libraries that are loaded in almost every running process. It is important to note that these core libraries typically maintain consistent API interfaces across Windows variants, which contribute to the compatibility of the proposed TC-RILC attack. In Table 1, we show an incomplete list of widgets that are used in our implementation. Our prototyping experience indicates that all functions except in the arithmetic/logic category are actually supported in Windows. This means that specific TC-RILC attack code can be readily ported to a different OS revision or even a different OS family altogether (as long as the environment supports the same standards). The

[8] The arithmetic/logic functions are not portable to Windows due to its lack of full POSIX compliance. Instead, we choose standard, cross-version Windows APIs in core Windows libraries to compensate for its limited POSIX support. Examples include `OffsetRect()`, `CopyRect()`, and `SetRect()` in *user32.dll* to provide the TC-RILC arithmetic operation. The functions in other categories are all supported in both Windows and Linux. Also note that `sigandset()` and `sigorset()` are Glibc extensions and are not part of POSIX standard.

```
#include <stdlib.h>
#include <string.h>

int main( int argc, char** argv ) {
        char buf[2048];
        strcpy( buf, argv[1] );
}
```

Fig. 2. The vulnerable application used to launch the example attacks.

changes needed are the adjustment of function entry points and, if necessary, the format of the `jmp_buf` data structure. This is in contrast to the ROP model, which requires analysis of individual binaries in order to locate and assemble specific snippets of machine code.

4 Implementation and Evaluation

To demonstrate the expressive power of the TC-RILC technique, we have developed two example stack-based buffer overflow attacks. The payload of the first attack is a RILC-based implementation of a universal Turing machine simulator while the payload of the second implements the selection sort algorithm. These two attacks were developed and tested on the 32-bit x86 version of Debian Linux 5.0.4, and solely used POSIX-compliant functions within the included libc binary.[9] After that, we also ported the attack technique to Windows in a straightforward manner. Our Windows platform runs Windows XP (with service pack 3), and the vulnerable application was compiled with Microsoft Visual C++ 6. The library functions employed were found in the standard runtime library for Visual C++ programs[10], and one core Windows library[11].

In both environments, the vulnerable program that was exploited to launch the attacks is given in Figure 2. In this program, the first command line argument is copied into a fixed-size stack buffer by `strcpy()`. Because this is done without bounds checking, an excessively long argument can overflow the return address of the `main` stack frame. This straightforward vulnerability allows an attacker to inject the RILC program into memory and hijack control flow in one step.

During our development, we experienced that both Linux and Windows have features intended to protect `longjmp()` from malicious exploitation. For example, in Linux, the values stored for `eip` and `esp` in the `jmp_buf` structure are rotated several bits and xored by a known value in order to "mangle" them, i.e., adjust them in a way unknown to the attacker. Unfortunately, this protection is not fully implemented, as the known value is currently a hard-coded constant instead of a per-process random value. Windows instead protects the `jmp_buf` structure by including a special "cookie" value within it. In theory, this would prevent the attacker from overwriting the structure, but this protection is flawed in a way similar to Linux: this value is a hard-coded constant. Therefore, these protection features do not prevent TC-RILC from being launched (as simple hacks involving `longjmp()` remain viable on both platforms).

[9] `/lib/i686/cmov/libc-2.7.so`, MD5 checksum: e4e7e3c6b4f1be983e00c0daafc3aaf3.
[10] `msvcrt.dll`, MD5 checksum: 355edbb4d412b01f1740c17e3f50fa00.
[11] `user32.dll`, MD5 checksum: b26b135ff1b9f60c9388b4a7d16f600b.

4.1 Universal Turing Machine Simulator

The term "Turing complete" is generally used as shorthand to indicate the capability for arbitrary computation. The set of Turing complete systems are equivalent in expressive power, and such systems are said to be *universal computers*. There are many ways to demonstrate a system is Turing complete. In this work, we opt for the most straightforward approach – a Turing machine simulator.

A Turing machine is a computer consisting of a tape `T` with a movable read-write head, an internal state register `Q`, and a fixed state transition table `A`. At each interval, the machine reads the current symbol, and, based on that symbol and the current internal state, updates the symbol, changes the state, and possibly moves the head one step left or right. This behavior is governed by the transition table, which constitutes the Turing machine's "program". A system which can simulate this behavior for an arbitrary tape and transition table is called a *universal Turing machine*.

We have developed a TC-RILC exploit that acts as a universal Turing machine, demonstrating the expressiveness of our technique. Instead of delving into the complexity and details of the binary form of this attack code, we choose to present an abstracted representation of our POSIX-based variant in Figure 3. In this figure, the memory state is shown in Figure 3(a). Each definition here indicates a pointer to a piece of attacker-controlled memory. These definitions are commented inline. We would like to draw special attention to the `jmp_buf` structure `jb`. This structure is crafted by the attacker so that, when passed to `longjmp()`, the CPU stack pointer will be redirected to the top of the main loop.

The string of malicious stack frames that make up the TC-RILC program itself is shown in Figure 3(b). For clarity, each stack frame is indicated with a line of C-like code. To better understand this particular exploit, we need to explain the mechanism that is used to store persistent data between function calls. Specifically, our mechanism relies on the use of environment variables and thus alleviates the need to rebuild the equation strings during each iteration of the Turing machine run. As indicated in Figure 3(b), the exploit uses specially-crafted strings of the form "`VARIABLE=VALUE`" that are updated with a new `VALUE` before being added to the environment via a call to `putenv()`. In addition, `wordexp()` caps the result of any arithmetic operation at 0x7fffffff – presumably in an attempt to avoid the ambiguity encountered when representing signed versus unsigned numbers. For this reason, all memory offsets are computed by referencing only the lower two addressable bytes in equation strings, then copying the result of the arithmetic operation into the lower half of a pointer which already refers to the stack region. For example, consider that the exploit is known to reside at a location spanned by addresses of the form 0xbfffXXXX. To successfully compute a memory offset, we must populate a known memory location with a value of this form, then copy the result of any arithmetic operation containing a memory address into the XXXX portion of this location. The resulting value can then be used as a pointer to the desired memory location.

The exploit therefore begins by initializing the environment with the variables `I` (the offset into the tape) and `Q` (the current state). Once these variables are

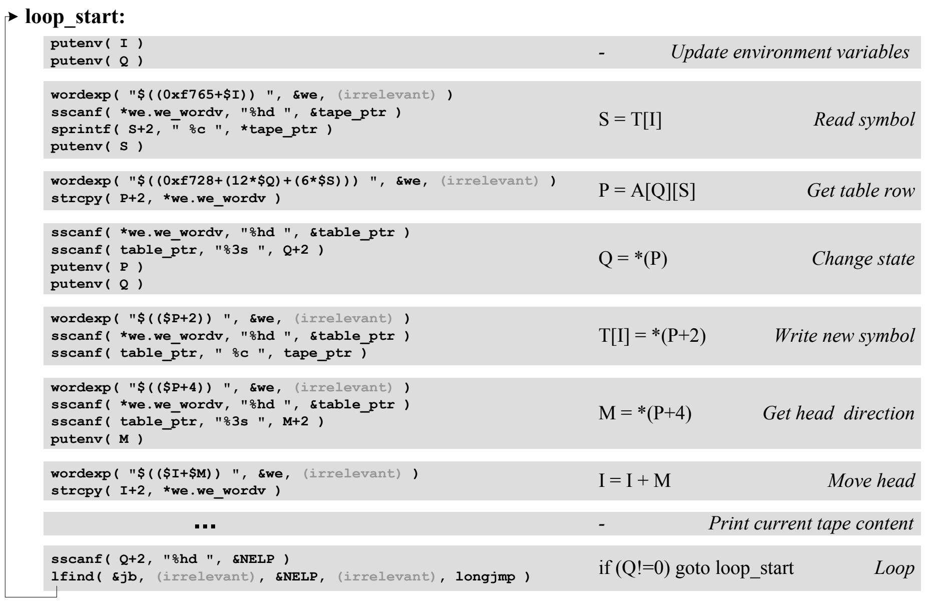
```
int        NELP      // Integer used with lfind
jmp_buf    jb = ...  // When passed to longjmp, the stack pointer will be moved back to loop_start
wordexp_t  we        // Result of a wordexp() operation

char* tape_ptr  = 0xbfffffff  // Pointer into tape T, lower two bytes will be adjusted
char* table_ptr = 0xbfffffff  // Pointer into table A, lower two bytes will be adjusted

char* T = "000000000000000000"  // Tape -- each byte is one symbol
char* A = " 0 0 0"              // State transition table.  Rows indexed by the state Q then symbol S.
          " 0 0 0"              //   Each row gives:
          " 2 1 1"              //     1. The new state Q
          " 2 1-1"              //     2. The symbol to be written
          " 1 1-1"              //     3. The direction to move the head pointer (-1, 0, or 1)
          " 3 0-1"              //   This particular table is a 4-state 2-symbol busy beaver.
          " 0 1 1"
          " 4 1-1"
          " 4 1 1"
          " 1 0 1"

char* I = "I=0xC "   // Current index into tape T
char* S = "S=0x0 "   // Symbol just read from the tape
char* Q = "Q=0x1 "   // Current state
char* P = "P=0x000 " // Lower 2 bytes of a pointer to a row within table A
char* M = "M=0x0 "   // Direction for head movement
```

(a)



`putenv( I )`
`putenv( Q )`
	-	*Update environment variables*

`wordexp( "$((0xf765+$I)) ", &we, (irrelevant) )`
`sscanf( *we.we_wordv, "%hd ", &tape_ptr )`
`sprintf( S+2, " %c ", *tape_ptr )`
`putenv( S )`
	S = T[I]	*Read symbol*

`wordexp( "$((0xf728+(12*$Q)+(6*$S))) ", &we, (irrelevant) )`
`strcpy( P+2, *we.we_wordv )`
	P = A[Q][S]	*Get table row*

`sscanf( *we.we_wordv, "%hd ", &table_ptr )`
`sscanf( table_ptr, "%3s ", Q+2 )`
`putenv( P )`
`putenv( Q )`
	Q = *(P)	*Change state*

`wordexp( "$(($P+2)) ", &we, (irrelevant) )`
`sscanf( *we.we_wordv, "%hd ", &table_ptr )`
`sscanf( table_ptr, " %c ", tape_ptr )`
	T[I] = *(P+2)	*Write new symbol*

`wordexp( "$(($P+4)) ", &we, (irrelevant) )`
`sscanf( *we.we_wordv, "%hd ", &table_ptr )`
`sscanf( table_ptr, "%3s ", M+2 )`
`putenv( M )`
	M = *(P+4)	*Get head direction*

`wordexp( "$(($I+$M)) ", &we, (irrelevant) )`
`strcpy( I+2, *we.we_wordv )`
	I = I + M	*Move head*

...	-	*Print current tape content*

`sscanf( Q+2, "%hd ", &NELP )`
`lfind( &jb, (irrelevant), &NELP, (irrelevant), longjmp )`
	if (Q!=0) goto loop_start	*Loop*

(b)

Fig. 3. A visual representation of the universal Turing machine simulator attack code. The attacker-controlled static memory is shown in (a). The tape `T` and table `A` constitute the program, while the environment variables `I`, `S`, `Q`, `P`, and `M` are used with `wordexp()` to do the bulk of the arithmetic and logic. When pointer indirection is needed, the lower two bytes are calculated by `wordexp()`, then converted to binary and written into the pointer variables `tape_ptr` and `table_ptr`. The stack frames are represented in (b) using a C-like notation where each line corresponds to an attacker-crafted stack frame. The frames are grouped by logical operation; within each group is its symbolic representation and description. This is the POSIX-based variant of the attack.

in place, we begin computing the locations of the elements needed to advance the Turing machine. Specifically, we determine the memory location and value of the current tape symbol `S`, then utilize this in conjunction with the current state `Q` to determine the location `P` of the relevant row in the state-transition table `A`. Given this memory location, advancing the machine is simply a matter of adding the correct offset to `P` in order to read the new symbol, state, and head movement direction `M`. Finally, we advance the head position `I` by `M`. Once these operations have been completed, the machine is ready to execute its next step. We use the value of the new state `Q` to determine whether or not the machine needs to continue. Recall that our approach to conditional branching makes use of a unique `lfind()`+`longjmp()` combination, and utilizes the `nmemb` parameter as its conditional value—specifically, the branch is taken only if `nmemb` is non-zero. In our Turing machine example, the final state is indicated by `Q`=0; thus, we can determine whether or not to continue looping by simply copying the value of the current state `Q` into the conditional parameter value.

To validate the correctness of our implementation, we configured the exploit to simulate a *busy beaver*—a special Turing machine that performs the greatest number of steps possible before halting [18]. Specifically, we simulate a 4-state 2-symbol busy beaver. In this exploit, there are in total 24 widgets used for the TC-RILC implementation of the busy beaver Turing machine. We also implement a Windows-variant of the same attack. The key difference from the POSIX-variant turns out to be the replacement of *wordexp()* (in Figure 3) with a few core Windows API functions. As mentioned earlier, though *wordexp()* is a POSIX function, it is unfortunately not supported in Windows. As such, we fall back on documented Windows APIs to emulate part of its functionality as needed for our busy beaver Turning machine implementation. In particular, our prototype makes use of two Windows API functions: *SetRect()* and *OffsetRect()*. The *OffsetRect()* function is used to implement addition in a straightforward manner with its effect similar to one simple C statement `A += B`. For multiplication, we achieve the same effect by controlling the number of loops (via `lfind()`+`longjmp()`) on an addition operation. More specifically, we use *SetRect()* to initialize a rectangle data structure which contains the value we want to multiply in a member field called `left`. Then, in the body of the loop, we repeatedly add to that field by using *OffsetRect()*. At the end of the loop, the `left` field will contain the multiplication result. In our current prototype, there was no need to develop support for division and complex logic operations. However, such support could be developed using the wealth of other Windows functions, including `ScaleViewportExtEx()` from `gdi32.dll`. In total, our Windows-variant of the busy beaver Turing machine contains 29 widgets. We have confirmed the successful run of the busy beaver program written entirely with library functions in various Windows systems, including Windows 95/98/2000/XP/Vista/7. The full detail can be found in [19].

4.2 Selection Sort

While the previous example is sufficient to demonstrate Turing completeness in theory, a Turing machine is not a very convenient model for practical computation. Therefore, to demonstrate the practicality of the technique, we also present a TC-RILC exploit that implements the *selection sort* algorithm,

The algorithm is basically implemented with two *for*-loops. The inner loop finds the minimum item by examining each one in the array. In the outer loop, each iteration exchanges the found minimum item with the first one so that subsequent iterations can exclude the first one to proceed with sorting. In other words, after the m-th iteration (of the outer loop), the array is divided into two parts: the first part contains the leftmost m items of the array, which is sorted while the remainder constitutes the second part, which is not sorted.

Just as a compiler can analyze this code and produce a series of primitive arithmetic, logic, and control flow machine instructions, we have been able to map the algorithm to a sequence of TC-RILC widgets. (The abstracted representation for the POSIX variant can be found in [19].) Specifically, we have two similar *for*-loops. The outer loop is the main loop, which will sort the first m items after m iterations. The inner loop instead is responsible for finding the minimum item in the array. Each loop, either outer or inner, needs to properly perform conditional control flow, which is fulfilled with the `lfind()`+`longjmp()` combination. In our exploit, we also apply several other techniques used in our universal Turing machine simulator (i.e., `wordexp()` for arithmetic operations and `sscanf()` for data movement). In total, there are 24 widgets used in the outer loop and 14 widgets used in the inner loop. The end result is a code-reuse exploit that can hijack our simple example program and sort an in-memory array.

We point out that implementing selection sort is not an end in and of itself, but it demonstrates the feasibility of TC-RILC: one can similarly craft complex, expressive attack codes by chaining entire functions to launch a TC-RILC attack.

5 Discussion

We have shown that the traditional return-into-libc attack, previously considered to be limited to straight-line code, is actually Turing complete. Given this, it is interesting to examine the reason why the traditional view of RILC attacks fails to properly recognize its expressive power and revisit the comparison between TC-RILC and ROP as they are now equivalent in expressive power.

Analyzing the commonly-held misconception The RILC attacks have been known for more than a decade [20]. However, the reason why we still suffer from this misconception is in part attributed to the lack of thorough understanding of the side-effects of legitimate C library functions. Specifically, we may have been used to providing normal input arguments to these functions and do not give careful and thorough consideration to all possible inputs. For example, the previously claimed incapability of RILC to perform conditional branching can be overcome by exploiting the side-effects of combining normal libc functions, i.e., `lfind()`+`longjmp()` or `lsearch()`+`longjmp()`. Also, the presence of a wealth of libc functions greatly facilitates the selection, construction, and integration of a variety of functional components in TC-RILC computation, including arithmetic/logic, data movement, memory access, and system calls. Moreover, thanks to the POSIX standard, the C library maintains a well-defined, consistent interface across various OS variants and families. This interface not only significantly contributes to the portability and compatibility of legitimate user programs, but also equally helps the portability and compatibility of developed TC-RILC attack code. For example, our Windows variant of the busy beaver Turing machine can run on all 32-bit Windows OSs from Windows 95 to Windows 7. Also, our experience indicates the cross-OS port is rather straightforward provided that they support the POSIX standard. The only limitation we have encountered so far is due to the lack of full POSIX-compliance in Windows.

From another perspective, one interesting open question is the issue of cross-architecture portability. We have shown that the technique can be used on different operating systems on the x86 32-bit architecture, but it is not clear yet how to carry the model to other CPU ISAs, especially RISC platforms. Our technique depends on the calling convention in use, which is influenced by the CPU architecture. For instance, the MIPS architecture passes most function parameters via registers rather than the stack, so applying TC-RILC in such an environment seems problematic. This remains an interesting problem which we leave to future work.

Revisiting the comparison with ROP Arguably due to its capability to perform arbitrary computation (in which traditional RILC was thought to be limited), ROP has recently attracted significant attention and development [2, 7–9, 11]. With this limitation in traditional RILC attacks removed, there is a need to re-assess the comparison between the two techniques.

As mentioned earlier, TC-RILC has several advantages. First, because it uses the intended behavior of functions to operate, attacks can be ported to different implementations by accordingly changing the function offsets and the format of data; this is true even between very different environments, provided that they support the same library functions. It is interesting to mention that most existing work on code-reuse attacks makes a probabilistic argument: if enough code is present, then it is likely that one can find enough code snippets to construct a Turing complete computation. In this work, however, we make a more concrete claim: because we rely mainly on the intended behavior of standardized functions, the TC-RILC technique is applicable to any standards-compliant OS environment. Second, because these functions are necessary for the normal operation of existing software[12], they cannot be simply taken away. This is in contrast with ROP, where the attacker is at the mercy of the specific machine instructions available in the binary. Also, a TC-RILC attack requires less information about the library than ROP: TC-RILC needs the locations of used library functions[13], whereas ROP requires an in-depth scan of the binary for useful instruction sequences. Third, certain existing anti-ROP defenses, i.e., DROP

[12] Our TC-RILC mainly relies on POSIX functions and does not utilize any "dangerous" functions such as `system()`, which may be removed by some security measures.

[13] They can be legitimately obtained by making certain library function calls. Examples include `dlsym()` in Linux and `GetProcAddress()` in Windows. Strictly speaking, traditional RILC attack also requires the location of `esp` lifting instructions. However, they can be replaced with the frame faking technique [1]. Possibilities also exist with the side-effects from misused library functions, e.g., `longjmp()`.

[12] and DynIMA [4] are defeated by the TC-RILC technique. These techniques observe the frequency or the presence of `ret` instructions and exploiting the fact that ROP gadgets are typically 2-5 instructions in length. Though these defenses are rendered ineffective by the recent ROP refinement [21], with the use of entire functions, TC-RILC is naturally immune from these defenses.

On the other hand, TC-RILC does have some disadvantages. First, a TC-RILC attack may require more stack space than an equivalent ROP attack. This distinction could be important when the vulnerability only permits overflows of a limited size. Second, our experience indicates that attacks based on TC-RILC could be more complex to construct *manually* than ROP attacks. This is primarily because of the complexity of storing data and operating control flow entirely through side-effects. In contrast, ROP programs can leverage the CPU registers to save state, and access memory only as needed. However, this complexity could be effectively reduced or even eliminated by developing a RILC-aware compiler, leveraging the same algorithms and techniques that produce ROP attacks. Third, while performance is not the primary aim of a TC-RILC attack, it is intrinsically computationally less efficient, especially when compared to native program execution.

To measure its computation overhead, we adapted our Turing machine example to compute a 5-state, 2-symbol busy beaver candidate, which runs for 47,176,870 steps, making it a much more computationally intensive program than our earlier example. For comparison, we developed a straightforward Turing machine simulator based on the same algorithm in both Python and C. The C version, which we use as a baseline, finished in 0.19 seconds, while the Python version took 42.75 seconds (225 times slower). The TC-RILC execution took 419.38 seconds, and is therefore over 2000 times slower than the C implementation. Such an overhead is to be expected, as the exploit is rife with memory copy and string processing operations which are unnecessary in a normal program.

It is also interesting to explore some possible defense mechanisms to counter TC-RILC attacks. At first impression, one may feel that removing vulnerable functions from applications that do not need them might defeat TC-RILC. This is indeed a generic approach to defend against traditional RILC attacks, though in practice, it may be challenging to deploy. Particularly, there are several difficulties: First, how do we know in advance those functions an application is going to use or not going to use? Second, this approach will not work if it turns out that the application itself needs those functions behind TC-RILC. One may also attempt to hinder TC-RILC attacks by trying to improve the protection mechanism used in `longjmp()` function. It is an open question, however, as to how effective it will remain when the attacker can almost always reverse-engineer the new mechanism and devise a method to craft the related `jmp_buf`.

From the attacker's perspective, there are several possible ways to improve TC-RILC attacks. One possible approach would be to extend the widget catalog. In the interest of time, our current prototype does not explore other libraries to find "abusable" widgets – as the current `longjmp()` and others are sufficient to demonstrate that RILC is indeed Turing-complete. Given the amount and size of installed libraries in a typical system (especially Windows), we strongly believe that similar functions could be found. It is also worthwhile to point out that in our current prototype, finding widgets requires manual analysis. More engineering effort will be needed to develop a scanner to harvest widgets from function specifications (e.g. header files).

We want to stress that, like ROP, TC-RILC is susceptible to some existing defense techniques. To be clear, the goal of this paper is not to cast TC-RILC as a threat without peer, but rather to reveal the unexpected fact that the RILC technique is more expressive and flexible than previously thought. The defenses available against this and other code-reuse attacks are explored and summarized in the following section.

6 Related Work

The original return-into-libc (RILC) attack was formalized as early as 1997, when Solar Designer introduced a single-call exploit which redirected control flow into the `system()` function of libc in order to launch a shell [20]. This technique was subsequently expanded to include multi-function chaining through the use of *`esp` lifters* and other techniques in 2001 [1]. This introduced the RILC technique as a mechanism for straight-line, chained execution of functions. Not satisfied with the limited expressive power that RILC was assumed to have, Shacham et al. put forth the notion of return-oriented programming (ROP) [2]. By arranging and chaining the execution of short code sequences ("gadgets"), ROP has been shown to be Turing complete. ROP was first introduced for the x86 and subsequently expanded to other architectures, including SPARC [7], ARM [8], and others. Further, Hund et al. presented a return-oriented rootkit for the Windows operating system that bypasses kernel integrity protections [9]. Castelluccia et al. similarly presented a ROP-based rootkit, but deployed it on embedded devices to attack existing software-based attestation techniques [10]. Checkoway et al. showed the feasibility of a ROP-based attack against electronic voting machines [11].

ROP attacks exhibit several peculiarities in their control flow and use of the stack; these features have been used to develop defenses against the ROP technique. For instance, ROPDefender [22] rewrites existing binaries to record a separate shadow stack which is used to verify that each return address is valid; this prevents return-based attacks, including both ROP and RILC. Other systems also make use of a shadow stack, either in hardware or software, and can be used to similarly enforce stack integrity [23–25].

Another interesting trait of ROP attacks is their reliance on gadgets—typically only 2 to 5 instructions in length. This means that the frequency of the `ret` instruction during the execution of a ROP attack is abnormally high. Capitalizing on this insight, DROP [12] and DynIMA [4] can detect a ROP-based attack. Because a TC-RILC attack makes use of whole function widgets, it does not exhibit this anomaly and is therefore indistinguishable from normal program execution to these defense schemes. From another perspective, the return-less approach [13] and G-Free [6] prevent return-oriented gadgets from being located or assembled. However, they only de-generalize the ROP back to (and will not block) the traditional RILC as they still provide the same function-level semantics.

Continuing the arm race between attackers and defenders, various forms of return-free code-reuses have been introduced. Checkoway et al. chain code snippets ending in a `pop/jmp` sequences to achieve arbitrary computation with ROP-like semantics [26]. Bletsch et al. introduce the concept of jump-oriented programming, which leverages indirect jump sequences instead of `ret` instructions to govern control flow [5]. Finally, Davi shows a jump-based attack on ARM is possible by using a special Branch-Load-Exchange (BLX) instruction [27].[14]

Other Defenses In addition to defenses that specifically target ROP, there are orthogonal defense schemes that protect against a variety of machine-level attacks. Address-space layout randomization (ASLR) randomizes the memory layout of a running program, making it difficult to determine the addresses in libc and other legitimate code on which code-reuse attacks rely [28, 29]. However, there are several attacks which can bypass or seriously limit ASLR, especially on the 32-bit x86 architecture [1]. Additionally, ASLR can be defeated by leakage of sensitive information about the memory layout of the process [30]. Therefore, while ASLR is certainly useful, it is not a silver bullet to the problem of code-reuse attacks. Instruction-set randomization (ISR) is another attempt at introducing artificial heterogeneity into program memory [31, 32]. Instead of randomizing address-space, ISR randomizes the instruction set for each running process so that instructions in the injected attack code fail to execute correctly. However, it is ineffective against code-reuse attacks, including ROP and RILC.

Many mechanisms have been proposed to enforce the integrity of memory. Program shepherding is a technique to allow the application of security policy to control flow transfers [33]. Abadi et al. introduce the notion of Control Flow Integrity (CFI), which seeks to ensure that execution only passes through approved paths taken from the software's control flow graph [34]. Subsequent work expanded on the notion of CFI to allow for other security features, such as Data Flow Integrity (DFI) [35]. If CFI is properly enforced, most, if not all, code-reuse attacks will be prevented. Unfortunately, systems which enforce CFI are not widely deployed, presumably due to issues of overhead and complexity.

7 Conclusion

Return-into-libc (RILC) is one of the most common forms of code-reuse technique, but has been long considered to be incapable of arbitrary computation. In this paper, we present the counterargument that, by chaining existing functions in unique ways, RILC can be made Turing complete. Specifically, we demonstrate that the generalized TC-RILC attack satisfies the formal requirements of Turing completeness. Moreover, by relying mainly on the well-defined semantics of libc functions, TC-RILC attacks are portable across OS variants and families and can also bypass some recent anti-code-reuse defenses that target the return-oriented programming technique. Our prototype development on both Linux and Windows demonstrates the expressiveness and practicality of this technique.

[14] Note that systems described separately in [26] and [27] are now merged [21].

Acknowledgments The authors would like to thank our shepherd, William Robertson, and the anonymous reviewers for their numerous, insightful comments that greatly helped improve the presentation of this paper. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), the US Air Force Office of Scientific Research (AFOSR) under Contract FA9550-10-1-0099, and the US National Science Foundation (NSF) under Grants 0852131, 0855297, 0855036, 0910767, and 0952640. 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 ARO, the AFOSR, and the NSF.

References

1. Nergal. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14*, 2001.
2. Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *14th ACM CCS*, 2007.
3. Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications, 2009. *http://cseweb.ucsd.edu/ hovav/dist/rop.pdf*.
4. Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense against Return-oriented Programming Attacks. In *4th ACM STC*, 2009.
5. Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *CSC-TR-2010-8, Department of Computer Science, NC State University*, April 2010.
6. Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating Return-Oriented Programming Through Gadget-less Binaries. In *26th ACSAC*, 2010.
7. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *15th ACM CCS*, 2008.
8. Tim Kornau. *Return-Oriented Programming for the ARM Architecture*. Master's thesis, Ruhr-Universität Bochum, January 2010.
9. Ralf Hund, Thorsten Holz, and Felix C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *19th USENIX Security Symposium*, August 2009.
10. Daniele Perito Claude Castelluccia, Aurélien Francillon and Claudio Soriente. On the Difficulty of Software-Based Attestation of Embedded Devices. In *16th ACM CCS*, New York, NY, USA, 2009. ACM.
11. Stephen Checkoway, Ariel J. Feldman, Brian Kantor, J. Alex Halderman, Edward W. Felten, and Hovav Shacham. Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. In *Proceedings of EVT/WOTE 2009*. USENIX/ACCURATE/IAVoSS, August 2009.
12. Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. DROP: Detecting Return-Oriented Programming Malicious Code. In *5th ACM ICISS*, 2009.

13. Jinku Li, Zhi Wang, Xuxian Jiang, Mike Grace, and Sina Bahram. Defeating Return-Oriented Rootkits with Return-less Kernels. In *5th ACM EuroSys*, 2010.
14. Dino Dai Zovi. Return-Oriented Exploitation. *Black Hat*, 2010.
15. The Austin Group. The Single UNIX Specification, Version 3 (POSIX-2001).
16. Microsoft MSDN. http://msdn.microsoft.com/en-us/library/dd1627462010.
17. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.
18. Busy Beaver. http://en.wikipedia.org/wiki/Busy_beaver.
19. Minh Tran and Mark Etheridge and Tyler Bletsch and Xuxian Jiang and Vincent Freeh and Peng Ning. On the Expressiveness of Return-into-libc Attacks. In *CSC-TR-2011-16, Department of Computer Science, NC State University*, June 2011.
20. Solar Designer. Getting Around Non-executable Stack (and Fix). *Bugtraq*, 1997.
21. Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *17th ACM CCS*, October 2010.
22. Lucas Davi, Ahmad-Reza Sadeghi, and Marcel 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.
23. Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *21st IEEE ICDCS*, April 2001.
24. Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *10th USENIX Security Symposium*, 2001.
25. Vendicator. Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux. *http://www.angelfire.com/sk/stackshield/info.html*.
26. Stephen Checkoway and Hovav Shacham. Escape from Return-Oriented Programming: Return-Oriented Programming without Returns (on the x86), February 2010. *http://cseweb.ucsd.edu/ hovav/dist/noret.pdf*.
27. Lucas Davi, Alexandra Dmitrienkoy, Ahmad-Reza Sadeghi, and Marcel Winandy. Return-Oriented Programming without Returns on ARM. In *Technical Report HGI-TR-2010-002, Ruhr University Bochum, Germany*, 2010.
28. PaX ASLR Documentation. http://pax.grsecurity.net/docs/aslr.txt.
29. S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *14th USENIX Security*, 2005.
30. G. F. Roglia, L. Martignoni, R. Paleari, , and D. Bruschi. Surgically Returning to Randomized Lib(c). In *25th ACSAC*, 2009.
31. Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *10th ACM CCS*, 2003.
32. Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. *10th ACM CCS*, 2003.
33. Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution Via Program Shepherding. In *11th USENIX Security Symposium*, August 2002.
34. Martín Abadi, Mihai Budiu, Úlfar Erilingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. In *12th ACM CCS*, 2005.
35. Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-Flow Integrity. In *7th USENIX OSDI*, November 2006.