Exploring Jump-Oriented-Programming in RISC-V

Daniel Starikov
Univeristy of Washington

Keanu Vestil
University of Washington

Abstract

This paper explores the viability of jump-oriented-programming (JOP) on RISC-V, a new open-source ISA that is gaining popularity in many different settings. We add to the code reuse attack repertoire by extending variations of return-oriented-programming from their initial architectures to RISC-V. We analyze `libc` for real JOP gadgets that can be used for these attacks, and find versatile gadgets that allow us to trivially issue arbitrary library calls such as printing strings using `putchar` or launching a shell using `execve`.

1 Introduction

RISC-V is a free and open instruction set architecture (ISA) introduced by UC Berkeley in 2010 [21]. Although it was initially designed for research and education purposes, it is being increasingly used in commercial settings [13]. RISC-V is a based on the reduced instruction set computer (RISC) paradigm. This aims to improve a machine's throughput by simplifying instructions and addressing modes, thus reducing the cycles needed per instruction and reducing the time per cycle [14].

Code injection attacks involve an adversary supplying arbitrary code through an innocuous input in a program, and later hijacking control flow to execute this code. While code injection attacks can be prevented by measures such as ASLR [20] and W$\oplus$X [6], code *reuse* attacks can bypass these defenses and be just as effective. Return-oriented-programming is an example of a code reuse attack.

Return-oriented-programming, or ROP, involves the reuse of short instruction sequences that end with a return instruction [19]. These short sequences are referred to as gadgets. The capability of a gadget can vary, but its functionality by itself is usually quite limited. However, when they used together in a sequence, it has been shown that an attacker can execute arbitrary operations. The attacker supplies addresses to these gadgets, and other values that they may acquire from memory. These gadgets are then executed in order as though they are instructions for a strange computer built on the misuse of registers and code fragments.

There have been many variations of ROP presented since its initial discovery. These include ROP without returns [4], sigreturn-oriented-programming [2], jump-oriented programming [1], and string-oriented-programming [15]. These variations seek to bypass defenses that focus on certain aspects of one or more of their peers. While ROP was originally proposed for the x86 architecture, it has been demonstrated on many other CISC and even RISC architectures [3, 9]. The variations of ROP have also been demonstrated on alternate architectures.

While there has been some recent work demonstrating ROP on RISC-V [7, 8], there has not been an explicit exploration into jump-oriented-programming on these systems. In this work, we explore jump-oriented-programming on RISC-V and show that it can be used to make arbitrary calls to `libc`. Furthermore, we present a new recipe of gadgets that uses a dispatcher gadget followed by a versatile initializer gadget, and demonstrate how they can be chained together.

The remainder of this paper is organized as follows: Section 2 walks through our approach for re-implementing ROP in RISC-V and implementing a closely related variant. Next, Section 3 explains how we implemented jump-oriented programming, its limitations, and how we searched for our gadgets. Section 4 discusses our results in general, and what we are able to do with them. Section 5 presents related work that is tangential to ours. Finally, Section 6 concludes this paper and posits ideas for future work.

2 Implementing Variations of ROP

We thought it would be beneficial to recreate previously demonstrated ROP attacks so that we could familiarize ourselves with the process. We began by testing the RISC-V ROP attack described in Gu and Shacham [7]. We cloned their publicly available GitHub repository to our machine and successfully replicated their ROP attack on our RISC-V VM. To replicate their attack on our VM, we found the exact ver-

sion of `libc.so` that the authors used and cross-referenced it with the addresses from their compiler to find their exact gadgets. We then implemented these gadgets in an assembly file and modified their compiler to use the addresses of our gadgets. With this in place we were able to successfully execute brainfuck [7, 12] programs through ROP on our VMs. Next we sought to port a variant of ROP, which does not use returns [4], from x86 to RISC-V.

At first, we reused the gadgets that we implemented in the aforementioned plain ROP attack. Instead of loading the return address into `ra`, we modified the gadgets to load it into `a6`. The `ret` and `jalr ra` instructions at the end of each gadget were replaced with `c.jalr a6`. Since `a6` was otherwise unused in our gadgets, this functionality is effectively the same as normal behavior with the exception that the `ret` and `jalr ra` mnemonics are never used. This was a successful proof of concept, but it was not realistic because it relied on gadget modification that we manufactured.

Checkoway et al. described a "trampoline" gadget that could be reused to achieve this `ret`-like behavior [4]. We were able to successfully implement a simplified version of this attack using a real `ret` gadget which incremented the stack pointer and performed a `ret`, allowing us to reuse the JOP versions of ROP gadgets that relied on loading from the stack. We modified our gadgets ending with `c.jalr a6` to no longer modify the stack pointer. We then rewrote the brainfuck to ROP compiler from [7] to use our updated JOP gadgets and chained them together using the new `ret` gadget. Finally, we demonstrated that this could extend to a full ROP-without-returns attack by replacing the real `ret` gadget with one that we wrote ourselves which increments the stack pointer and ends with an indirect jump. This attack is also not realistic as it relies on the modified gadgets that use the stack and requires an unrealistic gadget that increments the stack before performing an indirect jump.

3 Implementing JOP

In this section we describe our methodology for crafting a jump-oriented-programming attack. In addition, we discuss its capabilities, limitations, and possible real-world applications. Our approach is inspired by the previous work on this topic by Bletsch et al. [1]. Their research was done on x86, so part of our work was to translate their results to work in RISC-V. In their findings, they describe a *dispatcher* gadget and an *initializer* gadget. A dispatcher gadget increments a register that is used as a pseudo-program counter and then jumps to the address stored at that next address. An initializer gadget loads many (or perhaps all) registers that the proceeding gadgets rely on. In x86, the `popa` instruction loads every general purpose register from the stack. There is no such luxury in RISC-V, but we found a gadget that achieves a similar effect.

There is a powerful initializer gadget in the `setcontext` function from `libc`. It loads almost all[1] general purpose registers from memory, based on the address in the `t0` register, and then it jumps to the address that it just loaded into `t1`. The disassembly of this initializer gadget is shown in Figure 1 The ability to load so many register from memory and to subsequently jump to an address that we can arbitrarily define makes this gadget quite versatile. However, before we can utilize it, we must set `t0` to point to our dispatch table (which contains all the values that we wish to load) in memory.

Figure 1: Initializer gadget

```
ld      t1,176(t0)
ld      ra,184(t0)
ld      sp,192(t0)
ld      s0,240(t0)
ld      s1,248(t0)
ld      a0,256(t0)
ld      a1,264(t0)
ld      a2,272(t0)
ld      a3,280(t0)
ld      a4,288(t0)
ld      a5,296(t0)
ld      a6,304(t0)
ld      a7,312(t0)
ld      s2,320(t0)
ld      s3,328(t0)
ld      s4,336(t0)
ld      s5,344(t0)
ld      s6,352(t0)
ld      s7,360(t0)
ld      s8,368(t0)
ld      s9,376(t0)
ld      s10,384(t0)
ld      s11,392(t0)
jr      t1
```

We found dispatcher gadgets that set `t0` from one of the registers that we can control with our initializer gadget, and then jump to a different register that we control. Figure 2 shows an example of these dispatcher gadgets. We use one of these dispatcher gadgets as a prologue to the initializer gadget, as it allows us to prepare the dispatch table for the initializer gadget to load values from. Furthermore, we can endlessly chain these gadgets together by setting the source register of the next dispatcher gadget using the initializer gadget that precedes it. We (facetiously) refer to this combo as our *dispatchilizer* gadget.

[1] `gp`, `tp`, `t0`, `t2`, `t3`, `t4`, `t5`, and `t6` are the only registers that are not set by this initializer gadget.

Figure 2: Dispatcher gadget

```
mv      t0,s3
jr      a3
```

3.1 Finding Useful Gadgets

Our attack requires finding and using gadgets from an object file. There are various open source toolsets for this task, but none with current support for RISC-V. Initially, we manually searched for gadgets using regular expressions on the disassembled object files. This was both a cumbersome and time consuming task. We also reimplemented the Galileo search algorithm from [19], but later found an existing tool that we could contribute to. Ropper [18] is an open source project implementing a gadget scanner built on top of a multi-architecture disassembly framework, Capstone [16]. Ropper scans an object file for potential gadgets by looking for gadget epilogues such as returns or jumps. The latest versions of Capstone support RISC-V and allowed us to to extend Ropper to RISC-V by implementing a scanner for RISC-V jump instructions and adding bindings to the RISC-V Capstone library. We used our additions to Ropper to assist in finding useful JOP gadgets that have minimal side effects. Using this tool proved to be much more effective than manually searching.

3.2 Proof of Concept

As our initial attempt, we implemented a simplified JOP attack using only gadgets found from `libc.so.6`. The attack is initialized by a gadget from `longjmp` which loads a set of registers from the stack and returns to the first JOP gadget in our chain. We chained together multiple JOP gadgets to set up the necessary registers to open a shell with the `execve` function.

Next, we implemented a version of the JOP dispatch-table attack described in [1]. We made use of the dispatchilizer gadget that we previously described. After the first use of the dispatchilizer, we can jump to a short JOP chain that ends with a jump back to the dispatchilizer, allowing us to execute the next JOP chain. This allows us to trivially chain together multiple function calls by loading the the argument registers (`a0-7`) in the initializer portion, preparing the next jump to the dispatchilizer, and jumping to the function which then returns to dispatchilizer.

We demonstrated this capability by implementing a dispatch table compiler in Python which is able to chain calls to arbitrary functions and other gadgets. Figure 3 shows an example dispatch table that calls multiple short JOP chains and various `libc` functions. Each buffer contains the registers for the initializer gadget to load. The first buffer will load the dispatcher gadgets address into `t1`, causing execution to jump to the dispatcher. The dispatcher then moves `s3`, which

Figure 3: Dispatch table chaining function calls

buf1 (putchar)	`t1` = &dispatcher `a3` = &`putchar` `ra` = &initializer	`s3` = &buf2 `a0` = 'z'
buf2 (printf)	`t1` = &dispatcher `a3` = &`printf` `ra` = &initializer	`s3` = &buf3 `a0` = &"Enter a number: "
buf3 (scanf)	`t1` = &`scanf` `ra` = &mv_s2_to_a3_j_s1 `s1` = &dispatcher `s2` = &initializer	`s3` = &buf4 `a0` = &"%d" `a1` = &integer_ptr
buf4 (printf)	`t1` = &ld_a1_from_s0_j_a5 `a5` = &mv_s6_to_a0_j_s5 `s5` = &`printf`	`s0` = &integer_ptr `s6` = &"you entered: %d\n" `sp` = &printf_stack_buf
printf stack	`ra` = &mv_s2_to_a3_j_s1 `s1` = &dispatcher `s2` = &initializer	s3 = &buf5
strings buf	char* strings[] = {&str1, &str2, &str3, 0}	str1 = "/bin/bash" str2 = "-c" str3 = "<bash script>"
buf5 (execve)	`t1` = &dispatcher `a3` = &`execve` `ra` = &initializer	`s3` = &buf6 `a0` = &str1 `a1` = &s

contains the address of the next buf, into `t1` before jumping to `a3` which contains the address of `putchar`. Finally `putchar` will print the character in `a0` before returning to the address in `ra`, which contains the initializer gadgets address. The initializer will then load the register values to execute `printf` in the same manner. buf3 executes scanf and clobbers `t0` and `a3`. Thus a short JOP chain after the function call is required: first going to a gadget to setup the dispatcher jump address, then the dispatcher to setup the next dispatch table buffer, and finally the initializer gadget. buf4 demonstrates a function call using a dereferenced pointer as an argument. There is a short JOP chain prior to calling printf which loads memory from an address in register `s0` into `a1`, sets the argument in `a0` clobbered by the previous gadget, and finally calls printf with the stack pointer pointing to attacker controlled memory that will be restored to registers when printf returns. These registers are set in the printf_stack buffer and continue the JOP chain to the next initializer gadget. This gadget sets up the argument registers as pointers to an array of strings containing an executable and arguments to pass to execve. We show how this can be used to execute arbitrary executables or even shell scripts. We demonstrate this capability with an example attack that injects a JOP attack to run a script that opens a reverse-shell to the attacker over the network.

3.3 Limitations

In order to start this attack we need the ability to set `t0` to our first dispatch table. Next, we need a way to store the to-be-loaded values in memory. Finally, as with any attack that targets the control flow of a program, we need to redirect execution to our dispatchilizer gadget, or at least the initializer portion of it.

In the current state of our attack, we need the virtual ad-

dresses of the gadgets we use ahead of time. While this means our attack could be thwarted by ASLR, there are a number of ways to get around this that we consider out of scope for our research.

3.4 C Extension and Misinterpretation

The standard extensions of RISC-V—IMAFD, collectively identified as "G"—utilize exclusively 32-bit instructions. As such, instructions are required to be 4-byte aligned. If a branch or a jump leads to an address that is not 4-byte aligned, then an instruction address misaligned exception is raised [21]. The C extension of RISC-V introduces 16-bit alternatives for common instructions. Some examples are when one of the registers is `zero`, `ra`, or `sp`, and when the first source register and the destination register are identical. This extension reduces aforementioned 4-byte instruction alignment requirement to 2 bytes, in order to allow for better code density.

This relaxed requirement also allows the ability to misinterpret instructions. One example is that the latter 16 bites of a 32-bit instruction, in conjunction with the first 16 bits of the following instruction, could be interpreted as a valid, unintended 32-bit instruction. Another way this can occur is if the latter 16 bits of a 32-bit instruction encode a valid 16-bit instruction from the C extension.

Due to the limited possible sizes of RISC-V instructions, this misinterpretation is nowhere near as effective as that described in Shacham's original ROP paper [19]. However, Jaloyan et al. demonstrate that overlapping instruction caused by such misinterpretation can still be utilized in an attack [8]. We also make use of this strategy to look for additional jump-based gadget to add to our repertoire.

In order to search for these gadgets, we re-implemented the Galileo algorithm described in [19]. However, since our RISC-V instructions can only have sizes of 16 or 32 bits, the backtracking aspect is much simpler. Our implementation is as follows: We scan forward until we find a jump instruction. Then we work backwards, two bytes at a time, and try to disassemble instructions starting from the new address. Sometimes this can re-misinterpret the initial jump instruction that we started from, into a different instruction and the sequence becomes unusable. Other times we may encounter bytes that do not encode a valid instruction, and this also renders our potential gadget unusable. We hypothesize that, due to the sparsity of instruction on RISC-V, this approach can only find a limited number of usable instruction, let alone usable ones.

After some initial attempts with this technique, we were able to find a new gadget that could increase `t0` based on a register that we control, `s7`. Figure 4 shows the gadget that we found, and Figure 5 shows what the actual code (which starts 2 bytes before) is. This gadget ends with `ret`, which is equivalent to `jalr ra`. Since we can control `ra` with our initializer gadget, we can still make use of it. We recognize that this is not technically a JOP gadget, but it is possible that

Figure 4: Misinterpreted gadget

```
add     t0,t0,s7
li      a0,2
ret
```

Figure 5: Actual code sequence

```
jal     ra,0xb331c
li      a0,2
ret
```

there are other more complicated gadgets that accomplish this functionality with the use of `ret`. What we have shown is that misinterpreted gadgets can give us functionality that we otherwise did not previously have.

3.5 Following Relative Jumps

Our initial attempt to search for jump gadgets using our modified Galileo algorithm presented us with many gadgets that ended with relative jumps. At first glance, these gadgets are unusable because they redirect control flow to a destination that we do not control. However, it is possible that some of these gadgets jump to a short instruction sequence which ends with an indirect jump. If the indirect jump is determined by a register that we control, then these gadgets would possibly become of interest to us.

We developed a simple program to scan through the executable sections of a given object file. Upon reaching a relative jump instruction, we attempted to disassemble the next five instructions that would follow. In other words, we would seek to the address implied by the relative jump. If an invalid instruction is found after the relative jump, then that renders the entire gadget unusable so we skip it. We chose a limit of four instructions because large gadgets are likely to have undesirable register clobbering.

This program analyzed the same `libc.so.6` file that we used in our earlier gadget searching. We measured how many total relative jumps led to an indirect jump, within x instructions where $x \in [0,4]$. For completeness, we repeated the analysis twice: including and excluding `ret`. Our search results are presented in Table 1.

There were a total of 24,000 relative jumps in the text section of the `libc` file. This means that only approximately 1% of relative jumps lead to concise, desirable epilogues. Among the potential gadgets, the majority of indirect jumps relied on either the `t5` or the `t6` register. This posed an additional limitation because our dispatchilizer is not able to manipulate those registers by itself. As a result, we decided to forgo further investigation into following relative jumps during our gadget scanning. We do not rule out their potential utility en-

Table 1: Number of indirect jumps within x instructions of a relative jump

x	With `ret`	Without `ret`
0	38	16
1	81	23
2	334	276
3	334	276
4	334	276

tirely. However, we suspect that their scarcity and size would make them only usable in certain scenarios.

4 Discussion

We successully demonstrated the effectiveness of ROP and JOP attacks from [1,4,17] on the RISC-V architecture. RISC-V is a RISC-based ISA, like ARM, and the same limitations with translating x86 attacks to ARM apply to RISC-V as well. Implementing complex chains of JOP gadgets in RISC-V is made difficult for the same reasons as described in [1]: there is a layer of interdependency between JOP gadgets as certain registers need to be maintained to serve as the state of the dispatcher between short chains. We work around this limitation by using a initializer gadget which relies on a rarely-used register `t0` that always points to our dispatch table to load any values we want into nearly any register. This allows us to construct short length JOP chains before jumping back to the initializer gadget in order to setup the next JOP chain. This doesn't solve the problem of interdependency however as we now require that each short JOP chain moves `t0` forward in the dispatch table before jumping back to the initializer gadget. The benefit of this initializer gadget is that it makes calling arbitrary functions a simple task as it provides full control over function-argument registers in addition to the stack-pointer and return address. This allows us to implement powerful attacks by simply calling multiple functions rather than requiring a set of turing-complete chainable JOP gadgets.

AArch32, 32-bit ARM, supports variable length instructions: execution mode can be changed to either 32-bit ARM mode or 16-bit Thumb mode, and Thumb2 mode allows for executing both on a 16-bit alignment. This enables the processor to execute misaligned instructions and has been used in other code-reuse attacks such as JIT spraying [11]. AArch64, 64-bit ARM, on the other hand only supports single-sized instructions, limiting the set of gadgets available for ROP and JOP attacks in other literature. The RV64GC variant of RISC-V introduces a set of compressed instructions that allows for variable length instructions - this is less powerful than the variable length instructions of x86 but still enabled us to find additional useful gadgets which made our attack more powerful. In our initial JOP attack, the pointer to the next dispatch buffer needs to be loaded into `t0` before calling the initializer gadget. This incurs a limitation of needing to know the exact address at which the dispatch-table will be stored in memory when crafting the exploit. This is an unrealistic limitation given ASLR. We found very few usuable JOP gadgets in `libc` that gave us to control over `t0`. Searching for unintended compressed instructions revealed an additional set of short JOP gadgets, including ones that modified `t0`. We used one such unintended gadget to increment `t0` before jumping to another register. This allowed us to construct a dispatch table that did not need to contain pointers to the next dispatch buffer in the dispatch table, making our attack much more feasible against ASLR.

5 Related Work

There have been multiple papers investigating control-flow integrity (CFI) extensions to the RISC-V ISA [5, 10]. Zipper [10] implements a shadow stack to defend against ROP attacks by detecting when return addresses are overwritten by a buffer-overflow. This does not protect against purely JOP attacks which rely on jump addresses being overwritten. However, they do add additional protections for attacks exploiting Setjmp/Longjmp which would prevent our example attack against a program with a vulnerable call to `longjmp`.

Fixer [5] implements a tagged RISC-V architecture for CFI and also uses a shadow stack to protect against overwriting return addresses. They implement additional protections against forward-edge attacks such as indirect jumps by analyzing code both statically and at runtime to construct a control flow graph that acts as a policy matrix to validate jumps and function calls. This type of system should be effective at defending against the JOP attacks that we have demonstrated. One possible workaround would be to only use misinterpreted compressed instructions for jumps as these wouldn't be in the policy matrix. Being limited to only compressed instructions would greatly limit the effectiveness of a JOP attack. This could furthermore be defended against in by simply disallowing any jumps that to do not have any entries in the policy matrix.

6 Conclusion and Future Work

We have shown that on RISC-V it is possible to run a jump-oriented programming attack as well as a return-oriented-programming attack without any return instructions. These attacks can be mounted in a similar manner as seen in research on other architectures such as x86 and ARM on RISC-V. The same vulnerable code snippets from those attacks on other ISA's can likewise be exploited on RISC-V by following the same attack paradigms as demonstrated on ARM. Exploiting bufffer overflows with jump-oriented-programming is a diffi-

cult task due to the complexities of needing to manage and preserve registers across chains of gadgets with register interdependencies. We worked around this by finding and utilizing a very powerful initializer gadget, allowing us to easily chain multiple function calls. Unlike ARM, RISC-V has support for variable length instructions in the form of the Compressed RISC-V extension. We showed how this can be used to find additional useful gadgets that are otherwise unavailable, extending the toolset of gadgets available to attackers developing ROP or JOP attacks. JOP attacks are resilient to many of the defenses against ROP, but can most likely be prevented by code flow integrity checks in the processor.

Future work in this space would be to automate the process of finding gadgets and crafting attacks through the use of automated SMT solvers. Additionally, more research can be done into how Code Flow Integrity can be implemented to defend against the misuse of indirect jumps.

Availability

Our attacks were written for riscv64 Debian GLIBC 2.31-4 The implementations of these attacks can be found in the various branches of our repository on Github: https://github.com/dstarikov/riscv-jop

References

[1] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, page 30–40, New York, NY, USA, 2011. Association for Computing Machinery. https://doi.org/10.1145/1966913.1966919.

[2] E. Bosman and H. Bos. Framing Signals - A Return to Portable Shellcode. In *2014 IEEE Symposium on Security and Privacy*, pages 243–258, 2014. https://doi.org/10.1109/SP.2014.23.

[3] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. CCS '08, page 27–38, New York, NY, USA, 2008. Association for Computing Machinery. https://doi.org/10.1145/1455770.1455776.

[4] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 559–572, New York, NY, USA, 2010. Association for Computing Machinery. https://doi.org/10.1145/1866307.1866370.

[5] A. De, A. Basu, S. Ghosh, and T. Jaeger. FIXER: Flow Integrity Extensions for Embedded RISC-V. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 348–353, 2019. https://doi.org/10.23919/DATE.2019.8714980.

[6] Theo de Raadt. OpenBSD 3.3 announcement (public release of W⊕X), 2003. https://www.openbsd.org/33.html.

[7] Garrett Gu and Hovav Shacham. Return-Oriented Programming in RISC-V, 2020. https://arxiv.org/abs/2007.14995.

[8] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. *Return-Oriented Programming on RISC-V*, page 471–480. Association for Computing Machinery, New York, NY, USA, 2020. https://doi.org/10.1145/3320269.3384738.

[9] Tim Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010. http://www.handgrep.se/repository/ebooks/Security/Reversing/kornau-tim--diplomarbeit--rop.pdf.

[10] Jinfeng Li, Liwei Chen, Qizhen Xu, Linan Tian, Gang Shi, Kai Chen, and Dan Meng. Zipper Stack: Shadow Stacks Without Shadow, 2020. https://arxiv.org/abs/1902.00888.

[11] Wilson Lian, Hovav Shacham, and Stefan Savage. Too LeJIT to Quit: Extending JIT Spraying to ARM. In Engin Kirda, editor, *Proceedings of NDSS 2015*. Internet Society, February 2015. https://doi.org/10.14722/ndss.2015.23288.

[12] Urban Müller. brainfuck, An Eight-Instruction Turing-Complete Programming Language, 1993. https://www.muppetlabs.com/~breadbox/bf/.

[13] Jeffrey Osier-Mixon. Semico Forecasts Strong Growth for RISC-V, 2019. https://riscv.org/announcements/2019/11/9679/.

[14] David A Patterson and Carlo H Sequin. RISC I: A reduced instruction set VLSI computer. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 216–230, 1998. https://doi.org/10.1145/285930.285981.

[15] Mathias Payer and Thomas R. Gross. String Oriented Programming: When ASLR is Not Enough. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and*

Reverse Engineering Workshop, PPREW '13, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2430553.2430555.

[16] Nguyen Anh Quynh. Capstone. https://github.com/aquynh/capstone, 2013.

[17] Ali-Akbar Sadeghi, Farzane Aminmansour, and Hamid-Reza Shahriari. Tiny jump-oriented programming attack (A class of code reuse attacks). pages 52–57, 09 2015. https://doi.org/10.1109/ISCISC.2015.7387898.

[18] Sascha Schirra. Ropper. https://github.com/sashs/Ropper, 2014.

[19] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery. https://doi.org/10.1145/1315245.1315313.

[20] PaX Team. Address Space Layout Randomization (ASLR), 2003. 2003. https://pax.grsecurity.net/docs/aslr.txt.

[21] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. https://github.com/riscv/riscv-isa-manual.