RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64

Tobias Cloosters
tobias.cloosters@uni-due.de
University of Duisburg-Essen
Essen, Germany

David Paaßen
david.paassen@uni-due.de
University of Duisburg-Essen
Essen, Germany

Jianqiang Wang
jianqiang.wang@trust.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Oussama Draissi
oussama.draissi@uni-due.de
University of Duisburg-Essen
Essen, Germany

Patrick Jauernig
patrick.jauernig@sanctuary.dev
Technical University of Darmstadt
Darmstadt, Germany

Emmanuel Stapf
emmanuel.stapf@sanctuary.dev
Technical University of Darmstadt
Darmstadt, Germany

Lucas Davi
lucas.davi@uni-due.de
University of Duisburg-Essen
Essen, Germany

Ahmad Sadeghi
ahmad.sadeghi@trust.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

Return-oriented programming (ROP) is a powerful run-time exploitation technique to attack vulnerable software. Modern RISC architectures like RISC-V and ARM64 pose new challenges for ROP execution due to the lack of a stack-based return instruction and strict instruction alignment. Further, the large number of caller-saved argument registers significantly reduces the gadget space available to the attacker. Consequently, existing ROP gadget tools for other processor architectures cannot be applied to these RISC architectures. Previous work on RISC-V provides only manual construction of ROP attacks against specially crafted programs, and no analysis of ROP attacks has been conducted for ARM64 yet.

In this paper, we address these challenges and present RiscyROP, the first automated ROP gadget finding and chaining toolkit for RISC-V and ARM64. RiscyROP analyzes available gadgets utilizing symbolic execution, and automatically generates complex multi-stage chains to conduct arbitrary function calls. Our approach enables the first investigation of the gadget space on RISC-V and ARM64 real-world binaries. RiscyROP successfully builds ROP chains that enable an attacker to execute arbitrary function calls for the nginx web server as well as any binary that contains the libc library.

CCS CONCEPTS

• **Security and privacy → Systems security**; *Mobile platform security*; Vulnerability scanners.

KEYWORDS

RISC-V, ARM64, Return-Oriented Programming, Symbolic Execution, Exploitation

ACM Reference Format:
Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad Sadeghi. 2022. RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545948.3545997

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545997

1 INTRODUCTION

Return-oriented programming (ROP) [40] is a state-of-the-art attack exploitation technique to hijack modern software running on various architectures. As ROP only leverages existing benign code, it circumvents classic protection mechanisms such as non-executable data memory [45]. After exploiting a memory corruption error, a ROP attack executes a set of short code snippets (gadgets) to induce arbitrary malicious behavior and invoke arbitrary functions. The key part of ROP attacks is a stack-based return instruction, which dispatches execution from one gadget to the next gadget.

ROP attacks have been discussed extensively in academic literature [40, 19, 8, 43, 4, 42, 15, 7, 38] and several automated gadget tools as well as defenses have been proposed [45, 14, 33, 49]. On x86, the gadget space is significantly larger compared to RISC architectures because unaligned memory access produces many *unintentional* instruction sequences. Existing tools to automate ROP attacks on x86, such as Q [39], commonly utilize implementations based on the well-known Galileo algorithm, originally proposed by Shacham [40]. The Galileo algorithm searches for byte sequences reassembling a return instruction and analyzes the preceding bytes for valid (and useful) instructions. Even though Galileo-based algorithms have been used with great success to attack programs, this algorithm is unable to detect all types of gadgets on modern architectures such as RISC-V [24] (see Section 5). Hence, by utilizing existing tools, we may underestimate the capabilities of an attacker when analyzing a compiled binary for ROP gadgets. Nevertheless, RISC architectures

	x86	x86_64	ARM32	RISC-V	ARM64
① Writable Program Counter	○	○	●	○	○
② Stack-return Instruction	●	●	●	○	○
③ Max. Arguments in Registers	0	4–6	4	8	8
③ Mixed-purpose Registers	●	●	●	○	○
④ Instruction Alignment	1	1	4/2	4/2	4
⑤ Short Function Epilogue	●	●	●	○	○

Table 1: Properties and conventions of different architectures that pose challenges for return-oriented programming. The dot (●) indicates the presence of a property that is beneficial for return-oriented programming.

such as SPARC and ARM32 are similarly vulnerable to ROP attacks (though a smaller gadget space is available) because similar to x86 function return instructions can be exploited to chain gadgets [5, 8].

However, it is not yet clear to which extent ROP attacks are applicable to recent processor architectures such as ARM64 and RISC-V. ARM64 is widely used, e.g., in newer iOS devices from Apple, and RISC-V is a promising open source processor architecture that is gaining more and more traction. It is used in mobile devices, for example as a security chip in Google's Android phones [20] as well as in various products by Western Digital [30]. To our surprise, there is no thorough investigation whether real-world software executing on these modern architectures is susceptible to ROP. Due to the differences to existing architectures such as x86 and ARM32, ARM64 and RISC-V pose new challenges for ROP attacks. As we will elaborate in detail in Section 3, a number of architectural differences (cf. Table 1) make determining ROP gadgets challenging on these architectures: ① The program counter (`pc`) register is not a general-purpose register (unlike ARM32), ② there does not exist a stack-based return instruction, which is a key building block for ROP on x86 and ARM32, ③ arguments to functions are passed via (dedicated) processor registers instead of the stack, ④ memory alignment prevents execution of unintended sequences, and ⑤ function epilogue sequences introduce side effects by executing several instructions between loading a return address from the stack and actually using it for a control-flow transfer. These design features in combination severely limit the capabilities of an attacker as the only option available to the attacker is jump-oriented programming [4], which leverages sequences ending in indirect calls and jumps. While jump-oriented programming attacks have been demonstrated on x86 and ARM32 [8], they require a tedious manual gadget search. The result is, that—for both architectures—there remain only two major sources of gadgets: function epilogues and sequences ending in indirect jumps. The former is hardly usable for calling arbitrary functions as they rarely set registers which contain the function parameters. In contrast, the latter typically set function arguments in registers but do not read any data from the stack, i.e., they load arguments based on other registers. On top of all these challenges, we will empirically show that gadgets on these architectures are significantly longer and more complex compared to the convenient stack-based return sequences known from x86 and ARM32. In particular, the majority of gadgets on RISC-V and ARM64 introduce various side effects (i.e., changing many register values) which makes manual ROP-chaining highly challenging as useful gadgets are easily missed or deemed not useful when in reality they could be part of a ROP chain. Thus, we need to develop a new analysis approach to address the peculiarities of modern RISC architectures as existing tools cannot simply be re-purposed for ARM64 and RISC-V.

In this paper, we present RiscyROP, a novel and automated analysis approach that determines gadgets for ARM64 and RISC-V binaries allowing accurate risk assessment regarding ROP attacks on these architectures. Instead of a simple pattern search, our approach is based on symbolic execution. This allows us to leverage gadgets ending in any instruction that jumps to an attacker-controlled location. Furthermore, the symbolic execution approach allows us to easily collect path constraints to automatically exclude gadgets chains that are impossible to execute in practice. We study the prevalence of useful gadgets on ARM64 as well as RISC-V. We compare the results to traditional ROP gadget finders and demonstrate that they are rarely able to find ROP gadgets to construct a (malicious) function call on RISC applications.

In summary, we make the following contributions:

- We systematically analyze the complexity of finding gadgets and building ROP gadget chains for RISC-V and ARM64 (Section 3).
- We propose RiscyROP, an automated ROP gadget finder and chaining tool for RISC-V and ARM64 based on symbolic execution that is able to find gadgets despite the limited gadget space and increased complexity of resulting ROP chains (Section 5).
- We analyze real-world software from public repositories and show that RiscyROP is able to find gadgets to execute attacker-controlled function calls.
- We also provide a detailed evaluation based on a real-world ROP-based exploit using the popular nginx web server and the C standard library libc (Section 6).

2 BACKGROUND

To better understand the challenges of executing arbitrary function calls via a ROP attack, we introduce the registers and calling conventions of the RISC-V and ARM64 architectures.

2.1 RISC-V Architecture

RISC-V defines 31 general-purpose registers, which includes the stack pointer `sp` and argument registers `a0–a7`. Further, there is a constant zero register, 32 floating point registers and the program counter (`pc`). For function calls, arguments are copied to `a0–a7` prior to the jump to the destination. Calls are executed with the `jal/jalr` instruction, which stores the return address in the `ra` register. The function prologue on RISC-V decreases the `sp` register to allocate stack memory for local variables and stores (`sd`) the caller's frame pointer inside the newly allocated stack frame. Then the frame pointer (`s0/fp`) is adjusted to point to the new stack frame and subsequently used to reference local variables. Used callee-saved registers (`s0–s11`) are preserved in the stack frame. Upon function completion, the return value is stored in `a0`, all callee-saved registers are restored and `sp` is increased to reclaim the stack memory. The callee returns by jumping to the address stored in `ra`.

When a callee itself calls functions, the value of `ra` is saved on the stack prior to a nested function call.

2.2 ARM64 Architecture

ARM64 defines 31 general-purpose registers (`x0–x30`), 32 floating-point registers, and a set of special registers including the zero register, the program counter (`pc`) and the stack pointer (`sp`). The lower 32 Bit of the general-purpose registers can be addressed using `w0–w30`. On ARM64, the registers `x0–x7` are used for arguments. Functions are called using the `bl` instruction which stores the return address in the link register (`lr`) and then branches to the given address. The function prologue of the callee saves the return address (stored in `lr`), the frame pointer (`fp`), and the callee-saved registers (`x19–x28`) on the stack; the registers `x9–x15`, which are used for local variables, must be saved by the caller. Next, memory for local variables is allocated by decreasing the stack pointer. During the function epilogue callee-saved registers are restored. Then, return address and frame pointer are loaded from the stack and the stack pointer is increased using a single *load pair* instruction (`ldp fr, lr, [sp], #0x50`). Thereafter, the callee returns by branching to the return address stored in the `lr` register (`br lr`).

3 CHALLENGES

In the following we discuss the unique challenges of ROP attack on RISC-V and ARM64 in comparison to other architectures like x86, which are summarized in Table 1. The combination of all the described challenges is the reason that ROP attacks are so hard to execute on these architectures. Due to its unique design and utilization of symbolic execution, RiscyROP is able to solve the problems posed by RISC-V as well as ARM64.

Stack-Based Returns. The x86 `ret` instruction gives return-oriented programming its name as it dispatches execution to the gadget referenced by the (malicious) return address from the stack. RISC architectures commonly use a dedicated register to store the return address (`ra`/`lr`). However, for non-leaf functions (i.e., nested function calls), it is still required to store return addresses on the stack, which makes them potentially accessible to an attacker that exploits a security vulnerability. On ARM32, a stack-based return is usually implemented using `pop pc` which is—equivalent to the x86 `ret` instruction—a stack-based return instruction. This is possible because the program counter `pc` is a general-purpose register, writable by all instructions. However, on RISC-V and ARM64, the program counter is protected and can only be modified by jump instructions. As a result, RISC-V applications use the *load* instruction to restore `ra` at the beginning of the epilogue (`ld ra,0x10(sp)`), and on ARM64 the *load pair* instruction is used to restore the link register `lr` and frame pointer `fp` in one instruction directly before jumping/returning to the caller (`ldp fp, lr, [sp], #0x30; ret`).

The main challenge is that we need to consider complex side effects. In particular, between the loading of the return address from the stack and the actual jump to the intended gadget, callee-saved registers are restored (`s0–s11` on RISC-V). Note that none of the function epilogue sequences of the applications evaluated in Section 6 was free of side effects.

Memory Alignment. The gadget space on x86 highly benefits from unaligned single-byte instructions. For example, `pop rdi; ret` compiles to two bytes, which is already a useful gadget: It loads an argument register from the stack and returns to an address from the stack. On RISC-V, instructions are aligned to either 4 Bytes, or 2 Bytes for compressed instructions. This makes unaligned instructions possible but unlikely (unaligned sequences cannot go past two compressed instructions). In our empirical study in Section 6.3 we used one gadget that contained a single unaligned instruction. However, contrary to x86 we did not find any gadget compiled only of hidden instructions in real-world applications.

ARM64 has a fixed instruction size of 4 Bytes making unaligned sequences entirely impossible. Therefore, the gadget space is limited to the intended disassembly, i.e., we are limited to the intended function epilogue sequences, which—as mentioned above—always introduce side effects which need to be put into consideration.

Function Calls. Typically, ROP attacks build a gadget chain that sets arguments to call arbitrary functions. On RISC-V and ARM64 the arguments have to be loaded into *dedicated* argument registers (of which 8 are available on both architectures). Setting these argument registers is challenging because function epilogue sequences do not set these registers, i.e., they only set callee-saved registers. The apparent solution of using instructions from the function body introduces even more complexity, as the gadget has to be executed until the next *intended* jump instructions because stack-based return instructions do not occur unintentionally in real-world programs.

System Calls. Given the existing ROP attacks on programs compiled for x86, one might think that instead of calling a library function, the attacker could simply invoke a gadget that directly executes a system call. However, the system call instruction (`ecall`/`svc #0`) which has 4 Bytes on both architectures is very unlikely to be included unintentionally and therefore does not exist in most applications that use the libc as a system call wrapper. In fact, none of our real-world binaries analyzed in Section 6 includes such a gadget.

4 THREAT MODEL

RiscyROP analyzes the gadget space of real-world binaries and automatically generates ROP chains for RISC-V and ARM64. Hence, we build on a threat model and program state in which the execution of ROP chains is possible. This requires an attacker to exploit a program vulnerability (e.g., a buffer overflow vulnerability on the stack or heap) to control an indirect jump instruction and to ultimately gain arbitrary execution capabilities.

Note that return-oriented programming (ROP) [40] is a code-reuse attack that bypasses non-writable code sections (W⊕X) [45]. To do so, a ROP exploit requires exploiting a memory corruption vulnerability to overwrite a return address on the stack or a function pointer on the program's heap or stack. We assume that such a vulnerability is present in the programs we test. Binaries may be additionally hardened against buffer overflow vulnerabilities and code-reuse attacks using, e.g., stack canaries and address randomization such as ASLR. However, such defenses are typically bypassed by using an information leak vulnerability prior to the ROP attack. Therefore, we do not consider them for the sake of simplicity.

5 RiscyROP

RiscyROP is a novel ROP gadget finding and chaining tool based on symbolic execution targeting RISC-V and ARM64 to accurately handle the challenges described in Section 3, for which the commonly used Galileo algorithm is insufficient. It analyzes gadgets automatically to enable the generation of complex ROP chains to allow an attacker to perform arbitrary function calls. RiscyROP performs the following steps to construct such a ROP chain:

(1) RiscyROP first scans every aligned address in the target for usable gadgets.
(2) These gadgets are then evaluated for fitness using symbolic execution and subsequently stored in a database with their respective constraints (e.g., preconditions and effects on registers).
(3) Next, RiscyROP uses the database to construct ROP chains by stitching compatible gadgets. Candidate chains are verified using symbolic execution to ensure that they do not contain any breaking side effects. Gadgets are processed starting with the least constrained one to optimize the run time.
(4) Lastly, once a gadget chain has been successfully verified, RiscyROP generates the exploit payload using the argument values chosen by the attacker.

In the following sections, we elaborate in detail how each of these steps is executed by RiscyROP.

5.1 Gadget Finding

The gadget exploration of RiscyROP maximizes the usable gadget space to include complex gadgets that are essential for ROP chains on RISC-V and ARM64. Previous gadget finders [36, 6, 39] commonly find gadgets using a form of the Galileo algorithm [40], i.e., they search for a byte sequence or byte pattern that resembles a return instruction. Next, they decode instructions in the reverse execution order to find complete gadgets. In contrast, with symbolic execution, we are independent of specific byte patterns, which allows us to find *every* sequence of instructions that read an address from the stack and finally jump to this (attacker-controlled) address. Hence, we do not need to rely on known instruction patterns. This is especially interesting for RISC-V, since non-leaf functions (which must save the return address on the stack) include different instructions between restoring the return register `ra` and jumping to `ra` (cf. Section 3). In addition, symbolic execution allows us to also utilize other types of jumps. For example, there are load-jump sequences that effectively are stack-based returns but use different registers. The instructions `ld a7,0x40(sp) ... jalr a7` (taken from Figure 4, gadget 3) effectively perform a stack-based jump but using an argument register. Also, there are jump-to-register gadgets (indirect function calls) which can be chained in sequence to load-register gadgets to form a load-jump gadget. An example is gadget 5 in Figure 4 which jumps to `t1` and therefore requires a preceding gadget that loads an attacker-controlled value to `t1` (gadget 2). Thus, symbolic execution identifies gadgets by their effects independently of specific instructions.

We build RiscyROP on top of the angr framework [41] and its PCode engine. Since the support for PCode of angr is experimental and mostly untested on RISC-V, we had to implement the register and calling contention to execute RISC-V assembly within angr. In the exploration phase, RiscyROP symbolically analyzes every (aligned) address independently to enumerate potential gadgets. Our approach does not rely on any specific instruction (e.g., a stack-based return) to be present and at no point utilizes pattern matching. The analysis executes the following steps:

(1) Symbolically initialize registers and stack.
(2) Execute the instructions at this address until the control-flow changes to a symbolically unconstrained target, e.g., a return or jump to an unset register.
(3) Analyze the reached state and summarize the effect of the specific gadget on the program state.

During this process, the gadget finder enforces some timeouts and limits to exclude certain edge cases. First, there may be addresses that cause long-running loops: Thus, we abort the analysis process when a code sequence reaches 500 instructions. Second, we limit the maximum complexity of symbolic constraints of the found gadgets. These are the conditions that result from conditional branches within the gadgets that must be true to reach the desired target state, i.e., the end of the respective gadget. If the z3-solver [31] (used by angr) cannot determine if a target state is satisfiable within 250 ms, we assume that the analyzed path is not useful as a ROP gadget and discard it because the respective gadget is unlikely to be useful as part of a ROP chain due to the high complexity of its side effects. Upon completion, RiscyROP creates a database containing all viable gadgets along with the controllable registers, their constraints, and other side effects. Figure 1 shows an example of a gadget analyzed by RiscyROP including the extracted information.

Gadget Effects. The effects of a gadget are summarized to easily filter candidates for a ROP chain, i.e., the effects of a ROP gadget must not violate the requirements of succeeding gadgets. Hence, RiscyROP determines which registers:

- are overwritten with data from the stack,
- only depend on one different register (e.g., due to a move instruction: `mv a3, s20`),
- are set to a constant, and
- are changed otherwise in more complex calculations.

Figure 1 shows the results for one example gadget. In this case, the analysis executes until the `jalr a7` instruction, a jump to a register with an unconstrained value. Note that this passed the conditional branch at `0x5ee`. Thereafter, RiscyROP evaluates the changes of the symbolic state: `a3` and `a7` are set to values which are read from the stack, other argument registers are set from `s`-registers, and the jump-and-link (jalr) instruction stores the subsequent address in `ra` for returning. The stack pointer is unchanged, which must be considered for chaining because a following gadget could read the same values from the stack and cause problems. This is unlike classical ROP gadgets on x86 which commonly use `pop` instructions to read and clear stack values in one step and thus avoid such conflicts. Further, the analysis concludes that the terminating unconstrained jump of this gadget is controllable using offset `0x40` on the stack. Lastly, the constraints to reach this state require `s0` to be equal to the value on the stack at offset `0x28`, which requires `s0` to be controlled by a preceding gadget. This information is also stored in the database to be used in the chaining stage.

Analyzed gadget

```
0x5e6:  c.ldsp  a3,0x28(sp)
0x5e8:  c.ldsp  a7,0x40(sp)
0x5ea:  c.mv    a4,s3
0x5ec:  c.mv    a6,s0
0x5ee:  bne     a3,s0,0x5fe

0x5f2:  add     a5,s1,tp
0x5f6:  c.mv    a2,s2
0x5f8:  c.mv    a1,s5
0x5fa:  c.mv    a0,s4
0x5fc:  c.jalr  a7

0x5fe:  c.jr    ra
```

Extracted Information

Data flow:

- [sp + 0x28] → a3
- [sp + 0x40] → a7
- s0 → a6
- s1 + tp → a5
- s2 → a2
- 0x5fe → ra
- s3 → a4
- s4 → a0
- s5 → a1

Stack Pointer: sp += 0 (unchanged)
Constraints: [Stack + 0x28] == s0
Final Jump: [Stack + 0x40]

Figure 1: Example of the classification of a gadget by RiscyROP using the symbolic execution engine.

Gadget Preconditions. RiscyROP focuses on two types of preconditions: the return type of the gadget and the path constraints. The return type determines how a gadget can be chained to other gadgets. We differentiate between two broad classes: Either RiscyROP can prove that the jump target was read *from the stack*, then it can be simply chained using the stack, or the jump target originates from a *register value*, then the gadgets always need a preceding gadget to control the jump's target register. Hence, the first gadget in a chain always has to be in the stack-returning class. The extracted information is later leveraged in the gadget chaining process. For the path constraints, we store the preconditions that need to be fulfilled to make the gadget usable. As an example, this could be the condition of a conditional branch in the gadget. Additionally, the extracted constraints are also used for chaining as they indicate the complexity of a gadget.

5.2 Gadget Chaining

On the basis of the database gathered in the gadget finding stage, RiscyROP utilizes the gadget information to find candidate chains according to their requirements and effects. Each candidate chain is symbolically executed to verify it. The algorithm is depicted in Figure 2.

Specifically, RiscyROP constructs ROP chains for arbitrary function calls, thus stitches gadgets that control a set of (argument) registers and jumps to a given function. It takes the initial target registers (chosen by the attacker) as input and traverses the database to find gadgets that are suitable, i.e., gadgets that can control all or most of the target registers. Gadgets are processed by the following priority to find chains more quickly:

(1) Gadgets that load a target value from the *stack* are preferred over gadgets that load the value from another *register*.
(2) Gadgets that return to a stack-read address are preferred over register based returns (as these add another constraint).
(3) Gadgets are sorted based on the number of controlled target registers and their complexity (instruction count in this case) to keep the overall chain complexity to a minimum.

After a gadget is selected, the target registers are updated for the next iteration to include remaining registers and new dependencies, that are

(1) target registers that are unaffected by the selected gadget,

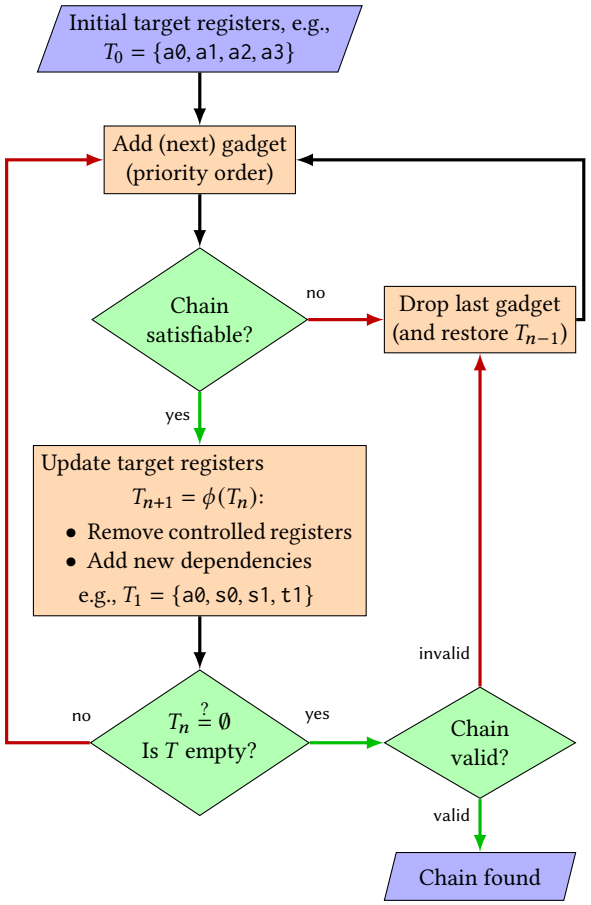


Figure 2: Simplified algorithm to generate a gadget chain as used in RiscyROP. Starting from the register set targeted by the attacker for the function call, gadget effects are applied until all requirements are resolved. The ROP chain is constructed in reverse execution order.

(2) source registers of move-gadgets (register-to-register), and
(3) the register to control the jump target of a gadget (if it is not a stack-returning gadget).

This procedure is done recursively until no registers remain (or a predefined chain length threshold is reached). This approach allows efficient generation of candidate chains and tackles the architectural challenge of missing stack-based returns.

In the next step, RiscyROP executes the full chain candidate using angr to test if the chain yields the desired effect (e.g., a function call with attacker-controlled arguments). The details of this execution step are similar to the gadget analysis, with the addition that the next gadget is inserted whenever the controlled jump (end of a gadget) is reached. In the final step, RiscyROP calculates the concrete exploit payload for the entire chain which is adapted to the attacker's argument values.

Optimizations. This chaining procedure aims to minimize the workload of the symbolic solver to efficiently find ROP chains: First, the precomputed gadget summaries in the database allow RiscyROP to generate chain candidates without invoking the symbolic engine. The iteration process prioritizes gadgets that are more likely to succeed. Lastly, RiscyROP checks if partial chains are satisfiable to reject contradicting chain suffixes. This avoids traversing the database for further gadgets, when the current partial chain already overwrites registers that are necessary as part of the ROP chain with values that cannot be controlled by the attacker.

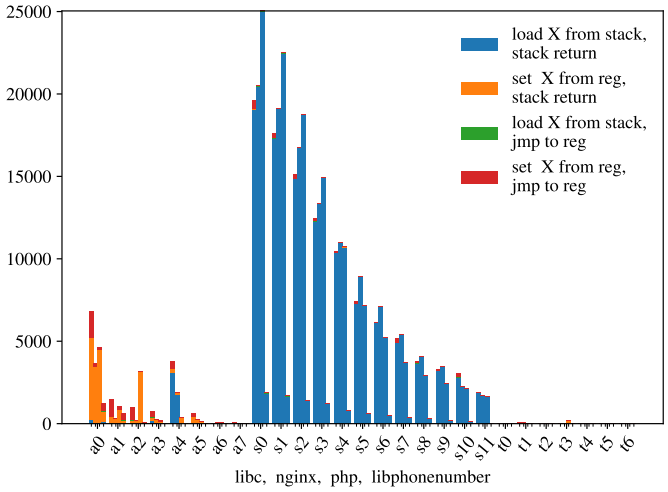
6 EVALUATION

We evaluate RiscyROP using applications and libraries from the official Ubuntu repository that are compiled for ARM64 and RISC-V and analyze, for the first time, their gadget space. We first compare a public exploit chain for x86 with chains for these RISC architectures to demonstrate the complexity of RISC gadgets. Thereafter, we quantify the gadget space showing that the available gadget space is indeed limited compared to x86. Lastly, we discuss a ROP chain on RISC-V in detail to elaborate on the principles to construct function call chains. In our evaluation, we target the C standard library *glibc*, which is commonly used for ROP exploits as it is loaded by most applications on Linux. Furthermore, we analyze two popular web applications namely nginx and php. Lastly, we include libphonenumber, a small library known from Android written by Google in C++. While C++ code tends to contain more indirection than C code, which generally leads to a larger gadget space, the small size of the library makes chaining of ROP gadgets very challenging.

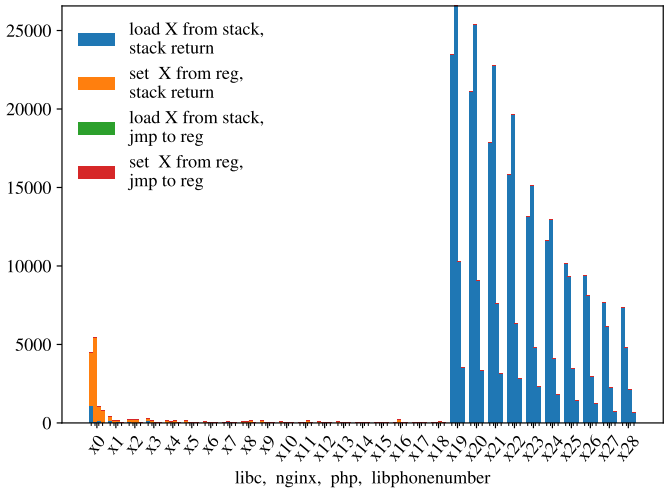
6.1 Exploiting nginx: x86, RISC-V and ARM64

nginx [32] is a popular HTTP server and proxy. We analyze a stack-based buffer overflow and corresponding ROP chain that was found for nginx and compare it with chains automatically generated by RiscyROP for RISC-V and ARM64. The goal of the ROP chain is a call to mprotect to make memory *writable and executable*, which is used to execute shellcode. Figure 5 shows the ROP chains in the context of the CVE-2013-2028 exploit [11] to construct a malicious call to mprotect. On x86, this chain consists only of unintended code sequences (due to unaligned memory access), which are mostly 2-byte `pop/ret` sequences that directly pop values into the argument registers. One can observe the side effect of the second gadget that needs `rax` to be a valid address. This is the reason to include the first gadget to set `rax` to a known address. In comparison, the gadget chains for RISC-V and ARM64 are more complex and therefore not as easy to validate manually. RiscyROP can prove using symbolic execution that the attacker can control the arguments with values from the stack and that it does not contain side effects that are out of control for an attacker (e.g., in the RISC-V chain, data is stored relative to `a0` which is controlled by the first gadget).

The conceptual details of a ROP chain on RISC-V are explained in Section 6.3. The core of this specific chain, which ultimately allows access to the argument registers, is a gadget that copies data from the stack to a given pointer while using *the argument registers as temporary variables*. The ARM64 chain uses an indirect function call (see Section 6.3); the challenging part is to find a gadget that



(a) RISC-V



(b) ARM64

Figure 3: Gadget space of official Ubuntu binaries (excluding dynamically loaded libraries): For every register, there is a stacked bar for each of the listed applications showing the *number of gadgets* present in the respective category. Note that stack-load stack-return gadgets are almost only found for callee-saved registers.

Target	Lang.	Version	Function Call Arguments	
			RISC-V	*ARM64*
glibc	C	2.31	≤ 7	≤ 7
nginx	C	1.18	≤ 3	≤ 6
php	C	8.0.8	≤ 4	≤ 4
libphonenumber	C++	8.12.16-4.2	≤ 6	≤ 4

Table 2: Maximal arguments of function call chains constructed by RiscyROP using the gadget space of official Ubuntu binaries (dynamically loaded libraries are excluded).

controls the target of that call (`x4`), which again needs a gadget to control its side effects on the `x0` register.

6.2 Gadget Space Statistics

Figure 3 shows an overview of the gadgets we collected during our analysis as classified by the symbolic execution engine. For every register and application, the graph shows a bar, which corresponds to the number of gadgets in the respective groups. The gadgets count as stack-return if the final jump is determined to be an address from the stack. Otherwise, it counts as register jump if the target is controllable by a register, e.g., not a constant value. A gadget is counted in multiple groups, if it sets multiple registers. First, one can clearly see the separation of argument registers `a0–a7/x0–x7` and callee-saved registers, which are restored by the function epilogues. As functions use the low-numbered registers first, there are more function epilogues that restore these registers. Second, gadgets that directly load an argument register from the stack are rare. Therefore, a pair of gadgets is needed to first load the values and then transfer them to argument registers (cf. Section 6.3). Third, the overall number of gadgets that set argument registers is rather small; especially when excluding libc. This number becomes even smaller when an attacker aims to control multiple registers because—considering the size of the gadgets—there often are side effects that overwrite other registers. However, using symbolic execution we can easily filter gadgets for such side effects. Table 2 shows the maximum number of function arguments that can be controlled using ROP chains discovered by RiscyROP. These chains confirm that RiscyROP tackles the challenges described in Section 3, but also shows that the gadget space is very limited. For some argument registers all available gadgets in the application binary have side effects that overwrite other arguments, which makes it impossible to control *all* arguments of a function call with a ROP chain. When exploits require more argument registers, an attacker can use additional gadgets from the loaded libraries, in particular the libc.

6.3 Function Call Chain in the RISC-V libc

In what follow, we analyze the libc standard library from the official Ubuntu repository for RISC-V. We use RiscyROP to create ROP chains which one can use to make malicious functions calls with attacker controlled arguments. The libc is an important target for ROP as it is commonly loaded into the address space of applications and its size offers a broad gadget space, thus it was often subject of previous work on ROP attacks [40, 5, 8]. The ROP chain we found in the libc for RISC-V demonstrates how RiscyROP uses complex gadgets to construct impactful exploit chains.

Figure 4 shows a chain that was generated by RiscyROP to perform a 7-argument function call which has similar characteristics as the chain we found on RISC-V. Gadget 4 and 5 are a loader and a dispatcher gadget which represent the basic building blocks for ROP chains on RISC-V and ARM64. Gadget 4 loads values from the stack and gadget 5 moves these values into the argument registers and calls a function. The former is a function epilogue which restores the callee-saved registers `s0–s10` from the stack and the latter is typically found in indirect function calls. Our tool selects these gadgets as a starting point to build the chain (in reverse order) because it can control five out of the seven target registers used to transfer function arguments. RiscyROP needs to use both gadgets as the libc for RISC-V does not contain any gadget which directly loads all necessary argument registers from the stack.

Gadgets 2 and 3 have similar functionality as gadgets 4 and 5, i.e, they first load a value from the stack and then copy them into the desired registers. In this case, gadget 3 has additional side effects that are beneficial for the chain because they reduce the dependencies on the loader: First, one argument register is popped directly from the stack. Second, the preface of the indirect function call (`jalr a7`) in this gadget loads the address from the stack. RiscyROP detects this automatically as a stack-based return and prefers such gadgets because they add fewer dependencies. Third, it sets `a0` from `s4` which eases controlling the branch in gadget 4. Gadget 2 resolves the dependency of gadget 5 on `t1`. Notably, this gadget is special because it starts with an *unintended instruction* prior to the usual function epilogue. The `mv t1, ra` is a 2-byte instruction that is part of a 4-byte jump instruction.

Lastly, we need to control the remaining registers `ra` and `s4`, which is achieved with gadget 1. However, `ra` is not only used to control `t1`, but also to chain gadget 2 to gadget 1. The result is, that the value of the jump target `t1` in gadget 5 can only be the address of gadget 2 (otherwise gadget 1 would not have jumped to gadget 2). The symbolic execution engine finds this constraint and automatically executes gadget 2 again after gadget 5, which allows us to control the final jump target via the stack.

After executing this gadget chain and concluding that it is satisfiable (cf. Section 5.2), RiscyROP generates the exploit payload. Note that during the generation of this ROP chain, RiscyROP assured that there are no dependencies on the memory other than the stack, i.e., all memory accesses are controllable by the attacker. This counteracts a common problem of symbolic execution-based ROP chain generation [37], like angrop [44], namely that the resulting ROP chains have side effects that cause the process to crash.

6.4 Run time

We measure the run time to quantify the performance of RiscyROP demonstrating that an elaborated symbolic execution analysis is feasible for an attacker. We execute the run-time evaluation in a virtualized environment on an Intel Xeon Gold 6326 using 64 threads (32 cores) and 64 GB of RAM. Table 3 shows the results of the measurements for our eight targets. The first columns show the size of the executable sections within the binary which maps to the range the analysis operates on, the time of the initial analysis, and the

```
; gadget 1
0xb3058:  c.ldsp     s4,0x28(sp) ; load s4 (for condition
0xb305a:  c.ldsp     ra,0x8(sp)            ;  in gadget 4)
0xb305c:  c.addi     sp,0x10
0xb305e:  c.jr       ra


; gadget 2
0xc1e82:  c.mv       t1,ra       ; control t1
0xc1e84:  c.ldsp     ra,0x8(sp)
0xc1e86:  c.ldsp     s0,0x0(sp)  ; load a6
0xc1e88:  c.addi     sp,0x10
0xc1e8a:  c.jr       ra


; gadget 3
0xb3b88:  c.ldsp     a3,0x28(sp) ; load a3 (becomes a5 in G4)
0xb3b8a:  c.ldsp     a7,0x40(sp)
0xb3b8c:  c.mv       a4,s3
0xb3b8e:  c.mv       a6,s0       ; set a6
0xb3b90:  add        a5,s1,tp
0xb3b94:  c.mv       a2,s2
0xb3b96:  c.mv       a1,s5
0xb3b98:  c.mv       a0,s4       ; set a0 (for condition
0xb3b9a:  c.jalr     a7                    ;  in gadget 4)
```

```
; gadget 4
0x7ba5c:  c.mv       a5,a3         ; set a5
0x7ba5e:  bne        a5,a0,0x7ba4a ; conditional branch
0x7ba62:  c.ldsp     ra,0x58(sp)
0x7ba64:  c.ldsp     s0,0x50(sp)
0x7ba66:  c.ldsp     s1,0x48(sp)
0x7ba68:  c.ldsp     s2,0x40(sp)
0x7ba6a:  c.ldsp     s3,0x38(sp)   ; load a1
0x7ba6c:  c.ldsp     s4,0x30(sp)   ; load a3
0x7ba6e:  c.ldsp     s5,0x28(sp)
0x7ba70:  c.ldsp     s6,0x20(sp)   ; load a0
0x7ba72:  c.ldsp     s7,0x18(sp)   ; load a2
0x7ba74:  c.ldsp     s8,0x10(sp)   ; load a4
0x7ba76:  c.ldsp     s9,0x8(sp)
0x7ba78:  c.ldsp     s10,0x0(sp)
0x7ba7a:  c.addi16sp sp,0x60
0x7ba7c:  c.jr       ra

; gadget 5
0xb3510:  c.mv       a4,s8       ; set a4
0xb3512:  c.mv       a3,s4       ; set a3
0xb3514:  c.mv       a2,s7       ; set a2
0xb3516:  c.mv       a1,s3       ; set a1
0xb3518:  c.mv       a0,s6       ; set a0
0xb351a:  c.jalr     t1          ; jump to gadget 2

; gadget 2 again
0xc1e82:  c.mv       t1,ra
0xc1e84:  c.ldsp     ra,0x8(sp)
0xc1e86:  c.ldsp     s0,0x0(sp)
0xc1e88:  c.addi     sp,0x10
0xc1e8a:  c.jr       ra
```

Figure 4: ROP chain for the RISC-V libc to perform a 7-argument function call generated by RiscyROP in execution order.

number of resulting gadgets. Our results show that RiscyROP has a reasonable run time (minimum one hour, maximum 14 hours) to complete its analysis which is similar to other symbolic execution approaches [39, 37].

The relation of run time and binary size is roughly proportional. We observe that the progression during the analysis depends on the complexity of the analyzed code. For example, the analyses executed on the php binaries spent about 3.5 h in a (symbolically) complex code range that yielded few gadgets. However, the overall run time is still reasonable due to the fact that an attacker only needs to run the analysis once to perform an attack.

The last three columns refer to the chaining algorithm. Note that the current implementation of RiscyROP runs the gadget chaining single-threaded as this part only requires a few seconds. Table 3 denotes the number of argument registers that a chain can set prior to the function call, i.e., the maximum number of parameters the attacker-called function can have, and the total chain length (number of gadgets and instructions, respectively). Note that the chain for nginx on RISC-V is limited to three arguments because the binary does not contain a gadget to control `a3` (the 4th argument register). Nevertheless, our PoC exploit in Section 6.1 demonstrates that an attacker can still mount reasonable attacks because three arguments is sufficient to manipulate the memory permissions using `mprotect` to load arbitrary shellcode. Additionally, if an attacker only needs to control some of the function call parameters, e.g., arguments 1, 3, and 5 they can use RiscyROP to explicitly search for this particular subset of registers. However, for the sake of simplicity we do not include such chains in Table 3. The results for libphonenumber show that the chaining time can increase notably with a more constrained gadget space like this small binary. Since the default threshold time of 60 seconds only produced chains for 3 and 5 arguments, respectively, we continued the analysis to find chains that control even more arguments. Hence, Table 3 includes multiple chains for the libphonenumber binary. This is a result of the priority order of the algorithm, which prioritizes chains which include gadgets it deems most useful. In this case, however, the binary contains multiple equally ranked gadgets, each of which sets only a subset of the target registers. Therefore, RiscyROP has to find and chain multiple loader/dispatcher pairs (cf. Section 6.3) to set all registers simultaneously. In this process, RiscyROP has to execute all candidate chains symbolically to ensure that the final chain does not overwrite data we later need to successfully execute a function call which increases the overall run time. However, the chaining process still only requires a few minutes and therefore is not an issue for an attacker. The complexity of the resulting chains also shows that manual chaining of these complex gadgets is not feasible for binaries which are compiled for modern RISC architectures.

x86_64

```
; avoid side-effect (next xor)
pop rax  ; some known address
ret

; third argument (0x7)
pop rdx
xor [rax-0x77], cl
ret

; second argument (0x1000)
pop rsi
ret

; first argument (shellcode dst)
pop rdi
ret
```

RISC-V

```
c.ldsp      a0,0x8(sp)
c.j         0x41c6f4  ; constant jump
c.ldsp      ra,0x48(sp)
c.ldsp      s0,0x40(sp)
c.ldsp      s1,0x38(sp)
c.ldsp      s2,0x30(sp)
c.ldsp      s3,0x28(sp)
c.ldsp      s4,0x20(sp)
c.ldsp      s5,0x18(sp)
c.addi16sp  sp,0x50
c.jr        ra        ; return

c.addi4spn  s1,sp,0x0
c.ld        a1,0x8(s1)
c.ld        a2,0x10(s1)
c.ld        a3,0x18(s1)
c.ld        a4,0x20(s1)
c.ld        a5,0x28(s1)
sd          a6,0x0(a0)
c.sd        a1,0x8(a0)
c.sd        a2,0x10(a0)
c.sd        a3,0x18(a0)
c.sd        a4,0x20(a0)
c.sd        a5,0x28(a0)
sd          a0,0x2a8(s0)
c.ldsp      ra,0x18(sp)
c.ldsp      s0,0x10(sp)
c.ldsp      s1,0x8(sp)
c.mv        a0,s2
c.ldsp      s2,0x0(sp)
c.addi16sp  sp,0x20
c.jr        ra        ; return
```

ARM64

```
ldr   x0, [sp, #0x40]
b     #0x2e548  ; constant jump
adrp  x1, #0x122000
ldr   x1, [x1, #0xa40]
ldr   x2, [sp, #0x48]
ldr   x3, [x1]
subs  x2, x2, x3
mov   x3, #0
b.ne  #0x2e5bc  ; conditional jump
ldp   x21, x22, [sp, #0x20]
ldp   x23, x24, [sp, #0x30]
ldp   fp, lr, [sp], #0x50
br    lr        ; return

ldr   x4, [sp, #0x60]
cbnz  x0, #0xafa90
ldp   x21, x22, [sp, #0x20]
mov   x20, #0x1f4
ldp   x23, x24, [sp, #0x30]
ldp   x25, x26, [sp, #0x40]
ldp   x27, x28, [sp, #0x50]
adrp  x0, #0x122000
ldr   x0, [x0, #0xa40]
ldr   x1, [sp, #0x1d8]
ldr   x2, [x0]
subs  x1, x1, x2
mov   x2, #0
b.ne  #0xaff34  ; conditional jump
mov   x0, x20
ldp   x19, x20, [sp, #0x10]
ldp   fp, lr, [sp], #0x1e0
br    lr        ; return

mov   x2, x22
mov   x1, x19
mov   x0, x26
blr   x4        ; indirect call
```

Figure 5: Comparison of ROP chains in the nginx binary for the three architectures as required for a 3-argument function call to mprotect. The two RISC chains contain several jumps making the gadgets non-linear. The x86 ROP chain was part of the CVE-2013-2028 exploit.

6.5 Related ROP assistance and chaining tools

Several tools for both ROP gadget finding [6, 1, 35] and chaining [44, 37] have been proposed over the last years. However, none of the existing chaining approaches supports RISC-V or ARM64 binaries as they mainly focus on x86. In contrast, gadget finding tools often support RISC-V and ARM64 because the underlying disassembler supports these architectures and the finding algorithm itself is sufficiently generic and architecture-independent. The algorithms leveraged for gadget finding are commonly based on the Galileo algorithm and are well suited for the short, and in particular unaligned, instruction sequences of x86. However, the complex gadgets of the RISC architectures we analyze require further analysis to make chaining feasible. In addition, there are barely any unintended gadgets in RISC-V, where a pattern-based approach could be beneficial. To the best of our knowledge there does not yet exist any other gadget chaining approach specifically for RISC-V/ARM64. Hence, we focus our comparison on the gadget finding approaches.

6.5.1 radare2/rizin. radare2 [1] and its fork rizin [35] are well-known tools for binary analysis for different architectures which includes finding ROP gadgets. We categorize the results using the RISC-V libc as an example. radare2 returns 62 134 gadgets of which:

12 330 (19.8 %) are unaligned with respect to the architecture and are therefore false positives. This is to be distinguished from unintended gadgets, which are in the architecture's alignment of the instruction pointer, but originally part of a longer instruction.

6894 (11.1 %) contain invalid instructions, i.e., bytes that radare2 is unable to disassemble.

37 574 (60.5 %) terminate with a jump to a constant address or offset, which makes them not controllable by an attacker and therefore not chainable. However, RiscyROP was able to show that 3771 (6.1 %) of these gadgets jump to code which ultimately lead to an attacker controlled jump and thus can be used as part of a ROP chain. RiscyROP is able to provide such insights because the symbolic execution engine can calculate the jump targets and

Target	Version	Executable Bytes	Run time *Finding*	Gadgets	Run time *Chaining*	Chain Arguments	Chain Length/ Instructions
glibc-riscv	2.31	736 926	5.02 h	56 035	25 s	7	5 44
glibc-aarch64		984 372	4.21 h	62 344	35 s	7	3 41
nginx-riscv	1.18	526 452	3.37 h	45 391	8 s	3	2 24
nginx-aarch64		742 224	4.49 h	48 844	33 s	6	3 21
php-riscv	8.0.8	2 000 844	14.04 h	144 773	48 s	7	3 39
php-aarch64		2 590 364	9.81 h	136 284	49 s	7	3 44
libphonenumber-riscv	8.12.16-4.2	166 184	0.52 h	7761	15 s	5	4 50
					3 m 23 s	6	4 38
libphonenumber-aarch64		218 272	0.49 h	9165	12 s	3	3 33
					2 m 14 s	4	3 45
					11 m 13 s	5	4 64

Table 3: Run-time evaluation of RiscyROP. *Chain Arguments* denotes that the chain controls the first n argument registers.

thereby continue the analysis beyond these jumps which is not possible with simple gadgets finders as found in radare2/rizin. An attacker is unlikely to identify these without symbolic execution. Note that such gadgets are not found by more advanced tools such as angrop, which ignores all gadgets containing (conditional) jumps.

4272 (6.9 %) are matching gadgets with RiscyROP.

In summary, 53 027 (85.3 %) of the gadgets found by radare2 are unusable.

The results of radare2 for the *libc on ARM64* consist of 24 418 gadgets of which at least 16 319 (66.8 %) are unusable, 3726 (15.3 %) are valid when extended beyond a constant jump, and 3323 (13.6 %) match with RiscyROP.

Next, we discuss the number of additional gadgets that RiscyROP finds. For this comparison, we define *unique gadgets* as a sequence of instructions that is not a suffix of another gadget. This definition matches the results of the algorithm used by radare2, which includes overlapping gadgets if each contains a different instruction at the same offset, e.g., due to differently aligned instruction bytes. Using this definition, the database of RiscyROP contains 13 862 unique gadgets for the libc on RISC-V and 9955 for libc on ARM64, of which all are proven to terminate with an attacker-controlled jump. Thus, RiscyROP does not only exclude a considerable number of unusable gadgets compared to radare2/rizin, but also identifies additional gadgets.

6.5.2 angrop. angrop [44] is the ROP chaining tool featured in the angr suite and aims to support all the architectures supported by angr (which, at the time of writing, excludes RISC-V). Note that we implement RISC-V support in angr based on its PCode engine, which we further use to implement RiscyROP (cf. Section 5.1). While this enables angr to execute RISC-V code, angrop still fails to handle RISC-V because it depends on the VEX engine and is incompatible with the PCode engine.

In contrast, ARM64 is supported by the VEX engine and thus can be analyzed using angrop. However, the optimizations of angrop for x86 limit its usefulness when analyzing binaries compiled for ARM64. For example, by default angrop uses a maximum basic block size of 12 Byte (3 instructions) for the full binary analysis (fast mode) and 32 Byte (8 instructions) for partial code sections. This is clearly insufficient, considering the typical length of function epilogues on ARM64 (cf. Figures 4 and 5 and Table 3), and leads mainly to trivial `ret` sequences without any useful effect. We disable these optimizations and manually increase the maximum block size to a sufficiently large for all ARM64 binaries. However, even after this optimization angrop is still not able to find the complex gadgets required for a successful attack because the gadget validation does not utilize symbolic execution but static analysis. Hence, angrop rejects all gadgets containing a constant jump (prior to the final attacker-controlled jump). Additionally, ARM64 binaries contain gadgets—including those shown in Figure 5—which include conditional branches and thus are ignored by angrop.

7 DISCUSSION

In this section, we discuss several practical aspects of RiscyROP, how existing mitigation technologies such as control-flow integrity can be applied to RISC-V and ARM64 to prevent RiscyROP-based attacks, and leveraging RiscyROP for automated exploit generation.

Case Study: Trusted Execution Environments. A trusted execution environment (TEE) allows software developers to store and execute security-critical data and code inside a strongly isolated environment, often referred to as an enclave. Keystone [28] leverages the *physical memory protection* (PMP) unit of RISC-V to implement trusted execution and secure enclaves on RISC-V processors. This enables isolated processes that cannot be manipulated, even if the attacker has compromised the operating system or hypervisor. However, the software within the enclaves is still prone to memory-corruption vulnerabilities and control-flow hijacks [3], and the binary interface of enclaves exposes a wide attack surface, which together introduces a high risk of vulnerabilities [47, 10]. For instance, enclaves developed for Intel's TEE system, called Software Guard Extensions (SGX), have been recently compromised since fingerprint driver software suffered from several memory

corruption errors [10, 9]. In contrast to SGX, Keystone enclaves do not share the address space with the untrusted world, which makes null-pointer dereferences hardly exploitable. Further, Keystone enclaves cannot access the standard system calls, but only application-specific functions implemented in the trusted OS layer. This limits the privileged calls available to enclaves, and shellcode injection attacks based on manipulation of memory permissions are usually not possible. Hence, elaborated ROP chains are the only option.

To verify a ROP chain on Keystone, we exploit the attestation demo application running within a Keystone enclave on a HiFive Unleashed RISC-V development board. The purpose of the demo application is to receive a nonce and attest its integrity to a remote party via network. We add a stack-based buffer overflow vulnerability and use RiscyROP to generate a ROP chain that calls the enclave's output function with the address of the secret buffer. This confirms that RiscyROP is perfectly capable of generating ROP chains for Keystone applications.

Finding even more gadgets. Currently, we focus on gadgets that allow controlling the registers to function calls, which is the main prerequisite to launch ROP attacks. However, it might be also interesting to extend RiscyROP such that memory write gadgets are supported as this would allow us to use, e.g., automated string arguments.

RiscyROP for x86. RiscyROP solves unique challenges (cf. Section 3) to make ROP chaining based on the limited gadget space of RISC-V and ARM64 binaries feasible and is designed to expand the available gadget space as far as possible to make complex gadgets usable. Hence, other tools targeting x86 are not applicable to these architectures. One might consider to use RiscyROP for x86 architectures because the design of RiscyROP is not limited to RISC-V and ARM64 but could be used for other architectures as well. However, short and unaligned instruction sequences make the gadget space of unprotected x86 binaries so large that such an in-depth analysis is not necessary. In fact, related tools [39, 37] conclude that a random sample of available gadgets is sufficient for ROP chaining on x86. Then again, binaries may be protected by CFI schemes that limit valid target addresses for control-flow transfers and thus reduce the available gadget space for ROP chains [15]. The result is that fewer gadgets are available, and they tend to be longer, more complex, and carry more side effects. In this case, the in-depth analysis of RiscyROP could be leveraged to compile ROP chains despite CFI.

Mitigations. Static CFI [26, 46], dynamic CFI [25, 48, 17] and pointer authentication [29, 18] are the main solutions to mitigate control-flow hijacking attacks. Both dynamic or static CFI focus on reducing the potential targets of indirect branch and therefore limit which gadgets can be chained. However, due to the well-known limits of program analysis for determining the valid targets of a control-transfer instruction, there is still a non-negligible amount of potential targets which make control-flow hijacking possible. Current pointer authentication solutions for ARM are based on a new processor feature, which protects the integrity of data and code pointers via a hash value encoded in the high bits of the address. In this way, the target address is checked before an indirect branch. Intel introduced Control-Flow Enforcement Technology (Intel CET) to defend against control-flow hijacking attacks. However, no similar native hardware anti-control-flow hijacking reinforcements have been introduced by RISC-V yet. Porting a similar scheme to RISC-V requires hardware changes [34, 16] or incurs additional performance overhead. For example, pointer authentication can be simulated by using the RISC-V hypervisor extension, which uses a hyper call for every calculation and validation of every pointer hash value and traps into the hypervisor to do the task. Even though it is convenient to issue a hyper call in both user space and kernel space, when deployed by large applications, it still costs non-negligible performance overhead. Simple stack canaries and shadow stack schemes can protect the return addresses from being smashed via a stack overflow vulnerability. However, as there is a sequence of register restoring instructions (*ldsp* instruction in Figure 4) between the check point and the real return instruction, the attacker is still able to reuse the return-based gadgets. The stack canary and shadow stack is not able to deal with call-oriented attack either. Prior works [25, 26] protect indirect call targets in the source code level regardless of the underlying architecture which includes RISC-V.

Automatic Exploit Generation (AEG). Symbolic execution is often used for automatic exploit generation [41, 2] for which an attacker can leverage RiscyROP. However, AEG is still an active research topic and the prevalence of security defenses (W⊕X, ASLR, stack canaries, CFI, sandboxing, secure enclaves) make AEG for end-to-end implementation use-cases increasingly harder. Thus, recent tools focus on specific challenges within an end-to-end scenario which can be combined to a complete AEG framework. Similarly, RiscyROP focuses on ROP chains and is capable of automatically generating exploit payloads based on a known vulnerability and assuming that additional defenses are bypassed, e.g., using an information leak, stack pivoting, or similar techniques.

8 RELATED WORK

For the x86 architecture Hund et al. [22] as well as Schwartz et al. [39] build programs that are able to automate ROP attacks. Both implementations use methods similar to the Galileo algorithm to search for ROP gadgets in vulnerable programs and are therefore limited in their usefulness for modern RISC processors such as RISC-V (see Section 3), which is the main focus of RiscyROP. Furthermore, these existing systems do not support pre-conditions for gadgets and therefore ignore gadgets that, e.g., copy a value into a register and subsequently jump to the address in that register. Our implementation makes use of symbolic execution to find suitable gadgets. Hence, we support any gadget that jumps to an address which can be controlled by the attacker.

BOPC [23] is a tool to generate data-only exploits for x86 64 bit. It uses symbolic execution to identify Data-Only Programming (DOP) gadgets in target binaries and compile CFI-conforming execution paths that emulate a given (Turing-complete) computation. The evaluation of BOPC shows great success at the emulation of isolated calculations using DOP gadgets, however, function call gadgets like *execve* are less likely to be found. While BOPC claims to be theoretically architecture independent, it is only implemented for x86 64 bit, and it relies on angr's VEX engine that does not support RISC-V. Hence, we could not evaluate its capabilities on our target

binaries. The extent to which the challenges of RISC, particularly the large number of dedicated registers, impact the DOP gadget space is part of future work. Lastly, DOP attacks are often not necessary in real-world scenarios since fine-grained CFI techniques are not commonly implemented. Thus, ROP exploits are still the prevalent method when exploiting a binary.

In academia, Jaloyan et al. [24] were the first to analyze ROP attacks on the RISC-V architecture. The authors show that it is possible to find gadgets in overlapping instructions and unintended compressed instructions which an attacker can misuse to hide a backdoor inside a program. Jaloyan et al. present an algorithm that simply disassembles the RISC-V code at any possible offset to find all existing code paths. They successfully attack a proof-of-concept binary with hidden ROP gadgets inside dummy functions. In contrast, our work does not focus on hidden backdoors in specially crafted binaries but rather on building automated fully-functional ROP chains for real-world programs. Thus, we have to solve different challenges. Namely, we have to work with code that does not include purposefully inserted gadgets, but only instructions generated by compilers from benign source code. Therefore, we implement an algorithm based on symbolic execution that can handle the complex side effects due to the added complexity for ROP chains in RISC-V.

Gu et al. [21] discuss the capabilities of ROP attacks on RISC-V using a libc binary. In contrast, we automate the analysis and selection of gadgets as well as taking care of side effects. Thus, our approach extends the usable gadget space and takes minimal manual effort when attacking other binaries. Furthermore, we generalized RiscyROP to also work on ARM64 and with any type of gadget, not just the ones ending in a return instruction in function epilogues.

Buchanan et al. [5] investigate the possibilities of ROP attacks on RISC architectures, more specifically the SPARC architecture. Contrary to ARM64 and RISC-V, SPARC processors use different sets of registers to store stack frame data and therefore have different requirements when building ROP chains. Furthermore, Buchanan et al. do not use any automation to take care of potential side effects and use only return-based gadgets, while our approach utilizes any attacker-controlled control-flow transfer instruction.

Kornau [27] describes how to perform ROP attacks on binaries compiled for the ARM32 architecture. The author discusses different basic gadget types for ROP attacks on ARM32 as well as the differences to attacks targeting the Intel x86 architecture. To find gadgets inside an ARM binary, Kornau uses an algorithm similar to the Galileo algorithm. Namely, it looks for a particular instruction (e.g., a return-like instruction as part of the function epilogue) and then traverses the instructions in reverse execution order. Davi et al. [13] and Checkoway et al. [8] also worked on ROP attacks on ARM32 and show that ROP attacks can be executed by exclusively using gadgets that end with an indirect jump instead of a return-like instruction, specifically the `blx` instruction, which is used to jump from one gadget to another. The required gadgets, including the chaining, have been constructed with a manual approach. Davi et al. [12] present a ROP chain to successfully attack an Android system running a vulnerable application. Contrary to these proposals, we focus on ROP attacks for the modern architectures, namely ARM64 as well as RISC-V instead of ARM32, which come with a different set of challenges. For example, the program counter on ARM64 and RISC-V is not a general purpose register as on ARM32 which prohibits useful ROP instructions such as `pop pc`. Therefore, ROP gadgets and resulting chains are more complex, which benefits our approach that automatically takes care of the added complexity via symbolic execution. Furthermore, RiscyROP finds and chains gadgets automatically, and not only detects gadgets that end in indirect jumps or simple return instructions, but further generalizes the search for gadgets by utilizes symbolic execution to extend the number of usable gadgets to *any* instruction that transfers the control flow to a target under the attacker's control.

Several tools such as radare2 [1] and ROPGadget [36] support searching for gadgets in RISC-V as well as ARM64. However, these tools commonly only support basic disassembly and pattern matching for RISC-V and ARM64, and are not adapted to the more specific characteristics of modern RISC architectures. Furthermore, they leverage designs based on the Galileo algorithm to find gadgets inside a binary. Hence, they are limited in finding gadgets [24] and cannot handle side effects of function epilogues. Additionally, they do not make use of symbolic execution to automatically find preconditions and complex gadgets. Therefore, these tools do not provide the same level of assistance as RiscyROP (see Section 6.5) which cannot only find gadgets but also generate ROP chains automatically.

9 CONCLUSION

Return-oriented programming (ROP) is the state-of-the-art memory corruption attack technique. However, it is unclear to which extent modern and emerging RISC architectures are vulnerable to ROP. Our analysis on RISC-V and ARM64 real-world binaries demonstrates that the gadget space available to the attacker is significantly reduced compared to x86 and ARM32 due to architectural differences. Further, available code sequences introduce many side effects that are almost impossible to resolve manually. As a consequence, new analysis approaches need to be developed to understand the risks of ROP attacks on these architectures. To do so, we develop the first automated ROP chain toolkit for RISC-V and ARM64. Our approach, called RiscyROP, uses symbolic execution to accurately determine the gadget space and automatically generate complex multi-stage chains for arbitrary function calls. We use RiscyROP to automatically generate working ROP chains for various real-world programs compiled for RISC-V and ARM64, including the standard library libc.

ACKNOWLEDGMENTS

This work has been partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972 and SFB 1119 – 236615297 within project S2.

REFERENCES

[1] Sergi Alvarez. *radare2*. 2008. URL: https://rada.re/n/radare2.html.

[2] Thanassis Avgerinos et al. "Automatic Exploit Generation". In: *Communications of the ACM* (2014).

[3] Andrea Biondo et al. "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *USENIX Security Symposium*. 2018.

[4] Tyler Bletsch et al. "Jump-Oriented Programming: A New Class of Code-Reuse Attack". In: *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2011.

[5] Erik Buchanan et al. "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC". In: *ACM Conference on Computer and Communications Security (CCS)*. 2008.

[6] Amat Cama. *xrop*. 2017. URL: https://github.com/acama/xrop.

[7] Nicholas Carlini et al. "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity". In: *USENIX Security Symposium*. 2015.

[8] Stephen Checkoway et al. "Return-Oriented Programming without Returns". In: *ACM Conference on Computer and Communications Security (CCS)*. 2010.

[9] Tobias Cloosters et al. "SGXFuzz: Efficiently Synthesizing Nested Structures for SGX Enclave Fuzzing". In: *USENIX Security Symposium*. 2022.

[10] Tobias Cloosters et al. "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *USENIX Security Symposium*. 2020.

[11] *nginx-1.4.0 for the analysis of CVE-2013-2028*. 2013. URL: https://github.com/danghvu/nginx-1.4.0 (visited on 03/19/2022).

[12] Lucas Davi et al. "Privilege escalation attacks on Android". In: *International Conference on Information Security*. Springer. 2010.

[13] Lucas Davi et al. *Return-Oriented Programming without Returns on ARM*. Tech. rep. HGI-TR-2010-002. 2010. URL: https://www.ais.rub.de/media/trust/veroeffentlichungen/2010/07/21/ROP-without-Returns-on-ARM.pdf.

[14] Lucas Davi et al. "ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks". In: *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2011.

[15] Lucas Davi et al. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". In: *USENIX Security Symposium*. 2014.

[16] Asmit De et al. "FIXER: Flow integrity extensions for embedded RISC-V". In: *IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019.

[17] Ren Ding et al. "Efficient Protection of Path-Sensitive Control Security". In: *USENIX Security Symposium*. 2017.

[18] Reza Mirzazade Farkhani et al. "PTAuth: Temporal Memory Safety via Robust Points-to Authentication". In: *USENIX Security Symposium*. 2021.

[19] Aurélien Francillon et al. "Code Injection Attacks on Harvard-Architecture Devices". In: *ACM Conference on Computer and Communications Security (CCS)*. 2008.

[20] Google. *opentitan*. 2022. URL: https://opentitan.org/.

[21] Garrett Gu et al. *No RISC No Reward: Return-Oriented Programming in RISC-V*. 2020. URL: https://arxiv.org/abs/2007.14995.

[22] Ralf Hund et al. "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms". In: *USENIX Security Symposium*. 2009.

[23] Kyriakos K Ispoglou et al. "Block oriented programming: Automating data-only attacks". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2018.

[24] Georges-Axel Jaloyan et al. "Return-Oriented Programming on RISC-V". In: *ACM Asia Conference on Computer and Communications Security (AsiaCCS)*. 2020.

[25] Mustakimur Rahman Khandaker et al. "Origin-sensitive control flow integrity". In: *USENIX Security Symposium*. 2019.

[26] Sun Hyoung Kim et al. "Refining indirect call targets at the binary level". In: *Symposium on Network and Distributed System Security (NDSS)*. 2021.

[27] Tim Kornau. "Return oriented programming for the ARM architecture". MA thesis. Ruhr-University Bochum, 2009. URL: https://zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf.

[28] Dayeol Lee et al. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *European Conference on Computer Systems (EuroSys)*. 2020.

[29] Hans Liljestrand et al. "PACStack: an Authenticated Call Stack". In: *USENIX Security Symposium*. 2021.

[30] Ted Marena. "RISC-V: high performance embedded SweRV™ core microarchitecture, performance and CHIPS Alliance". In: *Western Digital Corporation* (2019).

[31] Leonardo de Moura et al. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. 2008.

[32] *nginx*. 2019. URL: https://nginx.org/.

[33] Vasilis Pappas et al. "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization". In: *IEEE Symposium on Security and Privacy (S&P)*. 2012.

[34] Seonghwan Park et al. "Bratter: An Instruction Set Extension for Forward Control-Flow Integrity in RISC-V". In: *Sensors* (2022).

[35] *rizin*. 2020. URL: https://rizin.re/.

[36] Jonathan Salwan. *ROPgadget Tool*. 2011. URL: http://shell-storm.org/project/ROPgadget/.

[37] Moritz Schloegel et al. "Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains". In: *European Symposium on Research in Computer Security (ESORICS)*. 2021.

[38] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". In: *IEEE Symposium on Security and Privacy (S&P)*. 2015.

[39] Edward J Schwartz et al. "Q: Exploit hardening made easy". In: *USENIX Security Symposium*. 2011.

[40] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)". In: *ACM Conference on Computer and Communications Security (CCS)*. 2007.

[41] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy (S&P)*. 2016.

[42] Kevin Z. Snow et al. "Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization". In: *IEEE Symposium on Security and Privacy (S&P)*. 2013.

[43] László Szekeres et al. "SoK: Eternal War in Memory". In: *IEEE Symposium on Security and Privacy (S&P)*. 2013.

[44] angr Team. *angrop*. 2014. URL: https://github.com/angr/angrop/.

[45] PaX Team. *PaX non-executable pages design & implementation*. 2003. URL: https://pax.grsecurity.net/docs/noexec.txt.

[46] Caroline Tice et al. "Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM". In: *USENIX Security Symposium*. 2014.

[47] Jo Van Bulck et al. "A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes". In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[48] Victor Van der Veen et al. "Practical context-sensitive CFI". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2015.

[49] Chao Zhang et al. "Practical Control Flow Integrity and Randomization for Binary Executables". In: *IEEE Symposium on Security and Privacy (S&P)*. 2013.