ropbot: Reimaging Code Reuse Attack Synthesis

Kyle Zeng[†], Moritz Schloegel[‡], Christopher Salls[§]
Adam Doupé[†], Ruoyu Wang[†], Yan Shoshitaishvili[†], Tiffany Bao[†]
[†]*Arizona State University,* [‡]*CISPA Helmholtz Center for Information Security,* [§]*UC Santa Barbara*
[†]*{zengyhkyle, doupe, fishw, yans, tbao}@asu.edu,* [‡]*schloegel@cispa.de,* [§]*salls@ucsb.edu*

***Abstract*—Code reuse attacks are one of the most crucial cornerstones of modern memory corruption-based attacks. However, the task of stitching gadgets together remains a time-consuming and manual process. Plenty of research has been published over the past decade that aims at automating this problem, but very little has been adopted in practice. Solutions are often impractical in terms of performance or supported architectures, or they fail to generate a valid chain. A systematic analysis reveals they all use a *generate-and-test* approach, where they first enumerate all gadgets and then use symbolic execution or SMT solvers to reason about which gadgets to combine into a chain. Unfortunately, this approach scales exponentially to *the number of available gadgets*, thus limiting scalability on larger binaries.**

In this work, we revisit this fundamental strategy and propose a new grouping of gadgets, which we call ROPBlock, that exhibit one crucial difference to gadgets: ROPBlocks are guaranteed to be chainable. We combine this notion of ROPBlock with a graph search algorithm and propose a gadget chaining approach that significantly improves performance compared to prior work. We successfully reduce the time complexity of setting registers to attacker-specified values from $\mathcal{O}(2^n)$ to $\mathcal{O}(n)$. This yields a 2–3 orders of magnitude speed-up in practice during chain generation. At the same time, ROPBlocks allow us to model complex gadgets—such as those involving ret2csu or with conditional branches—that most other approaches fail to consider by design. And as ROPBlocks are architecture-agnostic, our approach can be applied to diverse architectures.

Our prototype, ropbot, generates complex, real-world chains invoking `dup-dup-execve` within 2.5s on average for all 37 binaries in our evaluation. All but one other approach fails to generate any chain for this scenario. For `mmap` chains, a difficult scenario that requires setting six register values, ropbot finds chains for 5x more targets than the second-best technique. To show its versatility, we evaluate ropbot on x64, MIPS, ARM, and AArch64. We added RISC-V support in less than two hours by adding twelve lines of code. Finally, we demonstrate that ropbot outperforms all existing tools on their respective datasets.

I. Introduction

Code Reuse (CR) attacks, such as Return-Oriented Programming (ROP) [1] or Jump-Oriented Programming (JOP) [2], are exploitation techniques that can turn a control-flow hijacking primitive into arbitrary code execution. Despite the presence of advanced protections, such as Address Space Layout Randomization (ASLR), Position-Independent Executable (PIE), Control-Flow Integrity (CFI), and others, CR attacks remain highly relevant and are becoming increasingly more powerful with fewer requirements on the target programs. For example, researchers have demonstrated ways to perform code reuse attacks without the need for `ret` instructions [2]–[4], and attackers can inject executable code into the target process and utilize on-the-fly gadgets [5].

However, synthesizing CR attacks is not only for malicious exploitation: it is a vital approach in the defender's arsenal to confirm exploitability, which is critical in prioritizing fixing severe bugs. At the time of writing, 52 out of 70 exploits accepted by Google's KernelCTF Vulnerability Rewards Program [6] use CR attacks to perform privilege escalation in the Linux kernel to demonstrate the exploitability of the reported vulnerabilities. Despite widespread use by both malicious and benign actors, constructing a CR payload (called a *chain*) is non-trivial, manual, and notoriously time-consuming. This unnecessarily delays the process of exploitability verification and requires domain experts. Hence, research has proposed to automate the task of generating CR chains. Attempts to solve this problem started with Q [7] and have seen much follow-up research, resulting in JIT-ROP [5], LIMBO [8], SGC [9], RiscyROP [10], Crackers [11], Arcanist [12] and more. Surprisingly, no approach has achieved widespread adoption in practice, and the problem of CR payload synthesis remains.

When carefully studying the underlying techniques of these tools, we find that all existing works rely on a *generate-and-test* (GAT) [13] scheme: given a target goal (*e.g.*, set registers `rax` and `rbx` to attacker-specified values), they first generate a list of gadgets that can potentially achieve the goal and then use a heavyweight approach, *e.g.*, symbolic execution or SMT solvers, to verify whether the list of gadgets can be chained and achieve the desired goal. This algorithm results in a time complexity of $\mathcal{O}(2^n)$, where n is the number of gadgets. Thus, it does not scale well with large binaries that have a large number of gadgets. As a balance between capability and speed, most existing works [9]–[12] decide to solve a *sub-problem* instead: they aim to find a chain that uses gadgets up to a maximum number (N). We call this approach

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240845
www.ndss-symposium.org

bounded generate-and-test (BGAT). This algorithm has a time complexity of $\mathcal{O}(n^N)$, which leads to the dilemma of choosing N. If N is too large, the algorithm will face state explosion; if N is too small, the tool cannot find useful chains. Despite known issues, the GAT scheme is still being used because techniques must (1) ensure all selected gadgets are chainable and (2) ensure that their effects do not conflict with each other but satisfy the attacker's goal.

With this observation in mind, we revisit the CR chain generation problem from a different perspective: what if we could group gadgets together in a way that guarantees chainability? Such a grouping bypasses chainability issues, and we can focus on finding groups that can satisfy the attack goal.

In this work, we present such a novel abstraction of gadgets that we call ROPBlock: a grouping of gadgets that guarantees chainability *between* ROPBlocks. This layer of abstraction reduces the classic CR chain generation task, namely register setting, into an $\mathcal{O}(n)$ graph search problem. In practice, it proves to be 2–3 orders of magnitude faster than the GAT approach widely used by *all* existing works [8]–[12], as demonstrated in our large-scale evaluation. In addition to performance improvements, working on the intermediate layer of ROPBlocks allows us to model complex gadget types, such as gadgets with conditional branches, ret2csu [14] gadgets, or GOT gadgets [15]. This significantly enhances the capabilities and real-world relevance of our technique. The concept of ROPBlock is architecture-agnostic (as is our algorithm to search for chains), which allows our approach to be applied to many architectures.

Based on our ROPBlock abstraction, we implemented a research prototype named ropbot that can generate CR chains in an architecture-agnostic manner. It is the first approach to support x86, x64, MIPS, ARM, AArch64, and RISC-V for both little-endian and big-endian. Our evaluation shows that even on x64, an architecture that has been extensively tested in the past, ropbot outperforms the state-of-the-art significantly for every CR chain generation task on the datasets proposed by prior work. For the task of generating `execve` chains, ropbot achieves a success rate of 89.7%, compared to 33.1% of the second-best technique, exrop. When trying to invoke `mmap`, which requires setting six arguments, ropbot finds CR chains for 5x targets compared to the second-best technique.

Beyond these evaluations, ropbot is ready for real-world use and can generate complex CR chains. For example, it can generate chains that invoke three function calls (`dup-dup-execve`) in 3.7s on average. ropbot has already been adopted in the real-world and is used by Google [16], [17], multiple vulnerability research companies, and serves as the foundation of some research prototypes [18], [19].

Contributions. We make the following key contributions:

- We revisit the challenges of synthesizing code reuse chains in an architecture-agnostic fashion to study the issue of efficiently synthesizing CR chains.
- Based on this analysis, we propose a new approach to understanding code reuse gadgets: an abstraction layer called ROPBlock that significantly reduces the complexity of CR chain generation.
- Building on all datasets used in prior work, we release a comprehensive dataset and evaluation framework that can assist the evaluation for future CR techniques.
- We implement our ROPBlock approach into a tool called ropbot, which, according to our evaluation, is able to effectively synthesize CR chains faster than prior work, on more/larger targets than prior work, and across more architectures than prior work.

To foster future research, we open source the artifact of ropbot at https://github.com/sefcom/ropbot.

II. Challenges

This paper aims to solve the problem of *automatically synthesizing a code reuse chain*: Given a vulnerable binary program with a control flow hijacking primitive and a target payload computation, our goal is to automatically generate a CR payload that carries out the payload computation in the vulnerable binary. Despite being around for more than a decade, there are many open challenges towards automatically synthesizing CR chains. We separate three different categories: diverse gadget type support, cross-architecture gadget chaining, and state exploration.

1) Diverse Gadget Types Support: Traditionally, CR attacks were limited to ROP, where a gadget is a series of instructions that ends in `ret`. As CR techniques evolved, more gadget types have been discovered. Examples include the gadgets used in the return-to-csu attack [14] and GOT gadgets [15]. These gadgets end with `call [mem]` or `jmp [mem]` and exist extensively in programs; for example, the gadget `mov rdx,r13; mov rsi,r14; mov edi,r15d; call [r12+rbx*8]` is inserted into *all* dynamically linked ELF executables by compilers. Combined with another gadget that sets the registers, it can be used to invoke a pointer in memory with attacker-controlled arguments. Despite giving an attacker extensive control, the required chaining of gadgets is too complex and most automated gadget chaining techniques currently ignore such gadgets ending in `call [mem]` entirely [10], [20] or impose additional constraints [9], [11], [12], such as needing an information leak.

One intuitive solution to automatically leverage such gadget types would be to dynamically generate them on-the-fly, i.e., to write the needed gadgets into memory first, and then invoke them. Although a presumably minor change, the generate-and-test (GAT) algorithm underlying all state-of-the-art works [8]–[12], [20] cannot handle it. Fundamentally, this is because the first step is to generate a list of available gadgets, which they need to ensure are *unconditionally chainable*. For example, for `pop rax; ret | pop rbx; ret`, no matter the values of `rax` and `rbx`, these two gadgets are chainable. But first writing data to memory and then invoking it as a gadget later creates a *conditionally chainable* situation. In other words, the list of gadgets may be chainable

```
block_A:            block_B:
    pop rax             ret
    jmp block_B
```

Listing 1: Gadget consisting of more than one basic block.

depending on *other* values in the chain. For example, consider the following list of the gadgets: `pop rax; ret| pop rbx; ret| mov [rax], rbx; ret| pop rcx; ret| pop rdi; call [rcx]`. They are only chainable if `rax == rcx`, and `rcx` is a gadget that cleans up the return address pushed to stack by the `call` instruction. Without the additional constraints, neither symbolic execution nor SMT solvers can verify the feasibility of this chain, thus rejecting it. As a result, existing works cannot model ret2csu and GOT gadgets without additional primitives.

2) Gadget Identification across Architecture: Many existing CR chaining tools [9], [21], [22] search for gadgets using the Galileo Algorithm [1]: They first identify "anchor instructions", such as `ret` or `jmp` on x86, and then search backwards to find potential gadgets. However, there are a few shortcomings associated with this approach:

- **Overlooking gadgets:** The intuition behind using such "anchor instructions" is that gadgets *usually* end with such instructions. However, not all potential gadgets necessarily do. For example, although most gadgets in ARM end with `bx` and `pop pc`, there is a significant number of instruction sequences that contain `ldr pc, [sp], #<X>`. While ignored by existing works [21], [22], they should be considered as gadgets.
- **Requiring domain knowledge and manual effort:** It is potentially possible to exhaustively collect all instructions that may serve as the end of a gadget. However, this approach requires domain knowledge and manual effort to support each architecture. Also, "anchor instructions" may differ across different compilers or calling conventions.
- **Overlooking multi-block gadgets:** Consider the example in Listing 1, where block A ends in a direct jump to block B, which ends in a `ret`. The reverse search algorithm will only mark block B as a gadget. In reality, any execution starting from block A will also end with `ret`, extending the pool of available gadgets. The underlying issue here is the reverse search starting from an "anchor instruction".

How to find all potential gadgets with minimal or no domain knowledge becomes even more challenging when attempting to find gadgets for more than one architecture. Currently, most tools support limited architectures.

3) Payload Chaining across Architectures: Payload chaining is an essential functionality to achieve arbitrary exploitation goals. For example, we may want to chain two payloads, where the first writes "/bin/sh" into memory and the second then invokes `execve` with "/bin/sh". Ideally, we have an architecture-agnostic solution to do so. However, previous work either does not support payload chaining [9]–[12] or supports only x86/x64 [20]. Here, chains can be just concatenated as strings due to its architectural design: for example, to combine two chains, `pop rax; ret | 0x41414141` and `pop rbx; ret | 0x42424242`, we can just concatenate them into `pop rax; ret | 0x41414141 | pop rbx; ret | 0x42424242`. This is no longer a trivial task when considering other architectures, such as ARM and AArch64. Consider the following ARM-based chain: `ldr r0, [sp,#4]; ldr lr,[sp],#4; add sp,#8; bx lr | 0x0 | 0x41414141`. This chain loads `r0` from `[sp+4]`, `lr` from `[sp]`, and then increases `sp` by 12. If we want to chain it with `pop.w {r1, lr}; bx lr | 0x42424242` to set both `r0` and `r1`, the final chain needs to be like this: `gadget1 | gadget2 | 0x41414141 | 0 | 0x42424242`, which is completely different from the linear layout that we usually see on x86/x64. Besides the difference in instruction sets, calling convention is another issue making payload chaining hard on non-x86 architectures. On AArch64, functions end with `jmp x30` (still rewritten as `ret`). In other words, invoking a function and continuing execution now involves two steps: set `x30` to a gadget we want to execute after the function call and then invoke the function.

In summary, different architectures and calling conventions make generic, cross-architecture payload chaining challenging.

III. Design

We design ropbot to overcome these challenges. Our design's central component is a new abstraction layer, *ROPBlocks*, which essentially is a special grouping of gadgets, such that any ROPBlock is guaranteed to be chainable. We then build a CR chain on top of ROPBlocks, deviating from the common wisdom that "chains consist of gadgets". In the following, we first discuss our threat model, define ROPBlocks, and then discuss how to find them in binaries and use graph-based search algorithms to generate CR chains.

A. Threat Model

We assume a threat model aligned to real attacks, where attackers control only the payload placed on the stack. This is the same threat model as exrop [20] uses. We specifically do *not* follow prior work here that assumes an attacker additionally knows the location of the payload [9], [12] or can even set arbitrary memory values [11] before executing the chain. These assumptions may not hold for real-world attacks and require additional primitives. Besides this difference, ropbot shares the same assumption as other tools: attackers know the address of executable code (by bypassing ASLR or disabling PIE) and backward-edge CFI (e.g., PAC, CET) is absent.

B. The Concept of ROPBlocks

ROPBlocks are our way of grouping gadgets, with their key property being guaranteed chainability. We design them based on an observation we made across CR chains, a concept we call *stack patch*.

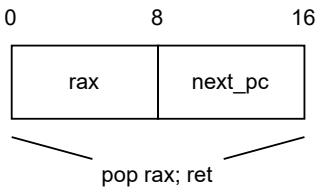

Fig. 1: Stack patch of a x64 gadget.

Stack Patch. Studying CR attacks, we observe that they consist of "frames", akin to function stack frames. To avoid ambiguity, we call them *stack patches* in the context of CR chains. More precisely, we define a stack patch as the data on the stack, bounded by the stack pointer *before* and *after* executing the gadget. The *change* of the stack pointer before and after gadget execution is referred to as *stack change*. In Figure 1, the gadget `pop rax; ret` is associated with a stack patch of 16 bytes (and the stack change is 16). During execution, it will pop the first 8 bytes into `rax` and then pop the second 8 bytes into PC, which enables gadget chaining.

Interestingly, a gadget does not guarantee a positive stack change (*e.g.*, `xor rax, rax; call rbx`), which implies there are gadgets *without* a stack patch. As an attacker controls solely the stack patch, these gadgets introduce additional constraints for chaining. To avoid the associated complexity, we propose a new concept, ROPBlocks.

ROPBlock. We say a gadget (or a sequence of gadgets) is a *ROPBlock* if and only if it

- has a positive stack change (thus having a stack patch),
- takes the PC from the stack patch, and
- does not conditionally branch.

By definition, a ROPBlock loads the PC from the stack patch, a region controlled by the attacker, thus, it is guaranteed to be chainable with any other ROPBlock. With this in mind, it is clear that some gadgets are ROPBlocks by definition (called *self-contained* gadgets), while others are not. For example, `pop rax; ret` is a ROPBlock while `pop rbx; jmp rax` is not (it takes the PC from `rax`). `cmp rax, 1; jnz <label>; pop rbx; ret` is not a ROPBlock either due to the conditional branch.

Crucially, however, gadgets that are *not* self-contained, i.e., that are no ROPBlocks on their own, can be turned into ROPBlocks by combining them with other ROPBlocks. For example, the gadget `cmp rax, 1; jnz <label>; pop rbx; ret` is not a ROPBlock. But if combined with a `pop rax; ret | 1` chain, it will no longer conditionally branch (but instead always take the same branch, as `rax` is set to `1`), thus becoming a ROPBlock.

We emphasize that part of a ROPBlock's properties is an implicit requirement in many past works. For example, SGC, exrop, and Crackers filter out gadgets containing conditional branches. RiscyROP and exrop do not utilize gadgets ending with `call [mem]` or `jmp [mem]` because the PC does not come from the stack.

C. Workflow

With the concept of ROPBlocks in mind, we focus on ropbot's workflow that consists of three steps:

1) Collect ROPBlocks by finding gadgets and identifying self-contained gadgets as the initial set of ROPBlocks.
2) Generate more ROPBlocks by turning non-self-contained gadgets into ROPBlocks by combining them with existing ROPBlocks.
3) Use a graph search-based algorithm to find a list of ROPBlocks that can achieve the attacker goal and turn it into a chain. This requires ROPBlocks to be chainable.

D. ROPBlock Collection

In the first step, ropbot finds gadgets in the target binary and identifies gadgets that are already ROPBlocks (self-contained gadgets).

1) Gadget Identification: To find gadgets in the target binary, we propose an algorithm that differs from the Galileo algorithm used by all prior work. We want to identify gadgets *without* heuristics or manually crafted "anchor instructions".

In this work, a gadget is a sequence of instructions that exists in the target binary, executes in order, and can potentially set the next PC to a fully attacker-controlled value. The relaxed notion of *potentially* setting the PC allows for including conditional branches. We implement our gadget identification using symbolic execution: given an executable address, ropbot symbolically executes a fully symbolized state (symbolized register, stack, and memory) starting from that address and checks whether the code moves any symbolic value into the PC. If so, it marks the starting address with that execution history as a gadget (one starting address may lead to different execution histories).

This new gadget identification algorithm has multiple advantages:

- It is architecture-*agnostic* and does not require architecture-specific domain knowledge.
- It allows us to find *all* instruction sequences that can be potentially used to an attacker's advantage to execute payload logic while maintaining the control flow, including ret2csu gadgets and GOT gadgets.
- It can find gadgets spanning multiple basic blocks, whereas the Galileo Algorithm is limited to one block. This leads to more usable gadgets.

When given a binary, ropbot exhaustively analyzes all executable addresses to find all potential gadgets using the aforementioned gadget identification algorithm. This algorithm is slow due to the speed of symbolic execution. To mitigate the impact, we design a caching mechanism. Essentially, we only analyze position-independent gadgets once and cache them. To determine whether a gadget is position-independent, we move the bytes into another address and see whether constants in the gadget change. For example, `pop rax; ret` will be considered a position-independent gadget because nothing changes if we move the bytes to another address. However, `lea rax, [rip+0x100]; jmp rax` will be considered

a position-*dependent* gadget, because `[rip+0x100]` will be resolved into another constant when the gadget is placed at a different address. Thus, `lea rax, [rip+0x100]; jmp rax` will not be cached, allowing us to reach different code regions.

2) Gadget Effect Analysis: For each found gadget, we create a fully symbolized state (symbolized registers, stack, and memory) and execute the gadget. By tracking the symbolic execution history, we can track the memory accesses and conditional branch information about the gadgets. By comparing the symbolic states before and after executing the gadget, we can also obtain the gadget's effects on registers.

Specifically, the stack change is calculated by subtracting the stack pointers, which is important for building stack patches later. We also track the register popping, moving, and changing behaviors. Importantly, we also track it when registers are set to concrete values, such as in `xor rdx, rdx; ret`, where `rdx` is set to 0. This is important to generate chains for tasks that want to set `rdx` to 0, such as `execve("/bin/sh", NULL, NULL)`.

Notice that all these "popping", "moving", or "changing" actions are gadget effects, not instructions, and thus architecture-agnostic. Although `push rax; pop rbx; ret` involves only `push` and `pop`, it will be considered a register move from `rax` to `rbx`. Also, "register pop" here means that the register values come from the stack (*e.g.*, `mov rax, [rsp+8]; add rsp, 0x10; ret`), so gadgets do not need to have a literal "pop" instruction.

3) Gadget Categorization: Based on the control flow transition types, we categorize gadgets into three types:

pop_pc. The next PC value comes from the stack, *e.g.*, `ret` or `ldr lr,[sp],#4;add sp,#8;bx lr`. Notice that all self-contained gadgets are `pop_pc` gadgets by definition (taking the PC from the stack patch).

jmp_reg. The next PC value comes from another register, *e.g.*, `jmp rax`.

jmp_mem. The next PC value comes from a specific memory location (*e.g.*, `call [rax+8*r12]` in ret2csu). By definition, ret2csu and GOT gadgets are all `jmp_mem` gadgets.

Note that we categorize gadgets by effects, not instructions. Even though a gadget such as `pop rax; jmp rax` ends with a jump to registers, it is considered as pop_pc, because at the end of the execution, the PC comes from the stack. Finally, we identify self-contained gadgets (pop_pc gadgets with no conditional branches) as the initial list of ROPBlocks.

E. ROPBlock Generation: Iterative Graph Optimization

In this step, we aim to generate more usable ROPBlocks by turning non-self-contained gadgets into ROPBlocks. This is achieved by utilizing a Register Moving Graph and the Gadget Normalization procedure.

At this point, we already have some usable ROPBlocks (the self-contained gadgets). This allows us to bootstrap and create a minimally functional chain builder and perform simple tasks,

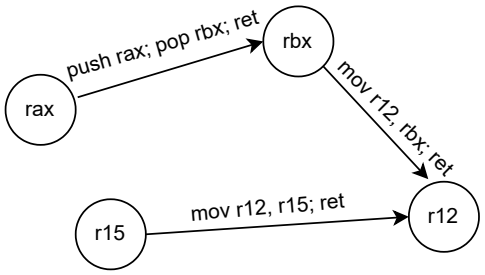

Fig. 2: Register Moving Graph.

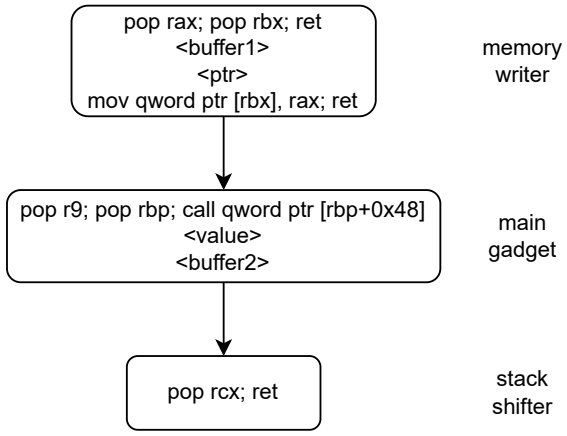

Fig. 3: How ropbot normalizes a `jmp_mem` gadget.

such as setting registers or writing to memory using the same algorithm described in Section III-F.

1) Register Moving Graph: We then build a directed graph modelling the register moving effect. As shown in Figure 2, we consider each register a node in the graph. We add an edge (Reg A, Reg B) if and only if there are ROPBlocks that move register A to register B. By using this graph, we can decide whether there exists a chain that can move any register to another. For example, in Figure 2, the chain builder can tell there is a chain `push rax; pop rbx; ret | mov r12, rbx; ret` that moves `rax` into `r12`. Note that edges are associated with ROPBlocks, so they can be easily chained together into a new ROPBlock (§III-F1).

2) Gadget Normalization: Then, we check whether any non-self-contained gadget provides unique capabilities (*i.e.*, sets a register that all existing ROPBlocks cannot set, or it performs a unique register move); if yes, we "normalize" them into ROPBlock for later use. Here, the goal is to group the non-self-contained gadget with other ROPBlocks into a new ROPBlock while maintaining its unique capability.

The gadget normalization procedure is as follows:

- If a gadget is a `jmp_reg` or `jmp_mem` gadget, prepend it with a ROPBlock that can set the target register or write to the memory (*e.g.*, prepend `pop rax; ret` to `pop rbx; jmp rax` to gain the capability of setting `rbx`).
- If a gadget has a non-positive stack change or takes `PC` outside its stack patch, chain it with a stack shift-

ing ROPBlock (*e.g.*, chain `pop rax; call rax` with `add rsp, 8; ret`).

- If a gadget contains conditional branches, prepend it with a ROPBlock that sets the correct constraints.

The result of the procedure described above is a list of gadgets and corresponding constraints (*e.g.*, where to write when normalizing `jmp_mem` gadgets). We turn the list of gadgets into a ROPBlock using symbolic execution. Specifically, we first create a fully symbolized state and symbolically execute the first gadget. When the execution finishes, the state will be at an "unconstrained" state because the PC comes from user-controlled data. Note that the gadget normalization process ensures that all the PC values can only come from the stack (if not, it will be prepended with a ROPBlock to set the register/memory, etc). Then, we can constrain the state PC to be the next gadget's address, which is essentially setting the correct stack data to the address. During symbolic execution, we apply the chain constraints so that the execution will only follow the chainable state. For example, when `angr` observes a write to a symbolic address, we concretize the variable using values from the provided constraints. By finishing symbolically executing the list of gadgets, we obtain a ROPBlock together with its stack patch. After this process, we also perform the effect analysis on the generated ROPBlock, just as for the gadget effect. As a result, if there are any conflicting constraints in the ROPBlock, they will be captured during symbolic execution and reflected in the ROPBlock effect result.

The symbolic execution step essentially embeds the constraints into each ROPBlock so that it can be chained independently of other ROPBlocks, which ensures that ROPBlocks are unconditionally chainable. For example, we can normalize `cmp rax, 1; jne <label>; pop rbx; ret` into `pop rax; ret | 1 | cmp rax, 1; jne <label>; pop rbx; ret`, which explicitly satisfies the constraint of `rax == 1`, so that the whole ROPBlock can execute correctly and deterministically without the help of other gadgets.

In Figure 3, we show how ropbot normalizes a `pop r9; pop rbp; call [rbp+0x48]` gadget to gain the unique capability of setting `r9` while maintaining the control-flow. We first prepend the main gadget with a memory writing chain and write the address of the stack shifter there. Then, we symbolically step through the main gadget, which sets `r9` to `<value>`, sets `rbp` to `buffer2`, and then invokes `[rbp+0x48]`. Here, ropbot will notice the symbolic read of the PC and redirect it to the previous symbolic write, adding the constraint `buffer1==buffer2+0x48`. Finally, the chain will execute the stack shifter, clean up the return address pushed onto the stack, and move fresh unconstrained stack data into the PC, thus maintaining the control-flow. After the normalization process, the whole sequence becomes a ROPBlock that has the ability to set `r9`.

By using the gadget normalization procedure, we turn gadgets that are hard to work with but provide unique capabilities into ROPBlocks so that they are usable by our chain builder. However, not all gadgets can be normalized if they require gadget setting capabilities that our chain builder does not currently possess.

3) Graph Optimization: Then, we utilize the Register Moving Graph to gain capabilities in terms of setting new registers. We use the graph to check whether there is any path from a register that we can set (*e.g.*, Reg A) to a register we cannot set (*e.g.*, Reg B). If such a path exists, we can combine the ROPBlock that sets Reg A and the ROPBlock that move Reg A to Reg B, and thus we obtain a new ROPBlock that can set Reg B.

As such, new edges in the Register Moving Graph may lead to new register setting capability; new register setting capability may enable the normalization of gadgets that provide unique capabilities. So, we perform the optimization iteratively until we can no longer normalize any gadget that provides unique capabilities.

F. Chain Generation: Graph Search

Finally, we need to craft a chain that fulfills the attacker's goals. We split the chain building task into sub-tasks: Register Setting, Register Moving, Function Invocation (Syscall Invocation), Memory Writing, and Stack Shifting.

Among them, Memory Writing and Stack Shifting are simple. Memory Writing involves choosing a ROPBlock that performs a memory write at a symbolic location. By correctly setting the registers using Register Setting capability, we can ensure that when chained with the register setting chain, the ROPBlock can perform a controlled write at the desired address. Stack Shifting is simply shifting the stack pointer without clobbering the existing state by choosing a ROPBlock that has a wanted stack change value. This module is needed for normalizing gadgets for stack cleaning as demonstrated in Figure 3.

Register Moving can be performed by using the Register Moving Graph built during graph optimization.

1) Payload Chaining: ropbot's Chain Builder only works with ROPBlocks. By definition, a ROPBlock has positive stack changes, thus it has a stack patch. The abstraction of the stack patch makes payload chaining naturally an easy task. As shown in Figure 4, it consists of two steps: 1. Concatenate the stack patches of two blocks, and 2. fix up the `next_pc` value in the stack patch to chain the two ROPBlocks. This abstraction makes it easy to generate CR chains that are not linear. One notable property of this payload chaining process is that the chaining result is also a ROPBlock, making it easy for further manipulation.

In this model, Return-Oriented-Programming on x86 becomes a special case where all gadgets/ROPBlocks have `next_pc` values in the last position of their stack patches (because `ret` moves the last value in the stack patch to PC).

2) Register Setting: Setting registers is the core challenge of CR chain generation. In our work, we consider three ways of setting registers:

- Register pop. This is the most common way of setting registers, *e.g.*, `pop rax; ret`. In our framework, we

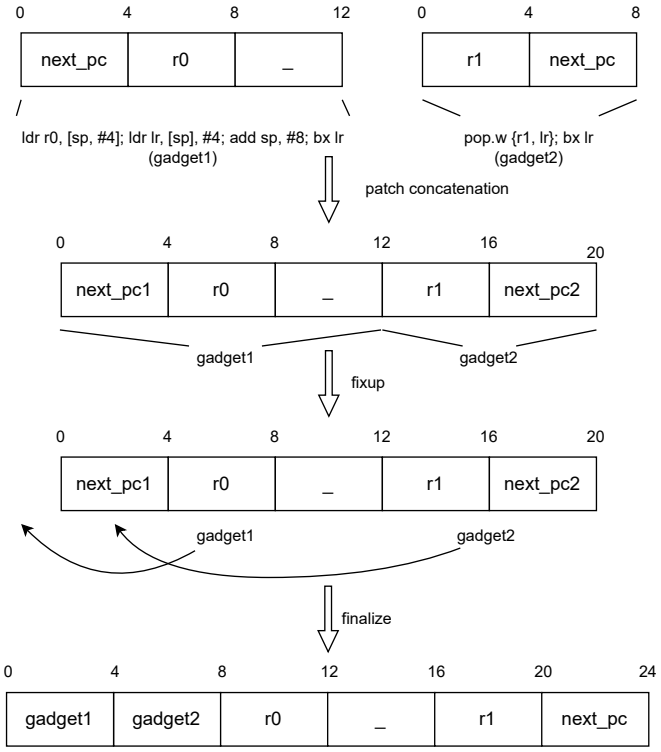

Fig. 4: The process of chaining two ROPBlocks.

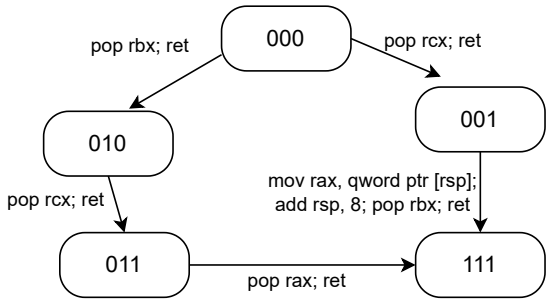

Fig. 5: Register Setting Graph for setting {rax,rbx,rcx}.

consider all memory loads from the stack as "register pop", including `mov rbx, [rsp+0x10]`.

- Register moves. It is common that not all registers can be set through register popping due to the lack of gadgets or constraints on inputs (*e.g.*, bad bytes). In those cases, we can set the value in one register using register pop and then move it to the target register. Gadgets with unique register moving capabilities will be turned into register pop ROPBlocks during graph optimization.
- Concrete values. In cases where a value cannot be set through register popping at all, we may resort to using existing concrete values in gadgets (*e.g.*, use `xor rax, rax; ret` to set 0 when `pop rax; ret` does not exist).

The core idea of our solution is to build a directed graph to represent the program state transition and find a path that can set all the requested registers. During Iterative Graph Optimization (§III-E), we gain the ability to set each individual register (in the form of register pops); we now need to find the shortest chain for a given set of registers with no conflict.

With all the ROPBlocks, we build a directed graph where each node represents the state of register control. Specifically, when a user wants to set `rax, rbx, rcx`, each node is a 3-boolean tuple that represents whether each register is already in our control. For example, `(True, False, True)` (written as 101), in the order of rax, rbx, rcx, is one of the nodes that means rax and rcx are under our control (can be popped), while rbx is not. Each edge means there is at least one ROPBlock that transitions from one state to another. Register pop ROPBlocks can clearly lead to state transitions, thus are added as edges in the graph. Besides them, we also check ROPBlocks with concrete value effects. For example, if the request is to set `rdx` to 0, this step will add `xor rdx, rdx; ret` as an edge because this ROPBlock can lead to the correct register control. As a result, our algorithm can utilize all three ways of setting registers (register move gadgets will be turned into register pop ROPBlocks during graph optimization) in a uniform way.

During the graph-building process, we model all ROPBlocks precisely. Besides their register pop effects and concrete values, we also check whether they clobber any wanted registers. For example, the `pop rcx; xchg rax, rsi; ret` ROPBlock sets `rcx` but clobbers `rax`. So it will add an edge 100-001 (in the order of rax, rbx, rcx) to the graph (besides other edges such as 000-001).

Now, the task of finding a chain that sets all the registers becomes finding a path from the Node representing no register control to the Node representing full register control. In the above example, it means finding a path from Node 000 to Node 111, which is visualized in Figure 5.

This graph takes $\mathcal{O}(n)$ to build, where n is the number of ROPBlocks, because we need to go through all ROPBlocks and add corresponding edges if needed. And it takes $\mathcal{O}(2^{2k})$ to solve, where k is the number of registers to set: Let V be the number of nodes in the graph, then $V = 2^k$. To solve the shortest path problem in the graph takes $\mathcal{O}(V^2)$, which is $\mathcal{O}(2^{2k})$. Since k is small and constant in practice, $\mathcal{O}(2^{2k})$ is negligible. As a result, the problem of register setting takes $\mathcal{O}(n)$ to solve using this graph search.

3) Function/Syscall Invocation: The goal of CR chains is to invoke functions or system calls such as `mprotect` (inject shellcode) or `execve` (execute "/bin/sh"). But this task is non-trivial across architectures due to the variety of calling conventions out there.

Here, we design a calling-convention-agnostic algorithm to handle function and syscall invocation by introducing an Invocation pseudo-gadget, a gadget that invokes a function or syscall. Essentially, the Invocation gadget is a pseudo-gadget that has different gadget effects under different architectures and calling convention configurations.

For function invocation, the Invocation gadget is a pseudo-gadget that points to the target function's address. Notice that, depending on the architecture and calling convention, an Invocation gadget may or may not be self-contained. For

example, on x64, a function Invocation gadget is as simple as the address of the function itself with a stack change of 8, and it is self-contained (functions end with `ret`, which is equivalent to `pop pc`). But on AArch64, it is no longer self-contained because its stack change is zero and functions end with `ret`, which is equivalent to `jmp x30`. This means that the function Invocation pseudo-gadget is a "jmp_reg" gadget. For syscalls, the Invocation gadget is just a normal gadget that invokes syscalls (*e.g.*, `syscall; ret`). Note that we identify system call invocation by checking the basic block's jump kind (syscalls are marked as `Ijk_Sys` in Valgrind [23]'s `vex` library, which is used internally by `angr`). So we do not rely on architecture-specific instruction identification.

With the Invocation gadget, we leverage the calling convention analysis in `angr` to map function arguments passed by users to registers and stack data. We can pass stack arguments as we control the stack and we utilize the previously described Register Setting capability to set register arguments. Then, we append the Invocation gadget. Finally, we normalize the gadgets to make the whole chain a ROPBlock. For example, on AArch64, a function Invocation gadget is a `jmp_reg` gadget, so we normalize it by setting `x30` before setting arguments and invoking the function. We do this for all calling conventions that return to a register. On x86, function Invocation needs to clean up the arguments on the stack, which can be done by setting the `next_pc` to a stack-shifting gadget. We infer the need for it by checking the number of stack arguments.

Note that the effects of Invocation gadgets are inferred from `angr`'s calling convention analysis (`angr` can analyze function and syscall calling conventions based on the architecture and operating systems), no manual effort is needed to craft the Invocation gadget on a per-architecture basis.

G. Scalability

Although ropbot uses symbolic execution in its design, we limit its use to simple tasks (*i.e.*, gadget effect analysis and turning lists of gadgets into ROPBlock). We do not utilize it for exploration, thus avoiding the notorious state explosion problem. This is evidenced by the fact that ropbot works for the Linux kernel, Chromium, and Firefox (Table V), three of the largest binaries.

H. Implementation

We implemented our prototype ropbot in $7,478$ lines of Python code, using `angr` as the symbolic executor and `networkx` as the graph library. The overall development lasted for around 10 years, but concepts presented in this paper, including ROPBlock, gadget identification algorithm, and the graph search algorithm were implemented within three months. Overall, ropbot is architecture-agnostic: To demonstrate its extensibility, we added support for RISC-V: The whole development process only took us two hours, involved changing 12 lines in ropbot, 88 lines in `angr`, and 152 lines in `archinfo`, one of `angr`'s dependencies. The changes were mostly to fix the wrong RISC-V specification used by `angr` and `archinfo`. The 12 lines of changes to ropbot are trivial changes, only for declaring RISC-V support.

IV. Evaluation

In the following, we evaluate ropbot to determine its efficiency and effectiveness in generating CR gadget chains. We first discuss our experimental setup, then conduct an experiment on x64, other architectures, and study two cases, highlighting how ropbot performs on the Linux kernel and across security-critical, widely used applications. Following our comparative evaluation, we conduct ablation studies to understand the performance improvement and new gadgets that only ropbot can use.

A. Experimental Setup

We compare ropbot against all existing open-source CR chain generation tools. We now briefly introduce these tools, with details and notable differences listed in Table I.

1) State-of-the-art Tools: We identify five open-source works relevant to our evaluation.

SGC [9] encodes all potential gadgets and their combinations as SMT formula that is then passed to an SMT solver. As many gadgets incur high solving overhead, SGC *samples* some gadgets in each round (generate) and checks if a valid chain can be formed for this subset (test).

Exrop [20] uses ROPGadget [21] to find gadgets statically and then uses Triton [24] to analyze the registers each gadget can set. It then enumerates all potential gadget lists that can satisfy a register setting request and uses Triton to verify the feasibility of the chain.

RiscyROP [10] uses an algorithm similar to exrop's with an improved enumeration algorithm so that valid chains are more likely to appear early in the enumeration.

Crackers [11] uses an SMT-based algorithm in an SGC-like fashion. Instead of directly encoding all native instructions in gadgets as SMT formula, it lifts native instructions to Ghidra's P-Code first and encodes P-Code. This extra lifting process provides the potential of working for different architectures.

Arcanist [12] was published concurrently to Crackers and essentially proposes the same algorithm and architecture. Though, it lifts native instructions to Binary Ninja Intermediate Language (BNIL) instead of Ghidra's P-Code and then encodes BNIL to SMT formulas.

When studying them (see Table I), we find that all except RiscyROP focus on x64, SGC and Arcanist additionally support x86, and Arcanist supports ARM as well. RiscyROP works only for AArch64 and RISC-V. All except Crackers and Arcanist allow to output the payload as bytes. Support for more complex gadgets, such as ret2csu ones, ones containing conditional branches, or ones spanning multiple blocks, is generally bad with exception of RiscyROP. Similarly, only exrop allows chaining multiple payloads, which is a convenience in practice. Interestingly, half of the approaches (SGC, Crackers, and Arcanist) sample from all potential gadgets before constructing a chain, increasing performance at cost of accuracy.

TABLE I: Capabilities and characteristics of open-source CR chaining tools. (B)GAT refers to the (bounded) generate-and-test algorithm. Attacker needs to: control Stack content (S), know stack location (L), control arbitrary memory values (Mem).

	SGC [9]	**exrop** [20]	**RiscyROP** [10]	**Crackers** [11]	**Arcanist** [12]	**ropbot**
Architectures	x86/x64	x64	AArch64/RISC-V	x64*	x86/x64/ARM	Multi
Algorithm	BGAT	GAT	BGAT	BGAT	BGAT	Graph
Byte Generation	✓	✓	✓†	✗	✗	✓
Conditional Branch	✗	✗	✓	✗	✓	✓
Multi-Block Gadget	✗	✗	✓	✗	✗	✓
Return-to-csu	✗‡	✗	✗	✗§	✗‡	✓
Payload Chaining	✗	✓	✗	✗	✗	✓
Uses Full Gadget Space	✗	✓	✓	✗	✗	✓
Attacker Requirements	S, L	S	S	S, L, Mem	S	S

* Crackers may support multiple architectures but was only evaluated on x64.
† RiscyROP can output payload in bytes only after we modified it.
‡ SGC and Arcanist can utilize ret2csu gadgets if the address of the payload on stack is known.
§ Crackers can utilize ret2csu gadgets only if all memory is addressable and controlled by attackers.

2) Attacker Capabilities: In line with our threat model, our evaluation assumes attackers do not control the initial program state except for the stack, where the attacker can place a gadget chain of arbitrary length. We use the same assumptions across all tools, with one exception: SGC and Arcanist additionally require knowing the location of the payload, which we provide. This gives SGC and Arcanist an unfair advantage, as other tools do not require or use this information.

3) Chain Verification: Generating a gadget alone is no guarantee that it fulfills attacker guarantees, as outlined in prior work [8], [9]. These chains can lead to crashes by accessing memory at an unconstrained address or lead to non-deterministic results when using gadgets with conditional jumps but failing to satisfy the jump constraints. To ensure correctness, we developed a symbolic verification framework on top of `angr` that simulates a control-flow hijacking scenario and loads the bytes produced by each tool into the symbolic state and starts execution. It considers a chain correct only if the execution is deterministic, all memory accesses are constrained to mapped regions, and the chain achieves the expected outcome (*e.g.*, execute an `execve` syscall). As Crackers and Arcanist cannot generate CR chains in byte form, we are unable to verify their generated chains; we conservatively use the number of generated chains instead.

4) Harware Setup & Parameters: All experiments were performed on a machine with 2 `Intel(R) Xeon(R) CPU E5-2670 v2 @ 2.50GHz` CPUs (20 cores, 40 threads, frequency scaling enabled) and 256GB RAM@1600MHz, with swap disabled All tools support concurrent computation and were configured to fully utilize the 40 cores. Following Crackers [11], we set a timeout of 30 minutes for each target. Time is measured by inserting Python `time.time()` calls before and after each target task.

B. x64 Evaluation

We first evaluate the tools on x64, as it is the most widely tested architecture. We first present our dataset and tasks, before then discussing results shown in Table II.

Dataset. Our dataset consists of 1016 random ALLSTAR [25] binaries used in Crackers and 6 binaries used in SGC. Note that the ALLSTAR dataset comprises real-world binaries from Debian. In total, we obtain a large and diverse set of 1022 binaries.

Task 1: facefeed. The first task is to invoke a function with three arguments: `0xfacefeed(0xdeadbeef, 0x40, 0x7b)`, modeling Crackers' evaluation. We use this task as a baseline to test each CR chain generation tool's functionality and perform it for all 1022 binaries.

Task 2: mmap. We target `mmap` with six arguments: `mmap(0x41414000, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANON | MAP_PRIVATE | MAP_FIXED, -1, 0)`, following SGC's evaluation. We use this task to stress each tool's ability to generate chains that set multiple registers. This chain is relevant in practice because one can use this function call to inject shellcode (combined with a memory write) and obtain arbitrary code execution. We perform this task on binaries containing `mmap` in the PLT, which is the case for only 64 of our binaries.

Task 3: execve. The third task is to invoke the `execve("/bin/sh", NULL, NULL)` system call, following SGC's evaluation. This checks each tool's ability to invoke system calls and write memory (binaries usually do not contain the "/bin/sh" string). We perform this task on all binaries with at least one "syscall" instruction, which results in 272 binaries.

Task 4: Full chain. Then, the last task is invoking a full chain: `dup2(3,0)+dup2(3,1)+execve("/bin/sh", NULL, NULL)`. This CR chain is commonly used in exploiting servers, known as the socket reuse payload [26]. We use it to test the tools' ability to generate realistic CR chains, and we run it on binaries containing both the `dup2` function call and "syscall" instructions, resulting in 37 binaries.

1) Results: For this dataset and these four scenarios, we study how many chains can be *generated*, how many of them can be verified (*success*), and how many are invalid (*false positives*) or the tool hit the 30-minute *timeout* before generation. We also track the time per stage (*gadget finding*, for ropbot graph *optimization*, and *chain-time*) as well as the time it took the tool to process all binaries in the dataset (*total time*). As shown in Table II, ropbot not only outperforms all

TABLE II: Chain generation capability for x64 binaries, detailing **Success**ful, verified chains out of **Generated** ones (invalid chains are **False Positives** and **Timeout**s are cases where no chain was produced in 30 minutes), average time taken per stage (finding gadgets, optionally optimizing the graph, and generating a chain), and **Total Time** needed to process full dataset.

Tool	Success	Generated	False Positives	Timeout	Avg Time per Stage			Total Time
					Gadget Finding	Optimization	Chaining	
facefeed (total: 1022 binaries)								
ropbot	**635**	**635**	**0%**	**0%**	11.5s	6.9s	**0.4s**	**5.3h**
exrop	357	358	0.3%	4.1%	**9.8s**	-	72.8s	23.4h
SGC	616	633	3.8%	14.8%	225.3s	-	312.6s	130.9h
Crackers	-	353*	-	3.9%	24.4s	-	237.7s	72.4h
Arcanist	-	416	-	20.2%	180.8s	-	457.2s	167.4h
mmap (total: 64 binaries)								
ropbot	**21**	**21**	**0%**	**0%**	**25.3s**	11.4s	**1.2s**	**0.7h**
exrop	4	4	**0%**	10.9%	56.7s	-	199.9s	4.5h
SGC	0	0	-	28.1%	932.1s	-	240.3s	17.5h
Crackers	-	0	-	62.5%	179.8s	-	1215.4s	22.1h
Arcanist	-	0	-	79.7%	903.0s	-	1453.8s	28.1h
execve† (total: 272 binaries)								
ropbot	**244**	**244**	**0%**	**0%**	**28.4s**	11.9s	**2.5s**	**3.2h**
exrop	90	117	23.1%	13.6%	33.6s	-	240.5s	20.9h
SGC	27	27	1.6%	80.9%	763.1s	-	1527.8s	123.0h
Crackers	-	0	-	21.0%	83.6s	-	695.9s	55.2h
fullchain (total: 37 binaries)								
ropbot	**37**	**37**	**0%**	**0%**	**81.8s**	12.9s	**3.7s**	**1.0h**
exrop	21	22	4.5%	5.4%	152.3s	-	92.7s	2.5h

* Crackers performs worse than in the original paper due to our stricter threat model: Attackers do not control arbitrary memory and registers initially.
† Arcanist does not support writing memory, thus we exclude it from the `execve` evaluation that requires writing "/bin/sh" to memory.

tools in terms of the success rate in all tasks, but it is faster as well. Still, exrop and Crackers are fast in finding gadgets. This is because they use static analysis to identify potential gadgets first, following the Galileo Algorithm (exrop uses ROPgadget internally) and use either symbolic execution or SMT solvers to analyze the gadgets. As mentioned in Section II-2, this approach misses gadgets (*e.g.*, gadgets spanning multiple basic blocks) by design, which significantly affects their ability to set registers as shown in the `facefeed` task. In our Ablation Study (§V), we observe that gadgets spanning multiple basic blocks are necessary for generating the `facefeed` chain for 51 binaries. ropbot is fast in gadget finding because of the position-dependent gadget caching mechanism (§III-D1).

In terms of chain generation, SGC performs exceptionally well in the `facefeed` task. This is because of two reasons. First, SGC generates multiple chains at once by design, while all other tools stop at the first chain. We consider one test a "Success" if any of the chains generated by SGC for the binary passes verification (and calculate *false positives* over all chains). Second, SGC has the unfair advantage of knowing where the payload is on stack so that it can utilize gadgets like `call [rbx]` by concretizing `rbx` to the payload address and place 0xfacefeed in the payload. In fact, we find that 94.1% of SGC's chains relied on knowing the address of the payload. ropbot can utilize these gadgets as well, but it needs to write to the location first due to not knowing the payload address.

Studying false positives, we find SGC incorrectly concretizes pointers (setting the upper bytes to `0xff`, thus rendering it invalid). exrop sometimes incorrectly models gadgets: Internally, it uses ROPgadget to find gadget candidates, which sometimes list a sequence of instructions that end with a constant jump (*e.g.*, `pop rax; jmp <label>`, which may not be a gadget for this tool's definition, depending on where it jumps to). But exrop treats the constant jump the same as a `ret`, failing to model the destination basic block, thus leading to crashes. As we cannot verify chains from Crackers or Arcanist, we are unable to check false positives.

TABLE III: Comparison to RiscyROP on AArch64 and RISC-V. *FPs = False Positives, Gen. = Generated, Opt. = ropbot's optimization stage, Chain. = gadget chaining time.*

Tool	Success	Gen.	FPs	Timeout	Avg Per-Stage Times			Total Time
					Finding	Opt.	Chain.	
AArch64 (total: 172 binaries)								
ropbot	**53**	**53**	**0%**	**2.9%**	**237.4s**	89.9s	**0.5s**	**13.9h**
RiscyROP	9	17	47.1%	28.5%	833.1s	-	176.6s	43.4h
RISC-V (total: 1000 binaries)								
ropbot	**362**	**362**	**0%**	**3.8%**	**192.7s**	133.3	**0.5s**	**77.1h**
RiscyROP	77	112	31.3%	46.6%	933.3s	-	485.8s	316.2h

C. Non-x64 Evaluation

In this section, we evaluate ropbot's capability on non-x64 architectures. Although Arcanist supports ARM, it is only configured for a few specific CVE targets, not ready for a generic evaluation on ARM. RiscyROP is the only other tool that supports non-x64 architectures, namely AArch64 and RISC-V.

TABLE IV: Evaluation result of the `0xfacefeed(0xdeadbeef,0x40,0x7b)` chain. We filter out PAC-enabled AArch64 binaries, leading to a smaller number compared to other architectures.

Arch	Success	Total	Rate	FP	Timeout	Avg Per-Stage Finding	Avg Per-Stage Opt.	Times Chain.	Total Time
x64	635	1,022	62.1%	0%	0.0%	11.5s	6.9s	0.4s	5.3h
MIPS	285	1,000	28.5%	0%	0.1%	11.2s	24.3s	0.8s	10.1h
AArch64	58	172	33.7%	0%	1.2%	24.3s	59.6s	0.6s	3.9h
ARM	660	1,000	66.0%	0%	0.0%	6.0s	8.1s	0.6s	4.1h
RISC-V	358	1,000	35.8%	0%	1.1%	34.7s	60.5s	0.5s	24.1h

To fairly compare ropbot against RiscyROP, we perform the evaluation with the CR chain generation task hardcoded in RiscyROP, which is to invoke `0xdeadbeef(0x40404040, 0x41414141, 0x42424242)` on both architectures. Here, we enabled ropbot's conditional-branch gadget support, which is off by default due to the fact that it significantly slows down gadget finding and chain building with no significant capability improvement (Table VIII in the Appendix).

Dataset. We are not aware of any large dataset of AArch64 or RISC-V binaries, so we sampled $1,000$ binaries from Ubuntu 24.04 for both architectures. However, we found that many of the AArch64 binaries had pointer authentication (PAC) enabled, leading to very few usable gadgets and, thus, low success rates for ropbot and RiscyROP; We filter out PAC-protected binaries, leaving us with 172 AArch64 binaries and $1,000$ RISC-V binaries.

1) Results: As shown in Table III, ropbot achieves 5x the success of RiscyROP under the same configuration. At the same time, RiscyROP shows a high false positive rate due to improper use of gadgets with conditional branches. RiscyROP works by identifying a potential chain, then symbolically executing it. It focuses on finding states that satisfy the expected outcome but does not reject non-deterministic chains.

D. Multi-Arch Support

Besides our comparison to other tools, we also evaluated the `facefeed` task across all supported architectures to investigate the impact of architecture designs on CR chain generation. The result can be found in Table IV and shows that ropbot can generate CR chains across architectures. However, its performance is significantly better on x64 and ARM. Studying gadget diversity, we observed that x64 binaries have 711 self-contained gadgets on average, and more than 90% of the binaries include gadgets that can pop `rdi` and `rsi`, and 32.8% can pop `rdx`—necessary prerequisites to set the arguments. Other architectures feature only around 200 self-contained gadgets on average, and 2%-20% of binaries include gadgets that can pop the target registers. This explains why ropbot performs worse on MIPS/AArch64/RISC-V, but it fails to explain why it does well on ARM. We inspected the chain ropbot generated for ARM and immediately noticed a pattern of using two gadgets: `pop {r3,r4,r5,r6,r7,r8,r9,pc}` and `mov r0,r7;mov r1,r8;mov r2,r9;blx r3`. These two gadgets alone are enough to generate the `facefeed` chain used in our evaluation, and they are pervasive in the target binaries. It is noteworthy that these are the equivalent of the ret2csu gadgets on ARM. Due to architectural features, they are compiled into `jmp_reg` gadgets, which makes them even more powerful for code reuse attacks.

Listing 2: A `commit_creds(prepare_kernel_cred(NULL))` chain generated by ropbot.

```
p = b""
p += p64(0xffffffff822a76a1)# xor edi, edi;
                            # jmp 0xffffffff822a7e60;ret
p += p64(0xffffffff8114d660)# prepare_kernel_cred
p += p64(0xffffffff8196eeeb)# pop rdx; pop rdi;
                            # jmp 0xffffffff822a7e60; ret
p += p64(0xffffffff82504104)# lea rsp,[rsp+0x8]; ret
p += p64(0xffffffff83600198)# <buffer>
p += p64(0xffffffff82211ea2)# mov qword ptr [rdi], rdx;
                            # xor edx, edx; xor edi, edi;
                            # jmp 0xffffffff822a7e60; ret
p += p64(0xffffffff8240191a)# pop rbp;
                            # jmp 0xffffffff822a7e60; ret
p += p64(0xffffffff83600150)# <buffer> - 0x48
p += p64(0xffffffff81aca5d2)# push rax; pop rdi;
                            # call qword ptr [rbp + 0x48]
p += p64(0xffffffff8114d3a0)# commit_creds
```

E. Case Study – Linux Kernel

We now study ropbot's capability of building a chain on large binaries by trying to generate the `commit_creds(prepare_kernel_cred(NULL))` chain for the Linux kernel. While it is commonly seen in Linux kernel exploitation, it is not an easy task due to the lack of a `mov rdi, rax; ret` gadget. Security researchers have resorted to finding sophisticated chaining methods to accomplish the same effect, such as using `push rax; jmp [rsi] | pop rdi;| ret` [27] and `pop rdi; ret | or rdi, rax; ret` [28]. These chains are time-consuming to craft manually due to the need to explore all possible ways of moving `rdi` to `rax`. In our experiment, ropbot can craft a similar chain in one second (with cached gadget analysis results, which takes 7 minutes to build), shown in Listing 2. ropbot's chain writes a `lea rsp, [rsp+8]; ret` gadget into a buffer and then invokes it using `push rax; pop rdi; call [rbp+0x48]` to achieve `mov rdi, rax` while maintaining the control flow. This is similar to what human researchers did in CVE-2023-3390 [27]. Another sample container escape payload can be found in Appendix-A.

Due to the Retpoline protection [29] adopted in the Linux kernel, all functions end with `jmp indirect_thunk`, which in turn executes `ret`. In Listing 2, this is `jmp 0xffffffff822a7e60; ret`, where `0xffffffff822a7e60` is the `indirect_thunk`. As a result, many existing gadget finders [21], [22] will misclassify these gadgets as gadgets ending with `jmp`, while in reality, they are equivalent to ending with `ret`. This demonstrates the importance of gadget effect analysis used in ropbot instead of instruction-based analysis and the need to analyze gadgets spanning multiple basic blocks.

TABLE V: Time needed to generate the `facefeed` chain (30min timeout; *fail* if the engine terminated unsuccessfully).

Binary	Size	ropbot	exrop	SGC	Crackers	Arcanist
TFTPD	80K	7.5s	**1.8s**	fail	fail	254.8s
Webkit_jsc	347K	32.4s	**9.7s**	timeout	238.8s	560.5s
OpenSSL	736K	**29.4s**	fail	1422.8s	timeout	439.1s
OpenSSH	900K	**26.4s**	38.5s	timeout	308.8s	586.5s
Dnsmasq	1.6M	**12.6s**	28.0s	fail	193.6s	648.6s
Apache2	2.8M	**48.1s**	69.0s	timeout	833.8s	1255.6s
Nginx	5.2M	**26.7s**	52.0s	timeout	1335.0s	865.1s
Linux kernel	64M	**192.0s**	fail	fail	timeout	timeout
Firefox	159M	**1252.0s**	timeout	timeout	timeout	timeout
Chromium	166M	**1626.0s**	timeout	timeout	timeout	timeout

TABLE VI: Evaluation on ropbot with different configurations. Base is ropbot with only single-block self-contained gadgets and does not perform graph optimization.

Config	Task	Success	Total	Rate
Base	execve	193	272	71.0%
Base+Graph_Opt	execve	215	272	79.0%
exrop	execve	90	272	33.1%
Crackers	execve	0	272	0%
SGC	execve	27	272	9.9%
Base	facefeed	319	1022	31.2%
Base+Graph_Opt	facefeed	391	1022	38.3%
Crackers	facefeed	353	1022	34.5%
exrop	facefeed	358	1022	35.0%
Full-got_gadgets	facefeed	574	1022	56.2%
Full-ret2csu_gadgets	facefeed	570	1022	55.8%
Full-multi_bb_gadgets	facefeed	570	1022	55.8%
Full-jmp_mem_gadgets	facefeed	457	1022	44.7%
Full	facefeed	635	1022	62.1%
Full	facefeed	55	172	32.0%
Full+cond_br_gadgets	facefeed	56	172	32.6%

F. Case Study – Security-Critical Dataset

We also evaluated all tools' capability on a selected subset of particularly exposed, security-critical binaries. Their nature makes them particularly interesting for attackers in practice. As shown in Table V, ropbot significantly outperforms all state-of-the-art works and is the only tool that can generate valid chains for all target binaries. In particular, it is the only tool to generate chains for Chromium and Firefox. Additionally, it is the fastest for all except the two smallest binaries, where exrop is faster. All other tools consistently timeout on large binaries, which supports our analysis that BGAT does not scale well with larger binaries.

V. Ablation Study

We now investigate ropbot's performance further through ablation studies, by turning off features. *Base* is an ablation of ropbot that only uses single-block self-contained gadgets and does not perform graph optimization (which we add to a subsequent ablations). The result can be found in Table VI, more details in Table VII in the appendix.

A. Graph Search vs Bounded Generate-and-Test

As the top part of Table VI shows, even *Base* can outperform other tools, despite these tools using `jmp_reg` gadgets that are not self-contained. Investigating this, we find that `execve` chains are usually long and require performing a memory write, setting four registers, and invoking the syscall gadget, which stresses the limit of bounded generate-and-test (BGAT). BGAT-based tools are at the risk of encountering state explosion when sampling longer lists of gadgets, leading to more timeouts. For the simpler `facefeed` chains, we observe fewer timeouts. At the same time, some tools limit chain length by design, also leading to lower success rates. For example, SGC allows at most 16 controlled words on stack, while 53.5% `execve` chains generated by ropbot are longer than 16 words.

As for the impact of our graph optimization, it boosts the success rate by 8% even when working with only single-block self-contained gadgets. This is because graph optimization allows ropbot to find a complicated chain that provides unique effects and use it as one ROPBlock. For example, ropbot can chain `pop rax;ret | mov rbx,rax;ret | mov rdx,rbx;ret` to set `rdx`.

B. New Gadget Categories

We also design a series of experiments to measure the impact of each newly supported gadget category on the evaluation results. Each experiment is performed by removing a specific gadget category and checking whether ropbot can still generate a CR chain. As shown in the bottom half of Table VI, each gadget type contributes uniquely to ropbot's success rate. Note that GOT gadgets contribute 5.9% and ret2csu gadgets contribute 6.3% to the success rate. But when they are combined, `jmp_mem` gadgets contribute 17.4% to the success rate. This suggests that there are 5.2% cases where CR chains can be generated with either GOT gadgets or ret2csu gadgets.

To evaluate the impact of gadgets with conditional branches, we design an additional evaluation on the 172 AArch64 binaries with a 30-minute chain generation timeout, with and without conditional branches. We did not perform the evaluation on the x64 binaries due to the fact that they have abundant self-contained gadgets, and ropbot prefers not to use gadgets with conditional branches because normalizing them takes up a considerable amount of payload space, which defeats the purpose of the evaluation. Our results show that utilizing gadgets with conditional branches hurts the overall performance (see Table VII in the appendix). This is because handling gadgets with conditional branches is slow compared to other gadgets due to state forking, leading to more timeouts, as evidenced by the elevated timeout rate (from 1.2% to 4.1%) and gadget finding time (from 32.3s to 237.6s). We also manually checked the one failed case and confirmed that it failed because of timeouts. By giving ropbot an infinite timeout, it

can generate five more (9% improvement) CR chains with the help of gadgets with conditional jumps. Overall speaking, although gadgets with conditional branches contribute unique capabilities to the chain generation engine, the performance hit outweighs their contribution.

VI. Discussion

In this section, we discuss real-world application, future work, and limitations of ropbot.

Real-world Application. We used ropbot successfully to generate privilege escalation payloads for 20 CVEs for the Linux kernel [18], which no other tool supports. To show its versatility on architectures not supported by other tools, we also used ropbot to generate RCE payloads for 37 ARM/MIPS firmware samples vulnerable to four CVEs, rehosted via Greenhouse [30]. Sample outputs can be found in Appendix-B.

More Gadget Types. Although ropbot supports many gadget types, including GOT gadgets, more exist: For example, some failed chains in our evaluation can be made possible by using gadgets that can load registers from a symbolic address (*e.g.*, `mov rax, [rbx]; ret`). These gadgets would require to prepend them with a chain that writes the wanted value to memory and sets `rbx` to the correct address.

Limitations. Although ROPBlock brings many advantages, it may lead to overlooking optimization opportunities offered by directly using gadgets and thus generating longer chains. For example, to use `cmp rax,1; jne <label>; pop rbx; ret`, `rax` needs to be set to 1. Setting it once can allow to use the gadget repeatedly, but turning it into a ROPBlock will prepend it with a chain that always sets `rax`, causing overhead.

Modern Defenses. Modern defenses raise the bar of automated code reuse payload generation but do not stop it. Position Independent Executable (PIE) [31] and Address Space Layout Randomization (ASLR) make an additional information leak primitive a necessity. Stack canaries [32] make hijacking control-flow using a stack-based vulnerability more difficult, but can be bypassed with an additional information leak primitive. Pointer Authentication Codes (PAC) and Control-Flow Integrity (CFI) [33] make obtaining PC-control much harder, but do not prevent code reuse attacks once the attacker controls the PC. This verification of pointers also reduces the amount of usable gadgets, which makes payload generation harder. Shadow stacks [34], [35] can stop ROP but not other types, such as JOP [2]; This would require modification of ropbot or other automated tools though. However, if both a shadow stack and CFI are deployed, control-flow hijacking will become impossible, rendering code reuse attacks and, thus, ropbot obsolete.

VII. Related Work

We propose a new technique to rethink code reuse attacks. Orthogonal research has focused on data-only attacks and automated end-to-end exploit generation.

a) Data-Only Attacks: Beyond code reuse, another class of attacks has recently gained popularity [36]–[39]: data-only attacks, or data-oriented programming. Other than code injection or code reuse attacks, where the attacker's first goal is to hijack the control flow, data-only attacks modify solely data without touching control flow-relevant information, such as return addresses. This way, they bypass any security policy monitoring the control flow, which is only indirectly affected through the modified data. Following an initial study [36], Hu et al. [40] proposed *data-oriented programming* and proved it to be Turing-complete for x86 programs in 2016. Subsequent work has proposed to automate the task of finding and stitching data-oriented gadgets [37], [38], with BOPC [38] being the first publicly available tool. Lately, Ling et al. [39] use programming language synthesis to craft data-only exploits more efficiently and effectively. Despite promising to overcome mitigations such as CFI or shadow stacks, DOP is more limited than code reuse in terms of the capabilities it provides an attacker with. It has yet to see widespread adoption in practice, and the current state of the art requires simplifying targets manually before attacks can be synthesized [39]. ropbot is orthogonal to DOP and considers only the generation of gadget chains under the traditional code reuse attack model.

b) Automatic Exploit Generation (AEG): Following a long history in memory corruption research [41], AEG [42] and Mayhem [43] pioneered the field of AEG, showing the possibility of generating end-to-end exploits automatically. During the study of AEG, researchers noticed the difficulty in automatically exploiting heap-based vulnerabilities due to the need of complex heap layout manipulation. Shrike [44] solved the problem by first identifying the heap operations associated with code fragments in interpreters (*i.e.*, allocation and release) and then use a random search algorithm to find code that can lead to desired heap layout. Gollum [45] improved on Shrike by additionaly identifying useful exploitation primitives such as function calls and use a genetic algorithm to derive the code fragments needed to achieve desired heap layout and obtain PC-control. However, Shrike and Gollum only work on interpreters because of their primitive analysis algorithm. Maze [46] and Gagua [47] generalized Shrike and Gollum's idea to event-loop-based application and analyze primitives in each event-loop iteration so that they work beyond just interpreters. Revery [48] approached the heap exploit generation problem differently, it used heap operation as a feedback metric and let a fuzzer to explore different heap layouts efficiently. Fuze [49] adopted a similar fuzzer-based exploration strategy in automatically generating exploits for the Linux kernel and used symbolic execution to explore crashing states and achieve control-flow hijacking. Koobe [50] modeled out-of-bound access vulnerabilities, explored primitives provided by the initial crash input, and successfuly showed that out-of-bound access vulnerabilities can be automatically tamed to provide PC-control. These works stop at gaining PC-control or generating a hardcoded and build-specific payload. They do not aim to generate code-reuse payload for different build and architectures, thus are orthogonal to ropbot.

VIII. Conclusion

In this work, we propose a new architecture-agnostic abstraction called ROPBlock and a graph-based CR chain generation model to tackle the code reuse attack synthesis problem.

We show that ROPBlock provides guaranteed chainability among gadgets, which leads to improved efficiency and effectiveness in CR chain synthesis. We perform extensive evaluation on our research prototype, ropbot, against all existing works, and ropbot achieves unparalleled results: ropbot outperforms all existing tools significantly on all CR chain generation tasks. We also performed an ablation study to investigate where the improvement comes from. We believe our work is a solid step towards solving the code reuse attack synthesis problem in general.

IX. Acknowledgement

We would like to thank the anonymous reviewers for their insightful feedback.

This material was supported by 2023 Google PhD Fellowship program, NSF 2232915, NSF 2146568, NSF 2247954, NSF 2442984, the Advanced Research Projects Agency for Health (ARPA-H) under contract number SP4701-23-C-0074, the Defense Advanced Research Projects Agency (DARPA) and Naval Information Warfare Center Pacific (NIWC Pacific) under contract number N66001-22-C-4026, the Air Force Office of Scientific Research under award number FA9550-24-1-0227, Department of Navy under award number N00014-23-1-2563, Department of Interior under Grant No. D22AP00145-00, and generous support from the US Department of Defense.

X. Ethical Considerations

Throughout this work, we have not uncovered new vulnerabilities. We have not interfered with external systems, and in particular, all generated CR payloads have been tested on our own systems, ensuring that no users have been harmed.

As our work automates the generation of exploits, thus raising potential concerns regarding its release. While we acknowledge that ropbot can be used by malicious actors, CR chain generation is by no means limited to nefarious purposes. Quite the contrary, it is actively used for exploit verification by companies to prioritize severe bugs. In fact, our tool is currently in use at multiple companies including Google for this very purpose. We stress that ropbot itself is neutral and provides means to an end. But as malicious actors already possess the capability to craft exploits, ropbot's release does not provide them with fundamentally new capabilities. Studying prior work in this domain, we find their tools have been publicly released, which enabled reproducibility of research and public understanding of attackers' capabilities. Thus, we believe the benefits of releasing our research outweigh its potential risks.

References

[1] H. 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.

[2] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-Oriented Programming: A New Class of Code-Reuse Attack," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[3] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-Oriented Programming without Returns," in *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[4] A. Sadeghi, S. Niksefat, and M. Rostamipour, "Pure-Call Oriented Programming (PCOP): Chaining the Gadgets using call Instructions," *Journal of Computer Virology and Hacking Techniques*, vol. 14, pp. 139–156, 2018.

[5] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "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.

[6] Google, "KernelCTF," 2023. [Online]. Available: https://google.github.io/security-research/kernelctf/rules.html

[7] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit Hardening Made Easy," in *USENIX Security Symposium*, 2011.

[8] E. J. Schwartz, C. F. Cohen, J. S. Gennari, and S. M. Schwartz, "A Generic Technique for Automatically Finding Defense-aware Code Reuse Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2020.

[9] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz, "Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains," in *European Symposium on Research in Computer Security (ESORICS)*, 2021.

[10] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A.-R. Sadeghi, "RiscyROP: Automated Return-Oriented Programming Attacks on RISC-V and ARM64," in *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2022.

[11] M. DenHoed and T. Melham, "Synthesis of Code-Reuse Attacks from p-code Programs," in *USENIX Security Symposium*, 2025.

[12] N. Bailluet, E. Fleury, I. Puaut, and E. Rohou, "Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives," in *USENIX Security Symposium*, 2025.

[13] J. Fandinno and L. Lillo, "Solving Epistemic Logic Programs using Generate-and-Test with Propagation," in *AAAI Conference on Artificial Intelligence*, 2025.

[14] H. Marco-Gisbert and I. Ripoll, "Return-to-csu: A New Method to Bypass 64-bit Linux ASLR," in *Black Hat Asia*, 2018.

[15] debugmen, "libC GOT chaining," 2024. [Online]. Available: https://debugmen.dev/pwn/2024/01/15/jump-planner.html

[16] Google, "XDK," 2025. [Online]. Available: https://xdk.dev/about/introduction.html

[17] ——, "XDK-rop-generator," 2025. [Online]. Available: https://github.com/google/kernel-research/blob/main/rop_generator/angrop_rop_generator.py

[18] K. Zeng, Z. Lin, K. Lu, X. Xing, R. Wang, A. Doupé, Y. Shoshitaishvili, and T. Bao, "Retspill: Igniting User-Controlled Data to Burn Away Linux Kernel Protections," in *ACM Conference on Computer and Communications Security (CCS)*, 2023.

[19] J. Miller, M. Ghandat, K. Zeng, H. Chen, A. H. Benchikh, T. Bao, R. Wang, A. Doupé, and Y. Shoshitaishvili, "System Register Hijacking: Compromising Kernel Integrity By Turning System Registers Against the System," in *USENIX Security Symposium*, 2025.

[20] d4em0n, "exrop," 2020. [Online]. Available: https://github.com/d4em0n/exrop

[21] J. Salwan, "ROPgadget," 2011. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget

[22] S. Schirra, "Ropper," 2015. [Online]. Available: https://github.com/sashs/Ropper

[23] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[24] Jonathan Salwan, "Triton," 2015. [Online]. Available: https://github.com/JonathanSalwan/Triton

[25] J. Staff, "Assembled Labeled Library for Static Analysis Research (ALLSTAR) Dataset," Dec 2019. [Online]. Available: http://allstar.jhuapl.edu/
[26] WangYihang, "Socket Reuse Shellcode," 2014. [Online]. Available: https://www.exploit-db.com/exploits/34060
[27] Google, "CVE-2023-3390," 2023. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-3390_lts_cos_mitigation/exploit/lts-6.1.31/exploit.c
[28] ——, "CVE-2023-4004," 2023. [Online]. Available: https://github.com/google/security-research/blob/master/pocs/linux/kernelctf/CVE-2023-4004_lts_cos_mitigation/exploit/mitigation-6.1/exploit.c
[29] P. Turner, "Retpoline: A Software Construct for Preventing Branch-target-injection," 2018. [Online]. Available: https://support.google.com/faqs/answer/7625886
[30] H. J. Tay, K. Zeng, J. M. Vadayath, A. S. Raj, A. Dutcher, T. Reddy, W. Gibbs, Z. L. Basque, F. Dong, Z. Smith *et al.*, "Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation," in *USENIX Security Symposium*, 2023.
[31] R. Hat, "Position Independent Executable," 2012. [Online]. Available: https://www.redhat.com/en/blog/position-independent-executables-pie
[32] P. Wagle, C. Cowan *et al.*, "Stackguard: Simple Stack Smash Protection for GCC," in *GCC Developers Summit*, 2003.
[33] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity: Principles, Implementations, and Applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.
[34] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.
[35] T. H. Dang, P. Maniatis, and D. Wagner, "The Performance Cost of Shadow Stacks and Stack Canaries," in *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.
[36] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *USENIX Security Symposium*, 2015.
[37] J. Pewny, P. Koppe, and T. Holz, "STEROIDS for DOPed Applications: A Compiler for Automated Data-Oriented Programming," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.
[38] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block-Oriented Programming: Automating Data-only Attacks," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[39] Y. Ling, G. Rajiv, K. Gopinathan, and I. Sergey, "Sound and Efficient Generation of Data-Oriented Exploits via Programming Language Synthesis," in *USENIX Security Symposium*, 2025.
[40] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.
[41] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.
[42] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.
[43] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on Binary Code," in *IEEE Symposium on Security and Privacy (S&P)*, 2012.
[44] S. Heelan, T. Melham, and D. Kroening, "Automatic Heap Layout Manipulation for Exploitation," in *USENIX Security Symposium*, 2018.
[45] ——, "Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters," in *ACM Conference on Computer and Communications Security (CCS)*, 2019.
[46] Y. Wang, C. Zhang, Z. Zhao, B. Zhang, X. Gong, and W. Zou, "MAZE: Towards Automated Heap Feng Shui," in *USENIX Security Symposium*, 2021.
[47] R. Li, B. Zhang, J. Chen, W. Lin, C. Feng, and C. Tang, "Towards Automatic and Precise Heap Layout Manipulation for General-Purpose Programs," in *Symposium on Network and Distributed System Security (NDSS)*, 2023.
[48] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou, "Revery: From Proof-of-Concept to Exploitable," in *ACM Conference on Computer and Communications Security (CCS)*, 2018.
[49] W. Wu, Y. Chen, J. Xu, X. Xing, X. Gong, and W. Zou, "FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities," in *USENIX Security Symposium*, 2018.
[50] W. Chen, X. Zou, G. Li, and Z. Qian, "KOOBE: Towards Facilitating Exploit Generation of Kernel Out-Of-Bounds Write Vulnerabilities," in *USENIX Security Symposium*, 2020.
[51] K. Zeng, "CVE-2022-1786," 2022. [Online]. Available: https://blog.kylebot.net/2022/10/16/CVE-2022-1786/

APPENDIX A
MORE PAYLOADS

A. CVE-2022-1786 Payload Generation

We visualize ropbot's capabilities using a case study of CVE-2022-1786. This is an invalid-free vulnerability in the Linux kernel. Crucially, it was originally exploited using a code reuse attack [51]. We tried ropbot on this vulnerability, and it successfully generated a container escape payload in 1.4 seconds (with cached gadget analysis, which takes 6min to build). The payload is shown as follow:

```
p = b""
p += p64(0xffffffff811cd195)# pop rdi;ret
p += p64(0xffffffff8265e460)# init_cred
p += p64(0xffffffff810ca820)# commit_creds
p += p64(0xffffffff811cd195)# pop rdi;ret
p += p64(0x1)
p += p64(0xffffffff810c0350)# find_task_by_vpid
p += p64(0xffffffff811cd2b8)# pop rsi;ret
p += p64(0xffffffff810bd36f)# pop rdx; ret
p += p64(0xffffffff8108dc40)# pop rdi;ret
p += p64(0xffffffff826000e0)# <buffer>
p += p64(0xffffffff810c8fd6)# mov qword ptr [rdi],rsi;
                            # pop rbp;
                            # jmp 0xffffffff81e03100;ret
p += p64(0xffffffff82600098)# <buffer-0x48>
p += p64(0xffffffff816fbbb2)# xchg rdi, rax;
                            # call qword ptr [rbp + 0x48]
p += p64(0xffffffff811cd2b8)# pop rsi;ret
p += p64(0xffffffff8265e080)# init_nsproxy
p += p64(0xffffffff810c86c0)# switch_task_namespaces
p += p64(0xffffffff81096080)# __x64_sys_fork
p += p64(0xffffffff811cd195)# pop rdi;ret
p += p64(0xffffffff)
p += p64(0xffffffff8112d760)# msleep
```

Listing 3: A container escape chain generated by ropbot.

This chain invokes `commit_creds` to escalate privilege first and then use `find_task_by_vpid` and `switch_task_namespace` to escape from the namespace container. Finally, it uses the telefork technique [51] to return back to userspace. This shows that ropbot can generate code reuse attack payloads in real-world exploit development scenarios.

B. CVE-2018-18706 Payload Generation

CVE-2018-18706 is a stack overflow vulnerability found in Tenda routers. Listing 4 shows a `system("/bin/sh")` chain generated by ropbot for an ARM router target.

```
p = b""
p += p32(0x5fe9c)    # pop {r1, r2, lr};
                     # mul r3, r2, r0;
                     # sub r1, r1, r3; bx lr
p += p32(0x0)
p += p32(0x6e69622f)# '/bin'
p += p32(0xbf30)     # pop {r3, pc}
p += p32(0x708a1)    # <buffer>
p += p32(0xe96c)     # str r2, [r3];
                     # pop {r3, r4, fp, pc}
p += p32(0x0)
p += p32(0x0)
p += p32(0x0)
p += p32(0x5fe9c)    # pop {r1, r2, lr};
                     # mul r3, r2, r0;
                     # sub r1, r1, r3; bx lr
p += p32(0x0)
p += p32(0xff68732f)# '/sh\x00'
p += p32(0xbf30)     # pop {r3, pc}
p += p32(0x708a5)    # <buffer+4>
p += p32(0xe96c)     # str r2, [r3];
                     # pop {r3, r4, fp, pc}
p += p32(0x0)
p += p32(0x0)
p += p32(0x0)
p += p32(0x60894)    # pop {r0, pc}
p += p32(0x708a1)    # <buffer>
p += p32(0xb784)     # system@plt
```

Listing 4: A `system("/bin/sh")` chain generated by ropbot.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

This paper describes a code reuse payload generation engine and the appendix provides detailed information on how to reproduce the results mentioned in the paper. The artifact consists of the source code and the the artifact has been uploaded to Zenodo.

1) How to access: The artifact can be downloaded from Zenodo at https://zenodo.org/records/17811054. It contains both the source code and the binary dataset.

2) Hardware dependencies: There is no special hardware dependencies to run the artifact. However, to reproduce the exact results described in the paper, it requires a machine with 40 cores and 256GB RAM.

3) Software dependencies: `docker` is required to run the evaluation.

B. Artifact Installation & Configuration

Running the `build.sh` script will build the `ropbot` docker container, which takes less than 5 minutes to run. Then you need to download the dataset and unpack it, which will generate a `dataset` folder. We will use `<dataset>` to represent the path of the dataset folder throughout this appendix.

After the docker image is successfully built, you can do `./run <dataset> ropbot x64 test` to test the installation. If the installation is successful, it should print a ROP chain that does `0xfacefeed(0xdeadbeef, 0x40, 0x7b)` and show the corresponding registers in the final symbolic state.

C. Experiment Workflow

At a high-level, you can use the `run` command to run all the experiments mentioned in the paper. The list of possible experiments is listed in `task_list.txt`.

D. Major Claims

- (C1): `ropbot` outperforms state-of-the-art systems for the code reuse payload generation capability. This is proven by experiment E1 and E2, whose results are reported in TABLE II in the table.
- (C2): `ropbot` can generate code reuse payload for multiple architectures: x64, MIPS, AArch64, ARM, and RISC-V. This is proven by experiment E3, whose results are reported in Table IV in the table.

TABLE VII: Finer-grained ablation breakdown. Graph search and chain generation is fast. Graph optimization is slower on AArch64 because of the lack of gadgets, which leads to more iterations to optimize the graph.

Config	Task	Gadget Finding Time	Opt-Time	Graph Search Time	Chain-Time	Success	Total
Base	execve	27.8s	-	0.004s	1.1s	193	272
Base+Graph_Opt	execve	27.8s	2.2s	0.006s	1.3s	215	272
exrop	execve	33.6s	-	-	240.5s	90	272
Crackers	execve	83.6s	-	-	695.9s	0	272
SGC	execve	763.1s	-	-	1527.8s	27	272
Base	facefeed	11.0s	-	0.002s	0.1s	319	1022
Base+Graph_Opt	facefeed	11.5s	1.7s	0.003s	0.2s	391	1022
Crackers	facefeed	24.4s	-	-	237.7s	353	1022
exrop	facefeed	9.8s	-	-	72.8s	358	1022
Full-got_gadgets	facefeed	11.3s	3.4s	0.007s	0.3s	574	1022
Full-ret2csu_gadgets	facefeed	11.5s	6.5s	0.013s	0.3s	570	1022
Full-multi_bb_gadgets	facefeed	11.5s	3.3s	0.009s	0.3s	570	1022
Full-jmp_mem_gadgets	facefeed	11.5s	2.2s	0.005s	0.2s	457	1022
Full	facefeed	11.5s	6.9s	0.003s	0.4s	635	1022
AArch64_Full	facefeed	32.3s	58.5s	0.021s	0.5s	56	172
AArch64_Full+cond_br_gadgets	facefeed	237.6s	101.4s	0.020s	0.6s	55	172

TABLE VIII: Gadget normalization with graph optimization can slightly reduce the length of generated chains. The use of gadgets with conditional branches increases run time significantly.

Config	Task	Chain Len(byte)	Runtime
Base	facefeed	81.0	11.6s
Base+Graph_Opt	facefeed	78.9	13.3s
Base+Graph_Opt+cond_br_gadgets	facefeed	79.0	36.5s
Base	execve	134.0	29.0s
Base+Graph_Opt	execve	128.6	30.5s
Base+Graph_Opt+cond_br_gadgets	execve	129.5	117.4s

E. Evaluation

1) Experiment (E1): [3 human-minutes + 6 compute-hours] This experiment aims to evaluate `ropbot`'s capability in generaing the `0xfacefeed(0xdeadbeef, 0x40, 0x7b)` chain for the `x64_dataset` binaries (1022 binaries). The performance varies according to the compute power of the running machine. On a machine with 40 cores and 256GB RAM, it should succeed for 635 ± 3 binaries.

[How to] To run the experiment, you can do `./run <dataset> ropbot x64 facefeed`. It will launch a docker container in the background and you can use `docker logs <container_id>` to check its progress. When the experiment finishes, it will print `Experiment finished` in the log.

[Results] Once the docker containers prints `Experiment finished` in the log, you can extract the results from the container by `docker cp <container_id>:/experiment/output.jsonl <dst_path>`. Running `python results/pretty_print.py <dst_path>` will print out the summarized results, which match what we report in the paper.

2) Experiment (E2): [3 human-minutes + 3.5 compute-hours] This experiment aims to evaluate `ropbot`'s capability in generaing the `execve("/bin/sh", 0, 0)` chain for the `syscall_dataset` binaries (272 x64 binaries with `syscall` instructions). The performance varies according to the compute power of the running machine. On a machine with 40 cores and 256GB RAM, it should succeed for 244 ± 2 binaries.

To run the experiment, you can do `./run <dataset> ropbot x64 execve`. The workflow and result intepretation is the same as E1.

3) Experiment (E3): [10 human-minutes + 8 compute-hours] This experiment aims to evaluate `ropbot`'s capability in generating the `0xfacefeed(0xdeadbeef, 0x40, 0x7b)` chain for different architectures. Since our original experiment takes more than 24 hours, we propose a scaled-down version of the experiment: only evaluate it on AArch64 and ARM besides x64 (evaluated in E1). We believe this scaled-down experiment is enough to show that `ropbot` works on different architectures.

To run the experiment, you can do `./run <dataset> ropbot <arch> facefeed`, where `<arch>` is either `aarch64` or `arm`. The workflow and result intepretation is the same as E1.

F. Customization

We welcome artifact evaluators to evaluate the results of other tools and other architectures as well. All available commands are listed in `task_list.txt`.