Spring 2020

Exploiting More Binaries by Using Planning to Assemble ROP Attacks

Daroc Alden
University of New Hampshire, Durham

Exploiting More Binaries by Using Planning to Assemble ROP Attacks

BY

Daroc Alden

BS in Computer Science, University of New Hampshire, 2019

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science

in

Computer Science

May, 2020

This thesis was examined and approved in partial fulfillment of the requirements for the degree of Master of Science in Computer Science by:

Thesis Director, Wheeler Ruml, PROFESSOR, Computer Science

Dongpeng Xu, ASSISTANT PROFESSOR, Computer Science

Laura Dietz, ASSISTANT PROFESSOR, Computer Science

On April 17, 2020

Approval signatures are on file with the University of New Hampshire Graduate School.

DEDICATION

For my wonderfully supportive family.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Professors Ruml and Xu, who advised me well and encouraged me to push my thesis to be the best it can be.

I would also like to thank Professor Dietz, for agreeing to be on my committee, even though my research is in no way related to her areas of expertise.

Thanks also to the OpenBSD project, for giving me this idea, and to the Angr project, for writing a powerful symbolic execution engine so that I didn't have to.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Exploiting More Binaries by Using Planning to Assemble ROP Attacks

by

Daroc Alden

University of New Hampshire, May, 2020

Return oriented programming (ROP) attacks have been studied for many years, but they are usually still constructed manually. The existing tools to synthesize ROP exploits automatically, such as ROPGadget and angrop, are very limited by their ad-hoc design: they rely on matching fixed patterns and assembling gadgets in fixed ways. We propose a new method, PEACE, that uses symbolic execution and partial-order planning to assemble gadgets more flexibly. Our method incrementally selects gadgets to address a need in the partially-constructed exploit, and infers ordering constraints over those gadgets based on their effects. This approach enables PEACE to create exploits for many more binaries than existing tools. By creating a more flexible and powerful ROP attack generation tool, we hope to raise awareness of how much code is vulnerable to ROP attacks and give researchers better tools to test detection and mitigation techniques.

CHAPTER 1

Introduction

Return Oriented Programming (ROP) attacks, a form of code-reuse attack, work by overwriting the stack with a set of return addresses that cause the program to do something malicious. ROP attacks leverage buffer overflows or other vulnerabilities to write return addresses and other data to the stack of a running process. The process then eventually executes a return instruction, and the attack gains control of the program's control flow. This allows an attacker to make a program perform malicious behaviors. First proposed by Shacham, ROP attacks have been widely employed to perform practical and versatile attacks (Shacham, 2007; Checkoway, Davi, Dmitrienko, Sadeghi, Shacham, & Winandy, 2010). The popularity of ROP has also spurred research on defense techniques, such as Address Space Layout Randomization (ASLR) (Team, 2003), Address Space Layout Permutation (ASLP) (Bookholt, 2005), shadow stack-based approaches (Sinnadurai, Zhao, & Fai Wong, 2008), compiler-based mitigations (Mortimer, 2019), and Control-Flow Integrity (CFI) (Abadi, Budiu, Erlingsson, & Ligatti, 2005).

Most ROP exploits are created by hand; existing tools to generate ROP attacks have significant limitations. Most of these techniques deal with the inherent complexity of assembling ROP attacks by searching for fixed patters of machine code that can be assembled in particular ways. Snippets of machine code that end with an indirect jump are called *gadgets*. Gadgets are the fundamental building blocks of ROP attacks. ROPGadget (Salwan, 2011) was the first tool of this type. Later work included angrop (Shoshitaishvili, Wang, Salls, Stephens, Polino, Dutcher, Grosen, Feng, Hauser, Kruegel, & Vigna, 2016) that improved on ROPGadget by using symbolic execution (based on the Angr framework) to defeat simple obfuscation techniques. However, the ROP attacks produced by these tools rely on a fixed library of patterns to identify gadgets (leading to stereotyped ROP exploits with distinctive signatures that make the attacks easy to identify).

Furthermore, compilers equipped with ROP defense methods can avoid emitting code that matches the templates these tools use, effectively crippling them. What originally sparked my interest in this thesis was a presentation by a contributor to the OpenBSD project that talked about how they had made changes to LLVM's register allocation code so that it would choose registers that were encoded in a way that ROP attacks could take advantage of less often. They were able to substantially reduce the number of gadgets in the OpenBSD kernel.

In this paper we propose PEACE (Planning Exploits by Assembling Chains Efficiently) to effectively construct complex and diverse ROP attacks. Our method treats the construction of ROP payloads as a planning problem and uses a specialized planning algorithm to assemble ROP payloads. More specifically, PEACE uses symbolic execution to infer properties of possible gadgets, and then uses that information to construct a payload via partial-order planning. PEACE incrementally selects gadgets to address a need in the plan, and infers ordering constraints over those gadgets based on their effects. This flexible system can plan around constraints that would stymie other techniques by using symbolic execution to account for syntactic obfuscation and by generating a binary-specific plan that can be augmented to avoid detection techniques. Our evaluation shows that the flexible, principled, inference-based approach of PEACE is able to find more exploits in standard Ubuntu binaries than existing techniques.

Our source code and evaluation benchmark are available at `https://setupminimal.github.io/thesis.tar.xz`.

CHAPTER 2

Background

In this section, we introduce ROP attacks and planning concepts. We briefly discuss existing mitigations against ROP attacks. We finish by explaining the components of a ROP attack and what PEACE is designed to generate.

2.1 Return Oriented Programming

ROP attacks remain one of the most common kinds of remote-code-execution vulnerability (Carlini & Wagner, 2014). ROP payloads allow an attacker to use an out-of-bounds write to direct the execution of the program. This is done by placing fake return addresses on the stack. When a program 'returns' to one of those addresses, it starts executing a series of *gadgets*, small sections of code that end with a return instruction or other indirect jump. This allows the attacker to direct the control flow in the application by placing more gadgets' addresses on the stack. Chaining these gadgets together can allow the attacker to perform any calculation, or execute syscalls to perform actions on the computer with the permissions of the suborned process. ROP has been shown Turing-complete in general (Shacham, 2007), although the gadgets available in any particular binary may not allow constructing the needed algorithms for simulating a Turing machine. This means that ROP attacks have very few theoretical limits.

ROP attacks are possible on many architectures, including Harvard architectures (Francillion & Castelluccia, 2008). On architectures that allow unaligned access to instructions (such as x86 and x86_64) they can be particularly devastating because of the possibility of *unaligned* or *unintended* gadgets. These are gadgets that result from an alternate decoding of the instruction stream that does not line up with what the compiler intended. These gadgets are important when considering ROP attacks, and should be kept in mind when considering the impact of the various mitigation

techniques that exist to hinder ROP attacks, because mitigation techniques involving modifications of the machine code must consider unintentional gadgets.

Exploiting a binary using a ROP attack consists of gaining arbitrary code execution, which an attacker may leverage to accomplish other aims. Ultimately, this is accomplished by one of a few methods. For example:

- Making a call to `execve` or other related procedures that allow an attacker to replace the running process with `/bin/sh` or some other known binary.
- Making a call to `mprotect` to mark a page containing attacker-controlled content executable, and then jumping to it.
- Making a call to `mmap` or `mremap` to map an attacker-controlled file as executable, and then jumping to it.
- Making a call to `fchown` or `fchmod` to change permissions on a file.

Our initial implementation of PEACE focuses on a `execve` call, but as will become clear, the technique generalizes easily to other approaches. In order to execute this syscall, a ROP attack needs to set the contents of specific registers and then jump to a `syscall` instruction.

2.2 Existing ROP Synthesis Methods

ROPGadget: ROPGadget (Salwan, 2011) was one of the first ROP synthesis methods. It focused on identifying occurrences of a pre-defined set of gadgets and then putting them into a pre-defined template. This approach works well on large binaries where instances of the gadgets it searches for are likely to be present, but is inherently very brittle.

Angrop: Angrop (Shoshitaishvili et al., 2016) uses a similar technique that leverages the Angr framework's symbolic analysis capabilities. Symbolic execution is a binary analysis technique where an emulated CPU executes a sequence of instructions with symbolic variables from a constraint solver as values, which can be used to discover generic information about that sequence of instructions. Angrop uses symbolic execution to match slight variations on its templates — for example,

ignoring `nop` instructions. Once it has identified gadgets, however, it still combines them with an inflexible template.

Both of these methods assume that they know the ASLR offsets or ASLP addresses of gadgets in the binary, as well as the location of at least one writable page. These assumptions are not too strong, and PEACE leverages the same information. We go into more detail about why this is reasonable in the discussion section.

2.3 Planning

Planning is an area of artificial intelligence focused on finding sequences of actions that lead through a space of possible states to a goal (Norvig, 2002; Ghallab, Nau, & Traverso, 2016). In the case of PEACE, a state is a partially constructed exploit and an action is a gadget that can be added to that exploit, so that a goal is a complete exploit of the binary. Planning has been used before for program synthesis, although usually at a relatively high level and not to our knowledge in the context of ROP attacks. For example, Bhansali (1991) uses hierarchical planning and analogical reasoning to generate simple Unix shell scripts from high-level specifications. Ireland and Stark (Ireland & Stark, 2006) use proof planning and partial-order planning to generate correct imperative programs from specifications. Planning has also been used to automatically synthesize workflows of web services (Bertoli, Pistore, & Traverso, 2010). In generalized planning, a planner attempts to find a small program, often expressed as a finite-state controller, that solves a class of problems instead of a single given problem (Srivastava, Immerman, & Zilberstein, 2011). In the context of computer security, planning has been used to generate attacks on models of computer systems and networks as an aid in penetration testing (Boddy, Gohde, Haigh, & Harp, 2005; Hoffmann, 2015). None of these works focus on code at the assembly code level or on ROP attacks.

The planning approach most relevant to PEACE is called heuristic graph search (Edelkamp & Schrödl, 2011). Graph search conceptually operates on a directed graph of discrete states. There is an arc from state A to state B if there is a valid action in state A that results in state B. The simplest form of graph search would be a breadth-first search, where we first examine every state distance 1 from our initial state, then every state distance 2, and so on, until we find a goal.

If we have no other information about the structure of the graph, this is the best that we can do (in terms of the number of states that we have to examine before finding a goal). When working on a specific problem, however, we often have *heuristics* that give us additional information about how likely a state is to lead to a goal. We can use these heuristics to guide the graph search so that it finds a goal much more quickly (Hart, Nilsson, & Raphael, 1968). Planning by examining the area of the graph near an initial state and progressing towards a goal state is called *progression* planning. Planning by examining the area near the goal and working backwards is called *regression* planning. As we will see in Section 3.6, in the case of PEACE, regression planning has an advantage because we can make better heuristics for it.

There is also an optimization that we can make to cut down the size of the graph. Planning can construct a sequence of actions in either a total order or a partial order. Partial order planning essentially takes the quotient of the graph by equivalence under re-ordering — i.e. we merge states that are identical when re-ordering actions that don't depend on each other (Weld, 1994). This makes it more costly to calculate the effect of actions, but can reduce the size of the graph by a factor of $d!$ in the best case, where d is the size of the solution found in number of actions. Partial order planning is a good fit for PEACE because we expect that many of the actions in the plans we explore will be independent.

CHAPTER 3

The PEACE Approach

In this section, we describe a novel method called PEACE to synthesize diverse and complex ROP chains. We first present a high-level picture of the whole methodology with an motivating example. Then we elaborate every component in detail.

3.1 Overview

PEACE constructs ROP exploits using symbolic execution and partial-order regression planning. Given a binary, it harnesses symbolic execution and a constraint solver to represent the functionality of every gadget in the binary. Next, the gadgets are selected and assembled by a planner to construct a complete ROP chain, which forms an exploit of the original binary.

The major research challenge is the enormous number of states which arise when searching for a valid ROP chain due to many possible gadgets in any given binary. We design powerful heuristics to guide the search to make it efficient, and use the techniques described in Section 2.3 to reduce the size of the state space to search by identifying plans that are equivalent under valid re-orderings and representing them as a single entity.

Figure 3-1 presents a high-level sketch of PEACE's architecture. PEACE includes the following five steps:

1. *Gadget Extraction.* PEACE extracts ROP Gadgets from a binary code, e.g., x86_64 ELF Linux executables. It seeks to decode from every valid position in the binary, until reaching certain forms of jump instruction. The valid starting positions depends on the details of the architecture of the binary, such as instruction alignment and design of instruction length.
2. *Loop Joining.* From the list of potential gadgets, PEACE selects the subset of gadgets that end in a controllable indirect jump. In particular, it determines that the ending jumps either

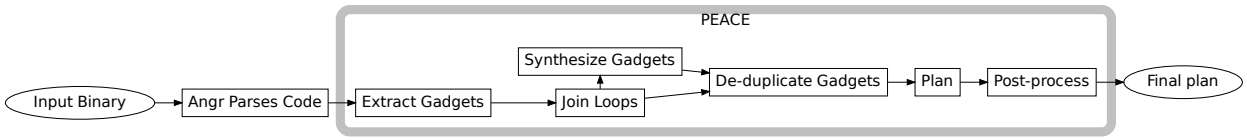

Figure 3-1: The high-level stages of PEACE.

have exactly one target, or can be fully controlled. To make analysis easier, any potential gadget that can only branch to one location is joined with the gadget from that location, and treated as a single gadget from then on. This lowers the number of gadgets that need to be considered for inclusion in the exploit. Any circular references — infinite loops — are dropped at this point.

3. *Library Filtration.* PEACE performs subsumption testing to winnow the list of gadgets down to a minimal subset. Gadgets are discarded if their function can be completely replicated by another gadget.
4. *Heuristic Search* PEACE chains the identified gadgets together as described in Section 3.6.2, resulting in a completed partial plan (described in Section 3.5). The output of this step is a set of gadgets and ordering constraints describing which gadgets should be executed, and in what order.
5. *Post-processing.* As the last step, PEACE transforms the planning result into a linear ROP payload. The payload uses existing gadgets in the binary to make the computer evaluate a syscall with arguments we control.

Note that PEACE only needs to find a feasible plan, instead of seeking the shortest or most efficient plan. Therefore, PEACE adopts optimizations to reduce the amount of time or memory required to construct a plan, at the expense of potentially finding more complex plans. One example of this is how PEACE handles detecting looping or redundant plans. In most planners, two partial plans with similar current open dependencies (see below) could still be distinct if neither could yet be shown to lead to the unambiguously better solution. PEACE can discard partially-formed plans more aggressively.

3.2 Motivating Example

We present a small motivating example to illustrate our method. I recommend skimming the motivating example, reading the rest of section 3, and then jumping back to the example when you want a concrete explanation of something. Consider the following example binary (x86_64 machine code): `48 31 c0 c3 ff c0 c3 5f c3 48 89 f7 c3 48 31 d2 c3 48 01 d6 c3 48 ff c0 29 d3 75 f9 c3 00`. After decoding into assembly language, Table 3-1 presents all gadgets that are not simply 32-bit versions of gadgets already present.

Synthetic gadgets: From these gadgets, PEACE first forms two additional synthetic gadgets: a combination of gadgets 1 and 2 to set the value in `%eax` (referred to as gadget 13), and a combination of 5 and 7 to generate the equivalent of `mov %rdx, %rsi` (referred to as gadget 14). This process is explained in more depth in Section 3.3. The key advantage of combining gadgets early like this is that it reduces the amount of work later phases have to do to chain gadgets together. As you will see over the course of the example, creating synthetic gadgets greatly decreases the number of interactions that have to be considered when planning, which lets the plans be shorter and more flexible in terms of ordering.

Library filtration: During the library filtration phase, gadgets 3 and 4 get combined to set the contents of `%rsi` (making gadget 15). Then gadget 15 can be combined with gadget 14 to set the value of `%rdx` (making gadget 16).

The library filtration phase also removes gadgets 2, 4, 5, 7, 8, 9, 10, 11, 12, and 14 for being redundant. This leaves us with the gadgets in Table 3-2. This is also the phase where the 32-bit versions of various gadgets actually get dropped if there are 64-bit versions available. They were not included earlier only because they would almost double the number of gadgets in the example, making it harder to follow.

Planning: In the planning phase, we start with an initial partial plan where the set of unfulfilled dependencies that we are seeking to resolve contains the goals listed in Figure 3-2. These goals make a syscall on Linux to `execve`. PEACE expands this plan by trying to find a way to set `%rdi`.

Table 3-1: Gadgets in the motivating example.

No.	Binary Code	ROP Gadget
1	48 31 c0	xor %rax, %rax
	c3	ret
2	ff c0	inc %eax
	c3	ret
3	5f	pop %rdi
	c3	ret
4	48 89 f7	mov %rsi, %rdi
	c3	ret
5	48 31 d2	xor %rdx, %rdx
	c3	ret
6	d2 c3	rol %cl, %bl
	48 01 d6	add %rdx, %rsi
	c3	ret
7	48 01 d6	add %rdx, %rsi
	c3	ret
8	48 ff c0	inc %rax
	29 d3	sub %edx, %ebx
	75 f9	jne <bad>
	c3	ret
9	c0 29 d3	shr BYTE PTR [%rcx], 0xd3
	75 f9	jne <bad>
	c3	ret
10	29 d3	sub %edx, %ebx
	75 f9	jne <bad>
	c3	ret
11	75 f9	jne <bad>
	c3	ret
12	f9	stc
	c3	ret

In this case, applying gadget 3 in Table 3-2 results in a partial plan of $\{gadget_3\}$ where goal 2 has been fulfilled.

Table 3-2: Remaining gadgets after the library filtration process.

No.	ROP Gadget	Behavior
1	`xor %rax, %rax` `ret`	Reset `%rax`
3	`pop %rdi` `ret`	Set `%rdi` to some value

6	`rol %cl, %bl`
	`add %rdx, %rsi`
	`ret`
		Rotate `%cl` and add to `%rdx`

13	`xor %rax, %rax`
	`ret`
	`inc %eax`
	`ret`
		Set `%eax` to some value

15	`pop %rdi`
	`ret`
	`mov %rsi, %rdi`
	`ret`
		Set `%rsi` to some value

16	`pop %rdi`
	`ret`
	`mov %rsi, %rdi`
	`ret`
	`xor %rdx, %rdx`
	`ret`
	`add %rdx, %rsi`
	`ret`
		Set `%rdx` to some value

PEACE continues to expand this plan, using gadgets 13, 15, and 16 to fulfill all the goals. Considering ordering constraints, gadget 16 interferes with gadget 15, and both interfere with

1	`%rax` = 57
2	`%rdi` points to the value ‘‘`/bin/sh`’’
3	`%rsi` points to 0
4	`%rdx` points to 0

Figure 3-2: An initial set of goals for executing an `excve` call.

gadget 3. Therefore, the following might be the order of gadgets produced: gadget 16, gadget 15, gadget 3, gadget 13.

Post-processing: From this partial plan, and given information such as ASLR offsets, PEACE produces a ROP payload that can be written to the stack. This payload includes addresses of gadgets intermixed with data needed for, e.g. `pop` instructions, as well as static data that needs to be referred to by a pointer.

We will now consider the major steps of PEACE in more detail.

3.3 Identifying Gadgets

As the first step towards a full ROP exploit, we seek available gadgets in a binary. This is the step labeled “Identifying Gadgets” in the motivating example. We start decoding gadgets from all valid positions in the binary, and use symbolic execution to quantify the gadgets according to several metrics. Unlike Angrop (Shoshitaishvili et al., 2016), we deliberately include gadgets that contain unconditional branches. The purpose of the loop joining step is to turn some of these into useful gadgets. Since ROP attacks can only continue to be effective if they maintain control over the instruction pointer, most techniques completely disregard gadgets where the IP is not fully controlled afterwards. By seeing where it is possible for gadgets ending in constrained jumps to jump to, we can potentially regain full control over the instruction pointer. The main advantage of this is that it allows PEACE to consider gadgets that contain loops or other complex flow control. Specifically, we recover this information about each gadget:

- Length in bytes

Length	4
Registers clobbered	$\emptyset$
Registers controlled	{`rdx`}
Type of jump	`ret`
Source location	16
Preconditions	$\emptyset$
Postconditions	$\texttt{rdx}_{\text{after}} = \texttt{rdx}_{\text{before}} + \texttt{rsi}_{\text{before}}$

Table 3-3: Example values for Gadget 7

- Registers clobbered
- Registers controlled
- Type of jump used
- Source location
- Symbolic constraints required for gadget to run successfully with the IP still controlled: requirements for register contents, and requirements for registers to contain readable/writable pointers. These are represented as symbolic variables and constraints in the underlying constraint solver, Z3 (De Moura & Bjørner, 2008) in our case.
- Symbolic constraints representing the effects of the gadget.

This information is stored in a data-structure representing the gadget. An example is given in Section 3.3 The symbolic constraints are stored in the following way: for every register in the current architecture we store two symbolic variables — the state before and after the gadget executes. The postconditions of the gadgets are represented as constraints relating the before and after variables. For example, if gadget 7 from the motivating example had $\texttt{rdx}_{\text{before}}$, $\texttt{rdx}_{\text{after}}$, and $\texttt{rsi}_{\text{before}}$ as symbolic variables (64-bit bit-vector), its effect would be recorded by the following

formula constraint in CVC (The CVC development team, 2020) format.

$$\texttt{rdx}_{\text{after}}, \texttt{rdx}_{\text{before}}, \texttt{rsi}_{\text{before}} \ : \ \texttt{BITVECTOR}(64)$$

$$\texttt{rdx}_{\text{after}} = \texttt{BVPLUS}(64, \texttt{rdx}_{\text{before}}, \texttt{rsi}_{\text{before}})$$

We also have some constraints that are generated by loads from or stores to memory, that require that a particular value is a pointer to a readable or writable page. To express this, we extend Z3's constraint language with an additional predicate, `POINTER`. Since analyzing whether an arbitrary address is readable or writable is difficult in general, PEACE restricts these reads or writes to occur on the stack.

Symbolic constraints representing preconditions are stored in a separate list, but still stated in terms of the before variables. For example, if gadget 10 from the motivating example had $\texttt{edx}_{\text{before}}$ and $\texttt{ebx}_{\text{before}}$ as variables, its precondition would be represented as $\texttt{edx}_{\text{before}} = \texttt{ebx}_{\text{before}}$. If the value of a register would depend on a read from memory that we control (e.g. the stack), the after variable is left unconstrained, so that it is free to take on whatever value is necessary in the rest of the plan.

We could restrict the set of gadgets we consider in a few ways, for example by only considering call-preceded gadgets, which are more likely to bypass CFI-based countermeasures. In principle, any ROP detection technique could be used as a black-box filter at this stage, to cause the planner to only consider gadgets that cannot be analyzed by the given technique. PEACE does not implement this, but it might be an interesting possible expansion.

One unique optimization PEACE makes is to synthesize additional *synthetic gadgets*. Synthetic gadgets are sequences of gadgets that are treated as a unit for the purposes of planning and that together perform some desirable behavior, such as "set `%rsi` to some value". We decided to include synthetic gadgets in the design of peace because they reduce the size of the exploits PEACE finds on the two small binaries that we were using for benchmarking during development by 54 gadgets, with a commensurate speedup. PEACE constructs synthetic gadgets using two different patterns: one that combines a gadget that zeros a register with one that increments that register, and one

that combines a gadget that zeros a register with ones that perform basic arithmetic manipulations with other registers as sources. These patterns are partially taken from existing techniques, and partially constructed to work well with PEACE. Note that synthetic gadgets are not actually gadgets present in the binary, but rather chains of gadgets that we represent as a single entity. One example of a synthetic gadget is gadget 16 in Table 3-2. By chaining four smaller gadgets together, it implements the behavior "set `%rsi` to some value".

This step greatly increases the efficiency of the search. In the language of search, it reduces the distance from the initial state to a goal state at the expense of increasing the number of available actions in each state. Practically, introducing more actions is not a problem both because of the library filtration step described below, and because if b is the average number of actions available in a state, the number of states within distance d of the initial state is b^d. Optimizations that decrease d are most important for keeping the size of the search feasible.

3.4 Library Filtration

The gadget extraction procedure described above results in an enormous number of gadgets in most binaries, the number of available actions in our search will be very high. The synthesis of useful synthetic gadgets is a technique to shorten the depth of the search, to keep the number of states PEACE needs to consider smaller. Another effective way to reduce the number of states is to remove redundant or irrelevant gadgets. We call this process library filtration — a kind of subsumption testing.

Subsumption Testing: One way to reduce the number of gadgets is by performing subsumption testing — for every pair of gadgets A and B, if A can produce all the same effects as B, B is redundant, and we can completely remove it from our library of available gadgets. This is determined by looking at each gadget's pre- and post-conditions, and seeing if there is a gadget with a subset of the preconditions and a superset of the postconditions. We can determine whether a condition is a sub- or super-set of another condition by querying the constraint solver to see whether $cond_1 \wedge \neg cond_2$ and $\neg cond_1 \wedge cond_2$ is satisfiable. If both queries are satisfiable, they are

not super or subsets. If both are not satisfiable, they are equal. Otherwise, we can determine which one is a subset of the other. Effectively this means that we keep exactly one gadget with each kind of effect (one `mov` between each set of registers, one `add` to each register, etc.). If there is a tie (i.e. A subsumes B and B subsumes A), we keep whichever of the gadgets is less amenable to analysis, to make detection techniques less effective. For example, the 64-bit version of an instruction almost always subsumes the 32-bit version of the instruction, because the set of states in which it could leave a register is a strict superset of the 32-bit version. An example of what this entails is given in the step labeled **Library filtration** in the motivating example. Empirically, it reduces the set of gadgets in our library by an average factor of 2.97, which yields a significant speed-up during search.

I think there might be a better way to perform the subsumption testing. Currently, have an $O(n^2)$ loop over the library of gadgets, comparing each one to the unique gadgets so far. Each subsumption test involves several queries to the constraint solver (one for each register the gadgets touch), which is expensive. A better implementation might involve using a disjoint-set data-structure. I didn't explore this because the naïve approach was sufficient on binaries that were small enough to actually perform planning on.

Library Representation: One final way to reduce the number of actions to consider at each step is to index the available gadgets by what registers they effect, which allows the planning phase to easily select which gadgets are relevant for achieving a given goal. Selecting gadgets in this way, instead of considering all gadgets in all states, is able to substantially reduce the branching factor of the search. In our implementation of PEACE, this is simply done by representing the library as a python dictionary keyed on register name.

Independent gadgets: Another technique to reduce the length of our search — that we didn't implement — is to scan through our library of gadgets to find gadgets without dependencies — that is, that don't require any particular pre-requisite to run correctly — and then combine these gadgets with other gadgets that use their effects to try to reduce each gadget's overall number of dependencies. We could turn each gadget that has a dependency that we can fulfill in this way

into a synthetic gadget that doesn't have that dependency by chaining it with the gadget with no dependencies. This reduces the number of inferential steps that our search needs to consider, and makes our library of gadgets more generally useful.

3.5 Dependencies and Partial Plans

We now use planning to combine the gadgets resulting from the previous phases. As discussed in Section 2.3, we use partial order planning, because it allows us to substantially reduce the search space. As we construct the partial plan, we keep track of which registers we still need to find a way to control at any particular point. We call these the *dependencies* of a gadget, because the gadget depends on them being fulfilled. For example, the gadget `xor %rax, %rax; ret;` does not have any dependencies, but the gadget `inc %rax, sub %edx, %ebx; jnz .somewhere_bad; ret;` requires `%edx` and `%ebx` to be equal in order to be usable. Every dependency is represented by a constraint in the underlying constraint solver's language that needs to be satisfied in order to use the gadget.

The nodes in PEACE's search space each represent a partial plan. Specifically, they contain a candidate attack represented as a 5-tuple $\langle \alpha, \beta, \gamma, \delta, \epsilon \rangle$:

- α is a set of *gadget instances*, which are particular instances of a gadget present in the binary.
- β is a set of ordering constraints over α. These ordering constraints are represented as a simple temporal network (Cesta & Oddi, 1996; Dechter, Meiri, & Pearl, 1991).
- γ is a set of causal links over α. A causal link (of the form $a_i \overrightarrow{p} a_j$) is a commitment by the planner that a given post-condition p of a gadget $a_i \in \alpha$ fulfills a dependency of $a_j \in \alpha$ or of the final goal.
- δ is a set of dependencies that have not yet been fulfilled.
- ϵ is a set of causal links that are currently *unsafe*, meaning that there exists a threatening action $a_k \in \alpha$ that would negate the post-condition of the causal link and could be ordered between a_i and a_j without violating the ordering constraints.

When there are no unfulfilled dependencies and no unsafe causal links, the node represents a complete plan that leads from any state of the computer to an exploit.

When representing the plan in our implementation, we represent a gadget instance as a separate data structure that points to the gadget, the time point representing where in the plan the gadget is executed, and how the gadget's dependencies are fulfilled. This lets us share static information about gadgets between uses, and use the same gadget multiple times in a plan unambiguously.

Our internal representation of partial plans is based on an immutable data-structure that allows related plans to share data. Unfortunately, representing simple temporal networks the same way proved problematic. PEACE represents STNs as an adjacency matrix, and uses the Floyd-Warshall all-pairs-shortest-path algorithm to check them for consistency. Unfortunately, this is $O(n^3)$ in the number of time points in the network. A better implementation would support partial updates and/or immutable representations of the STN(Cesta & Oddi, 1996). This didn't turn out to be too important, because by the time the size of the STN became problematic, the search-space of partial plans itself was too large to search in a reasonable amount of time.

3.6 Search

In this section we present how to construct partial plans and carry out the heuristic graph search. We first describe how to construct possible successor plans, then describe the actual search process, and end by discussing how to prevent infinite loops.

3.6.1 Successor Plans

Adding available actions to a plan is done by finding all valid successor plans. Given some particular partial plan, we create successor plans by selecting an open dependency in δ, and try to find gadgets that can fulfill it. If we fail, the plan is a dead-end and can be discarded. Otherwise, we generate one successor plan for each gadget that has a postcondition that can fulfill the open dependency. The postcondition is unified with the dependency by taking the symbolic variables representing the state of the relevant register and unifying them. So the $\texttt{reg}_{\text{before}}$ from the gadget with the open dependency is unified with the $\texttt{reg}_{\text{after}}$ from the gadget being added to the plan. Then we add the gadget to α, its dependencies to δ, an ordering constraint to β, and a causal link protecting the dependency to γ.

Usually in partial-order planners, successor plans are also constructed by seeing if there are additional causal links that can be added to the plan using the already instantiated gadgets. We don't need to do that because we're not trying to find the shortest plan, only a plan that will suffice. This reduces the number of potential successor plans.

When we are generating successor plans, we choose the dependency to focus on by selecting the dependency for which the fewest gadgets exist. The logic behind this choice is that it reduces our branching factor, and eliminates impossible plans quickly.

Generating a successor plan in this way may result in a plan where $\epsilon \neq \emptyset$. We then need to eliminate all of the *unsafe* causal links in ϵ by finding a partial order that doesn't violate any of the dependencies of the gadgets in the plan. We do this by choosing to either *promote* or *demote* each gadget that threatens a causal link, generating several new successor plans. Promoting or demoting a gadget involves adding a new ordering constraint that the gadget happen after the last gadget it threatens, or before the first, respectively. This prevents it from threatening the causal link. We expand all of these successor plans in this way until $\epsilon = \emptyset$.

3.6.2 Planning Algorithm

Now that we have a representation of a search space over possible plans, an initial node, and a successor function, we can turn to the algorithm for performing that search. Goal nodes are plans where $\delta = \emptyset$ (recall that $\epsilon = \emptyset$ due to threat resolution when successors are generated). Since each plan may have many possible successor plans, the key to our search algorithm is to find a goal node while expanding as few plans as possible, since each plan takes a non-trivial amount of computation to elaborate. We accomplish this by using a specific heuristic to choose which plan to elaborate next.

Algorithm 1 presents pseudocode for the whole planning process. The for loop beginning on line 9 enumerates the possible successor plans. Since we use regression search, meaning that we start from our desired exploit and work backwards, we first consider the plan consisting only of the dependencies needed for the exploit. For example, Figure 3-2 shows a possible initial state. The algorithm starts by inserting that initial state into a priority queue (lines 3 and 4). We use a greedy

Algorithm 1 PEACE Planning Algorithm.

Input: A goal G, and a library of gadgets L.
function SEARCH(G, L)
  queue ← EmptyQueue()
  Add EmptyPlan() to *queue* ▷ Section 3.5
  while *queue* is not empty **do**
    best ← pop(*queue*)
    register ← minBy(dependence(*best*), $\lambda r.\ len(L(r))$) ▷ $L(r)$ refers to those gadgets that can influence register r, as explained in Section 3.4
    gadgets ← $L(register)$
    for *gadget* in *gadgets* **do**
      plan ← addToPlan(*gadget*, *best*) ▷ Section 3.5
      if *plan* has unsatisfiable constraints **then**
        continue
      else if *plan* has no dependencies **then**
        return *plan*
      end if
      Add *plan* to *queue*
    end for
  end while
  return *failure*
end function

best-first search (implemented by the while loop beginning on line 5), meaning that the priority queue is ordered by our heuristic, described below. We pull plans from the priority queue one by one, expand them, and put the successor plans back in the queue. Lines 11 through 15 explain how the successor plans are evaluated and added to the queue. We do this until a complete plan is found.

We order the priority queue in the following way, in decreasing order of importance:

1. Increasing number of remaining dependencies. This causes the search to try first plans that are probably almost done.
2. Increasing number of constraints on symbolic variables present in the plan. This biases the search towards plans that deal with concrete values, because they are cheaper to enumerate and elaborate, while still allowing the planner to deal with symbolic variables when necessary.
3. Increasing complexity. This term is based on the sum of the numeric values of open dependencies, plus a small constant term for pointer-based constraints. This is useful because it generally biases the search towards numbers that are small, and therefore easy to generate from scratch with combinations of increments and mathematical operations.

This heuristic was chosen by observing how different possible heuristics influenced PEACE's behavior on a small collection of binaries that we used to help guide our implementation. Without item 1, PEACE has no reason to prefer plans that are nearly complete, and so frequently fails to terminate. Without item 2, PEACE will consider plans with increasing numbers of symbolic constraints. This alone is not a problem, but often there is a very similar concrete plan that would require fewer calls to the constraint solver. Including item 2 makes PEACE perform much better by resorting to plans with many symbolic variables only when necessary. Without item 3, PEACE tends to explore lots of possible mathematical manipulations of register contents, without any notion of which numbers are 'easier' to achieve. Since zeroing a register is a common operation, biasing the search towards low numbers helps PEACE find ways to set registers quickly.

One interesting possible extension to PEACE would be to design better heuristics. I think that heuristics that take the gadgets available in a binary into account can probably significantly outperform the heuristic we use here.

3.6.3 Dealing with Loops

Plans with identical dependencies are interchangeable for our purposes. Whenever we generate a plan, before we insert it into the priority queue, we check that it is not completely subsumed by another plan in the queue. Specifically, when there exists another plan with dependencies δ_1 which contains more general constraints than the new plan, the new plan can be considered superflous. In other words, whenever for every dependency d in δ_2, either δ_1 has no dependency on the register involved in d, or the set of values for that register that would satisfy the dependencies in δ_1 that depend on that register is a superset of the set of values that would satisfy d, the plan can be discarded. As in subsumption testing, this is determined for each register by using the constraint solver to determine whether there are any values that would satisfy the dependency from δ_1 but not d.

We also check each plan against its ancestor plans, which prevents the search from looping. Otherwise, there could be a pathological problem where two dependencies that can each be easily discharged by gadgets that introduce the other would cause a loop. Without tracking previously examined plans, gadgets of this kind would give rise to chains containing cycles of the form "A - B - A - B - A ...".

3.7 Post-processing

After finding a plan, we have a partial order between the gadgets involved. This is much more flexible than other tools — which usually result in a total order — because we can potentially disguise the ROP attack. By using the partial ordering information and the causal links, we can find places in the exploit where additional, extraneous gadgets can be inserted without threatening the purpose of the ROP chain. This can break up the profile of the attack, making it harder to detect. For example, many ROP detection techniques rely on global state, which may be confused by threading. Inserting calls to `yield()` can cause the program to switch threads, potentially dirtying the transition buffer that some techniques use (Pappas, 2012).

CHAPTER 4

Implementation

Our implementation of PEACE is remarkably slim, coming in at just under 1,500 lines of python on top of the Angr framework (Shoshitaishvili et al., 2016), which does the heavy lifting of parsing different binary formats, performing symbolic execution, and providing access to Z3 (De Moura & Bjørner, 2008) as a constraint solver. We do have to extend Vex, Angr's representation of Z3's constraint language, a little to be able to represent constraints on whether or not a register constains a pointer to a particular value. For example, executing a `excve` syscall requires a register to point to the value 0 in memory. Angr cannot natively represent this constraint, so we represent these kinds of constraints by wrapping them in a custom datatype. Vex is a generic AST for representing constraints; since we only care about whether values are valid pointers when dealing directly with the contents of registers, we can just wrap existing Vex expressions in an additional layer to represent pointer constraints.

We define two key data structures to use during the algorithm; the record of a gadget's attributes, and the data structure representing partial plans. We present them in Figures 4-1 and 4-2 to ease understanding and replication of our implementation — there is nothing special about this representation.

Since our implementation was mainly aimed at evaluating the feasibility of PEACE, several steps could be taken to improve the performance of the method. Notably, our representation of simple temporal networks was naive, and became the dominant factor affecting performance when the length of plans exceeded about 50 steps. Our search process only reached that depth on fairly small binaries (see Figure 4-3 for details on the size of exploits found), so it was usually not a limiting factor in PEACE's performance.

Addresses of all basic blocks in the gadget
The symbolic state of the machine before and after the gadget's execution
A map from a register to pre-requisite conditions
A map from a register to registers that can influence that register's final value
A set of registers that can be changed by executing the gadget

Figure 4-1: Fields in the gadget data structure

A simple time network (Cesta & Oddi, 1996) relating gadget instances
An open list of gadget instances with unresolved dependencies
A set of protected causal links
A pointer to the parent plan
A reference to the constraint solver

Figure 4-2: Fields in the partial plan data structure

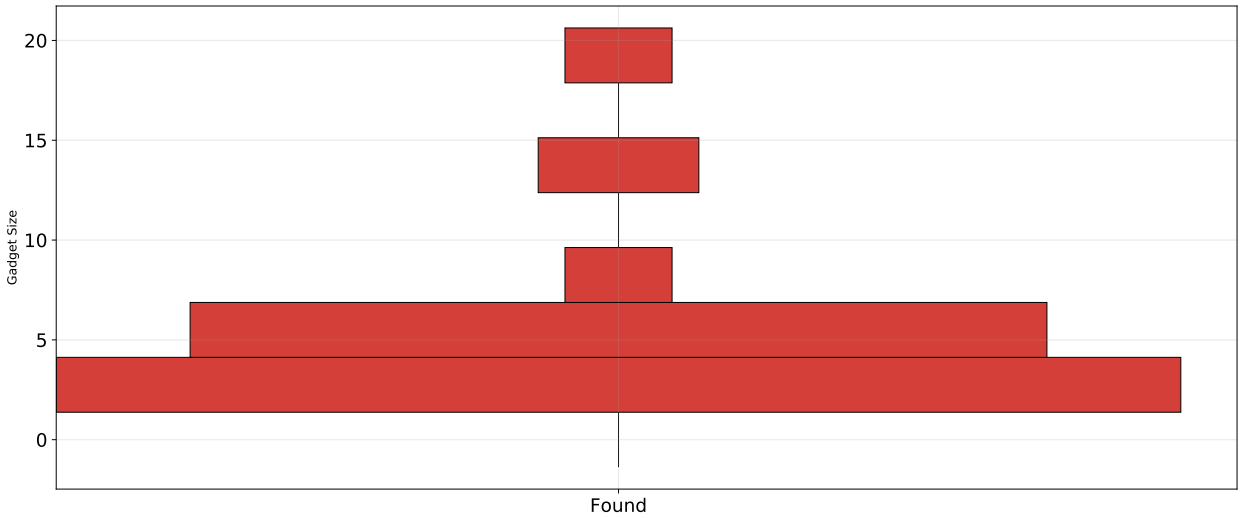

Figure 4-3: Size of ROP payloads found, in number of gadgets (including synthetic gadgets as one entry)

CHAPTER 5

Evaluation

We will begin by describing our experimental setup, and how the various previous methods can be compared to PEACE, and then present our results in detail.

5.1 Experimental Setup

There are several existing implementations of similar programs for automatically assembling ROP payloads.

There were several prior implementations that we were unable to evaluate. One used a very different technique and, as of this writing, is still under active development (Simplex, 2020). Another had a very detailed paper, but no available code (Follner, Bartel, Peng, Chang, Ispoglou, Payer, & Bodden, 2016). There were two other approaches that we could not compile, and they did not appear to have any published results (Souchet, 2017; Pakt, 2013). We carefully read their code, and concluded that it was structured largely similarly to Ropper (Schirra, 2019) and ROPGadget (Salwan, 2011), which we discuss in detail below.

Some of the tools did not assume that they would know the address of a writable section at runtime. Since this is one of the assumptions we made, where necessary we modified the tools to continue running as though they had found such an address. We believe that with that adjustment, all of the tools being compared make the same assumptions about what is needed for a successful ROP attack.

Our complete source code, along with a Nix expression for its dependencies, this set of binaries, pinned versions of the other tools we tested, and notes describing our experimental setup are available at `https://setupminimal.github.io/thesis.tar.xz`.

Table 5-1: Number of binaries successfully exploited by various programs.

Tool	Exploited (out of 361)	Percent(%)
ROPGadget (Salwan, 2011)	9	2.4
Ropper (Schirra, 2019)	0	0.0
Angrop (Shoshitaishvili et al., 2016)	0	0.0
PEACE	44	12.1

To systematically investigate the performance of PEACE versus previous work on a large corpus, we collected 361 binaries from the Ubuntu 18.04 Desktop installation image. We extracted every executable present in the image for use as our benchmark. Since Ubuntu is a very popular distribution, every binary in the benchmark is a widely-deployed open-source program.

5.2 Capability of Synthesizing ROP Chains

For each of the implementations that we were able to evaluate, we ran them over our corpus and counted how many binaries they were able to automatically exploit. On our corpus, PEACE was able to create ROP chains for 44 binaries, nearly 5 times as many as ROPGadget. (See Table 5-1). Success was self-reported by the tools. We considered ROPGadget failing only because it couldn't identify a writable page as a success, because that was a foundational assumption that other tools made. We did not attempt to actually exploit the binaries with the generated ROP chains because we did not have buffer overflows suitable for testing. An additional point of interest is that the sets of binaries exploited by ROPGadget and by PEACE were disjoint. This may be because ROPGadget performed well on large binaries where it could find many gadgets, but PEACE performed well on small binaries that were possible to analyze exhaustively.

We carefully investigated the existing work to analyze why they failed to synthesize as many ROP chains. Ropper and ROPGadget both take very similar approaches. They scan the executable starting from every possible address, looking for gadgets that match a set of templates. This

matching has to be exact. Inserting a `nop` instruction could prevent ROPGadget from recognizing a gadget. Then these gadgets are strung together in a pre-determined order.

Angrop, in contrast, leverages the capabilities of the Angr framework. It likewise scans all gadgets from every possible address, but then filters them by a few specific criteria. For Angrop to consider them, gadgets must meet the following requirements: be fewer than 20 bytes long, perform fewer than 2 memory writes, perform fewer than 2 memory reads, end with a return instruction, and not contain a jump. These restrictions significantly reduce the number of gadgets that Angrop can use, and makes it easy for ROP detection techniques like kBouncer to detect that many indirect jumps in a row are occurring. The attributes of each gadget were discovered using symbolic execution. This prevents the tool from being deterred by `nop`s and other irrelevant instructions. Then the gadgets are assembled in a pre-determined order.

PEACE includes gadgets that contain all kinds of jump, including unconditional and conditional branches — for example, gadgets that include `if` statements or `for` loops. This allows PEACE to make use of more complex gadgets than Angrop, which in theory makes ROP attacks generated by PEACE more difficult for detection techniques to identify.

PEACE was able to show that for 54 binaries, there was no sequence of gadgets that could lead to calling `execve` in a single-threaded context. The fact that PEACE can make this claim is worth highlighting, because it is a property that other methods do not share — when PEACE does not find an exploit, it is because one does not exist, subject to the assumptions PEACE makes.

Another interesting result is how important gadget-reduction is to PEACE. On average, the library filtration phase reduces the number of gadgets that must be considered by a factor of 2.97 (Minimum: 1.54, 25th percentile: 2.53, Median: 2.93, 75th percentile: 3.29, Maximum: 7.62). This reduces the average runtime of PEACE substantially. PEACE's runtime is $O(b^d)$; this optimization reduces b, which has an outsize effect on the runtime.

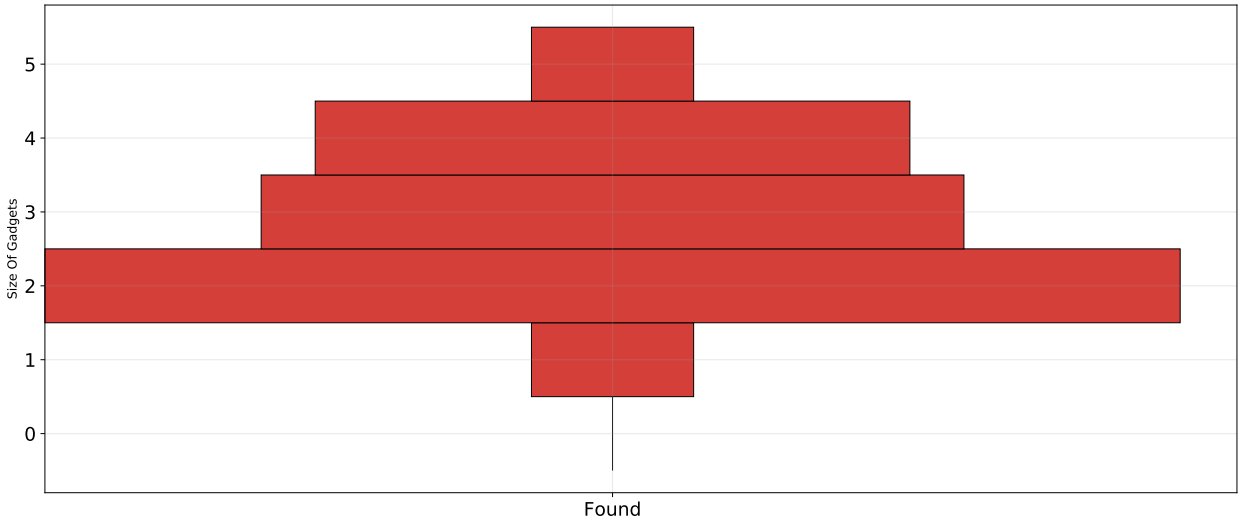

Figure 5-1: The average size of gadgets in exploits assembled by PEACE.

5.3 ROP Chain Diversity

This section discusses the diversity of ROP chains produced by PEACE. We measure the attributes of the gadget chains generated by PEACE and present the results as kebab plots. The plots show a granular representation of the underlying distribution, akin to a histogram.

Figure 5-1 shows the distribution of sizes of the gadgets used by PEACE to assemble its exploits. The average gadget was 3.3 instructions. On average, PEACE used more complex gadgets than ROPGadget, which used gadgets with an average length of 2.0 instructions. The number of gadgets in exploits PEACE generated varied according to the following distribution: Minimum: 5; 25th percentile: 6; Median: 6; 75th percentile: 10; Maximum: 22. A search of depth 22 is quite remarkable given that the average branching factor was 5.48, showing that our priority queue ordering heuristics were highly effective.

Figure 5-2 shows the size of the exploits PEACE found in instructions. Compare this varied distribution with ROPGadget, which usually constructs exploits (for x86_64) that are between 126 and 134 instructions long (Salwan, 2011). The fact that PEACE is able to generate exploits that are so short is weak evidence for the claim that gadgets are usually independent, and therefore that

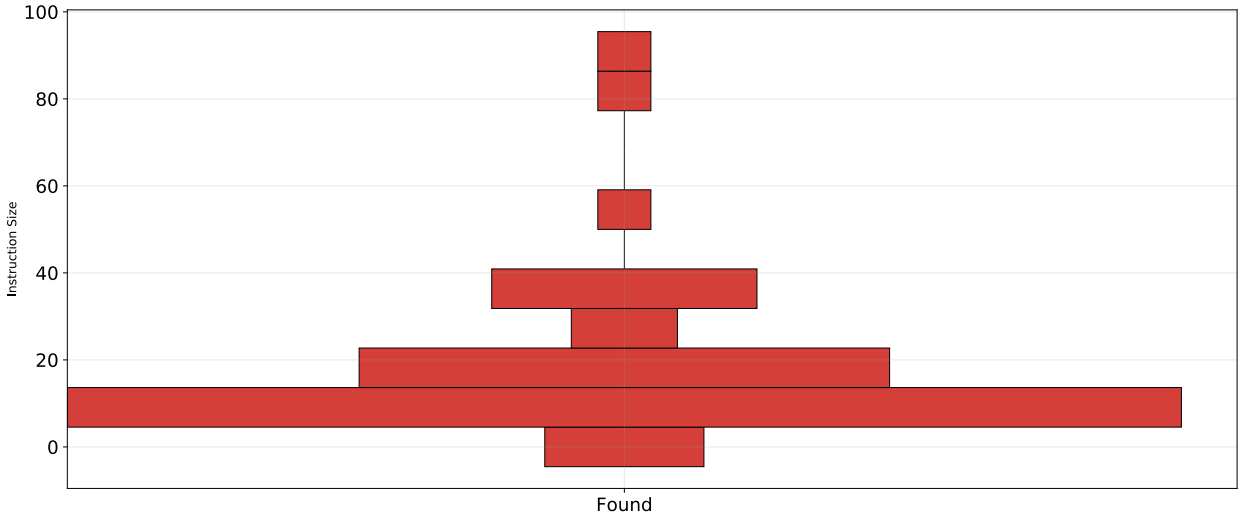

Figure 5-2: Total number of instructions in all ROP payloads found.

partial-order planning is a good fit for this domain. If gadgets interfered with each other more, we would expect to have long sequences of gadgets present to work around unhelpful side effects.

Figure 5-3 shows the average branching factor vs. the number of gadgets after library filtration. The branching factor was weakly positively correlated ($R^2 = 0.284$) with the number of gadgets present in the binary. This makes sense because when there are more gadgets available in total, there are more likely to be gadgets that effect each register.

Figure 5-4 shows the number of nodes that PEACE expanded during its search, both in cases where an exploit was ultimately found, and in those where it was not. As the distributions show, the difference between binaries was extreme, with some binaries having many plans explored before they were found not to contain an exploit. The average number of expansions overall was 296.5.

Figure 5-5 shows how many gadgets PEACE was able to extract from the binaries in the corpus. This plot shows that those binaries where PEACE was able to show that no exploit exists were usually smaller in terms of number of gadgets. This makes sense, because exhaustively searching all possible ROP exploits in a binary is more complex than trying to find a single example.

To evaluate the performance of PEACE, we allowed up to 12 hours for gadget extraction and library filtration, and a further 2 hours for planning. Most binaries in the corpus did not take

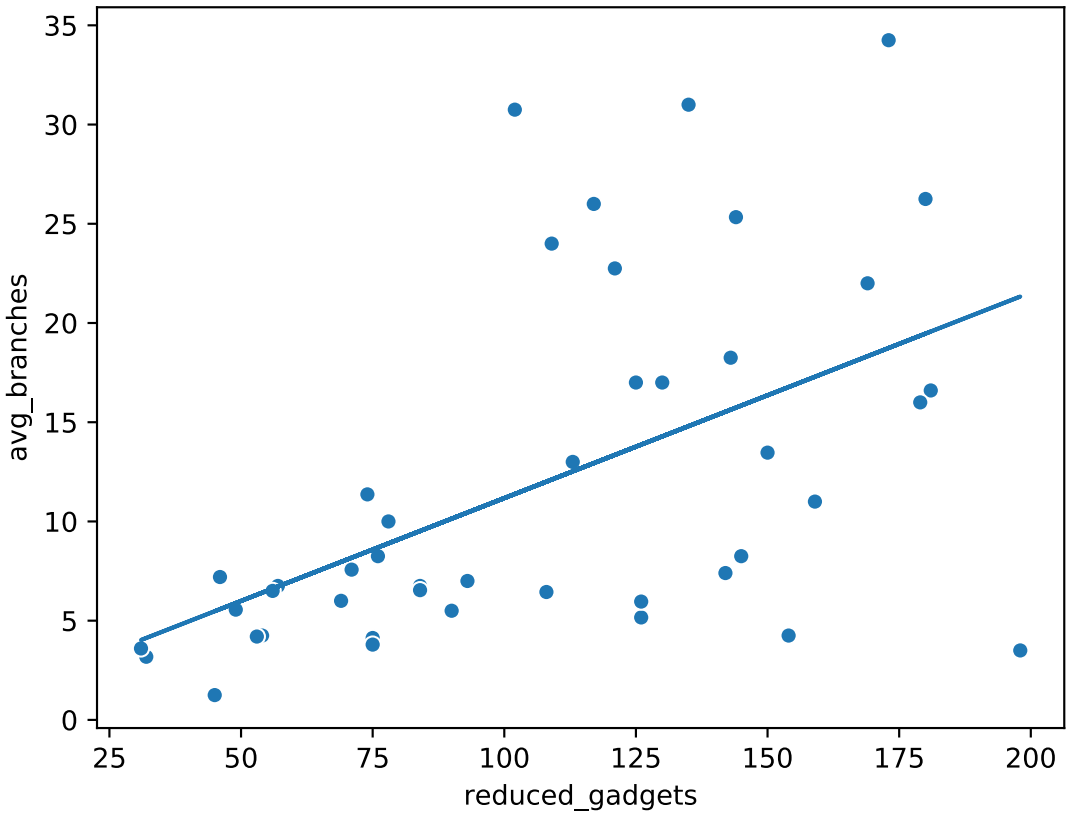

Figure 5-3: The average branching factor vs. number of gadgets found.

the full time for gadget extraction, but did take the full time for planning. Figure 5-6 shows that PEACE was finding exploits over a wide range of times, even up to the 2-hour cutoff. This suggests that given sufficient time, PEACE would be able to construct ROP payloads for more binaries. All of the previous work that we evaluated executed in less than 3 hours total on our test machine, and did not consume significant memory. The performance difference versus existing methods comes from PEACE's use of planning. This is still an acceptable trade-off because PEACE can provide more complex and resilient ROP chains on binaries that stymie other techniques. It also suggests that a hybrid method might be an interesting area of research.

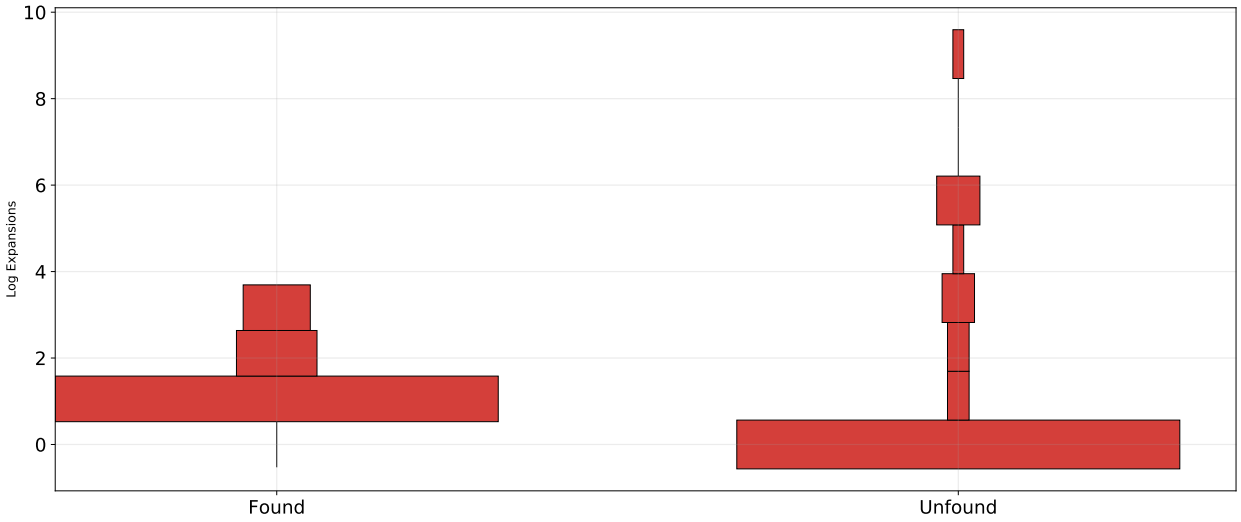

Figure 5-4: Number of expansions performed, for executions that found a plan versus those that did not, on a log scale.

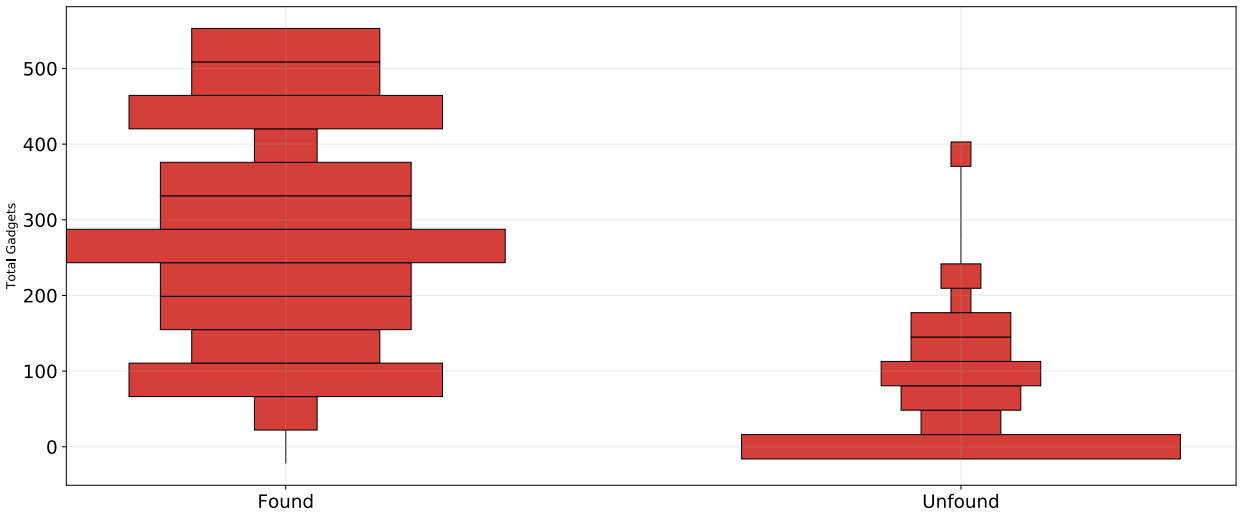

Figure 5-5: The number of gadgets present before library filtration for executions where a plan was found versus not.

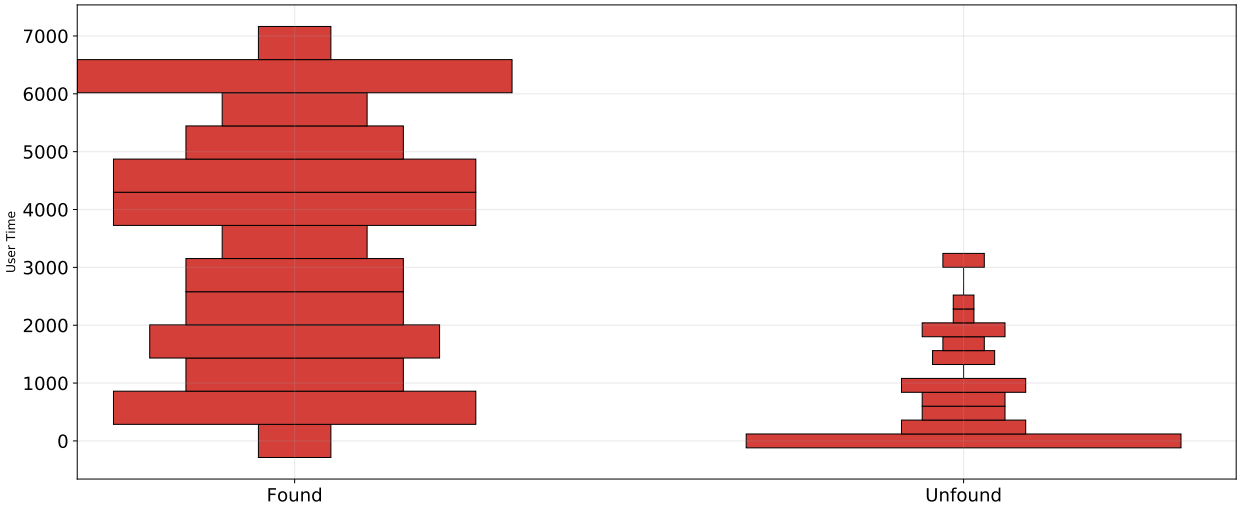

Figure 5-6: Runtime of PEACE comparing executions that found a ROP payload to those that did not.

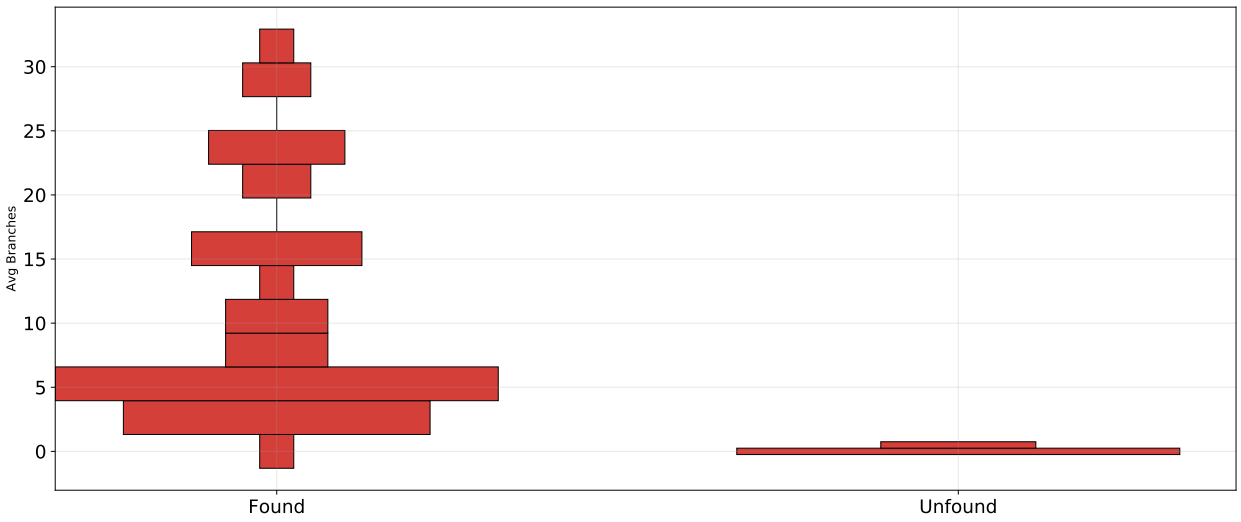

Figure 5-7: Average number of branches seen while expanding plans, for executions that found a plan versus those that did not.

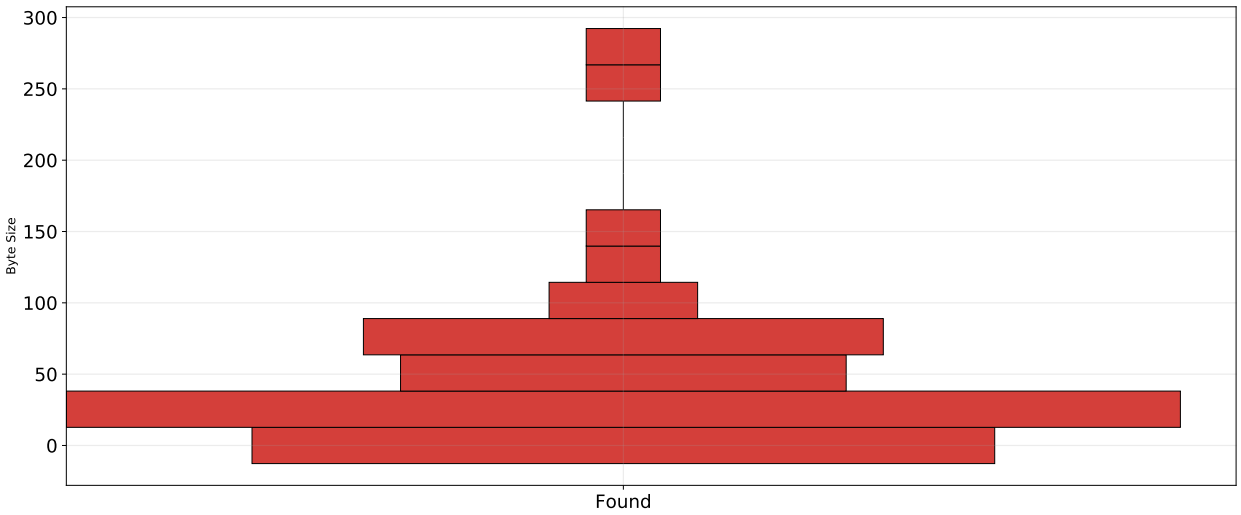

Figure 5-8: Size of ROP payloads found, in bytes.

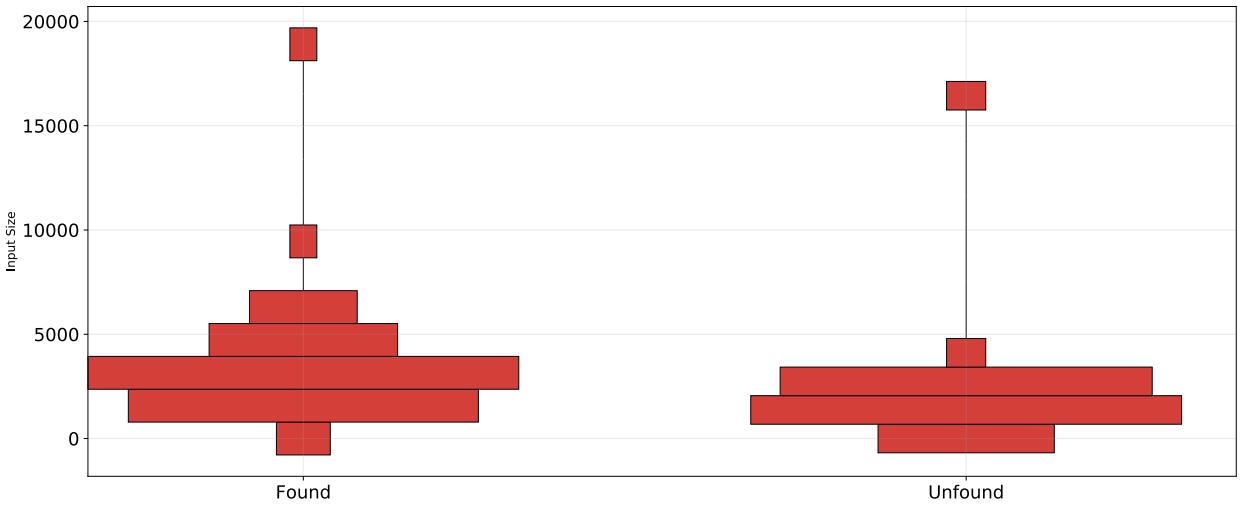

Figure 5-9: Size of files for which a ROP payload was found versus not, in bytes

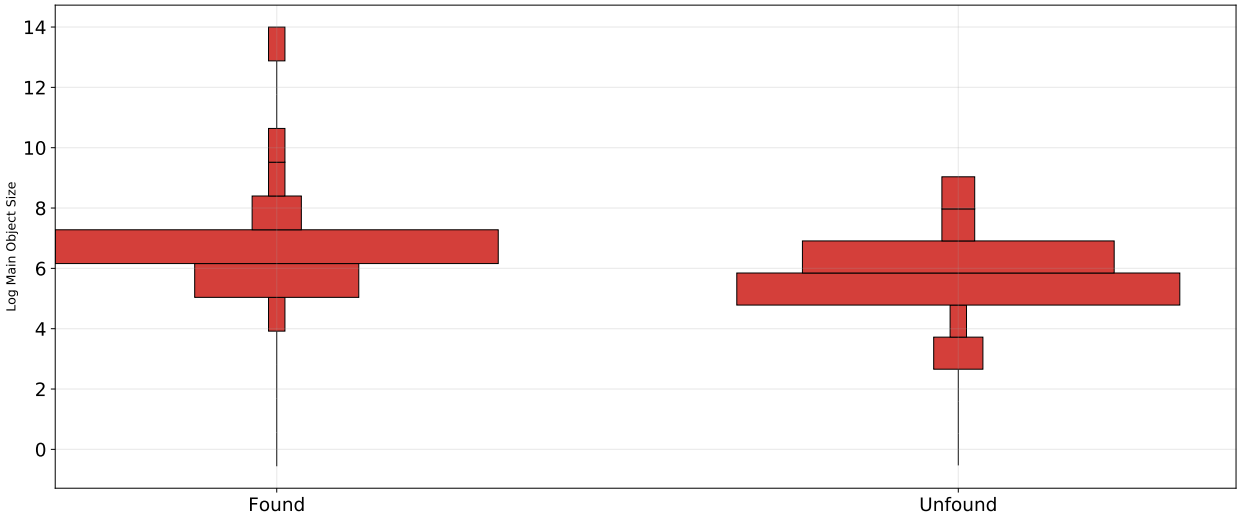

Figure 5-10: Size of the main object loaded from the file for instances where a plan was found versus not, in bytes, log scale.

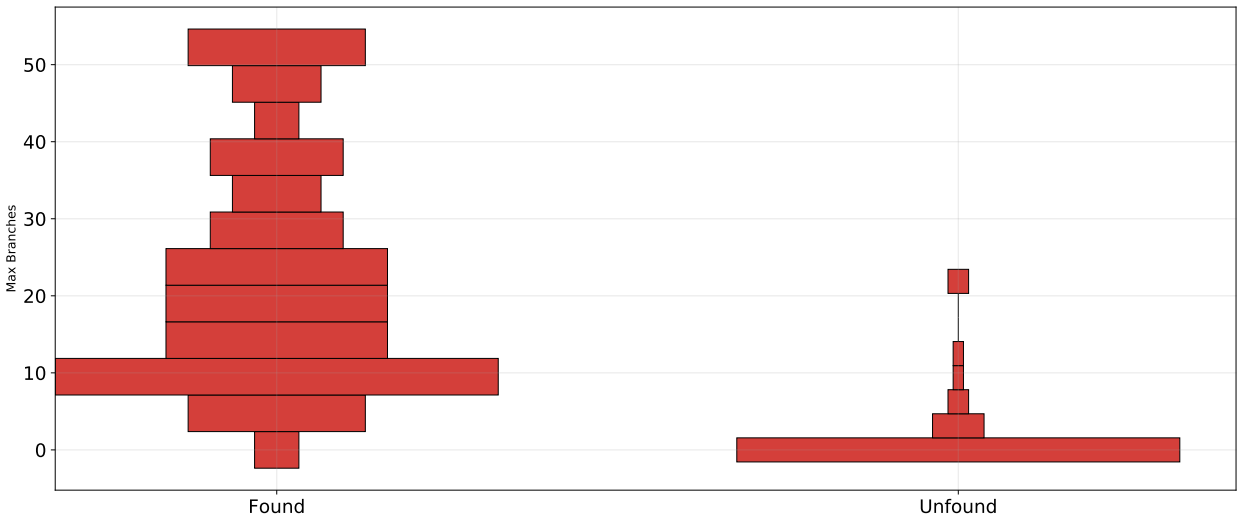

Figure 5-11: The maximum number of actions available in some state in each execution where a plan was found versus not.

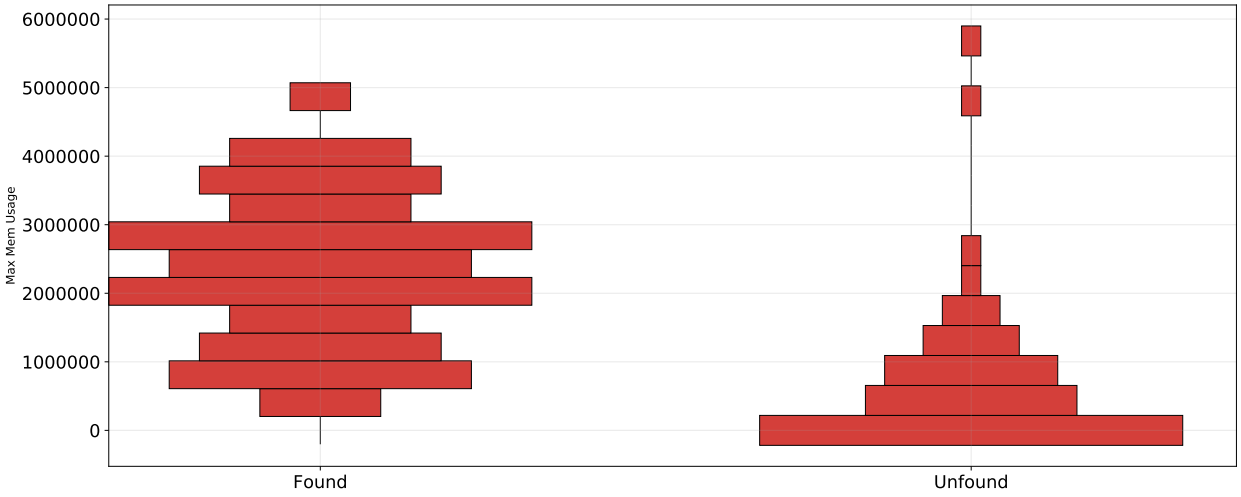

Figure 5-12: The maximum resident set size of PEACE for executions where a plan was found versus not, in bytes.

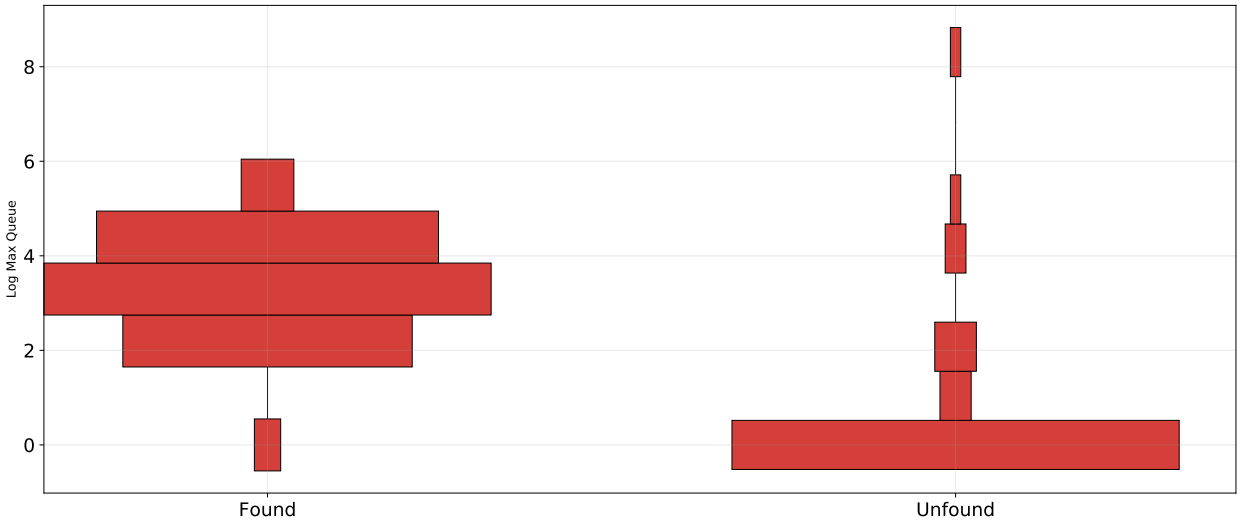

Figure 5-13: The maximum size of the queue for executions where a plan was found versus not, log scale.

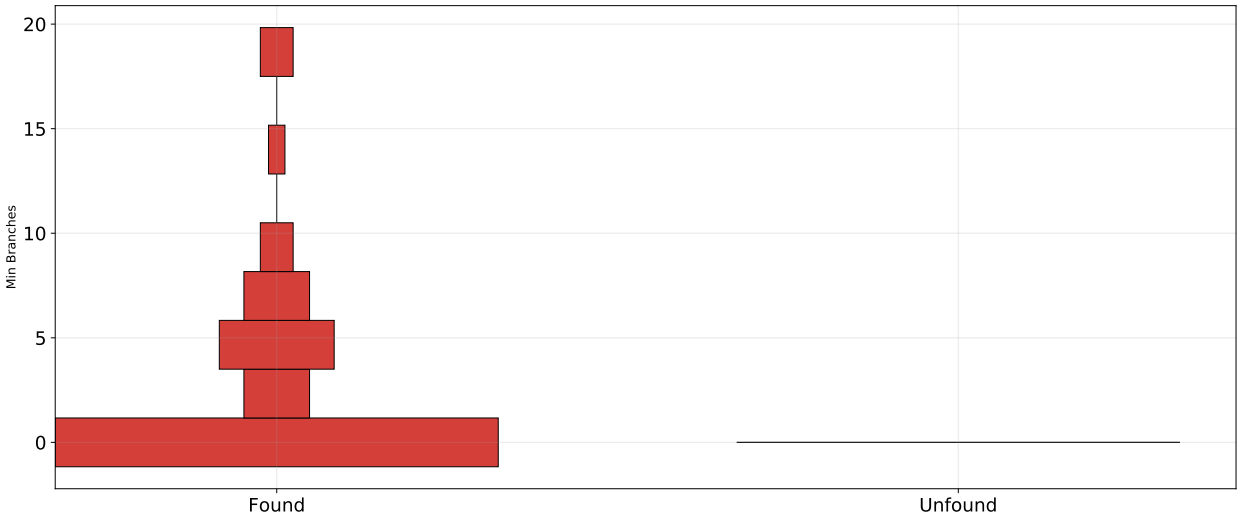

Figure 5-14: The minimum number of actions available in some state for executions where a plan was found versus not.

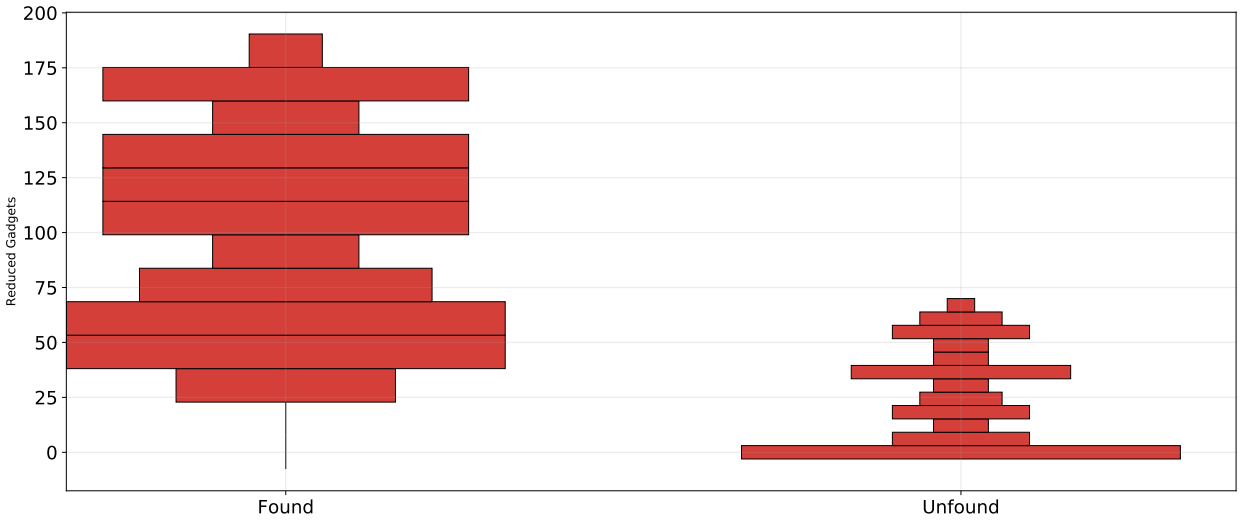

Figure 5-15: The number of gadgets present after library filtration for executions where a plan was found versus not.

CHAPTER 6

Discussion

ROP attacks have been widely studied within the security community, and there is a lot of existing work dedicated to mitigating them. However, these techniques are orthogonal to PEACE's contribution. While we hope that research like ours can spur researchers to develop as many ROP countermeasures as practical, there are plenty of theoretical issues with detecting or preventing ROP attacks, which we briefly review. Detection techniques can use common characteristics (i.e. frequency of indirect jumps, repetitive constructs, etc.) of ROP payloads to identify them before they reach the target application, or can use architectural details to detect them as they execute. Changing how code is compiled can reduce or eliminate gadgets entirely (Mortimer, 2019). ASLR (Address Space Layout Randomization) can prevent the payload from being assembled unless private information is known. We briefly discuss each of these techniques.

6.1 Detection Techniques

Several detection techniques are based on Control Flow Integrity, or CFI (Abadi et al., 2005; Mashtizadeh, Bittau, Mazieres, & Boneh, 2014; Payer, Barresi, & Gross, 2015). This technique analyses a binary to identify the potential control flow that the program could take, and then detects when a ROP attack causes it to differ. This is a difficult problem because identifying the CFI graph perfectly is in theory Turing Complete, and in practice quite difficult or inefficient. Therefore most such protections use a conservative approximation of the true CFI graph. This is usually insufficient to prevent code reuse attacks (Carlini & Wagner, 2014; Davi, Sadeghi, Lehmann, & Monrose, 2014; Checkoway et al., 2010; Göktas, Athanasopoulos, Bos, & Portokalidis, 2014a).

Practically, many such protections check that indirect jumps are *call-preceded*, meaning that they jump to an instruction directly after an instruction that could have called a function. There are

several such instructions in x86_64, for example, but the most common one is `call`. Other analogous instructions exist in every architecture. A possible extension to PEACE would be to optionally consider only call-preceded gadgets, which should allow it to circumvent these techniques.

Another common prevention first introduced in kBouncer (Pappas, 2012) is to check for many indirect jumps happening in a row using something like Intel's Last Branch Record. However, it turns out that common application patterns can have control flow that looks similar to ROP attacks, and that ROP attacks don't have to rely only on short, easily identified gadgets (Göktaş, Athanasopoulos, Polychronakis, Bos, & Portokalidis, 2014b).

Other detection techniques include *shadow stack* based approaches (Sinnadurai et al., 2008; Davi, Sadeghi, & Winandy, 2011). These track the call stack separately from the program's main stack, which makes it more difficult to overwrite return addresses. This originally had a significant runtime overhead, although modern techniques have reduced it to a few percent (Dang, Maniatis, & Wagner, 2015). This is a very promising defense against ROP attacks, although it is possible to construct ROP attacks without using return instructions, instead relying on other kinds of indirect jumps (Checkoway et al., 2010). Other kinds of indirect jumps are much rarer in practice, so shadow stack based defenses remain a strong contender, but most ROP attack construction tools — including PEACE — are capable of using other kinds of indirect jumps. Shadow stacks are also usually less strict than they could be due to non-local control constructs such as C++ exceptions or `setjump` and `longjump`.

PEACE addresses shadow stack based approaches by considering non-return indirect jumps, as in JOP, and using unaligned instruction sequences to bypass function epilogues, which can circumvent some shadow stack-based techniques (Dang et al., 2015; Sinnadurai et al., 2008).

6.2 Prevention Techniques

No existing method completely prevents ROP attacks, other than eliminating buffer overflows as memory-safe languages do. The techniques that do exist operate by making it more difficult to construct ROP attacks by reducing the number of gadgets, limiting indirect control flow, or requiring attackers to have privileged information.

For example, the OpenBSD Project was able to significantly decrease the number of gadgets in the OpenBSD kernel by changing LLVM's register selection algorithm to choose registers that result in fewer unaligned ROP gadgets (Mortimer, 2019). They also used *nop-sleds* to make unaligned access to function epilogues harder, by padding the function with one-byte `nop` instructions to prevent unaligned instruction decoding, and then using specially-designed function epilogues that don't contain many useful operations for a ROP attack. Both of these techniques substantially decrease the number of viable gadgets available to all ROP methods, including PEACE, but do not eliminate them. These changes have been included in recent versions of LLVM and GCC, so many binaries present in our corpus were built with some form of this mitigation enabled. Based on our results, these mitigations don't remove enough gadgets to prevent ROP attacks on commonly distributed binaries.

These techniques can decrease the potential for ROP attacks, but cannot provably eliminate them, because programs must always include indirect jumps to be useful. The only statically sound way of eliminating ROP attacks is to completely prevent buffer overflows and other techniques that allow malicious data to be used to influence the control flow of the program.

6.3 ASLR

Address Space Layout Randomization and its cousin Address Space Layout Permutation seek to make ROP difficult by adding randomness to the location of code while the program is running. However, numerous papers have shown that there are ways to use various side channels to leak ASLR or ASLP information (Rudd, Skowyra, Bigelow, Dedhia, Hobson, Crane, Liebchen, Larsen, Davi, Franz, Sadeghi, & Okhravi, 2017; Snow, Monrose, Davi, Dmitrienko, Liebchen, & Sadeghi, 2013; Strackx, Younan, Philippaerts, Piessens, Lachmund, & Walter, 2009; Hund, Willems, & Holz, 2013). Since side channels such as timing are exploitable even over networks that add a significant amount of jitter and noise (Brumley & Boneh, 2003), it seems likely that ASLR and its variants will always be susceptible to this kind of disclosure if running on a networked computer. PEACE therefore assumes that as well as being able to write to the stack and control the instruction pointer, we know the base offset at which the executable was loaded. Automated fuzzing techniques (Lcamtuf, 2014)

can already find ways to satisfy the first two automatically, and related work usually makes the same assumptions (Shoshitaishvili et al., 2016; Follner et al., 2016; Schirra, 2019; Salwan, 2011).

CHAPTER 7

Conclusion

ROP attacks remain a popular area of research, with many interesting offensive and defensive techniques. However, the creation of ROP exploits is still largely manual. Existing automatic ROP attack synthesis tools are far from mature. They still rely on matching fixed patterns and assembling gadgets in fixed ways. This dearth of flexible tooling not only limits the development of ROP attacks, but also restricts the evaluation of defensive techniques. In this work, we propose a novel method combining symbolic execution and partial-order planning to automatically produce diverse exploits from a binary file. PEACE is a more principled and flexible technique for assembling ROP attacks, and our evaluation shows that it substantially outperforms the previous techniques in exploiting widely distributed binaries. More broadly, our research suggests that there is exciting work to be done on ROP attacks, particularly in adapting techniques from the planning and formal methods communities to security research.

List of References

Abadi, M., Budiu, M., Erlingsson, U., & Ligatti, J. (2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, pp. 340–353, New York, NY, USA. ACM.

Bertoli, P., Pistore, M., & Traverso, P. (2010). Automated composition of web services via planning in asynchronous domains. *Artificial Intelligence*, *174*(3), 316 – 361.

Bhansali, S. (1991). Domain-based program synthesis using planning and derivational analogy. *AI Magazine*, *12*(3).

Boddy, M., Gohde, J., Haigh, T., & Harp, S. (2005). Course of action generation for cyber security using classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Bookholt, C. G. (2005). *Address space layout permutation: Increasing resistance to memory corruption attacks*. MS thesis.

Brumley, D., & Boneh, D. (2003). Remote timing attacks are practical. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pp. 1–1, Berkeley, CA, USA. USENIX Association.

Carlini, N., & Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pp. 385–399, Berkeley, CA, USA. USENIX Association.

Cesta, A., & Oddi, A. (1996). Gaining efficiency and flexibility in the simple temporal problem. In *Proceedings Third International Workshop on Temporal Representation and Reasoning (TIME '96)*, pp. 45–50.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., & Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference*

on Computer and Communications Security, CCS '10, pp. 559–572, New York, NY, USA. ACM.

Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., & Xie, L. (2009). Drop: Detecting return-oriented programming malicious code. In Prakash, A., & Sen Gupta, I. (Eds.), *Information Systems Security*, pp. 163–177, Berlin, Heidelberg. Springer Berlin Heidelberg.

Cheng, Y., Zhou, Z., Yu, M., Xuhua, D., & Deng, R. (2014). Ropecker: A generic and practical approach for defending against rop attacks. In *NDSS Symposium 2014*.

Dang, T. H., Maniatis, P., & Wagner, D. (2015). The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pp. 555–566. ACM.

Davi, L., Sadeghi, A.-R., Lehmann, D., & Monrose, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pp. 401–416, Berkeley, CA, USA. USENIX Association.

Davi, L., Sadeghi, A.-R., & Winandy, M. (2011). Ropdefender: a detection tool to defend against return-oriented programming attacks. In *AsiaCCS*.

De Moura, L., & Bjørner, N. (2008). Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer.

Dechter, R., Meiri, I., & Pearl, J. (1991). Temporal constraint networks. *Artificial intelligence*, *49*(1-3), 61–95.

Edelkamp, S., & Schrödl, S. (2011). *Heuristic Search: Theory and Applications*. Morgan Kaufmann.

Felner, A., Goldenberg, M., Sharon, G., Stern, R., Beja, T., Sturtevant, N., Schaeffer, J., & C Holte, R. (2012). Partial-expansion a* with selective node generation. *Proceedings of the National Conference on Artificial Intelligence*, *1*.

Follner, A., Bartel, A., Peng, H., Chang, Y.-C., Ispoglou, K., Payer, M., & Bodden, E. (2016). Pshape: Automatically combining gadgets for arbitrary method execution. In Barthe, G.,

Markatos, E., & Samarati, P. (Eds.), *Security and Trust Management*, pp. 212–228, Cham. Springer International Publishing.

Francillion, A., & Castelluccia, C. (2008). Code injection attacks on harvard-architecture devices. In *CSS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 15–26. ACM.

Ghallab, M., Nau, D., & Traverso, P. (2016). *Automated Planning and Acting*. Cambridge University Press.

Göktas, E., Athanasopoulos, E., Bos, H., & Portokalidis, G. (2014a). Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy*, pp. 575–589.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., & Portokalidis, G. (2014b). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, pp. 417–432.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, *SSC-4*(2), 100–107.

Hoffmann, J. (2015). Simulated penetration testing: From "dijkstra" to "turing test++". In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Hund, R., Willems, C., & Holz, T. (2013). Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security*, pp. 191–205.

Ireland, A., & Stark, J. (2006). Combining proof plans with partial order planning for imperative program synthesis. *Automated Software Engineering*, *13*, 65–105.

Lcamtuf (2014). American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`.

Mashtizadeh, A. J., Bittau, A., Mazieres, D., & Boneh, D. (2014). Cryptographically enforced control flow integrity. arXiv:1408.1451.

Mortimer, T. (2019). Removing rop gadgets from openbsd. In *AsiaBSDCon 2019*.

Norvig, R. P. (2002). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

Pakt (2013). ropc: A turing complete rop compiler. https://github.com/pakt/ropc.

Pappas, V. (2012). kbouncer: Efficient and transparent rop mitigation. http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf.

Pappas, V., Polychronakis, M., & Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pp. 447–462, Berkeley, CA, USA. USENIX Association.

Payer, M., Barresi, A., & Gross, T. R. (2015). Fine-grained control-flow integrity through binary hardening. In Almgren, M., Gulisano, V., & Maggi, F. (Eds.), *Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 144–164, Cham. Springer International Publishing.

Polychronakis, M., & Keromytis, A. D. (2011). Rop payload detection using speculative code execution. In *2011 6th International Conference on Malicious and Unwanted Software*, pp. 58–65.

Rudd, R., Skowyra, R., Bigelow, D., Dedhia, V., Hobson, T., Crane, S., Liebchen, C., Larsen, P., Davi, L., Franz, M., Sadeghi, A.-R., & Okhravi, H. (2017). Address-oblivious code reuse: On the effectiveness of leakage-resilient diversity. In *NDSS Symposium 2017*.

Salwan, J. (2011). Ropgadget. http://shell-storm.org/project/ROPgadget.

Schirra, S. (2019). Ropper. https://scoding.de/ropper.

Schwartz, E. J., Avgerinos, T., & Brumley, D. (2011). Exploit hardening made easy. In *Proceedings of the 2011 USENIX Security Symposium*.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pp. 552–561, New York, NY, USA. ACM.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., & Vigna, G. (2016). SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*.

Simplex, O. (2020). Roper. https://github.com/oblivia-simplex/roper.

Sinnadurai, S., Zhao, Q., & Fai Wong, W. (2008). Transparent runtime shadow stack: Protection against malicious return address modifications. https://zatoichi-engineer.github.io/assets/docs/10.1.1.120.5702.pdf.

Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., & Sadeghi, A. (2013). Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*, pp. 574–588.

Souchet, A. (2017). rp++. https://github.com/0vercl0k/rp/.

Srivastava, S., Immerman, N., & Zilberstein, S. (2011). A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, *175*(2), 615 – 647.

Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., & Walter, T. (2009). Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pp. 1–8, New York, NY, USA. ACM.

Team, P. (2003). Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

The CVC development team (2020). About CVC4. https://cvc4.github.io/.

Turing, A. (1936). On computable numbers, with an application to the entscheidungsproblem..

VII, T. M. (2016). Abc: A c compiler for printable x86. In *Sigbovik*.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, *15*(4), 27–61.

Yoshizumi, T., Miura, T., & Ishida, T. (2000). A* with partial expansion for large branching factor problems. In *AAAI/IAAI*.