USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Synthesis of Code-Reuse Attacks from p-code Programs

Mark DenHoed and Tom Melham, *University of Oxford*

https://www.usenix.org/conference/usenixsecurity25/presentation/denhoed

This paper is included in the Proceedings of the
34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Proceedings of the
34th USENIX Security Symposium is sponsored by USENIX.

Synthesis of Code-Reuse Attacks from p-code Programs

Mark DenHoed
University of Oxford

Tom Melham
University of Oxford

Abstract

We present a new method for automatically synthesizing code-reuse attacks—for example, using Return Oriented Programming—based on mechanized formal logic. Our method reasons about machine code via abstraction to the p-code intermediate language of Ghidra, a well-established software reverse-engineering framework. This allows it to be applied to binaries of essentially any architecture, and provides certain technical advantages. We define a formal model of a fragment of p-code in propositional logic, enabling analysis by automated reasoning algorithms. We then synthesize code-reuse attacks by identifying selections of gadgets that can emulate a given p-code reference program. This enables our method to scale well, in both reference program and gadget library size, and facilitates integration with external tools. Our method matches or exceeds the success rate of state-of-the-art ROP chain synthesis methods while providing improved runtime performance.

1 Introduction

Return Oriented Programming (ROP) is a method of leveraging a memory corruption vulnerability to perform arbitrary computations, even in the presence of executable space protection (commonly and generically known as 'W $\oplus$ X'). Paired with an information disclosure vulnerability, ROP can be achieved on systems featuring W $\oplus$ X and Address-Space Layout Randomization (ASLR). It does this by chaining together short function suffixes already existing in memory (referred to as *gadgets*) to execute a *chain*: an attacker-specified computation. Since its systematization by Shacham [1], ROP has received substantial industrial and academic attention and become the prevailing method to leverage memory corruption.

Defensive efforts have produced several proposals for software-level protections, such as kBouncer [2] and ROPecker [3], as well as micro-architectural protections, such as Intel CET [4]. Offensive efforts have produced the family of *code-reuse attacks*, exploitation methods with equivalent power to ROP but without reliance on the ret primitive [5] [6], or even on unintended code sequences [7]. For the remainder of this paper, we will use the term ROP to refer to this wider class of exploitation technique unless otherwise specified.

The automation of code-reuse attacks has also been a fertile field of research. The most notable automation efforts are Q [8], PSHAPE [9], and SGC [10]. The approaches described in these three papers differ in their particulars, but they share a common theme: using automated reasoning in symbolic logic and decision procedures to automatically synthesize code-reuse attacks.

In this paper, we present a new method for automatically synthesizing code-reuse attacks, also based in part on mechanized formal logic. Our method is built on the p-code intermediate language of Ghidra, a well-established software reverse-engineering framework from the National Security Agency Research Directorate [11]. Ghidra's code translator, SLEIGH, produces p-code to express the semantics of assembly instructions. Ghidra distributes SLEIGH definitions for over 30 processor architectures. Our synthesis method, and its reference implementation, can therefore be applied to binaries from any architecture that SLEIGH supports.

Our method synthesizes code-reuse attacks by generating and evaluating assignments of gadgets to steps of a *reference program*, itself composed of p-code operations. This approach yields many desirable properties. We do not require that gadgets cleanly fit into any predefined functional category. In addition to constraints automatically generated by our reference program, we allow for asserting arbitrary pre- and post-constraints on the system state, as well as state transition invariants. We allow for gadgets to perform unspecified (but still constrained) side-effects. Gadgets need not be in one-to-one correspondence with reference program operations: one gadget may perform many roles. Synthesis from a reference program scales more gently with reference program length and gadget library size than a purely pre- and post-constraint system would; while this will not allow as many solutions, we do not think this is a problem in practice.

Since the reference program is a block of p-code opera-

tions, it may be constructed manually or produced from native code using SLEIGH. This opens the door to powerful integrations with other tools built around native code.

We present the following contributions:

- A formal model in propositional logic for a significant segment of Ghidra's p-code intermediate language. We define our model in terms of the standard FixedSizeBitVectors and ArraysEx theories of SMT [12, 13] allowing p-code programs to be analysed by off-the-shelf automated reasoning tools.

- A novel architecture-agnostic algorithm for synthesis of code-reuse attacks from a p-code reference program. Our algorithm decomposes the reference program into sub-goals and performs a SAT-driven search for combinations of gadgets with compatible p-code semantics. Dividing the problem this way makes it feasible to reason about larger sets of gadgets and more complicated reference programs than would otherwise be feasible.

2 Background

Ghidra [11] is a well established and widely used reverse engineering tool produced by the United States National Security Agency. Ghidra is primarily a static binary analysis tool that ingests binaries and produces an interactive presentation of their disassembly and decompilation.

Ghidra's decompilation procedures are implemented using an intermediate language called 'p-code' for a simple abstract virtual machine. Binary programs are translated by the tool from their source architecture into p-code according to a specification written in SLEIGH, an NSA-developed language for capturing the binary encoding, the syntax, and the semantics of machine code across a wide range of architectures. It is an extension of SLED [14] and employs certain concepts from the Semantic Syntax Language [15, 16].

Instructions in the p-code language, called 'operations', are very elementary. A machine instruction in the source architecture typically translates to a block of several, more fine-grained p-code operations. And complicated machine instructions may translate to a block of p-code that employs internal control-flow to implement their semantics.

The p-code language is an excellent abstraction for synthesis for the same reasons it is an good basis for a decompiler: it fully models ISA-level instructions, without the extra complication of microarchitectural details; and all p-code operations have an explicit representation of the parts of system state they depend on and mutate (with some minor exceptions). This uniformity allows us to find ROP gadgets and synthesize ROP chains without sorting gadgets into pre-defined roles.

2.1 The p-code Virtual Machine

In this section, we present a brief overview of the p-code virtual machine and how it models real processor architectures. The following description of p-code is based largely on the SLEIGH documentation [17].

2.1.1 p-code Address Spaces

The p-code virtual machine represents system state as a collection of *address spaces*. An address space is similar to a processor's RAM: it holds mutable data comprising a certain number of bit-vector '*words*', which are individually accessed through an *address*. Each p-code address space (or just 'space') has an associated word size, the number of bits held at each address, and an address size, the number of bits needed to encode every address uniquely.

The number and dimensions of the p-code spaces in the virtual machine is not fixed. These parameters are furnished by SLEIGH, based on the machine architecture it is lifting. There are, however, two special-purpose spaces that are always defined, regardless of architecture:

const A space with a special interpretation for holding constant values.

unique A space used for storing intermediate results during the execution of the block of p-code that models a single machine instruction. These intermediate results are temporary, and the unique space is effectively cleared between machine instructions.

While the p-code spaces in the virtual machine are not fixed, almost all SLEIGH architectures create these two spaces:

register A space representing the registers of the source architecture.

ram A space representing a processor's main memory, generally identified with the virtual address space of a loaded executable. The size of a 'word' in the ram space will not necessarily be 32 bits, but will depend on the physically addressable units in the memory being modelled.

SLEIGH architectures are free to create further p-code spaces, which provides the flexibility to model less-common processor paradigms, such as Harvard architecture machines with separate code and data storage.

2.1.2 Varnodes

A *varnode* is a 3-tuple $(name, offset, size)$ and represents a contiguous sequence of words within the address space identified by *name*. A varnode represents only the location of a block of data in system state; it does not impose an interpretation on the data or include any type information. All

access to and mutation of system state by p-code instructions is expressed in terms of varnodes. This uniformity means p-code operations need not distinguish between, for example, reading a register and reading RAM. Both are varnodes. This property proves very convenient when modeling p-code with propositional logic and standard theories of SMT.

2.1.3 p-code Operations

A p-code *operation* is an atomic instruction of the p-code virtual machine. Each such operation may read blocks of system state designated by one or more *input* varnodes and optionally updates a block of system state designated by an *output* varnode. No p-code operation may use a varnode in the const space as an output.

Most p-code operations are required to conform to certain constraints on the number and sizes of their inputs to be considered well-formed. For example, any p-code COPY instruction must have exactly one input and one output varnode of the same dimensions. There is no explicit type system that enforces these restrictions and runtime size mismatches are considered a bug in SLEIGH.

Some p-code operations apply specific interpretations to the varnodes they reference. There are four such interpretations: *unsigned integer*, *signed integer*, *floating point*, and *boolean*. These interpretations manifest only in the semantics of the execution of a p-code operation; they are not explict in the representation of state. In this work, we do not consider p-code operations that make floating point interpretations. We do, however, account for the p-code virtual machine's configured endianness when modeling p-code operations that interpret varnodes spanning multiple addresses as integers.

Almost all p-code operations depend on only their input varnodes and mutate system state only at their output varnode, ensuring that all effects are explicit in the code. This makes a particularly uniform logical encoding of p-code possible and also provides a mechanism for our code-reuse synthesis algorithm to greatly narrow the search space. There are a few operations that break this pattern:

LOAD/STORE These operations are indirect memory reads and writes, respectively. The indirect access is represented by an input varnode representing a pointer location, a special const varnode input that is interpreted as a numerical identifier for the space in which the read or write should occur, and a size. To aid in later discussion, we define the 3-tuple of these quantities to be an *indirect varnode*.

CALLOTHER This instruction is used to represent any type of unmodeled functionality, including system calls. The p-code documentation states that CALLOTHER may do undefined mutation on parts of the state outside the region defined by the output varnode.

Finally, control-flow is expressed as follows. Every p-code operation is uniquely addressed by a tuple (*address*, *offset*), where *address* is the address of the machine instruction that was translated into the block of p-code containing the operation, and *offset* is the operation's index within that block of p-code. During execution, a non-control-flow p-code operation falls through to the p-code operation at the next *offset*, if one exists. Otherwise, it falls through to first p-code operation in the block for the next machine instruction.

The control-flow operations of p-code are: BRANCH, CBRANCH, BRANCHIND, CALL, CALLIND, RET and CALLOTHER. Of these, BRANCH and CALL are direct branches. CBRANCH represents a direct branch to a location that is taken only if a condition (given as an argument) evaluates to true. BRANCHIND, CALLIND and RET are indirect branch to machine addresses within the same p-code space. CALLOTHER, also referred to as USERDEFINED in the p-code documentation, represents system transitions too complex to model in p-code (e.g. computing a Fast Fourier Transform) as well as unmodeled functionality such as atomic memory barriers and system calls.

p-code branches, like any other operation, use varnodes as input. A branch target in the const space represents a relative jump of the p-code *offset* within the block of p-code for the machine instruction it represents. This is used for modeling both straightforward conditional control-flow and complicated assembly semantics, such as the x86 rep prefix.

Our current work is limited to assembly instructions that generate a single p-code basic block: a sequence of non-control-flow p-code operations, optionally ending with a single control-flow p-code operation.

3 A Formal Model of p-code

Our algorithm for code-reuse attack synthesis is based on a formal model of a fragment of p-code into standard formal theories in first-order logic that can be handled by the class of automated reasoning methods called Satisfiability Modulo Theories (SMT) [12, 13]. The model is constructed to reflect the semantics laid out in the p-code reference documentation [17]. The approach is inspired by the construction of software bounded model checkers, such as CMBC [18], which build a model as a transition relation following the 'semantics laid out in the C language standards' [19].

Our formal model of p-code enables automated reasoning about the semantic suitability of a given candidate code-reuse chain, with respect to the supplied reference program. Our synthesis procedure, explained in Section 4, separately employs a solver for Boolean Satisfiability (SAT) to generate and refine candidate chains.

In this section, we briefly explain our model of p-code. This is relatively simple, because it needs to cover only the specific p-code operations and restricted forms of p-code block for which a formal representation amenable to automated reasoning is needed by our synthesis algorithm. Each

non-control-flow p-code operation is represented formally by the logical relation that establishes the system state before and after its execution. The p-code control-flow operations needed for synthesis are represented formally by an expression that captures their (possibly symbolic) branch target addresses.

3.1 State

As explained, the state of the p-code virtual machine is given by a collection of address spaces. Viewed mathematically, an address space is essentially a mapping from addresses to words. The standard theories for SMT [13] include a theory of functional arrays with extensionality: ArraysEx. This gives a direct formalization of such a mapping, and therefore is the basis for our formal model of p-code state. Extensionality just means that the formalization includes equality on arrays, defined in the obvious component-wise way.

ArraysEx is an abstract theory, parameterized by the types of the index into the array and the elements it contains. For modelling p-code, these will both be fixed size bit-vectors, which are formalised for SMT in the standard theory of FixedSizeBitVectors. This theory is central to many automated reasoning applications for computer systems [20], such as formal verification of hardware designs, and so well supported by many off-the-shelf SMT solvers.

3.1.1 Reading and Writing State

Operations in p-code interact with state through their input and output varnodes. Reading and writing an address space at the block of words indicated by a varnode is straightforwardly modeled using the select and store functions of the ArraysEx theory. Given an array a and index i, the expression (select a i) denotes the element of a located at index i. In our model, this will be a 'word' of the address space modeled by a. And for a given word w, (store a i w) denotes the array that is identical to a except that it contains w at index i. In our model, this denotes an updated address space.

The word sizes of p-code address spaces can vary across machine architectures being modeled. For uniformity, our implementation concatenates the results of all multi-word address space reads into single bit-vectors. Likewise, the formal representation of the bit-vector result that would be produced by executing a p-code operation is broken into consecutive words and expressed as a address space update by cumulative applications of the store function.

This paradigm is a well-established approach to engineering systems for automated reasoning about low-level code [10, 21]. In our case, modeling is not restricted to blocks of words with a power of two size, though in practice most are. We also take the architecture's endianness into account in the conversions to and from bit-vectors.

The above describes how all varnode reads and writes are modeled—with the exception of const, explained below. A processor's RAM and its registers are both modeled as address spaces, so our encoding does not need to embody assumptions about the arrangement, roles, or names of architectural registers. It also seamlessly handles sub-registers (e.g. rax vs eax vs ah), correctly constraining overlapping registers.

Modeling reads from the const address space requires interpreting const varnodes as immediate values. We assume an unsigned integer interpretation on all reads from the const space. We further assume that if SLEIGH needs to represent a negative constant, it will do so by emitting a two's-complement encoding of that constant. We represent a p-code read of the varnode $(\texttt{const}, n, k)$ as a constant bit-vector of value n and width $k*8$ bits. Writes into the const space are explicitly forbidden in the SLEIGH documentation, and our implementation exits if it encounters one.

3.2 Modeling p-code Blocks

Executing a p-code operation produces a transition of the p-code virtual machine from one state—comprising all the declared address spaces—to the next. In our model, we represent a p-code operation as a logical relation between a collection of functional arrays, representing the address spaces of the initial state, and corresponding but separate functional arrays representing the address spaces of the final state.

To formalize the effect of the operation, we build symbolic expressions denoting the state updates that it will make. These will be expressed as functions of the symbolic representations of the bit-vectors obtained through its input varnodes, which refer to the initial state arrays. The expressions are constructed by our implementation using the rich collection of operations over bit-vectors formalized in the standard SMT FixedSizeBitVectors theory. We then build expressions that denote the final state arrays through suitable applications of the store operation for each block of state addressed by the output varnode. The logical relation that holds between an arbitrary initial state and its corresponding final state models the p-code operation to capture its semantics.

As an example, consider the x86 instruction MOV EBX,EDI. This instruction is represented in p-code with a single operation: register[0xc]:4 = COPY register[0x1c]:4. Figure 1 shows the relation describing the modified register space after the execution of this instruction.

We can iterate through a block of p-code operations and model their cumulative semantics by applying each operation's relation on the final state of the previous one. This allows us to model a p-code basic block; as noted in Section 2.1.3, we create a formal model only for sequences of non-control-flow p-code operations containing no outgoing branches, optionally terminated by a final branch.

```
(store
 (store
  (store
   (store register!0
    #x0000000c (select register!0 #
        x0000001c))
    #x0000000d (select register!0 #
        x0000001d))
    #x0000000e (select register!0 #
        x0000001e))
    #x0000000f (select register!0 #
        x0000001f))
```

Figure 1: An SMT encoding of an `x86` register move.

3.3 Modeling Control-Flow

We now explain how we account for the control-flow behavior of a basic block of `p-code` in our model. A basic block is a sequence of non-control-flow `p-code` operations ending with a single `p-code` control flow operation. These are the only kind of block that we will encounter in our synthesis algorithm. We do not consider the `CBRANCH` conditional branch, and provide limited support for `CALLOTHER`. We further assume that code will only branch within a given space (e.g. all code is in the `ram` space).

Given a `p-code` block of the kind just defined, we model its control-flow behavior as a bit-vector expression that represents the address of the target machine instruction.

If a block ends with an unconditional `p-code` branch, then the target address produced is simply a constant bit-vector giving the offset of the branch destination. If the `p-code` block contains an indirect `p-code` branch, then the target address is constructed by reading the destination varnode from the final state of the instruction preceding the branch, using the formal representation described in the previous section.

We also provide limited modeling of certain uses of the `CALLOTHER` operation. The `p-code CALLOTHER` operation is underspecified and does not lend itself well to a general formalization. But we must consider at least a limited case, because many code-reuse attacks have the explicit goal of executing a system call, which `SLEIGH` represents through `CALLOTHER` operations.

Our model treats the `CALLOTHER` operations we need to handle as direct branches. In general, `p-code` uses `CALLOTHER` for many things, distinguishing between different cases by the form of its input varnodes. To represent the case we need in our model, we perform a hash of its varnode arguments to produce a constant bit-vector address, which we take to represent the `CALLOTHER` 'branch'.

Since we are modeling `CALLOTHER` branches as producing bit-vector address representations, we must provide a way to distinguish between the addresses produced by `CALLOTHER` and the addresses produced by the other `p-code` control-flow operations we model. We do this by adding a one-bit tag for each `p-code` block's branch destination address. When a block ends with `CALLOTHER`, this tag is set to 1 and is set to 0 otherwise. We then define equality for `p-code` block branch destinations; their bit-vector target addresses must be equal, and their tags must be equal.

4 Synthesis

We present `crackers`, our algorithm for synthesizing code-reuse attacks. We first provide a general overview of the algorithm, which is shown as Algorithm 1. We then provide details of each of the main steps.

`Crackers` synthesizes ROP chains from a library of gadgets (Section 4.1), such that the chains refine a linear `p-code` *reference program* (Section 4.2) that is provided as an input. The reference program is split into a sequence of *steps*, each representing the effects of one or more `p-code` operations (Section 4.2.1). A chain synthesized from a sequence of N steps will have exactly N gadgets. `Crackers` iterates over all possible permutations of steps, ordered by ascending length, seeking the shortest possible chain for a program.

`Crackers` attempts synthesis of a given sequence of steps by selecting a set of *candidate* gadgets for each step (Sec-

Input: A target binary B, a `p-code` reference program P, a table of chain constraints E
Output: A satisfying chain if satisfiable, UNSAT if not

```
1  L := MakeLibrary(B);
2  foreach steps in SequencePartitions(P) do
3      candidates := MakeCandidates(steps, L);
4      if ¬candidates then
5          continue;
6      end
7      solver := SatSolver();
8      InitSolver(solver, candidates);
9      while chain := SelectGadgets(solver);
10     do
11         result := Check(chain, steps, E);
12         if SAT(result) then
13             return chain;
14         else
15             core := GetUnsatCore(result);
16             clause := GetConflictClause(core);
17             AddClause(solver, clause);
18         end
19     end
20 end
21 return UNSAT;
```

Algorithm 1: `crackers`

tion 4.3). Each step's candidates are then associated with a unique Boolean proposition, and a SAT solver generates a selection of candidate gadgets using these (Section 4.4). The `p-code` of the selected gadgets is used to build a logical model of a ROP chain (Section 4.5) suitable for analysis with an SMT solver. This model is constrained to ensure each gadget fulfils the post-conditions of its step, as well as certain other requirements. An SMT solver is used validate the model. If the solver returns SAT, `crackers` succeeds. If the solver returns UNSAT, the UNSAT core of the chain model is used to further constrain subsequent gadget selection (Section 4.6). If a candidate cannot be selected for any gadget, the algorithm moves to the next partition of reference program steps. If all step partitions fail, the algorithm fails.

4.1 Generating a Gadget Library

The first step of our algorithm is to build a library of gadgets from the provided binary code, as shown in Procedure 2. Our approach to collecting gadgets is simple: for every address within an executable segment of a binary, we attempt disassembly with `SLEIGH`, up to a set number of instructions, yielding a block of `p-code`. We perform syntactic validation of each potential gadget's `p-code` before we accept it. The gadget must contain only `p-code` operations that are covered by our model and must terminate in an indirect `p-code` branch. This filters out gadgets with floating-point operations, conditional branches, and other forms of internal control-flow. Gadgets passing these requirements are accepted, as `p-code` blocks, into the library and associated with the address of their first machine instruction.

Our approach differs from that of Shacham [1] and Schloegel et al. [10] in that it works forward from candidate addresses rather than backward from `ret` and `jmp` instructions. This frees us from architecture-specific assumptions that specific patterns of bytes represent indirect branches.

Input: A target binary B, a max gadget length N, a set of allowable `p-code` operations P
Output: A library of gadgets

```
1  library := [];
2  foreach segment in B do
3      if IsExecutable(segment) then
4          foreach address in segment do
5              gadget := Disassemble(address, N, P);
6              if ¬ gadget then
7                  continue;
8              end
9              if SyntacticCheck(gadget) then
10                 library += gadget;
11             end
12         end
13     end
14 end
15 return library;
```

Procedure 2: MakeLibrary

4.2 Reference Program

Our algorithm attempts to synthesize a ROP chain from a `p-code` reference program—a blueprint for the sequence of effects we would like our ROP chain to achieve. This reference `p-code` may be manually written or expressed first as assembly code, perhaps even automatically produced by an assembly synthesis tool, and then assembled and translated to `p-code` by `SLEIGH`.

Like our gadgets, the reference program is required to be a single basic block of `p-code`: a sequence of non-control-flow `p-code` operations, generally ending in a branch or syscall.

Figure 2 shows a block of assembly code that might be used to produce a `p-code` reference program: a simple invocation of `execve` on `x64` linux. In this example, `rax`, `rsi`, and `rdx` are set to the concrete values necessary to invoke `execve`. This reference program requires that `rdi` is written, but does not require it to contain any particular value (as `rbx` has no constraints on its value). To properly execute the syscall, this value must be a pointer to readable memory containing a null-terminated ASCII representation of the path of an executable on the system (e.g. '/bin/sh').

Figure 2: A x64-linux code-reuse-attack reference program

```
mov     rax, 0x3b
mov     rsi, 0x0
mov     rdx, 0x0
mov     rdi, rbx
syscall
```

4.2.1 Reference Program Steps

`Crackers` attempts to synthesize the shortest possible chain for a given reference program by combining the operations of the reference program into a shorter sequence of compound steps. For example, a program with operations $[a,b,c]$ will generate the following step partitions: $[[a,b,c]]$, $[[a],[b,c]]$, $[[a,b],[c]]$, $[[a],[b],[c]]$. By ordering these partitions by their number of members, `crackers` ensures it attempts to synthesize shorter ROP chains first.

4.3 Gadget Candidate Identification

Our algorithm identifies a pool of candidate gadgets for each step of the reference program. Procedure 3 describes how candidate gadgets are identified and selected. This procedure collects candidates until it exhausts the gadget library or reaches a configured number of candidates for each step.

The procedure iteratively selects a random gadget from the library, without replacement, and evaluates its suitability for each step of the reference program. If a gadget is found to be suitable for a step, it is added to a pool of candidate gadgets for that step. A single gadget can be accepted as a candidate for more than one step, allowing the same gadget to appear multiple times in a chain.

Suitability is determined syntactically by analysis of the *output signatures* of the `p-code` blocks that form the step and the gadget being considered. The output signature of a `p-code` block is the set of output varnodes and indirect varnodes written by a `p-code` block. A gadget is suitable for a step if it mutates at least all the outputs of the step.

Informally, we say that an output signature A *covers* an output signature B if, for every varnode b in B, there exists a varnode a in A that contains all the addresses of b, and if, for every indirect varnode b_i in B there exists an indirect varnode a_i in A with equal or greater size to b_i and the same pointer varnode as b_i.

To be accepted as a candidate for a given step, a gadget must satisfy the following:

- The gadget's output signature must cover the output signature of the step.
- If the step ends with a `CALLOTHER` `p-code` operation, the gadget must end with an *identical* `CALLOTHER` operation.

To avoid unnecessary work for our algorithm, we further filter this set to remove candidates with obvious semantic incompatibilities. For example, the gadget `MOV EAX, #1; RET` is clearly incompatible with the step `MOV EAX, #0`. We accomplish this by building a logical expression relating the formal model of the candidate gadget to the step. (This same expression is used in Section 4.5.1 to evaluate a chain.)We use Z3 to perform a *syntactic* simplification of this expression. If the expression can be syntactically simplified to `false`, we discard the candidate.

This syntactic filtering, while coarse, provides vital guarantees about the gadgets accepted as step candidates. The fact that all candidates write to all outputs of their step prevents our chain validation from conjuring unrealistic favorable memory models. This was only possible to ensure thanks to the explicit, uniform nature of `p-code` operation side effects.

If Procedure 3 is unable to identify any candidates for at least one step, it fails and `crackers` continues to the next combination of reference program steps.

Input: a sequence of reference program steps S, a gadget library L
Output: An enumerated set of candidate gadgets for each step of the reference program, NULL if any step has no candidates

```
1  candidates := [];
2  for gadget in RandomOrder(L) do
3  |  foreach step in S do
4  |  |  if SuitableForStep(gadget, step) then
5  |  |  |  candidates += (gadget, step);
6  |  |  end
7  |  end
8  |  if SufficientCandidates(candidates) then
9  |  |  return candidates;
10 |  end
11 end
12 foreach step in S do
13 |  if StepHasNoCandidates(candidates, step)
   |    then
14 |  |  return NULL;
15 |  end
16 end
17 return candidates;
```

Procedure 3: MakeCandidates

4.4 Chain Assignment Generation

A *gadget assignment* (i, j) is the selection of the j^{th} candidate gadget for reference program step i. A *chain assignment* is a sequence of such assignments, one for each step.

We use an off-the-shelf SAT solver (Z3 [22]) to generate chain assignments by encoding possible gadget assignments as boolean propositions and making a series of assertions encoding the 'rules' of gadget selection. As our algorithm iterates over chain assignments, it derives additional constraints from the validation failures of chain models, progressively refining subsequent assignments (Section 4.5).

To encode the gadget assignment problem in propositional logic, we define for each possible gadget assignment (i, j) a unique boolean proposition a_{ij}. We can then encode the set of all possible gadgets assignments from among the n options for step i as follows:

$$g_i = \bigvee_{j<N} a_{ij}$$

Using this encoding, we can assert that *at least* one gadget must be selected for each reference program step with the conjunction $\bigwedge_i g_i$. We can further assert that *at most* one candidate is to be selected for each step by the formula

$$\bigwedge_i \bigwedge_j \left(a_{ij} \implies \bigwedge_{k \neq j} \neg a_{ik} \right)$$

In our implementation, we encode this 'exactly one' requirement using the `SMT-LIB2` `pbeq` operator.

A SAT result (and subsequent model) from the SAT solver is translated to a chain assignment. The chain denoted by this assignment is then modeled and validated (see Procedure 4).

Since chain assignments are validated individually, they can be validated in parallel. Our implementation supports

the parallel validation of multiple chains via worker threads. Each worker thread validates a unique chain assignment and, in the event of a validation failure, generates a new constraint on gadget assignments that is immediately asserted by the main thread into the SAT problem. This allows the gadget assignment problem to combine constraints found by multiple workers and ensure that each worker is given gadget assignments that fulfil all constraints found by all workers.

If the SAT solver returns UNSAT, then gadget selection can not be made for the given set of steps and our algorithm continues to the next partition of reference program steps.

4.5 Validating Candidate Gadget Chains

A chain assignment selects a gadget for each step of the `p-code` reference program that mutates (at least) the same part of system state that the step mutates. For these gadgets to constitute a viable code-reuse attack, as modelled by the reference program, they must produce a compatible sequence of state updates if executed in sequence.

In our synthesis algorithm, a *chain model* is a logical formula that encodes these constraints. It asserts that the intermediate state resulting from each gadget in the chain equals the initial state of the next gadget. The model therefore encodes the correct sequential execution of the gadgets. It further asserts that the `p-code` model of the each of the chain's gadgets matches the `p-code` model of the corresponding step of the reference program. The formal models of gadgets are additionally constrained with any user-provided relational constraints. This is typically used to limit the areas of memory a ROP chain can access.

Our framework also allows additional and arbitrary constraints—so-called *specification* constraints—to be imposed on the chain model. This allows, for example, assumptions about the initial state to be made.

In our implementation, the resulting formula is checked for satisfiability by an off-the-shelf SMT solver (Z3 [22]). If the formula is satisfiable, the chain model is a realizable characterization of the sequence of states of the `p-code` virtual machine that would arise during a complete gadget chain execution emulating that of the reference program. If the formula is unsatisfiable, the SMT solver will produce an UNSAT core. Our algorithm then uses this to derive further constraints to refine the gadget assignments in the chain assignment generation step.

Procedure 4 shows how we construct the chain model and check its satisfiability. We now explain the procedure in detail.

Input: a candidate gadget chain G, a table of specification constraints E, a sequence of reference program steps S
Output: SAT if satisfiable, an UNSAT core if not

```
1  solver := SMTSOLVER();
2  ASSERT(solver, E.pre(FIRST(G)));
3  ASSERT(solver, E.post(LAST(G)));
4  foreach (gadget1, gadget2) in PAIRWISE(G) do
5  |  a := FINALSTATE(gadget1);
6  |  b := INITIALSTATE(gadget2);
7  |  ASSERT(solver, a = b);
8  |  branch := BRANCH(a) = ADDRESS(gadget2);
9  |  ASSERT(solver, branch);
10 end
11 foreach (gadget, step) in ZIP(G, S) do
12 |  ASSERT(solver, OUTPUTSEQUAL(gadget, step));
13 |  ASSERT(solver, E.relation(gadget));
14 |  if BRANCH(step) then
15 |  |  branch :=
   |  |    BRANCH(step) = BRANCH(gadget);
16 |  |  ASSERT(solver, branch);
17 |  end
18 end
19 if CHECKSAT(solver) then
20 |  return SAT;
21 else
22 |  return UNSATCORE(solver);
23 end
```

Procedure 4: Check

4.5.1 The Constraint Components of a Chain Model

The chain model built by our procedure encodes different types of logical properties that layer constraints on the chain. These are denoted by the ASSERT statements in Procedure 4. To aid in analysis of chain validation failures, every time we add a constraint we give it a specific label, indicating its role in the model and what gadget it constrains.

Our labels distinguish five roles: *memory*, *control-flow*, *semantic*, *precondition*, and *postcondition*. We explain each of these below.

Memory. To encode the sequential execution of our gadgets, the memory constraints assert that the final state of each gadget in the chain is equal to the initial state of its successor (if one exists). We exclude one modeled `p-code` space: `unique`. The `unique` space is meant to be a 'scratch pad' space for storing intermediate results in the `p-code` translation of complicated assembly instructions. As such, `SLEIGH` assumes that it is cleared between the decoding of every assembly instruction.

Control-Flow. Since our `p-code` block model does not represent `p-code` control-flow, we must separately assert the control-flow behavior of each gadget in our chain. The control-flow constraints assert that each gadget in the chain, other than the final gadget, must branch to its successor.

Semantic. To express the requirement that the chain emulates the reference program, the semantic constraints assert that each gadget is a refinement of its reference program step.

For each direct and indirect output of a reference program step, we assert that that the bit-vectors addressed by the output in the final state of the step and in the final state of the gadget are equal. For indirect outputs of the reference step, we additionally assert that the relevant pointer varnode bit-vectors in both states are equal.

We do not assert semantic equality for writes to the `unique` space. We do, however, assert equality of pointers used for indirect accesses, even if they reside in `unique`. If a reference step has a `p-code` branch (which will happen only in the last step), then we additionally constrain the branch destination of the gadget to be equal to that of the reference step.

We additionally overlay any user-provided constraints onto the formal model of each gadget, expressed as a predicate over the gadget's `p-code` operations. This is most often used to enforce that gadgets access only specific regions of memory.

Specification Constraints. These allow our algorithm to impose pre-conditions and post-conditions on the state of a gadget chain. This is useful for specifying implicit facts about the program under exploitation. For example, 'when this vulnerability is triggered, `rax` is always 0'. It is also used for asserting requirements not already captured by the `p-code` representation of the reference program. For example, '`rdi` must point, in the final state, to the string `"/bin/sh/\x00"`'.

We express pre- and post-constraints as sets of predicates over `p-code` states. The chain evaluation procedure asserts precondition predicates on the initial state of the first gadget (labeled as *precondition* constraints), and postcondition predicates on the final state of the final gadget (labeled as *postcondition* constraints). Our implementation exposes a configuration file that supports specifying simple register and memory constant-value equality constraints, from which we derive corresponding state predicates.

4.6 Conflict Clauses

We assert each of the constraints that make up the chain model in a SMT solver. When the SMT solver returns UNSAT, it also returns an *UNSAT core*: a subset of assertions that are responsible for the UNSAT result. We can inspect the labels of the propositions in the UNSAT core to determine both the set of gadgets that participated in the failure and the role of each gadget in the failure.

We use this association to create a *conflict clause*: a list of gadget assignments (see Section 4.4) identifying problematic combinations of gadgets. The gadget assignments in the conflict clause are translated back into the Boolean propositions used by the gadget assignment SAT solver. The negation of the conjunction of these propositions is added to the constraints of the SAT solver; this prevents the set of candidate gadgets in the conflict clause from being used together in future gadget assignments.

The strongest possible conflict clause corresponds to the full gadget assignment of the chain: every gadget participated in the UNSAT core. This produces a conflict clause ruling out a single full chain assignment, thus still guaranteeing forward progress of the algorithm.

The weakest possible conflict clause contains only a single gadget assignment, preventing that gadget candidate from being used in any future chain assignments.

4.6.1 Heuristic Conflict Clause Weakening

Consider a chain with three gadgets, represented by propositions g_a, g_b, g_c. And suppose a precondition constraint sets the memory used by g_c in such a way that g_c fails its semantic check. In this example, the UNSAT core of the chain model would contain four terms. The first term would identify the *precondition* constraint. The second term would identify the *memory* equality constraint between g_a's final state and g_b's initial state. Likewise, the third term will identify the link between g_b and g_c. The fourth term would identify the *semantic* assertion of g_c. The conflict clause generated from this core would be the conjunction $g_a \wedge g_b \wedge g_c$.

In this example, however, gadget c is unlikely to work in any chain, regardless of the choices made for gadgets a and b. We would therefore like to heuristically rule out all chains containing g_c. This does risk missing solutions, as it is *possible* that there are valid chains containing g_c. But these solutions are likely to be rare and there are probably more productive areas of the search space to explore.

We implement this heuristic by filtering the terms used to build the conflict clause, including only UNSAT core terms with a *semantic* or *control flow* label. Since gadgets a and b only feature through *memory* assertions, they will not appear in the weakened conflict clause, which is now a single term: g_c.

The intent of this heuristic is to reduce the average solving time without greatly impacting the algorithm's success rate. In our evaluation (Section 5), we perform an ablation study to determine whether this goal is achieved.

The presented heuristic is a baseline for analysis that can be done on chain model failures. With additional assertions and labels, it may be possible to separate the side-effects of gadgets from the effects of gadgets that are refinements of the reference program step, allowing for finer-grained analysis. And, in the gadget assignment procedure, a pre-analysis of all candidate gadgets may allow for a type of 'theory propagation' of conflict clauses: allowing a clause to constrain selections of equivalence classes of gadgets rather than individual gadgets.

5 Algorithm Evaluation

We seek to answer several questions regarding our algorithm:

- How does `crackers` compare to existing approaches in terms of speed and rate of success?

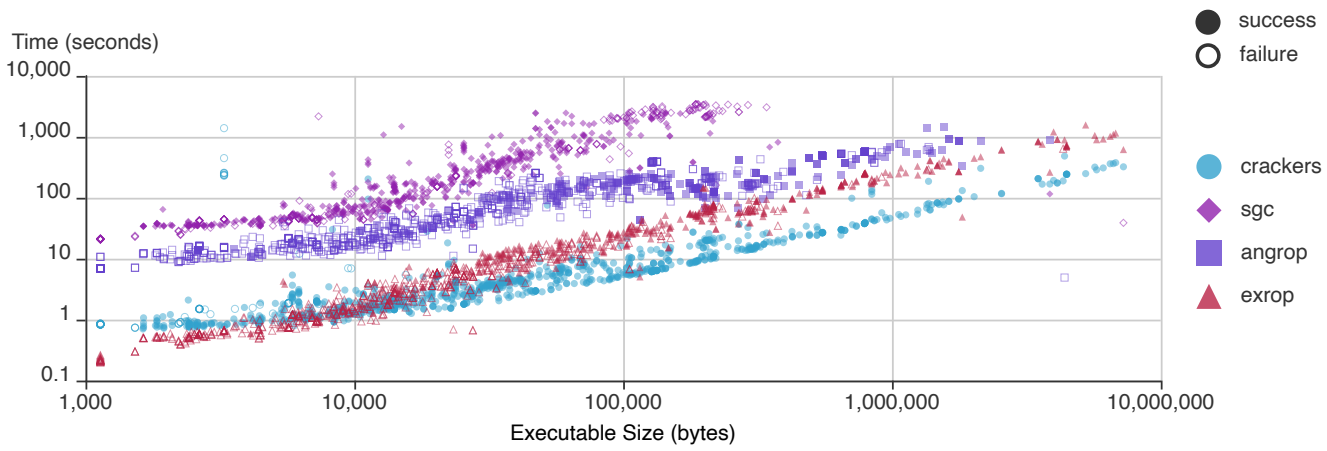

Figure 3: ROP Tool Benchmark Comparison

Table 1: ROP Tool Statistics

Tool	Success Rate	Failure Rate	Timeout Rate	Median T_{kB}(s/kB) (success)
`crackers`	92.42%	7.48%	0.10%	0.11 ($IQR = 0.10$)
`exrop`	34.55%	61.12%	4.33%	0.24 ($IQR = 0.09$)
`angrop`	25.39%	70.87%	3.74%	1.03 ($IQR = 1.23$)
`sgc`	52.66%	25.39%	21.95%	9.71 ($IQR = 6.45$)

- Does the usage of a reference program provide any benefit in speed or rate of success over simply constraining pre-and-post conditions?
- Does the conflict clause heuristic provide any benefit in speed? Does it negatively impact the rate of success?
- Can `crackers` synthesize working chains subject to the constraints of a real-world exploit?

5.1 ROP Tool Comparison

We evaluated `crackers` relative to existing ROP tools over a large corpus of test binaries. The raw results of this evaluation are captured in Figure 3 with summary statistics in Table 1.

Dataset. The ALLSTAR [23] dataset is a repository of Debian Jesse build artifacts. We chose the first 1000 binaries from ALLSTAR, ordered by package name, as our test corpus. This sample contains a diverse collection of executables and shared objects with anywhere from a few kilobytes up to tens of megabytes of executable space. As only `crackers` and `angrop` support architectures other than `x86`, we limited our consideration to `x86-64` ELF binaries in this test.

ROP Tools. We evaluated ROP tools capable of automatically synthesizing a function call, a task that requires setting registers and then jumping to an arbitrary address. We chose a function call rather than `execve` or `mprotect` as both those chains require the target binary to contain certain instructions or symbols and the various tools handle this requirement in ways that are difficult to compare. We also required that the tool be runnable by our Docker-based measurement setup, and that it succeed on a simple test case. `ROPGadget` [24] and `Ropper` [25] cannot synthesize arbitrary function calls, so were excluded. We were able to run `ROPium` [26], but were unable to synthesize trivial chains on a test file. We therefore evaluated the remaining ROP tools: SGC [10], `angrop` [27], and `exrop` [28]. For a general introduction to these tools, and comparison with our work, see Section 6.

Evaluation Setup. We ran each ROP tool against each binary in our dataset, with a timeout of 30 minutes. Each tool was configured to synthesize a function call `0xfacefeed(0xdeadbeef, 0x40, 0x7b)`, with no extra constraints on the system state. To the degree that we were able, we disabled multithreading or multiprocessing in every tool to reduce noise in the measurements. Each tool was run by a testing harness in a Docker container and timed with the linux `wait4` syscall, providing cumulative CPU usage of each tool's process tree. This approach was derived from guidelines by McGeoch [29]. We ran this evaluation on a workstation with 24 CPU cores and 256 gigabytes of RAM.

We made a minor alteration to the above test plan to accommodate SGC. SGC iterates over a matrix of run parameters, but does not terminate when a set of parameters succeeds. Our test runner externalizes this iteration by invoking SGC multiple times with individual run configurations. As SGC caches its analysis, this approach allows us to time it without

penalty. Each invocation of SGC is given a 30 minute timeout, so it is possible for SGC to get over 30 minutes of runtime per test-case, as occasionally happened in our test suite.

Limitations. It is important to recognize the limitations of this test. Each tool differs both in its synthesis algorithm and many engineering decisions made in its implementation. Our algorithm, `crackers`, is a native Rust [30] executable while each other tool evaluated is a Python script with some native C/C++ routines. The tools all have differing start-up costs and runtime efficiencies; we hope that by sampling on a wide range of binary sizes we can still identify meaningful trends.

As this test was run without imposing realistic state constraints, it is likely that each tool reported positive results that, while semantically allowable, would not be practically reachable from an exploit. Design choices may also impact success rates: `crackers` by default models all memory as accessible unless otherwise constrained, while `angrop` uses a heuristic that requires memory to be addressable from the stack pointer. This design choice likely increases `crackers`' reported success rate relative to `angrop`'s.

Analysis. To analyze the performance of each tool, we define a measure T_{kB}: the runtime in seconds per kilobyte of executable space in a binary. This measure provides a way to measure the speed of ROP chain synthesis across test cases with varying input sizes. The raw runtimes, and thus the T_{kB}, both contain numerous outliers, so we use the median and the Inter-Quartile Range (IQR) as summary statistics.

By T_{kB}, `crackers` shows a clear performance improvement over its competitors, though `exrop` has equivalent or better performance on small files, likely indicating that `crackers` has a higher initial startup cost than `exrop`. Surprisingly, SGC has a much higher failure rate than `crackers`. Inspection of the graph shows a high percentage of failures on larger file sizes. This may indicate that SGC hit an internal timeout before our testing harness detected a timeout, artificially inflating its failure rate and lowering its timeout rate. `angrop` and `exrop` both struggled, relative to SGC and `crackers`, with finding solutions in smaller binaries. We suspect that this is due to SGC and `crackers` not imposing functional categories on gadgets, allowing individual gadgets to fulfil multiple roles.

5.2 Ablation Study

We wish to study the performance implications of using reference programs and conflict clause heuristics. To this end, we ran an ablation study over variants of our algorithm.

`crackers_a`. The variant `crackers_a` does not apply the conflict-clause weakening heuristic. This represents a more conservative choice, as the heuristic errs on the side of ruling out too many candidate combinations.

`crackers_b`. This is a significant change to `crackers`, fully removing the reference program from the algorithm. Instead of selecting gadgets from the library via an output signature, gadgets are now selected using the configured chain post-conditions. This filtering is performed by syntactic comparisons over the logical model of the gadget, similar to those described in Section 4.3. Without reference program steps, `crackers_b` uses a single set of candidate gadgets and these gadgets may appear in any order in the chain. Only relational invariants, pre-and-post conditions, and memory constraints are enforced. This design is similar to what is used by SGC except that SGC builds a single SMT model while we maintain separate SMT chain modeling and SAT conflict resolution.

Test Setup. This test is intended to evaluate the performance of conflict clause resolution among the variants. To better measure this, we make two changes to the `crackers` variants' configuration to increase the number of evaluated conflict clauses. All variants are configured to synthesize chains with exactly 5 gadgets to ensure a shorter chain is not accepted. Additionally, pre-condition constraints have been placed on most general purpose registers and memory access is restricted to a several-kilobyte window. These changes ensure that the algorithm will encounter many conflicts. We ran all variants on a (stable) random subset of 500 binaries from the initial 1000 with a timeout of 30 minutes per run.

Analysis. The `crackers_b` variant performed poorly, timing out almost 80% of the time. By removing conflict clauses tied to specification program steps, we removed the mechanism by which many chain validation errors could be localized to individual gadgets. Almost all conflict clauses in `crackers_b` will then be the strongest possible: ruling out a single full gadget assignment. The results for `crackers_a`'s are more nuanced. `crackers_a` did show a modest increase in success rates, as it is more conservative in ruling out gadget assignments. However, this was at the cost of increasing the rate of timeouts at least as much, and greatly increasing the the spread of solution times as measured by T_{kB}. While the performance of `crackers` relative to `crackers_a` is not strong enough to claim a strict improvement, it is strong enough to advocate for leaving the conflict clause weakening heuristic in the algorithm as an optional element.

5.3 Real-World Usage

To evaluate the real-world utility of our algorithm, we adapted Google's proof of concept for triggering CVE-2017-14493 [31], a Remote Code Execution vulnerability in `dnsmasq` (previously used to demonstrate SGC [10]). We wrote a proof-of-concept that exploits this vulnerability with a ROP chain synthesized on-demand by `crackers`.

CVE-2017-14493 is a classic stack-buffer overflow. When `dnsmasq` processes a DHCP6 [32] 'Relay Forward' message with a `CLIENT_LINKLAYER_ADDR` option, it uses an attacker-controlled length field to `memcpy` attacker-controlled data into a fixed-length stack buffer. To analyze this vulnerability,

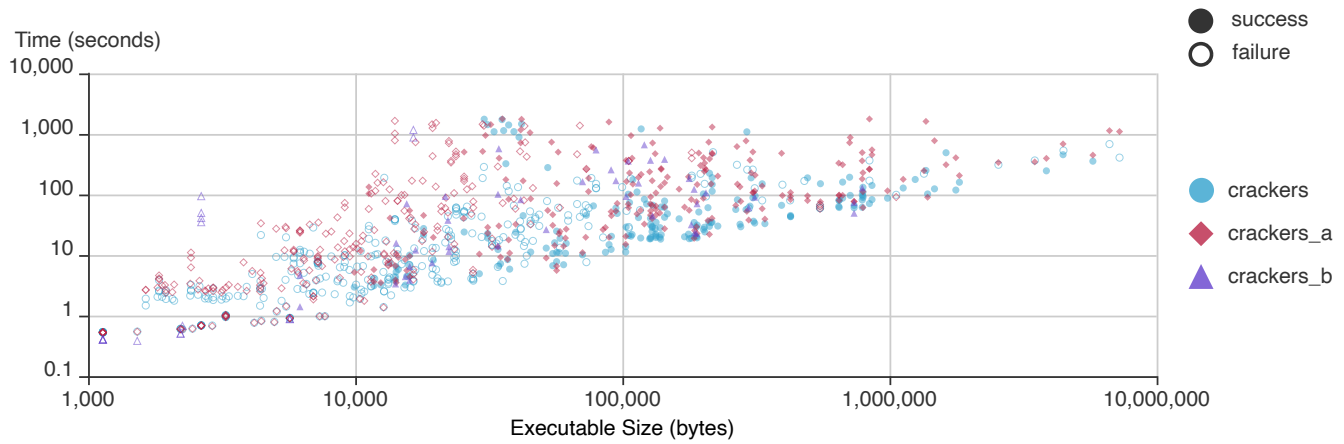

Figure 4: `Crackers` Ablation Study

Table 2: Ablation Study Statistics

Tool	Success Rate	Failure Rate	Timeout Rate	Median T_{kB}(s/kB) (success)
`crackers`	35.60%	62.40%	2.00%	0.20 ($IQR = 0.30$)
`crackers_a`	41.40%	47.80%	10.80%	0.52 ($IQR = 1.33$)
`crackers_b`	13.00%	7.20%	79.80%	0.53 ($IQR = 1.33$)

we first produced an `aarch64` build of `dnsmasq`. To simplify the exploit and ease debugging, we disabled ASLR and stack canaries and ran `dnsmasq` within `gdb` in a docker container. We triggered the vulnerabilty with a specially crafted `CLIENT_LINKLAYER_ADDR` buffer containing a non-repeating pattern. When `gdb` detected a crash, we inspected the final register and memory state.

From this system state, we found a set of logical relationships between our chain's initial CPU state, the initial memory state, and the contents of the DHCP6 packet. For example, we could see that the value of `pc` was equal to the sequence of 8 bytes at offset `0x32` of the `CLIENT_LINKLAYER_ADDR` buffer as well as the sequence of 8 bytes in memory at offset `sp` − `0x168`. For every register with a value derived from our test bytes, we enforced the respective equality as a precondition constraint for our algorithm. For every other register, we asserted they contained the concrete value found in the crash dump, preventing `crackers` from assigning arbitrary initial values. We configured `crackers` to read from RAM only within the `CLIENT_LINKLAYER_ADDR` buffer. These complex equality constraints would be difficult or impossible to express in any other ROP synthesis tool.

We synthesize a chain with a visible effect: making `dnsmasq` exit cleanly with an error code of 1, represented by the reference program `mov w0, #1; b exit;`. Once `crackers` synthesizes a chain, we employ its memory model to derive the contents of the `CLIENT_LINKLAYER_ADDR` buffer. We send this packet to `dnsmasq` and verify that it exits cleanly with an error code of 1.

6 Related Work

6.1 `p-code` Semantics and Modeling

Naus et al. [33] formulate an operational semantics for 'high `p-code`', an extension of `p-code` used in Ghidra's decompiler. A semantics of this kind enables proofs at the language level, but is less suitable for reasoning about specific code than a model, as developed in our work. This work is, however, a useful point of reference for `p-code` semantics in general.

GhiHorn [34] is a path analysis plugin for Ghidra that uses decision procedures to reason about `p-code` programs. GhiHorn operates in the context of a single decompiled function in Ghidra. Ghidra's decompilation, exposed through its API as high `p-code`, is processed to produce a set of Horn clauses—logical implications that encode control flow constraints. GhiHorn asserts these terms in Z3, allowing it to expose a reachability analysis to reverse engineers.

While GhiHorn models `p-code` programs in the language of SMT, it does not define a viable approach for our own work, as the high `p-code` it uses abstracts away many of the incidental side-effects that matter in ROP chain generation.

6.2 Code-reuse Attack Synthesis

There are three notable academic approaches to automating code-reuse attacks: Q [8], PSHAPE [9], and SGC [10]. All three use an IL to represent the effects of machine instructions and use decision procedures to reason about these effects.

Q. Q [8] accepts programs written in a high-level domain-specific language and attempts to compile them into a virtual assembly language. To synthesize a ROP chain, individual gadgets are chosen to represent virtual assembly instructions. Q uses concrete execution and decision procedures to determine whether a gadget implements a virtual assembly instruction, and whether a given scheduling of gadgets implements the desired semantics of the chain. As Q's implementation was not published, we could not evaluate it.

Q's approach mirrors that of a compiler's code generator, imposing stringent semantic requirements on every gadget it uses. This approach would likely allow it to quickly synthesize complex chains. But there are also drawbacks, which our semantically looser approach avoid. Q's set of virtual assembly instructions is fixed, so gadgets must fit into predefined categories. Q requires a gadget's semantics to match those of a virtual instruction in all input states with no side-effects. These strict requirements reduce the set of usable gadgets. They also incentivize Q to use very short gadgets. Many ROP mitigations developed over the last decade either detect sequences of very short gadgets or make it infeasible to use them, perhaps partially in response to Q.

Q used BAP [35] to reason about gadgets and chains.

PSHAPE. PSHAPE [9] partially automates the synthesis of ROP attacks for library calls (e.g. `mprotect`). PSHAPE is positioned as an aid to a human analyst, not an end-to-end synthesis technique. It provides written 'summaries' of the semantics of gadgets it identifies, and generates short chains to initialize registers, while meeting logical constraints.

PSHAPE's automated chain synthesis has several limitations. It is strictly limited to synthesizing ROP chains that perform function calls on `x86 linux`, using hardcoded calling conventions. Additionally, PSHAPE omits return instructions from its chain modeling. These missing constraints could easily lead to false positives. Schloegel et al. [10] discovered a similar issue: almost all PSHAPE chains crashed in their evaluation due to missing constraints on pointers.

PSHAPE models gadgets via the VEX IR from `valgrind` [36] and reasons about gadgets with Z3 [22].

SGC. SGC [10] is a system for automatically synthesizing code-reuse attacks that transition from a constrained start state to a constrained end state.

SGC produces a single logical formula to express the entire synthesis problem. Given N gadgets and a chain length M, SGC encodes all possible assignments of N gadgets into all chain positions up to M. This formula is given to `boolector` [37] in an external process. A SAT result produces a chain and a model of the memory and register states induced at every step of the chain's execution.

While SGC properly constrains synthesis, its design limits the scale of problems it can practically solve and the types of programs it can synthesize. In our evaluation, SGC was by far the slowest tool tested. Our algorithm matches the expressiveness of SGC while greatly improving on its performance.

The *size* of SGC's `SMT-LIB2` formula grows proportional to $N \times M$, making it quickly infeasible to scale either dimension of the problem. Pre- and post-conditions impose no constraints on the order of operations in a chain. This approach is very flexible, but makes it impossible to require that a location holds multiple values at different points in a chain. This precludes synthesizing chains that can perform 'cleanup' or perform more exotic operations (e.g. toggling memory-mapped control registers in a firmware context). While SGC supports parallelism through multiprocessing, it does not share state between workers, limiting the benefit.

SGC uses a customized version of the `miasm` [38] binary analysis framework to model gadgets. Reasoning about the models is done using the `boolector` [37] SMT solver with `picosat` [39] as the SAT backend.

6.2.1 Exploit Practitioners

In addition to the work cited above, there are several open-source ROP tools made by the community of Exploit Practitioners. As observed by Dullien [40], such tools often represent the state of the art in computer exploitation. Code-reuse attack tools from the community tend to be built as interactive aids to a human analyst. Of particular interest are `ROPGadget`, `Ropper`, `ROPium`, `angrop`, and `exrop`. Because this research is often done outside the academic process, much of our discussion below has necessarily been based on studying the source code, with the limitations that entails.

`ROPGadget`. `ROPGadget` [24] is the most popular (by Github stars), and oldest, open-source ROP tool. Its main use is the efficient identification of ROP gadgets, and is as such used as a component in many other ROP chain synthesis tools. `ROPGadget` itself contains its own rudimentary support for synthesis of `execve` chains on `x86` linux. This synthesis searches for gadgets containing specific `x86` instructions sequences (e.g. `pop rdx; ret;`), which it then inserts into a pre-configured arrangement to form a chain. This synthesis method is exclusively written for the `execve` chain, and relies on the existence of gadgets matching particular templates, severely limiting its flexibility.

`Ropper`. `Ropper` [25] is another popular open-source ROP tool. `Ropper`'s chain synthesis is slightly more flexible than that of `ROPGadget`, allowing for synthesis of `execve`, `mprotect`, and `VirtualProtect` chains. `Ropper`'s synthesis relies on a categorization of gadgets. Gadgets are disassembled, and the textual disassembly is evaluated against a set of regular expressions to determine the possible roles of the gadget (e.g. writing a value to `rdx`). `Ropper` models a chain as a sequences of steps, where each step is associated with a gadget category. To synthesize a chain, `Ropper` iterates over all possible permutations of candidate gadgets for each step and verifies that no gadget overwrites the output of any other.

This synthesis method also lacks flexibility: it can only synthesize three specific chains, it requires that gadgets match a hard-coded set of templates, and there is no attempt to resolve semantic conflicts between gadgets.

ROPium. ROPium [26] is a ROP synthesis tool that uses symbolic execution to synthesize ROP chains. Similar to Ropper, it categorizes gadgets into several higher-level functional categories. Chains are represented as a list of higher-level commands (e.g. "move rdx to rax, then call foo"), which themselves are translated into sequences of gadget steps. ROPium then performs a depth-first search for a satisfiable scheduling of its candidate gadgets for the given specification sequence. Unlike Ropper, ROPium symbolically executes candidate chains to verify their end states match the specification. ROPium, due to its use of symbolic execution, is also able to express initial memory constraints on candidate chains. ROPium uses its own custom symbolic executor, which appears to be an early version of Trail of Bits' MAAT [41].

Angrop. Angrop [27] is a ROP synthesis tool distributed as part of the angr [42] binary analysis framework. Angrop, like ROPium, expresses ROP chains as a high-level series of steps (e.g. writing memory, setting registers, calling functions). Each of these specification steps has its own strategy to identify candidate gadgets for that step from the overall gadget pool. Angrop performs a breadth-first search of its candidate gadgets when building a chain, using the angr symbolic executor to validate the semantics of constructed chains and enforce simple precondition constraints. While angrop is a well-written and capable tool, its reliance on placing gadgets into functional categories likely harms its ability to form chains in highly constrained scenarios.

Unlike every other tool, angrop provides some level of multi-architecture support, including support for x86/x64, arm32/64, and mips. This is enabled by angr's use of Valgrind's [36] VEX IR. As angr supports additional architectures, angrop could likely be extended to these as well.

Exrop. Exrop [28] is a now-deprecated ROP synthesis tool touting its usage of SMT to synthesize chains on x86-64. Exrop uses the Triton [43] symbolic executor to express and solve the values of registers during the building of a chain. A chain specification is translated into a dictionary associating register names and memory locations with desired concrete values. Exrop then performs a depth-first search of gadget assignments, ensuring that each gadget sets at least one register to its intended value. In the event that a gadget overwrites an already-set register, exrop re-adds the register onto its search queue and recurses (up to a maximum depth), allowing for conflict resolution between gadgets.

Our evaluation shows that this methodology is clearly performant, as exrop came the closest to matching our algorithm. However, as exrop struggled disproportionately on smaller binaries, we suspect it is less effective at handling synthesis with small pools of gadgets.

7 Discussion and Outlook

We demonstrated a novel method for synthesizing code-reuse attacks from a reference program. By formally modelling p-code at the level of operations and varnodes, we assure that our technique is not limited to binaries of any particular operating system, calling convention or even architecture. We demonstrated that this flexibility did not compromise the runtime performance of our method, as it matched or outperformed all evaluated ROP chain synthesis tools.

7.1 Future Work

We see opportunity to develop many aspects of our work:

p-code Modeling. Our p-code model has many limitations. We separately model data movement in p-code basic blocks and control-flow between basic blocks. By encoding p-code data movement and control-flow in a more integrated way, we could widen the set of p-code constructs we support to include conditional control-flow and bounded loops.

Such an extension to our model would also allow for bounded model checking of p-code control-flow graphs. This could be a powerful analytic tool for reverse-engineers to use alongside Ghidra's existing control-flow analysis. Recent work has demonstrated bounded model checking on LLVM bitcode [44], which operates at a similar level of abstraction to p-code, showing the promise of this direction.

Our model could also be extended to support additional p-code operations, such as floating-point operations.

Code-Reuse Attack Synthesis. Further development of our formal model of p-code would enable enhancements to crackers. It may be possible to synthesize chains for reference programs with conditional control flow (e.g. 'if EAX equals 0, execute a syscall') or bounded loops. It may also express 'conditional gadgets': directed graphs of p-code basic blocks that behave as a gadget only under certain inputs.

Our algorithm's representation of gadget assignments and chain conflict clauses could also be refined. By using multiple propositions to encode gadget assignments, it would be possible to have a conflict clause encode the reason for the gadget's inclusion in the clause. This in turn would allow for a type of 'theory propagation': using learned constraints from one gadget to constrain an equivalence class of gadgets.

Agentic Reference Program Synthesis. Our method's usage of p-code programs for ROP chain specifications raises the possibility of integration with other tools. We developed a prototype integrating a Large Language Model (LLM) with our algorithm, enabling synthesis of a ROP chain from a verbal specification. The LLM synthesizes reference programs for use with our algorithm and refines them in response to synthesis failures. This setup performed well in our initial testing and we suspect that further integration will be possible.

8 Ethics Considerations and Compliance with the Open Science Policy

8.1 Ethics

All research into offensive techniques carries the risk of enabling bad actors. Such research contributions must be weighed against the potential harm they might cause. We believe the benefit of our research outweighs these risks.

Our formal model of `p-code` in logic , taken on its own, merely provides a way to reason about binary code. While our `p-code` modeling is novel, the concept of modeling native code with propositional logic is well-established and extensively used in tools such as the popular symbolic executor `angr`.

We believe there is a net benefit in releasing our code-reuse attack synthesis algorithm.

There is a long history of interplay between offensive and defensive security research. Providing new methods for code-reuse-attack synthesis may shed light on new mitigations, or provide new tools for those developing defenses against these techniques. Additionally, while this work presents novel techniques and applications, it does not uncover previously unknown vulnerabilities in computer systems.

8.2 Open Science

We will submit a comprehensive set of artifacts for this paper.

Firstly, we will provide the full source code for our `p-code` modeling, discussed in Section 3. This code is primarily implemented in the Rust language with some C++ components for interfacing with `SLEIGH`. This software artifact also includes a small command-line-interface allowing for manual inspection of the logical formulas generated for any machine instruction supported by `SLEIGH`.

We will provide the full source code for our code-reuse-attack synthesis algorithm, discussed in Section 4. This code is also written as a Rust library. We provide a command-line-interface allowing for the usage of a configuration file to initialize the synthesis algorithm. This is useful for quick evaluations of chains and binaries, but we envision actual usage of this tool being through its software API.

We will provide all source code needed to produce the graphs and statistics used in our evaluation (minus minor stylistic tweaks), as well as a copy of the raw data collected during the evaluation. We will additionally provide all the code making up our evaluation framework (including a custom Rust API for ALLSTAR, a frontend library for programmatically interacting with our evaluation data from a redis database, the evaluation utility used to run all the ROP tools, and the docker compose project that runs it all). This is sufficient to fully rerun our evaluation and verify our results.

We will provide the demonstration setup for synthesizing a working chain in the context of an exploit of CVE-2017-14493. This is a Docker Compose project containing the vulnerable service (`dnsmasq`) and a small tool that uses `crackers` to synthesize a working ROP chain within a set of constraints imposed by the vulnerability.

All of the above artifacts are publicly available here: https://zenodo.org/records/14738161

References

[1] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, 2007.

[2] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP exploit mitigation using indirect branch tracing," in *22nd USENIX Security Symposium (USENIX Security 13)*, (Washington, D.C.), pp. 447–462, USENIX Association, Aug. 2013.

[3] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, The Internet Society, 2014.

[4] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[5] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), p. 30–40, Association for Computing Machinery, 2011.

[6] 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, no. 2, pp. 139–156, 2018.

[7] 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 *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 969–986, IEEE, 2016.

[8] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy," in *20th USENIX Security Symposium (USENIX Security 11)*, (San Francisco, CA), USENIX Association, Aug. 2011.

[9] A. Follner, A. Bartel, H. Peng, Y.-C. Chang, K. Ispoglou, M. Payer, and E. Bodden, "Pshape: Automatically combining gadgets for arbitrary method execution," in *Security and Trust Management* (G. Barthe, E. Markatos, and P. Samarati, eds.), (Cham), pp. 212–228, Springer International Publishing, 2016.

[10] M. Schloegel, T. Blazytko, J. Basler, F. Hemmer, and T. Holz, "Towards automating code-reuse attacks using synthesized gadget chains," in *Computer Security – ESORICS 2021* (E. Bertino, H. Shulman, and M. Waidner, eds.), (Cham), pp. 218–239, Springer International Publishing, 2021.

[11] "Ghidra." https://ghidra-sre.org/. Accessed: 2024-07-11.

[12] D. Monniaux, "A survey of satisfiability modulo theory," in *Computer Algebra in Scientific Computing* (V. P. Gerdt, W. Koepf, W. M. Seiler, and E. V. Vorozhtsov, eds.), pp. 401–425, Springer International Publishing, 2016.

[13] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.6," tech. rep., Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[14] N. Ramsey and M. F. Fernández, "Specifying representations of machine instructions," *ACM Trans. Program. Lang. Syst.*, vol. 19, p. 492–524, may 1997.

[15] C. Cifuentes and M. Van Emmerik, "Uqbt: adaptable binary translation at low cost," *Computer*, vol. 33, no. 3, pp. 60–66, 2000.

[16] C. Cifuentes, *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994.

[17] "Ghidra sleigh documentation." https://github.com/NationalSecurityAgency/ghidra/blob/2760eebc929d4b0ac99a7c5aff201eaf0db3e967/GhidraDocs/languages/html/sleigh.html. Accessed: 2024-07-18.

[18] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004* (K. Jensen and A. Podelski, eds.), vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176, Springer, 2004.

[19] D. Kroening, P. Schrammel, and M. Tautschnig, "CBMC: the C bounded model checker," *CoRR*, vol. abs/2302.02384, 2023.

[20] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[21] C. Sinz, S. Falke, and F. Merz, "A precise memory model for Low-Level bounded model checking," in *5th International Workshop on Systems Software Verification (SSV 10)*, (Vancouver, BC), USENIX Association, Oct. 2010.

[22] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (C. R. Ramakrishnan and J. Rehof, eds.), (Berlin, Heidelberg), pp. 337–340, Springer Berlin Heidelberg, 2008.

[23] JHU/APL Staff, "Assembled labeled library for static analysis research (allstar) dataset." https://allstar.jhuapl.edu/, Dec 2019.

[24] J. Salwan, "Ropgadget." https://github.com/JonathanSalwan/ROPgadget. Accessed: 2024-09-03.

[25] S. Schirra, "Ropper." https://github.com/sashs/Ropper. Accessed: 2024-09-03.

[26] B. Milanov, "Ropium." https://github.com/Boyan-MILANOV/ropium. Accessed: 2025-01-06.

[27] "angrop." https://github.com/angr/angrop. Accessed: 2024-09-03.

[28] "exrop." https://github.com/d4em0n/exrop. Accessed: 2024-09-03.

[29] C. C. McGeoch, *A Guide to Experimental Algorithmics*. Cambridge University Press, 2012.

[30] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, pp. 103–104, ACM, 2014.

[31] Google, "CVE-2017-14493 proof of concept." https://github.com/google/security-research-pocs/blob/master/vulnerabilities/dnsmasq/CVE-2017-14493.py. Accessed: 2025-01-06.

[32] T. Mrugalski, M. Siodelski, B. Volz, A. Yourtchenko, M. Richardson, S. Jiang, T. Lemon, and T. Winters, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)." RFC 8415, Nov. 2018.

[33] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, "A formal semantics for p-code," in *Verified Software. Theories, Tools and Experiments.* (A. Lal and S. Tonetta, eds.), (Cham), pp. 111–128, Springer International Publishing, 2023.

[34] J. Gennari, "Ghihorn: Path analysis in ghidra using smt solvers." Carnegie Mellon University, Software Engineering Institute's Insights (blog), Oct 2021. Accessed: 2024-09-03.

[35] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification* (G. Gopalakrishnan and S. Qadeer, eds.), (Berlin, Heidelberg), pp. 463–469, Springer Berlin Heidelberg, 2011.

[36] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[37] R. Brummayer and A. Biere, "Boolector: An efficient smt solver for bit-vectors and arrays," in *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*, pp. 174–177, Springer, 2009.

[38] C. I. Security, "miasm - reverse engineering framework in python." https://github.com/cea-sec/miasm. Accessed: 2024-08-05.

[39] A. Biere, "Picosat essentials," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2-4, pp. 75–97, 2008.

[40] T. Dullien, "Weird machines, exploitability, and provable unexploitability," *IEEE Transactions on Emerging Topics in Computing*, vol. 8, no. 2, pp. 391–403, 2020.

[41] B. Milanov, "Maat." https://github.com/trailofbits/maat. Accessed: 2025-01-06.

[42] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, pp. 8–9, IEEE, 2017.

[43] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, pp. 31–54, 2015.

[44] S. Priya, Y. Su, Y. Bao, X. Zhou, Y. Vizel, and A. Gurfinkel, "Bounded model checking for llvm," in *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 214–224, 2022.