Towards Automating Code-Reuse Attacks Using Synthesized Gadget Chains

Moritz Schloegel, Tim Blazytko, Julius Basler,
Fabian Hemmer, and Thorsten Holz

Ruhr-Universität Bochum, Germany
firstname.lastname@rub.de

Abstract. In the arms race between binary exploitation techniques and mitigation schemes, *code-reuse attacks* have been proven indispensable. Typically, one of the initial hurdles is that an attacker cannot execute their own code due to countermeasures such as *data execution prevention* (DEP, `W^X`). While this technique is powerful, the task of finding and correctly chaining gadgets remains cumbersome. Although various methods automating this task have been proposed, they either rely on hard-coded heuristics or make specific assumptions about the gadgets' semantics. This not only drastically limits the search space but also sacrifices their capability to find valid chains unless specific gadgets can be located. As a result, they often produce no chain or an incorrect chain that crashes the program. In this paper, we present `SGC`, the first generic approach to identify gadget chains in an automated manner *without* imposing restrictions on the gadgets or limiting its applicability to specific exploitation scenarios. Instead of using heuristics to find a gadget chain, we offload this task to an SMT solver. More specifically, we build a logical formula that encodes the CPU and memory state at the time when the attacker can divert execution flow to the gadget chain, as well as the attacker's desired program state that the gadget chain should construct. In combination with a logical encoding of the data flow between gadgets, we query an SMT solver whether a valid gadget chain exists. If successful, the solver provides a proof of existence in the form of a synthesized gadget chain. This way, we remain fully flexible w.r.t. to the gadgets. In empirical tests, we find that the solver often uses all types of control-flow transfer instructions and even gadgets with side effects. Our evaluation shows that `SGC` successfully finds working gadget chains for real-world exploitation scenarios within minutes, even when all state-of-the-art approaches fail.

1 Introduction

Early exploitation techniques relied on code-injection attacks, where an attacker injects shellcode into the memory space of an application and then executes it. However, quickly established mitigations forced attackers to adapt. Especially the introduction of the `W^X` policy (commonly referred to as *data execution prevention* (DEP)) made the execution of injected code infeasible, as memory is marked as either writable or executable. This forced attackers to develop

novel exploitation techniques that *reuse* already existing code (e. g., *return-to-libc*) [26, 30, 32]. As an additional line of defense, modern operating systems randomize a program's address space layout (ASLR). Still, a single *information leak* or small, non-randomized parts of the executable often provide an attacker the capability to mount their attack. In the past years, control-flow integrity (CFI) [1] has gained popularity. This technique enforces the property that only legitimate control-flow transitions inside a benign set required by the program are performed. While greatly limiting the attacker's freedom to chain arbitrary code snippets, so-called *code-reuse attacks* are still feasible in practice [11, 24, 35]. In general, code-reuse attacks have been shown to be Turing complete [22, 23]. Note that in practice, attackers often only need to disable `W^X` before they can execute arbitrary shellcode in the context of the exploited program. This is commonly achieved by chaining so-called *gadgets*, (short) sequences of instructions ending with an indirect control-flow transfer such as `ret` [26]. Even medium-sized programs contain thousands of gadgets, making the process of extracting and finding a suitable combination cumbersome. Various techniques to automate the process were proposed: Initial attempts used pattern-matching-based strategies to identify a chain [20, 21]; later approaches [2, 8, 17] make use of symbolic execution to classify gadgets and identify undesirable side effects, e. g., writing values to memory. However, even the most advanced approaches to date rely on various heuristics to confine the large search space [11, 24, 35]. While sometimes effective, pruning may lead to false negatives: these heuristics try to find generic chains to work across many targets, but in some cases no such chain exists.

In this paper, we propose a novel method to find gadget chains efficiently *without* pruning the search space. One category of tools that particularly excels at finding solutions for decision problems involving a large search space are SMT solvers [33]; they check if a (potentially large) set of logical formulas—so-called *constraints*—can be satisfied [15]. By building a logical formula that describes (1) the CPU and memory state before executing the first gadget, (2) the CPU and memory state desired by the attacker, and (3) the data flow between gadgets, we can model the gadget chain synthesis as a reachability problem and use an SMT solver to decide it. This approach is similar to bounded model checking [27], a software verification technique used to determine whether a system meets a given set of requirements: it combines a set of assumptions that have to hold before execution *(preconditions)* and a set of requirements that have to hold after execution *(postconditions)* with a logical encoding of the program semantics and then queries an SMT solver. If the solver returns `SAT` (satisfiable), it provides a *model* representing a concrete variable assignment that satisfies the given constraints. In our case, this implies that the solver successfully synthesized a gadget chain. If the result is `UNSAT` (unsatisfiable), the SMT solver mathematically proved that the constraints cannot be satisfied and, thus, no chain can exist for the given set of gadgets.

We introduce the design and implementation of `SGC`, a generic approach capable of automatically identifying gadget chains without relying on any classification or heuristics to prune the search space. At the same time, the logical

formula offers a framework to specify target-specific constraints. Our evaluation demonstrates that `SGC` not only outperforms all state-of-the-art tools with regard to finding gadget chains, but the synthesized chains always work in real-world scenarios. For instance, we demonstrate how we can craft a gadget chain that spawns a shell for a stack-based buffer overflow in `dnsmasq`: After defining the concrete CPU state as preconditions, we encode the target state right before executing the system call `execve(&"/bin/sh", 0, 0)`; running `SGC` provides us with a gadget chain spawning the shell without requiring any other information. We further demonstrate that even complex constraints (e. g., the sum of all values in the gadget chain must be equal to a specific value) can be satisfied by `SGC`.

In summary, our main contributions are:

– We introduce a generic approach to synthesize gadget chains in an automated way based on bounded model checking. Our approach does not require heuristics or pruning of the search space; instead, the SMT solver provides a proof of existence in the form of a gadget chain or proves that no gadget chain can be found for the given gadgets and constraints.
– We present the design and evaluation of our prototype `SGC`. We show that it not only outperforms all state-of-the-art approaches, but also works in real-world settings.
– Our approach provides unprecedented flexibility: `SGC` allows an attacker to specify arbitrary constraints and, thus, model even complex or unusual exploitation scenarios.

To foster further research in this area, we open-source `SGC` at https://github.com/RUB-SysSec/gadget_synthesis.

2 Shortcomings of State-of-the-Art Approaches

In the following, we discuss state-of-the-art approaches from academia and industry that can be used in practice to generate gadget chains automatically and analyze their shortcomings in this regard (cf. Table 1). We find that existing tools can be separated into two categories, based on their gadget chain generation:

Hardcoded Chaining Rules. `Ropper` [21] and `ROPgadget` [20] both fall into this category. Their main task is to extract gadgets, but both require hardcoded rules based on regular expressions to chain gadgets. While `ROPgadget` only supports a single exploitation scenario (i. e., building a system call to `execve(&"/bin/sh\0", 0, 0)`), `Ropper` allows system calls to `mprotect` as well. As a result, these tools are inflexible in practice.

Symbolic Exploration. `angrop` [2] and `ROPium` [17] operate on an *intermediate representation* of gadgets, which allows them to symbolically determine side effects and perform a classification. To this end, gadgets are first lifted, then analyzed, and chained together in the last step. The latter usually involves an algorithm such as *depth-first search* (`ROPium`) or *breadth-first search* (`angrop`) to identify a sequence of gadgets that fulfills the attacker's specifications, such as specific argument values. While vastly more flexible than approaches using

Table 1: Features of different tools capable of automatically chaining gadgets.

	SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
supports chains without `ret`	✓	✗	✗	✓	✓	✓
no hardcoded chaining rules	✓	✓	✓	✓	✗	✗
no classification needed	✓	✗	✗	✗	✗	✗
supports arbitrary postconditions	✓	✗	✗	✗	✗	✗

hardcoded rules, these tools are no panacea. They still rely on a classification of gadgets, and while they provide greater flexibility by allowing simple memory and register constraints, they lack support for more elaborate constraints. `P-SHAPE` by Follner et al. [8] also uses a symbolic exploration approach. However, it only focuses on finding gadgets useful for constructing library calls. It does neither provide a full gadget chain nor allows an attacker to specify any constraints.

Overall, all approaches lack flexibility; especially, they fail to support arbitrary postconditions (cf. Table 1). Instead, they rely on a classification of gadgets and pre-defined strategies to identify a gadget chain. Even when finding a chain, we empirically observe that they often crash the targeted program, e. g., through invalid memory accesses. Despite this, no tool makes any attempt at verifying the correctness of the generated gadget chains.

3 Design

In the following, we present a gadget-agnostic design that does not perform any pre-classification of gadgets while providing high flexibility by allowing to specify arbitrary, complex constraints. The nature of our approach overcomes the limitations of existing approaches. Most importantly, we can enforce an arbitrary CPU register and memory state before and after the exploitation—our design will identify a gadget chain facilitating the transition from the initial to the desired state using any gadgets available, including such using `jmp` and `call` instructions. To this end, our approach encodes the search of the gadget chain as a synthesis problem that an SMT solver decides. More specifically, our design is based on bounded model checking: preconditions and postconditions are represented by the initial and desired CPU state, while a logical formula encodes the possible gadget chain that facilitates the transition between both states.

Recall that bounded model checking is usually applied to a well-defined unit of code, such as a function with specific conditions. The goal of bounded model checking is to qualitatively assert that no diversion from the specified postconditions is possible (i. e., any diversion implies a bug that must be fixed). In other words, the goal is to find a counterexample *violating* the postconditions. For the use case of synthesizing a gadget chain, the scenario is slightly different: There is no well-defined unit of code such as a function, but a large number of individual gadgets that can be executed in an arbitrary order. As a consequence, we are not interested in knowing whether specific postconditions can be *violated* (as this most certainly is the case given the number and nature of the gadgets); instead, we are interested in whether there exists a chain of gadgets that *satisfies* the postconditions. In other words, we task the SMT solver with finding a

satisfying assignment for $preconditions \wedge gadget_chain \wedge postconditions$. If the solver finds such an assignment, the produced model contains concrete values for all variables—including stack or other attacker-controlled buffers—which describe the chain of gadgets. Thus, once a model is found, converting the values into a chain becomes a trivial task. In the following, we present these steps in detail.

3.1 Gadgets

First, we must extract gadgets from the target program, which can then be further processed. This step is independent of the subsequent encoding and is covered in detail by previous works in this area [3, 4, 6, 9, 26]. As such, we omit it here for brevity. Note that we do not require the gadget extraction to be exhaustive or classify gadgets, as long as these sequences of instructions end with an indirect control-flow transfer. As assembly instructions commonly have side effects (e. g., `mul rbx` implicitly modifies the `rdx`, `rax`, and `rflags` register), we disassemble and lift the gadgets to an *intermediate representation (IR)* with explicit side effects. An example for two gadgets is visible in Figure 1a. Noteworthy, each IR instruction has no implicit side effects. We reiterate that—other than most state-of-the-art tools—our design imposes no restrictions, ranking, or classification on the gadgets.

3.2 Logical Encoding

Given a pool of gadgets, we want to query an SMT solver to find a chain of gadgets that transitions the initial program state (formulated as preconditions) into the desired program state (formulated as postconditions). For this, we need to logically encode the semantics of gadgets and chains. Especially, we must model the semantics of gadgets, the data flow between instructions, and the data flow between gadgets. Once we have encoded all components, we must combine them into a single formula, which we then pass to an SMT solver. To construct such a formula, we connect each statement through conjunctions. In the following, we first describe how individual gadgets are encoded and then explain how gadgets are interconnected to form a chain.

Instructions and Gadgets. To use a gadget in the logical formula, we must first model all implicit state transitions on the instruction level: While a CPU executes a sequence of instructions in a row, it implicitly tracks state changes in registers and memory. To represent this behavior in a logical formula, we must explicitly model it on the instruction and inter-instruction level. To address the instruction level, recall that we lift instruction into an IR form that explicitly handles side effects. For the latter, we have to model the data flow between instructions, e. g., when a register is assigned to another register or is defined more than once. To achieve this, we make variable assignments stateful by converting IR instructions into *static single assignment (SSA)* form [7]. This implies that each variable *definition* is assigned a new unique index, while *uses* always use the last defined index. To differentiate between gadgets, we prefix SSA variable names with an identifier that is unique to each gadget. If a gadget

```
1  gadget_a:
2    mov rbx, [rsp+8] ; rbx := @64[rsp + 8]
3    mov [rsp], rdx   ; @64[rsp] := rdx
4    ret              ; rsp := rsp + 8
5                     ; rip := [rsp - 8]
```

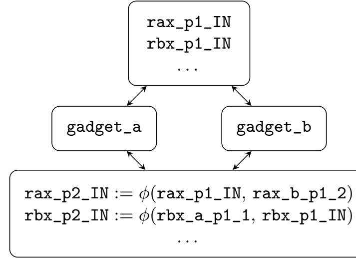
```
1  gadget_b:
2    pop rax ; rax := @64[rsp]
3            ; rsp := rsp + 8
4    inc rax ; rax := rax + 1
5    jmp rbx ; rip := rbx
```

(a) Assembly code and the corresponding intermediate representation (IR) of the instructions as comments. Note that side effects are explicitly modeled in the IR, thus a single assembly instruction may result in multiple IR instructions.

```
1  gadget_a:
2    rbx_a_1 := read(M_IN, rsp_IN + 8, 64)
3
4    M_a_1 := write(M_IN, rsp_IN, rdx_IN, 64)
5
6    rsp_a_1 := rsp_IN + 8
7    rip_a_2 := read(M_a_1, rsp_a_1 - 8)
```

```
1  gadget_b:
2    rax_b_1 := read(M_IN, rsp_IN, 64)
3    rsp_b_1 := rsp_IN + 8
4
5    rax_b_2 := rax_b_1 + 1
6
7    rip_b_2 := rbx_IN
```

(b) SSA form of the IR representation. The variable's locality is specified by an unique identifier, here `_a` or `_b`. Suffix `_IN` represents the initial definition.



(c) Structural overview of the final SMT formula, assuming a chain of two gadgets.

Fig. 1: The high-level idea of our logical encoding: We lift assembly gadgets to an intermediate representation, make the variable and memory accesses stateful (via static single assignment form) and encode the data flow between gadgets using ϕ-functions.

uses a variable that was not defined previously within this particular gadget, we postfix it by `_IN` to indicate that the value has been defined outside of the gadget's scope. In other words, it is an input to the gadget.

Example 1. Figure 1b shows how `rip_a_2` depends on the memory at address `rsp_a_1 - 8` (line 7), which itself can be calculated as `rsp_a_1 = rsp_IN + 8` (line 6). Note the identifier `_a`, which distinctly marks this variable as belonging to gadget `a`, and the postfix `_IN` indicating that this instruction depends on `rsp`'s definition outside this gadget.

Memory. Similar to registers, we apply SSA to memory to make it stateful, as otherwise, the SMT solver has no context information about memory addresses and values. To transform memory into SSA form, we define memory read and write accesses as explicit operations: `v_j := read(M_i, address, size)` and `M_i+1 := write(M_i, address, value, size)`. Given a stateful memory variable `M`, we read from and write to this variable at a given address with a given access size. Note that the write operation is stored in a new memory variable `M_i+1` that encodes the previous write. Internally, these operations are expressed within a

byte-wise memory model similar to the work of Sinz et al. [27], in which memory accesses with larger sizes are translated to nested byte-wise memory reads or writes. For a formal definition, we refer the interested reader to Appendix A. We initialize all memory addresses to contain the value 0.

Interconnecting Gadgets. Up until now, we described how to encode data flow within a single gadget using SSA for registers and memory. However, our goal is to combine multiple gadgets in a chain of length n without making assumptions on neither the order of gadgets nor the particular gadgets used. Especially, we allow gadgets to occur more than once in the chain. Thus, in the next step, we have to logically encode the data flow between gadgets. To achieve this, we first have to ensure that all variables are unique. So far, variables are only unique with respect to their gadget due to the SSA form's unique identifier. However, to encode the order of execution, each variable must also be unique with regard to the gadget's position within the chain. Therefore, we also include the position as index within the SSA variable names: `variable_gadgetId_position_definitionIdx`. This way, we can use any gadget at any position in the chain.

Example 2. If we consider the gadget for the first position in the chain, the definition `rbx_a_1` (line 2 in Figure 1b) becomes `rbx_a_p1_1` (with `p1` representing the first position). This way, we can use the gadget in position 2 as well, as `rbx_a_p2_1` is a distinct variable.

Naturally, our encoding must consider that a gadget can be used at any position in the chain, while, at the same time, we cannot know which gadget is at a specific position within the chain. In other words, `gadget_a` and `gadget_b` can both be at positions 1 and 2, but at the time of formula generation, we do not know which of these gadgets will be at which position in the chain synthesized by the SMT solver. Therefore, we must ensure that the gadget at position $i+1$ uses the values derived by the gadget at position i; a scenario strikingly similar to the problem of merging control flow in SSA form (for which ϕ-functions are used). We must merge the state of all gadgets at chain position i such that it can be used as input for the gadgets in the subsequent position. To achieve this, we apply the following for each register and memory variable: We first determine the variable's last definition in each gadget for position i. Then, we merge the last definitions from all gadgets via a ϕ-function and define a new variable that is used as input for the next position.

Example 3. Assume that we want to encode the gadgets for a chain of length 2 (cf. Figure 1c). For each register, we create a ϕ-function that merges the last definitions of these variables. In the following, we consider this process exemplary for `rax` at position 1. The initial value of `rax` is `rax_p1_IN`—the input of `rax` for the first gadget position. Since we do not know if `gadget_a` or `gadget_b` is the first gadget in the chain, we must account for both possibilities and merge their last definitions of `rax` in a ϕ-function. `gadget_a` does not modify `rax`, thus we use `rax_p1_IN`; for `gadget_b`, we use its latest definition, `rax_b_p1_2`. Finally, we define a new variable—`rax_p2_IN`—that encodes the merged variables and is used as input for the second position in the chain: $\texttt{rax_p2_IN} := \phi(\texttt{rax_p1_IN}, \texttt{rax_b_p1_2})$.

To model the data flow between gadgets, the logical formula has to connect each input variable of the ϕ-function with the gadget that defined the corresponding variable. On a technical level, we translate this abstract ϕ-function into nested `If-Then-Else` expressions that select the corresponding variable based on the program counter, which has to be equal to one of the gadget addresses. This way, we ensure that the conditions are mutually exclusive (as the program counter can only point to a single gadget) and, thus, each register's value can always be uniquely determined. This approach is based on work by Sinz et al. [27].

3.3 Preconditions and Postconditions

Following the logical encoding of the gadget chain, we now describe how to set the initial state (preconditions) and the targeted state (postconditions).

Preconditions. These conditions allow setting the initial state at the time when the attacker can divert execution flow to the gadget chain. They constraint the inputs of the first position in the gadget chain, e. g., we can encode relevant context from the target program, such as the value of specific registers or memory areas (e. g., by using a debugger). Additionally, we must specify the location where the SMT solver should place the synthesized gadget chain (and how many bytes are available), e. g., by choosing an attacker-controlled buffer on the stack. This area is then considered a free variable in the formula, such that the SMT solver can place gadget addresses and data there. We can also enforce specific characteristics for any attacker-controlled areas, such as constraining memory buffers to hold only values within a certain range.

Postconditions. While the preconditions outline the initial position, postconditions describe the desired state that the program should reach after executing the gadget chain. More specifically, we can set any register or memory address to a specific value (e. g., the system call we would like to execute and its arguments). We encode these postconditions by asserting that the outputs (i. e., register and memory variables) of the last position in the chain are equal to the given values.

Furthermore, we also support indirect constraints, so-called *pointer constraints*. These constraints support common constructs, where a reference to a specific value or string (e. g., "/bin/sh") in memory needs to reside in a specific register. To this end, we add an assertion that the memory address pointed to by this register must contain the desired value(s). This does not require us to specify the memory address itself, but we can leave the task of choosing a suitable memory address to the SMT solver. On a technical level, the values are constrained as byte-wise memory read operations relative to the address chosen by the solver.

Notably, the flexibility of our approach allows us to enforce arbitrary constraints between registers and memory locations. For instance, we could enforce that (1) certain register values must be odd, (2) the sum of registers must be equal to a specific value, or (3) the sum of two specific registers must be prime. To put it differently, our design allows to constraint exotic, target-specific conditions that may be useful in some exploitation scenarios.

3.4 Formula Generation

Our final formula consists of three main components: preconditions, gadget chain, and postconditions. The preconditions describe the initial state, which is used as input for the chain's initial gadget. The chain contains the encoding of individual instructions, the data flow between instructions within a gadget, and the data flow between gadgets—in short, the complete semantics of the gadget chain. Finally, the postconditions define the state which should be reached after executing the gadget chain. Here, the attacker encodes the desired CPU state. We combine these three components with logical conjunctions to the formula:

$$formula := preconditions \land gadget_chain \land postconditions$$

We then pass this formula to an SMT solver that supports the combined quantifier-free theory of fixed-size bit vectors (registers) and arrays (memory), `QF_ABV` [28]. If the solver finds a satisfying assignment, it provides a model, i. e., concrete values for each relevant variable in the formula. For all variables of gadgets that are *not* relevant for the synthesized gadget chain, no values are assigned. As a consequence, the model describes not only the initial state (e. g., values on the stack) but register and memory values for each gadget in the chain; in other words, we receive sort of an instruction trace that includes the intermediate values for each variable in the chain. In a final step, we can extract the initial values for each controlled buffer and use them as exploitation payload. When the payload is inserted, the gadget chain is executed as described in the model. Because a satisfying assignment produced by an SMT solver is a proof of existence, the gadget chain is guaranteed to reach exactly the specified postconditions. This is in strong contrast to state-of-the-art approaches, which often use heuristics rather than proofs to construct a gadget chain.

3.5 Algorithm Configuration

A few parameters define the performance of our approach, most of which affect the SMT solver: (1) For larger numbers of gadgets, the SMT solver needs more time in its decision process. To reduce its runtime, we can sample a small subset of gadgets (e. g., 300 gadgets as determined in empirical tests). (2) Due to our logical encoding, the chain length must be defined beforehand. While this may appear inflexible, our evaluation shows that testing different chain lengths is feasible in practice; if a shorter chain is possible, the SMT solver places semantic no-operations as padding gadgets in the chain. (3) To avoid excessive runtimes, we define upper time limits for the initial gadget extraction as well as for the SMT solver. While limiting the initial gadget extraction may reduce the number of available gadgets, this has no major impact if we only sample a subset.

4 Implementation

To demonstrate the practical feasibility of our proposed approach, we implemented a prototype of `SGC` in roughly 5,000 lines of Python code (see https://github.com/

RUB-SysSec/gadget_synthesis). While SGC's initial gadget extraction is based on Binary Ninja [34] (version 2.3.2660), all further steps are built on top of Miasm [5] (commit 218492cd). Especially the logical encoding of gadgets is facilitated in Miasm's IR. We extended its internal memory model to be stateful. The logic formula generated in the encoding step is then passed to the SMT solver Boolector [18], which is particularly suited to solve problems within the domain theory of bit vectors and arrays [36]. As Boolector supports the const-array extension [31], we use it to model memory and initialize it with a default value of 0. As memory accesses should not happen in read/write-restricted regions, we allow the user to specify which memory addresses may be accessed. In general, the user can add any constraint they need, such as excluding specific bytes from the chain (so-called *bad bytes*).

5 Evaluation

Based on the prototype implementation of SGC, we answer the following questions:

1. Is SGC capable of automatically finding valid gadget chains in diverse exploitation scenarios? How does it compare to state-of-the-art tools?
2. How does SGC perform in real-world exploitation scenarios?
3. How flexible and target-specific are SGC's chains in comparison to other approaches?
4. In what regard do SGC's generated gadget chains differ from the ones found by state-of-the-art tools?

To answer these research questions, we conduct the following experiments.

5.1 Setup

All our experiments were performed using Intel Xeon Gold 6230R CPUs at 2.10 GHz with 52 cores and 188 GiB RAM, running Ubuntu 20.04 on x86-64. To facilitate a deterministic analysis, we disable ASLR. Even if present, we only require an attacker to leak the base address, e. g., via an information leak, which is a weaker requirement than other approaches make [11, 35].

We compare SGC against the state of the art discussed in Section 2. While these tools work deterministically and take all gadgets into account, SGC does not: To keep the runtime of the SMT solver manageable, a subset of gadgets is randomly sampled for a provided seed. As a consequence, the sampled gadgets may be insufficient to fulfill the attacker's goals. To mitigate this problem, SGC uses by default ten different seeds, running them in parallel and reporting the first chain found. To add further variety, SGC attempts to find a chain of length 3 and 5, both for 100 and 300 gadgets, while not using more than 128 bytes of the attacker-controlled buffer. These values have been empirically chosen (cf. Section 5.6) In summary, 40 configurations are executed in parallel. For our evaluation, we run all configurations until completion for later analysis instead of returning the first gadget chain found. As all other tools operate deterministically,

we only run them once. We emphasize that all tools are provided equal resources, i. e., CPU cores and RAM. While we restrict `SGC` to one hour for disassembly and the SMT solver, we define a timeout of 24 hours for all other tools. To verify whether a generated chain is *valid*, we use `GDB` to place it in the attacker-controlled buffer within the program and then execute the chain. This way, we ensure that the gadget chain works in practice.

As targets, we use a diverse set of programs. In a first step, we replicate the experiments of Follner et al. [8] on recent versions of `chromium` (version 88.0.4324.182), `apache2` (version 2.4.46), `nginx` (version 1.19.9), and `OpenSSL` (version 1.1.1f). All of these targets are dynamically linked and we configure `SGC` to ignore shared libraries, simulating a scenario where only the base address of the main executable is known but no locations of libraries. To cover scenarios where `libc` is present, we create an empty wrapper program that is statically linked against `glibc` version 2.31. To evaluate whether `SGC` can be used to exploit real-world vulnerabilities, we use `dnsmasq` (version 2.77).

5.2 Finding a Chain

Based on the experiments by Follner et al. [8], we evaluate whether `SGC` is capable of finding valid gadget chains. While a multitude of possible attacker goals exists, in reality, attackers mostly aim at either calling library functions such as `mprotect` (to change the protection flags of memory regions) and `mmap` (to map a RWX page in which their shellcode can be placed), or at executing system calls, such as `execve` with the parameter `/bin/sh` that spawns a shell. Therefore, we pick three exemplary attacker goals, namely (1) a library call to `mprotect(addr, len, prot)` with three parameters, (2) a library call to `mmap(addr, length, prot, flags, fd, offset)` with six parameters, and (3) a system call to `execve(path, argv, envp)` with four parameters (one being the system call number) and the requirement to place a string in memory. On the `x86-64` architecture, these arguments are passed via registers [16]. As parameters, we use fixed exemplary values that are common in real-world exploitation scenarios, such as `execve(&"/bin/sh", 0, 0)` to spawn a shell or setting `prot` in `mprotect` to `RWX`, such that an attacker could place and execute arbitrary shellcode. To compare the tools, we run each of them in the same configuration, analyze whether it finds a chain, and check—based on our verification tooling—if the chain is *valid* in practice. Table 2 depicts the results of this experiment. As `ROPgadget` only provides fixed heuristics for `execve`, we exclude it from the other attacker goals. Similarly, `Ropper` is limited to `mprotect` and `execve`, and `P-SHAPE` focuses on library calls.

Most tools find a chain for `mprotect`, which is the easiest goal since only three registers have to be set. `angrop` struggled both with `chromium` and `OpenSSL` and crashed during the attempt to locate gadget chains. Likewise, `P-SHAPE` crashed for `chromium`. Although `P-SHAPE` found a chain for four targets, none of them were valid in real-world scenarios: Manual verification revealed that they cause segmentation faults (e. g., due to write attempts to inaccessible memory regions). For `mprotect`, only `SGC` identifies a valid gadget chain for all targets.

Table 2: Capability of finding a valid gadget chain to call `mprotect`, `mmap`, or `execve`. Legend: ✓ = valid chain, (✓) = chain found but crashes program, ✗ = no chain found, [1]) = chain found when increasing timeout to 5h, [2]) = `SGC` *proves* that no chain exists.

		SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
mprotect	chromium	✓	✗	✗	✓	-	✗
	apache2	✓	(✓)	✓	✓	-	(✓)
	nginx	✓	(✓)	✓	✓	-	✗
	OpenSSL	✓	(✓)	✗	✗	-	✗
	libc	✓	(✓)	✓	✓	-	✓
mmap	chromium	✓[1]	✗	✗	✓	-	-
	apache2	✓	✗	✗	✓	-	-
	nginx	✓	(✓)	✗	✗	-	-
	OpenSSL	✗[2]	✗	✗	✗	-	-
	libc	✓	(✓)	✗	✓	-	-
execve	chromium	✓	-	✗	✓	✓	✗
	apache2	✓	-	(✓)	✓	✗	(✓)
	nginx	✓	-	(✓)	✓	✗	✗
	OpenSSL	✓	-	✗	✗	✗	✗
	libc	✓	-	✓	✓	✓	✓

In comparison to `mprotect`, finding a chain for `mmap` is significantly more challenging since six register arguments have to be set, and thus more suitable gadgets are required. While all chains found by `P-SHAPE` crashed again, `ROPium` produced valid chains for three targets. However, this was only possible after we fixed a bug in its source code. `SGC` found four out of five valid chains. For `chromium`, we had to increase the timeout for disassembly and solving to 5h, since we initially did not find suitable gadgets to set `r8` and `r9`, the fifth and sixth argument to `mmap`. We discuss the shortcomings of our disassembly and random sampling in more detail in Section 6. For `OpenSSL`, no tool was able to produce a chain. To get more insights, we performed another experiment in which `SGC` was given access to all 3045 available `OpenSSL` gadgets (instead of choosing a random subset). After 226s, the SMT solver returned `UNSAT`, which can be understood as proof of non-existence. In other words, `SGC` was able to assert that no chain for the provided gadgets exists that fulfills the postconditions. This saves the user valuable time as they are guaranteed that even manual analysis will be fruitless.

The last attacker goal, `execve`, models the common scenario where a shell is spawned via a system call. It differs from the previous goals in the fact that not only four register values must be prepared, but the string `/bin/sh\x00` must be placed in memory. To express this behavior in `ROPium`, the user has to manually set a suitable memory address at which the string should be placed in memory. As such, the gadget chain construction is not completely automated. However, we include it since it is the only tool besides `SGC` that succeeds in finding valid chains for almost all targets.

In summary, these experiments answer research question 1: `SGC` outperforms all state-of-the-art approaches and manages to find valid gadget chains for all targets, even when other tools fail. For the only case where it did not find a chain, it even provided formal proof that no chain for the available gadgets can exist.

5.3 Real-World Applicability

To answer research question 2, we are interested in whether `SGC` proves helpful towards finding gadget chains in real-world exploitation contexts. To this end, we conduct a case study for CVE-2017-14493 [25], which describes a stack-based buffer overflow in `dnsmasq` (up to version 2.77) [12]. In essence, an attacker can craft a malicious DHCPv6 packet that, when received by `dnsmasq`'s DHCP server, triggers an overflow in the `dhcp6_maybe_relay` function, where the length and data of a `memcpy` can be controlled by the attacker. This bug allows for the injection of gadget chains of arbitrary length; if ASLR is present, an attacker can exploit an information leak in the same version, assigned CVE-2017-14494, to leak the base address [25]. For simplicity, we assume ASLR is already bypassed.

Our goal is to craft a gadget chain that calls `execve(&"/bin/sh", 0, 0)` to spawn a shell. Following the System V AMD64 ABI calling convention [16], register `rax` needs to hold the `execve` system call number (`0x3b`), while the registers `rdi`, `rsi`, and `rdx` pass the arguments to `execve`. Therefore, we set the postconditions accordingly. To define the preconditions, we have to inspect the program state at the time when the attacker can divert execution flow to the gadget chain. In detail, we dump the CPU state with `GDB` and constraint register values accordingly. After defining preconditions and postconditions, we logically encode the gadget chain and query the SMT solver with the formula. `SGC` finds a gadget chain after approximately 8m. A shell is spawned after embedding the gadget chain in a DHCPv6 packet and sending it to `dnsmasq` For a detailed explanation of the bug and chain found by `SGC`, we refer to Appendix B. To conclude research question 2, `SGC` assists in real-world exploitation scenarios. It only requires the initial CPU state as preconditions and the desired target state.

5.4 Target-Specific Constraints

To answer research question 3 that addresses the flexibility of our approach, we conduct two experiments that model different exploitation scenarios. In the first experiment, we aim at crafting chains that do not include so-called *bad bytes*. Such bytes cannot be used in an exploit payload since they act as terminators in the underlying program (e. g., `\x00` in C strings). We can avoid using such bytes in our payload by adding the constraint that each byte in the attacker-controlled buffer must be different from specific byte values. In this experiment, we try to craft valid gadget chains that call `mprotect`, `mmap`, and `execve` in the statically-linked `libc` wrapper, where `\x0a` and `\x0b` are considered as bad bytes. `SGC` produced a valid gadget chain within, on average, 512s; similarly, all other tools (excluding `P-SHAPE`, which does not support bad bytes) were able to produce gadget chains. This is not surprising, as avoiding bad bytes is a common requirement for many

Table 3: Statistics over all valid chains generated during experiments in Section 5.2.

		SGC	P-SHAPE	angrop	ROPium	ROPgadget	Ropper
avg. instructions		5.9	-	2.9	2.4	2.0	2.6
gadgets w/ mem. write		9%	-	7%	6%	3%	14%
└ excluding `execve`		9%	-	0%	0%	-	0%
gadgets w/ mem. reads		30%	-	7%	0%	0%	0%
└ excluding `execve`		32%	-	0%	0%	-	0%
CF types	`ret`	68%	-	100%	97%	100%	100%
	`call MEM`	10%	-	0%	0%	0%	0%
	`call REG`	20%	-	0%	3%	0%	0%
	`jmp REG`	2%	-	0%	0%	0%	0%

exploits and most tools consider this in their heuristics. Then, we slightly modify this experiment: We include one of the parameter values passed to the functions as a bad byte (essentially prohibiting the tools from using this specific value directly), such that the tools must construct the value indirectly via the gadget chain. In this scenario, only `ROPium` and `SGC` manage to find valid gadget chains. This shows that even a standard feature can be problematic for heuristics-based tools.

In the second experiment, we add a more complex constraint: We require that the sum of all values (quadwords) in the attacker-controlled buffer (where the addresses and data for the gadget chain are placed) must be equal to the value `0xdeadbeef`. While this constraint seems artificial, similar constraints can be found in commercial DRM systems that perform integrity checks over specific memory regions. While no other tool provides the flexibility to model this behavior, we can enforce this within a few lines of code in `SGC` and produce valid gadget chains for the same setup as before (within, on average, 527s).

Overall, we conclude that `SGC` provides great flexibility and allows to model complex constraints. Thus, it covers even unusual exploitation scenarios.

5.5 Chain Statistics

To answer research question 4, in what regard differ our gadget chains from the ones found by state-of-the-art approaches, we inspect which types of gadgets and instructions are used in the generated chains. To this end, we analyze each valid chain found during our experiment in Section 5.2. Since `P-SHAPE` found only invalid chains that crashed the program, we exclude it from this experiment.

As visible in Table 3, `SGC`'s gadgets contain on average almost six instructions, whereas the other tools use two to three instructions per gadget. Further, `SGC` is the only approach that makes use of explicit memory reads and writes (excluding instructions such as `push` and `pop`); all other tools only use it in the case of `execve` to place the string `/bin/sh` into the memory. Similarly, most of the tools rely exclusively on return-oriented gadgets; only `ROPium` uses call-oriented programming for 3% of its gadgets. Contrary, `SGC` only uses return-oriented

Table 4: SGC's timings for initial disassembly and chaining.

	Disassembly	Chaining	Total
mprotect	1845s	363s	2207s
mmap	1617s	2667s	4284s
execve	1845s	494s	2338s

programming in 68% of the cases, while it deploys call and jump-oriented gadgets in 32%. In summary, SGC has on average longer gadgets, uses more memory reads/writes, and has a significantly higher amount of non-return-oriented gadgets; in short, it includes gadgets specific to the target with side effects that are disregarded by other approaches due to their generic heuristics.

Another relevant aspect is SGC's runtime (cf. Table 4). The disassembly step is comparably slow; the time required for instruction lifting, encoding, and SMT solving is significantly lower. Our disassembly relies on a combination of Binary Ninja and Miasm: we first analyze the whole binary and disassemble then individual functions in Miasm. As it is not a focus of this work, we consider improving our disassembly component as future work. Only for mmap, finding the chain takes significantly more time since the SMT solver has to find a valid chain that prepares six function arguments. For reference, the other tools find a chain on average within 319s. However, this ignores the runtime when they found no chain (e. g., Ropper hit the timeout of 24h twice), which was often the case, especially for mmap. In summary, SGC manages to find a valid chain within minutes.

5.6 SGC's Configuration

After successfully answering all research questions, we would like to give a better intuition of the configuration parameters relevant for SGC. As described before, our approach is probabilistic: it randomly samples only a small subset of gadgets. As a result, the chosen subset may not be sufficient to generate a chain that fulfills the postconditions. We can select another subset of the same size or a larger number of gadgets to overcome this. The latter, however, increases the time required by the SMT solver to decide the chain synthesis problem. To get a better feeling for this trade-off, we vary the chain length and number of sampled gadgets and analyze how often the solver succeeds in deciding the synthesis problem, i. e., it finds a chain or returns UNSAT within one hour. For each configuration, we run the solver ten times with different seeds such that diverse gadgets are sampled. We do this for all target programs from Section 5.2 and count how often the solver finds an answer or timeouts in the process of finding chains for mprotect. In total, we perform 50 independent runs (ten different seeds for five different targets) for each configuration.

As Table 5 shows, the chain length and the number of gadgets determine the SMT solver's performance: For a small number of gadgets and chain length of 1, the solver always finds an answer. However, for longer chains or more

Table 5: Number of gadget chains the solver decided (i. e., considered `SAT` or `UNSAT`) vs. timeouts when building a chain to `mprotect` for the targets in Section 5.2 with ten different seeds each. Format is `#Decided by SMT solver`/`#Timeout`. We color the prevalent outcome.

	Chain Length							
#Gadgets	1	2	3	4	5	6	7	8
100	50/ 0	50/ 0	49/ 1	31/ 19	24/ 26	16/ 34	15/ 35	12/ 38
300	50/ 0	50/ 0	37/ 13	20/ 30	13/ 37	10/ 40	7/ 43	6/ 44
500	50/ 0	44/ 6	31/ 19	16/ 34	10/ 40	8/ 42	5/ 45	4/ 46
1000	50/ 0	31/ 19	25/ 25	11/ 39	9/ 41	2/ 48	0/ 50	0/ 50

sampled gadgets, the number of timeouts increases. While the solver can decide some chains of length six or higher, it increasingly triggers the timeout of one hour. Similarly, for a larger gadget pool (e. g., 1000 gadgets), the solver already struggles for chains of length three. While the strategy of randomly sampling a small number of gadgets proved effective, an attacker can always increase the number of gadgets and set higher timeouts for the SMT solver.

6 Discussion

Limitations of `SGC`. While `SGC` has proven overall effective, various aspects can be improved: (1) Our currently used disassembly is naive since we only consider regular instruction offsets. As an improvement, we can search unaligned gadgets since any sequence of bytes can be interpreted as instructions on `x86-64`. (2) The SMT solver is the most significant performance bottleneck of our design as it may require a large amount of time to identify valid gadget chains. However, as our evaluation shows, randomly selecting a subset of gadgets provides an effective strategy to reduce `SGC`'s runtime. In this scenario, an `UNSAT` provided by the SMT solver is *not* a formal proof that no gadget chain exists, as it only proves that no chain for the selected subset of gadgets exists.

Mitigations. To prevent exploitation, various mitigations have been proposed. (1) `W^X` prevents execution of injected code, however, it is ineffective against *code reuse* attacks and thus `SGC`. (2) Address space layout randomization (ASLR) shuffles the program's memory layout such that an attacker cannot rely on addresses. `SGC` requires only the base address of the code section and does not require shared libraries to find valid gadget chains, thus a single information leak suffices. (3) Lastly, control-flow integrity (CFI) prevents the redirection of control flow to arbitrary code locations. This severely hampers code-reuse attacks such as `SGC` because only specific gadgets can be chained together. However, related work has shown that even fine-grained CFI is insufficient to prevent code-reuse attacks in general [11, 24]. We believe that an attacker could add constraints modeling the enforcement policies such that the SMT solver will only select gadget chains that pass the CFI enforcement policy. We leave this as interesting future work.

7 Related Work

After initial techniques in the domain of code-reuse focused on functions from `libc` [30], the concept was generalized to re-use small snippets of existing code [14, 26]. These small snippets are often chained via `ret` instructions (ROP) [26], but other control-flow transfers work as well (JOP [3,6] and COP [4,9]). Mitigations such as ASLR have been shown to be insufficient [29]. Moving forward with new mitigations such as control-flow integrity (CFI) [1], even more advanced approaches have been proposed, e. g., counterfeit object-oriented programming (COOP) [22] or data-oriented programming (DOP) [10]. Even fine-grained CFI solutions fail to stop attackers from finding gadget chains [35].

In parallel, various techniques to automate the cumbersome task of identifying suitable gadgets have been proposed. Early approaches use pattern matching to search for desired gadgets [13, 19]. Other approaches tackle the task of automating the attack itself: One of the earliest approaches, `Q` [23], uses software verification methods instead of pattern matching to achieve this goal. Using identification and chaining of gadgets similar to `Q`, Wollgast et al. [37] automate COP, which allows them to bypass coarse-grained CFI implementations. Tackling the problem imposed by fine-grained CFI solutions, Ispoglou et al. [11] propose an approach, `BOPC`, which automates data-only attacks. Further improving this avenue, Schwartz et al. [24] propose a generic approach, `Limbo`, capable of constructing chains using ROP, JOP, COP, or DOP. Their approach is similar to ours in the spirit of maintaining a generic approach to code-reuse attacks. However, their focus is on the construction of CFI-compatible gadget chains. Internally, their search relies on concolic execution and hard-coded heuristics. In contrast, our approach does not tackle the problem of identifying CFI-aware gadgets but maintains generality without relying on hard-coded heuristics. Further, `Limbo` only works for 32-bit Linux executables, which limits their real-world applicability. As no code is published, we cannot evaluate against `Limbo`.

8 Conclusion

In this paper, we presented a generic and flexible approach to automate the task of finding gadget chains. With our prototype implementation, we have shown that `SGC` outperforms state-of-the-art tools. It not only finds gadget chains where all other approaches fail but also allows to model complex constraints.

Acknowledgements This work was supported by the German Research Foundation (DFG) within the framework of the Excellence Strategy of the Federal Government and the States—EXC 2092 CaSa—39078197.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-Flow Integrity Principles, Implementations, and Applications. ACM Transactions on Information and System Security (TISSEC) **13**(1) (2009)

2. angr team: angrop. https://github.com/angr/angrop
3. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-Oriented Programming: A New Class of Code-Reuse Attack. In: ACM Conference on Computer and Communications Security (CCS) (2011)
4. Carlini, N., Wagner, D.: ROP is Still Dangerous: Breaking Modern Defenses. In: USENIX Security Symposium (2014)
5. CEA IT Security: Miasm – Reverse Engineering Framework. https://github.com/cea-sec/miasm
6. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-Oriented Programming Without Returns. In: ACM Conference on Computer and Communications Security (CCS) (2010)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An Efficient Method of Computing Static Single Assignment Form. In: ACM Symposium on Principles of Programming Languages (POPL) (1989)
8. Follner, A., Bartel, A., Peng, H., Chang, Y.C., Ispoglou, K., Payer, M., Bodden, E.: PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution. In: Security and Trust Management Workshop (2016)
9. Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of Control: Overcoming Control-Flow Integrity. In: IEEE Symposium on Security and Privacy (2014)
10. Hu, H., Shinde, S., Adrian, S., Chua, Z.L., Saxena, P., Liang, Z.: Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In: IEEE Symposium on Security and Privacy (2016)
11. Ispoglou, K.K., AlBassam, B., Jaeger, T., Payer, M.: Block-Oriented Programming: Automating Data-only Attacks. In: ACM Conference on Computer and Communications Security (CCS) (2018)
12. Kelley, S.: dnsmasq. https://thekelleys.org.uk/dnsmasq/doc.html
13. Kornau, T.: Return-Oriented Programming for the ARM Architecture. Master's thesis, Ruhr-Universität Bochum (2010)
14. Krahmer, S.: x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique (2005)
15. Kroening, D., Strichman, O.: Decision Procedures. Springer (2016)
16. Matz, M., Hubicka, J., Jaeger, A., Mitchell, M.: System V Application Binary Interface. AMD64 Architecture Processor Supplement, Draft v0 **99** (2013)
17. Milanov, B.: ROPium. https://github.com/Boyan-MILANOV/ropium
18. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 53–58 (2014)
19. Roemer, R.G.: Finding the Bad in Good Code: Automated Return-Oriented Programming Exploit Discovery. Master's thesis, UC San Diego (2009)
20. Salwan, J.: ROPgadget. https://github.com/JonathanSalwan/ROPgadget
21. Schirra, S.: Ropper. https://github.com/sashs/Ropper
22. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In: 2015 IEEE Symposium on Security and Privacy. pp. 745–762. IEEE (2015)
23. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit Hardening Made Easy. In: USENIX Security Symposium (2011)
24. Schwartz, E.J., Cohen, C.F., Gennari, J.S., Schwartz, S.M.: A Generic Technique for Automatically Finding Defense-Aware Code Reuse Attacks. In: ACM Conference on Computer and Communications Security (CCS) (2020)

25. Serna, F.J., Linton, M., Stadmeyer, K.: dnsmasq stack-based buffer overflow (CVE-2017-14493). `https://security.googleblog.com/2017/10/behind-masq-yet-more-dns-and-dhcp.html`
26. Shacham, H.: The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In: ACM Conference on Computer and Communications Security (CCS) (2007)
27. Sinz, C., Falke, S., Merz, F.: A Precise Memory Model for Low-level Bounded Model Checking. In: International Conference on Systems Software Verification (2010)
28. SMT-LIB: Logics. `https://smtlib.cs.uiowa.edu/logics-all.shtml#QF_ABV`
29. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In: IEEE Symposium on Security and Privacy (2013)
30. Solar Designer: Return-to-Libc (1997)
31. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: IEEE Symposium on Logic in Computer Science (2001)
32. Szekeres, L., Payer, M., Wei, T., Song, D.: Sok: Eternal war in memory. In: IEEE Symposium on Security and Privacy (2013)
33. Vanegue, J., Heelan, S., Rolles, R.: SMT Solvers in Software Security. In: USENIX Workshop on Offensive Technologies (WOOT) (2012)
34. Vector 35 Inc.: Binary Ninja. `https://binary.ninja/`
35. van der Veen, V., Andriesse, D., Stamatogiannakis, M., Chen, X., Bos, H., Giuffrdia, C.: The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In: ACM Conference on Computer and Communications Security (CCS) (2017)
36. Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A., Reger, G.: The SMT competition 2015-2018. J. Satisf. Boolean Model. Comput. **11**(1) (2019)
37. Wollgast, P., Gawlik, R., Garmany, B., Kollenda, B., Holz, T.: Automated Multi-architectural Discovery of CFI-resistant Code Gadgets. In: European Symposium on Research in Computer Security (ESORICS) (2016)

A Memory Modeling

Byte-wise memory reads and writes are modeled using single *select* and *store* operators, respectively. Larger reads are modeled by concatenating multiple *select* expressions, which we define recursively in terms of smaller read operations. Reads smaller than 64-bit into a 64-bit register are zero-extended by using *concat* with the zero bit vector bv_0. Larger writes are similarly modeled using the composition of multiple *store* expressions. Table 6 and 7 show memory accesses of various sizes. Given an array m, address k and value v and bit size $n \in (8, 16, 32, 64)$, we use the names $mem_read_n(m, k)$ and $mem_write_n(m, k, v)$ to substitute the longer SMT expressions from these tables.

B `dnsmasq` CVE-2017-14493

In the following, we analyze the `dnsmasq` bug in more detail. The stack-based buffer overflow in `dnsmasq` is caused by the absence of a length check of the data copied to a static buffer on the stack. Figure 2 shows the vulnerable call to

Table 6: Encoding of memory reads of various sizes, returning a value from memory m at address k.

Name	SMT encoding
$mem_read_8(m,k)$	$select(m,k)$
$mem_read_{16}(m,k)$	$concat(mem_read_8(m,k), mem_read_8(m,k+1))$
$mem_read_{32}(m,k)$	$concat(mem_read_{16}(m,k), mem_read_{16}(m,k+2))$
$mem_read_{64}(m,k)$	$concat(mem_read_{32}(m,k), mem_read_{32}(m,k+4))$

Table 7: Encoding of memory writes of various sizes, returning a memory with value v at address k.

Name	SMT encoding
$mem_write_8(m,k,v)$	$store(m,k,v_{0:7})$
$mem_write_{16}(m,k,v)$	$mem_write_8(mem_write_8(m,k,v_{0:7}), k+1, v_{8:15})$
$mem_write_{32}(m,k,v)$	$mem_write_{16}(mem_write_{16}(m,k,v_{0:15}), k+2, v_{16:31})$
$mem_write_{64}(m,k,v)$	$mem_write_{32}(mem_write_{32}(m,k,v_{0:31}), k+4, v_{32:63})$

`memcpy` in function `dhcp6_maybe_relay`. Sending a malicious DHCPv6 packet allows an attacker to gain control over the instruction pointer by overflowing the `mac` buffer of static size `DHCP_CHADDR_MAX` (16) in the `state` structure present on the stack.

```
206    /* RFC-6939 */
207    if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
208      {
209        state->mac_type = opt6_uint(opt, 0, 2);
210        state->mac_len = opt6_len(opt) - 2;
211        memcpy(&state->mac[0], opt6_ptr(opt, 2), state->mac_len);
212      }
```

Fig. 2: Vulnerable `memcpy` in file `rfc3315.c` triggering the overflow of the `mac` buffer in struct `state`.

The proof-of-concept (PoC) provided alongside the bug report [25] builds up such a DHCPv6 packet containing an `OPTION6_CLIENT_MAC` option holding data of excessive length. While the PoC overwrites the instruction pointer with a dummy value, injecting an arbitrary amount of bytes is possible. As long as the stack is not exhausted, the packet's content is copied and remains untouched until the instruction pointer is overwritten.

In order to synthesize a gadget chain, the information needed to specify preconditions and postconditions is gathered by extracting the program state before hijacking the control flow through `GDB`. Table 8a shows the preconditions set for `dnsmasq`. The initial `ret` instruction, which redirects the control flow to the chain's first gadget (`gadget_0`), is specified by preconditioning `rip`. The

Table 8: Preconditions and postconditions used for `dnsmasq`. Registers not mentioned in the preconditions are free variables, i. e., registers an attacker controls and can set to an arbitrary value.

(a) Preconditions

Register	Value
rip	0x33dfb
rax	0x223
rcx	0x0
rdx	0x5a
rdi	0x22
r8	0x7fffffffe0e0
r9	0x0
r10	0x7fffffffbc50

(b) Postconditions

Register	Value
rip	0x461d0
rax	0x3b
rsi	0x0
rdx	0x0
rdi	&"/bin/sh"

stack pointer `rsp` points to the part of the controlled buffer, where the gadget chain will be copied. In the logical formula, this stack area is a free variable.

Since we want to execute a system call to `execve` to spawn a shell, the final register values which the gadget chain needs to reach are specified accordingly. Table 8b shows the postconditions in preparation for calling `execve(&"/bin/sh", 0, 0)`. Here, `rip` holds the address of a `syscall` instruction available in the program. Using the default configuration described in Section 5.1, `SGC` finds a gadget chain consisting of four gadgets within approximately $8m$. While most gadgets are straightforward, `gadget_3` (shown in Figure 3) writes a value to the stack outside the attacker-controlled buffer, a side effect that does not harm the chain. The arithmetic operations of the first four instructions do not change register `rax`' value of 0. In line 6, the `lea` instruction is used to add `0x5` to the value present in `rbp` = `0x55555559a1cb`. The resulting address, `0x55555559a1d0`, is a `syscall` instruction; the address is placed on the stack at address `0x7fffffffe240` present in register `rbx`. As this address is writable memory, no harm results from this side effect.

As mentioned earlier, the PoC crafts a rogue DHCPv6 packet. In order to construct the payload with our synthesized gadget chain, the length parameter is adjusted and the dummy value is replaced with the data of the gadget chain. Sending this packet to the `dnsmasq` DHCP server successfully spawns the shell.

```
1  0x55555558a009:
2      movzx     rax, ax
3      imul      rax, ax, 0x1DCB
4      shr       eax, 0x15
5      movzx     eax, ax
6      lea       rax, qword ptr [rax + rbp + 0x5]
7      mov       qword ptr [rbx], rax
8      pop       rbx
9      pop       rbp
10     pop       r12
11     ret
```

Fig. 3: `gadget_3` of the gadget chain used to spawn a shell in `dnsmasq`.