USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives

Nicolas Bailluet, *Univ Rennes, Inria, CNRS, IRISA;* Emmanuel Fleury, *Univ Bordeaux, CNRS, LaBRI;* Isabelle Puaut and Erven Rohou, *Univ Rennes, Inria, CNRS, IRISA*

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

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.

Nothing is Unreachable: Automated Synthesis of Robust Code-Reuse Gadget Chains for Arbitrary Exploitation Primitives

Nicolas Bailluet
Univ Rennes, Inria, CNRS, IRISA

Emmanuel Fleury
Univ Bordeaux, CNRS, LaBRI

Isabelle Puaut
Univ Rennes, Inria, CNRS, IRISA

Erven Rohou
Univ Rennes, Inria, CNRS, IRISA

Abstract

Automating gadget chaining is a challenge that has attracted significant attention since the introduction of code-reuse attacks. Influenced by the primitives offered by stack-overflow vulnerabilities, several approaches were proposed that required the attacker to control the stack. Since then, most proposed approaches have had strong requirements on the capabilities of the attacker. However, during the last decade, a plethora of new attack primitives have emerged – e.g. use-after-free, heap-overflow – often breaking the requirements of existing approaches – e.g. controlling the stack. This paper presents a new approach to synthesizing code-reuse gadget chains that supports *arbitrary exploitation primitives and layouts*. We thoroughly compare the performance of our approach to the state-of-the-art. We show its ability to outperform its competitors by supporting intricate exploitation primitives and layouts that other approaches cannot. Especially, we demonstrate its real-world applicability by synthesizing gadget chains for ten real-world vulnerabilities with diverse exploitation primitives that competing tools struggle with. Among them is our case study: CVE-2022-46152 – which targets a widely used trusted execution environment.

1 Introduction

Code-reuse attacks are techniques that allow an attacker to achieve malicious behaviors once having a way to hijack the control-flow of a program. Execution is diverted to use small pieces of code present in executable parts of the memory (code, libraries). Each time such fragment of code is executed, control is then handed to another fragment, chaining them together. Code reuse attacks have first been unveiled in 1997 by A. Peslyak as *ret-to-libc* [42]. In 2007, H. Shacham generalized it by introducing the concept of *return-oriented programming* (ROP) [49] based on injecting a fake call-stack to chain *gadgets* – pieces of executable code that end with a return instruction [6, 7, 33]. Since then, other works showed that a fake call-stack was not needed to conduct code-reuse attacks [9, 10]. The only requirements for an attacker are to control the *code-pointer* and a memory buffer – mainly to store the addresses of gadgets.

When aiming to achieve a precise goal, such as starting a program, a single gadget is not sufficient. In such a case, it becomes necessary to build a *gadget chain*, which is a sequence of gadgets executed consecutively. Finding such chains is a complex task, often stated as a reachability problem [2, 12, 46, 47]. Numerous approaches have been proposed [57] that mostly rely on *explicit path exploration* [2, 12, 48], which is often driven by hard-coded heuristics that limit the exploration. Recently, the advent of SMT solvers [14, 41] has prompted the adoption of symbolic execution [29] as a prominent exploration technique [2, 12, 35].

Mitigations like *Control-Flow Integrity* (CFI) were introduced to prevent code-reuse attacks. Research efforts were made in automating attacks when CFI is enforced – so-called *data-only attacks* [27, 28, 43, 48]. Yet, their threat models require *write-what-where* capabilities – a very strong primitive offered only by a few vulnerabilities. Additionally, even though CFI is getting in popularity, it is still not a standard in industry, and code-reuse attacks are still relevant – as demonstrated by recent exploits [31, 32, 52].

Yet, the ability of an attacker to chain gadgets highly relies on their capabilities. That is, which registers and memory regions are attacker-controllable, and which values the attacker can write in there – we will refer to this as the *exploitation layout*. An attacker that controls the stack can chain gadgets easily – in ROP fashion – whereas one that controls a heap-buffer cannot leverage the same return-ended gadgets.

Overall, existing gadget chaining approaches have strong requirements on the capabilities of the attacker in their threat models, and do not take into account the exploitation layout. Instead, they are mostly limited to generating a fake call-stack – which requires to overwrite the stack [2, 12, 35, 46, 47, 57]. Controlling the stack has been a very common primitive in early attacks due to *stack-overflow* vulnerabilities [42]. Nowadays, an attacker rarely has total control of the stack [36, 55]. In the last decade, we have seen a shift in the type of

vulnerabilities present in software [36, 55]. A plethora of new powerful exploitation primitives have emerged, often targeting the *heap* [36] – e.g. *heap-overflow, use-after-free* – and existing approaches are unable to handle their associated exploitation layouts.

In this paper, we propose an approach to gadget chaining based on *robust reachability* [22], mixed together with *component-based program synthesis* [23]. Compared to existing works, it adapts to any user-defined exploitation layout – whether it involves the stack, the heap, or anything – without requirements on the capabilities of the attacker or the exploitation primitive. We let the user define which regions are attacker-controllable, and which constraints apply on these data. Eventually, we tell if it is possible to fulfill an attacker-defined goal, and how to reach it – by outputting which values to write to the attacker-controllable regions only. We rely on SMT solving and leverage techniques inspired by *taint propagation* to ensure robust reachability. Related works for code-reuse and data-only attacks already made extensive use of SMT solving [2, 12, 43, 46, 48], and some also leveraged taint propagation [28, 56]. The novelty of our work rather lies in the combination of *robust reachability* and *program synthesis* techniques.

Contrary to standard reachability, *robust reachability* [22] takes into account that only some specific variables in the program are attacker-controlled. Therefore, it tells whether a point in the program can be reached, whatever the value of uncontrolled variables – the reachability is said to be robust against variations of uncontrolled variables.

Component-based program synthesis [23] is the problem of finding a sequence of so-called *components* that, chained together, form a program whose result always fits a semantic specification. We straightforwardly see the connection between gadget chaining and program synthesis: in our situation, the components are the gadgets.

The benefits of this mix are twofold. Firstly, it adapts to any user-defined exploitation layout via explicit handling of attacker-controlled values versus uncontrolled data. Secondly, like regular program synthesis [23], we fully delegate the arrangement of gadgets (components) to an SMT solver. All that we request is the goal to reach. As a result, free from heuristics, our approach produces intricate chains on its own, without us explicitly requesting for it – e.g. *stack-pivoting*, *jump-oriented chaining*, *predicated instructions*. Overall, here are our contributions:

Contribution 1. We introduce a formalization of code-reuse gadget chaining that takes inspiration from both *robust-reachability* [22] and *program synthesis* [23]. Using techniques inspired by *taint propagation*, we build one-shot SMT queries that fully delegate the gadget chaining to an SMT solver – no explicit path exploration is done. Compared to the state-of-the-art, our approach can adapt to any exploitation layout – no assumption is made on the attacker's capabilities or the exploitation primitive. Additionally, it is able to autonomously leverage complex chaining techniques, without an explicit request from our side.

Contribution 2. We implement our approach in a *proof-of-concept* tool named ARCANIST, supporting `x86`, `x64` and `Arm` architectures, and easily extensible to others. We evaluate its performance and compare it with open-source related works – Ropium [35], Angrop [2] and SGC [46]. We show that ARCANIST outperforms its competitors by synthesizing attacks for exploitation layouts that others do not support. Especially, we demonstrate its applicability on 10 real-world vulnerability cases – involving various non-trivial exploitation layouts and diverse attack primitives – that competitors are unable to exploit.

Contribution 3. We thoroughly study a real-world vulnerability (CVE-2022-46152) targeting OP-TEE – a widely used trusted execution environment for `Arm` architectures. We show that no state-of-the-art approach can model the case-specific complexities involved in its non-trivial exploitation layout. Eventually, we successfully demonstrate the ability of ARCANIST to synthesize attacks for CVE-2022-46152.

Paper organization

The rest of this paper is organized as follows. Section 2 first introduces some background information on the concepts used in the paper. Section 3 details the threat model we considered. Section 4 then introduces our motivations through the case-study of a real-world CVE. Sections 5 to 7 detail the proposed approach. Section 5 defines the gadget chain synthesis problem as a robust reachability problem. Section 6 presents how gadgets are coded into SMT formulas. Section 7 describes how the SMT solver is queried and introduces some optimizations for better scalability. Section 8 is devoted to an extensive experimental evaluation. Finally, we discuss the limits of our approach in Section 9, and survey related research in Section 10.

2 Background

Satisfiability Modulo Theory (SMT). SMT-solvers address the problem of determining the satisfiability of logical constraints with respect to background theories. Unlike traditional Boolean satisfiability (SAT) solvers, which solely supports Boolean constraints, SMT-solvers extend to several theories from various domains – e.g. arithmetic, bit-vectors, arrays, etc. These extensions make it a powerful tool to reason about program semantics. Especially, SMT finds applications in formal software verification, symbolic execution [29] and synthesis of correct-by-design programs [23].

Taint propagation. Taint propagation is a security analysis that allows tracking the data-flow inside a program. The term *taint* typically denotes data that originates from untrusted sources – e.g. user inputs. Taint propagation techniques aim

to identify how this tainted data spreads through a program's execution. By analyzing how taint flows through the program, it allows identifying which parts may be affected by untrusted data. Notably, taint propagation is widely used in software verification and vulnerability analysis [11].

3 Threat Model

In this paper, we consider an attacker that is able to rewrite the *code-pointer* and store some *controlled* data in memory. Our goal is to tell whether this attacker can reach a specific goal, starting from the moment they overwrite the *code-pointer*. Rewriting the *code-pointer* is usually done using a memory corruption vulnerability. However, discovering of such vulnerabilities is orthogonal to our work, and we do not aim to do automated end-to-end exploit generation (AEG) – this is discussed in Section 9. We consider – of course – that NX/DEP is activated. Like any code-reuse attack, if ASLR is enabled, the attacker must bypass it before running the attack – e.g. via an information leak vulnerability. Regarding *Control-Flow Integrity* mitigations, we consider they are not enforced – this is discussed in Section 9.

4 Motivations

In this section, we elaborate about our motivations. We compare the workflow of our synthesis procedure with our main competitor, SGC [46] – a state-of-the-art code-reuse chain synthesizer – and highlight the benefits of our approach. Finally, we illustrate our motivations through the study of a real-world vulnerability – CVE-2022-46152 [39].

4.1 Diversity of Exploitation Layouts

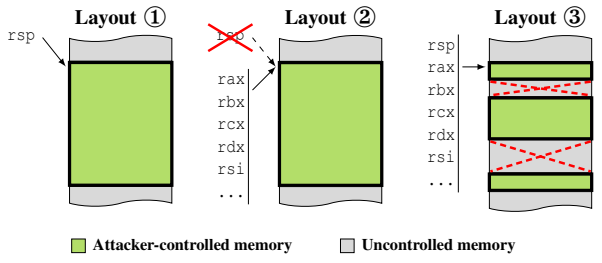


Figure 1: Various exploitation layouts – ① stack-based buffer, ② heap-based buffer & ③ sparse attacker-controlled values.

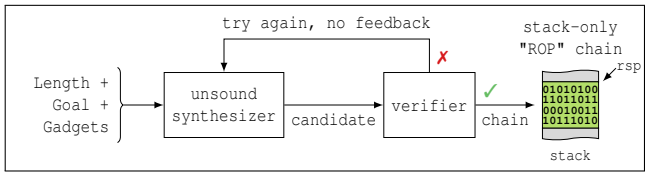
The ability of an attacker to chain gadgets depends on the exploitation layout. Our approach supports arbitrary layouts, but for the sake of clarity, and to illustrate our points, we focused on three kinds of layout as depicted in Fig. 1.

Layout ①. This layout models most stack-based vulnerabilities – e.g. *stack-overflows*. The attacker has control of a contiguous stack-buffer and `rsp` points to attacker-controlled data. This setup is well-suited to classical ROP attacks, gadgets can be sequentially chained via a fake call-stack using return-ended gadgets.

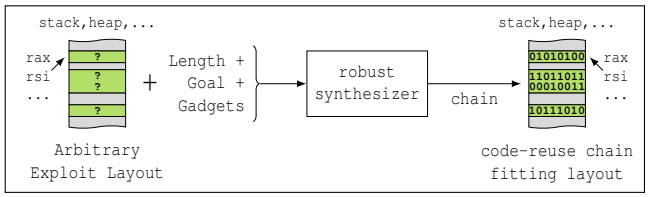
Layout ②. This layout models typical situations that involve the heap – e.g. *heap-overflows*, *use-after-free*. The attacker controls a buffer on the heap, and some registers – other than `rsp` – point to attacker-controlled data – preventing the use of return-ended gadgets. These registers are case-specific and depend on the vulnerability being exploited.

Layout ③. This layout models situations where the attacker has limited control over the memory. They can inject controlled data, but in very specific locations that are interspersed with uncontrollable data. For instance our case-study CVE-2022-46152 involves this kind of layout – see Section 4.3.

4.2 Better Adaptability & One-Shot Synthesis



(a) SGC: needs a verification loop & only supports hard-coded layout ①.



(b) ARCANIST: synthesizes attacks in one shot & adapts to any layout.

Figure 2: Synthesis workflow of (a) SGC and (b) ARCANIST.

Most existing approaches require to be in layout ① [2, 12, 35, 46, 47, 57], making them unable to support other layouts. Especially, our main state-of-the-art competitor (SGC) only supports layout ①. In Fig. 2, we compare the synthesis workflow of SGC with our own – referred to as ARCANIST.

In ARCANIST, we designed a synthesis procedure that takes into parameter the exact exploitation layout – provided by the user. Consequently, the synthesizer knows exactly which regions it can use, which register points to these regions, and how to load the appropriate gadgets. Thereby, it enables the support for layout ①, ② and ③, and for other *arbitrary layouts*.

Furthermore, SGC uses an *unsound* synthesis procedure. The chains generated by the synthesizer are sometimes invalid – e.g. they rely on uncontrollable values – and need a *post-synthesis* verification pass. During our experiments in Section 8, we observed that almost 30% of SGC candidates

were rejected by the verifier. On the contrary, with ARCANIST, we propose a *sound* synthesis procedure based on *robust reachability* – see Section 5 – that allows synthesizing chains that are guaranteed to be valid in one shot.

4.3 Real-World Case-Study

In this section, we introduce an example of real-world vulnerability – CVE-2022-46152 – leading to an intricate layout of type ③ – not supported by competitors. Its unique exploitation primitive does not fulfill the usual requirements of existing chain synthesizers, making it a good example to illustrate our points.

OP-TEE [30] is a framework for developing trusted applications – for Arm processors. Prior to version 3.18, OP-TEE's kernel `smc` command handler was vulnerable to an improper validation of array indices – CVE-2022-46152 [15, 39]. An attacker can use this CVE to trick OP-TEE into treating an attacker-controlled value as a pointer to an internal object – so-called `mobj`. The `mobj` structures are reference-counted, and the attacker can prompt OP-TEE to decrement the reference count of the fake `mobj` – thereby allowing to decrement values at arbitrary memory locations. We identified the stack of the `smc` command handler as a good candidate region to target in order to compromise the control-flow.

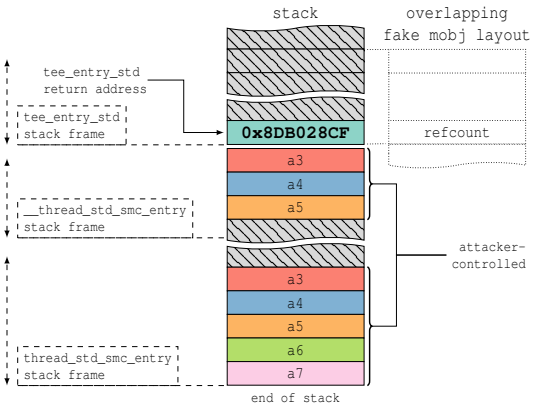


Figure 3: Stack layout of OP-TEE's `smc` command handler when exploiting CVE-2022-46152. Identical values have the same color. Grayed-out and hatched means uncontrolled.

The associated memory layout is given in Fig. 3. In the middle is represented the stack layout. Since the attacker controls the arguments passed to the `smc` command handler, they can control multiple values pushed on the stack, we denote them `a3` to `a7`. On the left, we give the names of the functions called by the `smc` command handler – just for reference. On the right we represented the layout of a fake `mobj` whose reference count overlaps the return address of `tee_entry_std` on the stack. An attacker can repeatedly exploit the vulnerability to decrement the return address as much as they want. Doing so, they could take over the execution of the program when `tee_entry_std` returns. However, they cannot decrement it further when it reaches zero because it is treated as a reference count. Hence, they can only replace it with values less than the original one – i.e. less than `0x8DB028CF`.

This situation fits layout ③. The attacker's control of the stack is limited to a few values in memory that are mostly bound to each others. Furthermore, uncontrolled values interpose between attacker-controlled arguments.

5 Robust Gadget Chain Synthesis

We aim to determine if there exists a sequence of gadgets that, starting from an initial state of the program – when the attacker gains control over the execution – leads to a state that fits the attacker's goal. However, we also need the chain to fit in the specific exploitation layout given by the attacker. Furthermore, the chain must remain valid *for any instance* of the unpredictable values in the layout. We achieve this by stating the gadget chain synthesis as a mix of *robust reachability* [22] and *component-based program synthesis* [23].

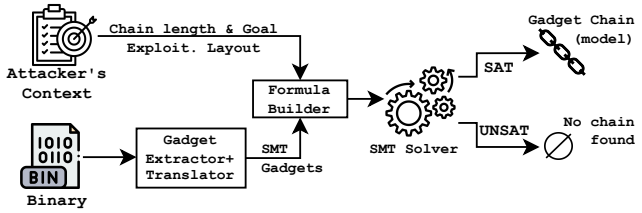


Figure 4: High-level overview of our approach.

An overview of our approach is depicted in Fig. 4. We solve the problem by reducing it to an SMT problem. We fully delegate the chaining of gadgets to the solver. If the solver finds a solution, it indicates exactly which values must be written in attacker-controlled areas to properly execute the synthesized chain.

5.1 Notations & Modeling

In order to accurately model and capture the exact semantics of gadgets, we use the quantifier-free theory of fixed-size bit-vectors and arrays of bit-vectors – a.k.a. `QF_ABV` [53].

States. The *bit-vectors* theory allows modeling the CPU registers and the fundamental operations performed during program execution – such as arithmetic, bitwise and shift operations – while the theory of *arrays of bit-vectors* enables the modeling of memory. We represent a state as a tuple of SMT variables: $s = \langle reg_0, \ldots, reg_n, Mem \rangle$. Each reg_i is a bit-vector representing a CPU register, and Mem is an array of 8-bit bit-vectors representing the memory [50].

We note s_0 and call *initial state* the state from which the attacker takes control of the execution flow of the program. Since the attacker has limited control over the initial state,

some variables may have unpredictable values at runtime. Consequently, we split the initial state in two: $s_0 = \langle \vec{p}, \vec{u} \rangle$. We call *predictable variables* and note $\vec{p}$ the list of variables whose value is controlled by the attacker or predictable at runtime. We call *unpredictable variables* and note $\vec{u}$ the list of variables whose value is unpredictable at runtime.

For instance, if the attacker has control over a 16-byte buffer in memory, only those 16 bytes in *Mem* would be put in $\vec{p}$, the rest would be put in $\vec{u}$.

Transitions. We consider gadgets as the sole means of transitioning between states. We represent a gadget g as a `QF_ABV` formula: $g(s,s') \triangleq g_{pre}(s) \wedge g_{post}(s,s')$, with:

$$g_{pre}(s) \iff s \text{ satisfies the pre-conditions to execute } g$$
$$g_{post}(s,s') \iff s \text{ transitions to } s' \text{ by executing } g$$

We detail the SMT encoding of g_{pre} and g_{post} in Section 6. We also define $s_0 \rightarrow^n s_n$, meaning that s_0 leads to s_n after executing n gadgets:

$$s_0 \rightarrow^n s_n \triangleq \exists s_1 \dots s_{n-1}. \exists g_0 \dots g_{n-1}. \bigwedge_{i=0}^{n-1} g_i(s_i, s_{i+1})$$

Predicates. Let $\vec{v}$ be a list of SMT variables and $\vec{\mathrm{v}}$ be concrete values for $\vec{v}$. We call $\vec{\mathrm{v}}$ an assignment of $\vec{v}$. Let $\psi(\vec{v})$ be a predicate on $\vec{v}$, and $\vec{\mathrm{v}}$ be an assignment of $\vec{v}$. We write $\vec{\mathrm{v}} \models \psi(\vec{v})$ to state that the assignment $\vec{\mathrm{v}}$ satisfies ψ.

5.2 Robust n-Gadget Satisfiability

A major concern is that the attacker has limited control over s_0. That is, the content of memory (and registers) is only partially attacker-controlled. Speculating about unpredictable values would result in the unsound synthesis of incorrect chains.

To address this issue, we formalize in Definition 1 the problem of chaining gadgets – a mix of the *robust reachability problem* defined by Girol et al. [22] and *component-based program synthesis* introduced by Gulwani et al. [23]. We introduce the *assumptions* predicate $\mathcal{A}$ to represent the constraints that apply on s_0. That is, $\mathcal{A}$ encodes the context in which the chain executes – e.g. the location of attacker-controlled data and which register points to these data. We also introduce the *specification* predicate $\mathcal{S}$ to encode the constraints on the final state reached by the chain. That is, $\mathcal{S}$ encodes the goal that we want to reach – e.g. a function or system call.

Definition 1 (Robust n-Gadget Satisfiability)**.** Consider an initial state $s_0 = \langle \vec{p}, \vec{u} \rangle$. Let $\mathcal{A}$ be the *assumptions* and $\mathcal{S}$ be the *specification*. We define $\mathcal{S}$ to be n-gadget satisfiable (GS_n) from s_0 under $\mathcal{A}$, if there exists an assignment $\vec{\mathrm{p}}$ of $\vec{p}$ such that s_0 satisfies the assumptions $\mathcal{A}$, and regardless of the assignment of $\vec{u}$, the program reaches a state s_n that satisfies the specification $\mathcal{S}$, after executing n gadgets:

$$\vec{\mathrm{p}} \models \forall \vec{u}.\ GS_n(\vec{p}, \vec{u}) \text{ with}$$
$$GS_n(\vec{p}, \vec{u}) = \exists s_n.\ \mathcal{A}(s_0) \wedge s_0 \rightarrow^n s_n \wedge \mathcal{S}(s_n) \quad (1)$$

5.3 Eliminating the Quantifiers

One significant concern regarding Definition 1 is the presence of a $\forall$ quantifier on $\vec{u}$ in Eq. (1). Although SMT solvers are now proficient at solving quantifier-free formulas, they tend to perform poorly with quantifiers. Farinier et al. [18] showed how to solve quantified problems by reducing them to *quantifier-free* problems by leveraging *taint propagation*.

We introduce additional constraints to prove that the goal reached by the gadget chain is independent of unpredictable variables. Similarly to a *taint propagation*, we track which variables remain predictable after the execution of each instruction, directly in the SMT formula. Yet, contrary to a classical taint propagation, we do not track which values get polluted by the attacker, but rather which attacker-controlled variables get polluted by unpredictable values at runtime.

For each SMT variable x – whether it is a register or a byte in memory – we add a new Boolean variable $x^\bullet$ to represent its taint. We define that $x^\bullet$ holds only if x is predictable, whereas $\neg x^\bullet$ means that x may be unpredictable at runtime. For a state s, we note its taints $s^\bullet \triangleq \langle reg_0^\bullet, \dots, reg_n^\bullet, Mem^\bullet \rangle$. For each gadget, we introduce two new predicates $g_{pre}^\bullet$ and $g_{post}^\bullet$. We encode the pre-conditions on taints in $g_{pre}^\bullet$ and the taint propagation between two states in $g_{post}^\bullet$ – see Section 6 for details on their encoding. We also introduce the assumptions and specifications on taints as $\mathcal{A}^\bullet$ and $\mathcal{S}^\bullet$. Specifically, we encode which values are part of $\vec{p}$ and $\vec{u}$ using $\mathcal{A}^\bullet$. Finally, we derive the problem from Definition 1 into a *quantifier-free* problem using taints:

$$\langle \vec{\mathrm{p}}, \vec{\mathrm{u}} \rangle \models GS_n(\vec{p}, \vec{u}) \wedge GS_n^\bullet(\vec{p}, \vec{u})$$
$$\text{with } GS_n^\bullet(\vec{p}, \vec{u}) \triangleq \exists s_n^\bullet.\ \mathcal{A}^\bullet(s_0^\bullet) \wedge s_0^\bullet \rightarrow^n s_n^\bullet \wedge \mathcal{S}^\bullet(s_n^\bullet) \quad (2)$$

GS_n models the semantics of the chain, whereas $GS_n^\bullet$ ensures that the proper execution of the chain does not rely on unpredictable values. Any solution $\langle \vec{\mathrm{p}}, \vec{\mathrm{u}} \rangle$ to the quantifier-free Eq. (2) implies that $\vec{\mathrm{p}}$ is a solution to Eq. (1) from Definition 1 – the proof is analogous to that of Farinier et al. [18].

Refer to Appendix A for an example of assumptions and specifications encoding for CVE-2022-46152.

6 From Gadgets to SMT Formulas

Gadgets can be extracted using several methods [2, 12, 49]. This section provides an overview of how we build the g_{pre} and g_{post} SMT formulas for each gadget. For the sake of clarity, we introduce an example gadget **G** in Listing 1 – found in the `libc` (`x86_64`) – that we will use in the rest of this section to illustrate our points. On the left are given the assembly instructions of the gadget together with their addresses, whereas on the right are given insights on the semantics of the instructions.

Listing 1: Assembly instructions of gadget **G**.

```
1 0x50138: mov    edx,1                    set edx to 1
2 0x5013d: cmovne ecx,edx          copy edx to ecx if zf==0
3 0x50140: test   ecx,ecx              check if ecx == 0
4 0x50142: jne    0x56148          jump out if ecx != 0
5 0x50144: ret                        return otherwise
```

6.1 Semantics Extraction

We analyze and process each gadget to extract its explicit semantics – see Section 8 for details on the tools we use.

Lifting. In the lifting phase, the gadgets are translated to an *intermediate representation* (IR). The IR explicitly represents all hidden side effects of instructions, guaranteeing an accurate representation of their semantics. Additionally, this conversion to an IR facilitates subsequent analysis by abstracting the underlying complexity of the architecture – thereby enabling easier portability to other architectures.

CFG Construction. Afterwards, we recover the *Control-Flow Graph* (CFG) of each gadget. Figure 5 showcases the translated version of gadget **G** into the IR, and the explicit encoding of its control flow. Specifically, lines 2-4 show the translation of the predicated `cmovne` instruction with a conditional `if` (at line 2). It also demonstrates the explicit handling of *condition flags* from line 5 to 11 – set by the `test` instruction. The branch to **BB3** corresponds to the jump to `0x56148` in gadget **G**. It is considered as undefined behavior because it branches outside the gadget boundaries.

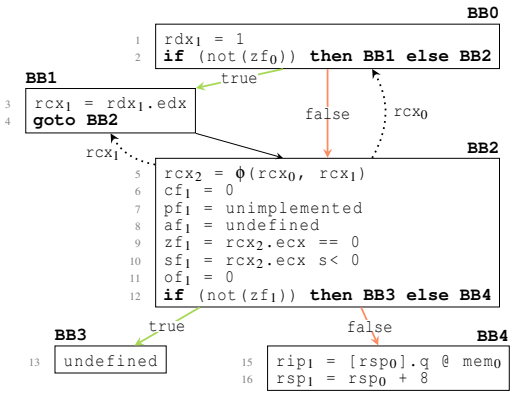


Figure 5: CFG of gadget **G** in GSA form.

GSA Form. To properly model the *data-flow* of gadgets, we convert the IR into a *Gated Single Assignment form* (GSA) [24, 54] – an extension of the *Static Single Assignment form* (SSA) [13]. In SSA, variables are made *stateful* by labeling them at each assignment. Control flow splitting and merging are handled by introducing so-called ϕ-nodes, whose purpose is to select between different potential candidates for an assignment. Additionally, the GSA encodes the semantics of ϕ-nodes by associating each block with a *gating function* that specifies the condition under which the block is reached, and from which block it is entered. For instance, the gating function of **BB2** and **BB4** in Fig. 5 are as follows:

$$\underbrace{\gamma_{\texttt{BB2}}(\texttt{zf}_0, \texttt{BB0}, \texttt{BB1})}_{\substack{\text{if } \texttt{zf}_0 = 1, \text{ then } \texttt{BB2} \text{ is entered} \\ \text{from } \texttt{BB0}, \text{ else } \texttt{BB2} \text{ is entered from } \texttt{BB1}}} \qquad \underbrace{\gamma_{\texttt{BB4}}(\texttt{zf}_1, \texttt{BB2}, \varnothing)}_{\substack{\text{if } \texttt{zf}_1 = 1, \text{ then } \texttt{BB4} \text{ is entered} \\ \text{from } \texttt{BB2}, \text{ else } \texttt{BB4} \text{ is not reached}}}$$

6.2 Pre-Conditions

The pre-conditions are meant to ensure the proper execution of gadgets. They prevent program crashes and undefined behaviors – e.g. invalid memory accesses We consider various kinds of pre-conditions, particularly:

Control-Flow. We prevent undefined behaviors. For example, in Fig. 5, **BB2** has only one valid outgoing edge leading to **BB4** – the other one leads to undefined operations. Thus, we enforce the conditions for branching from **BB2** to **BB4**.

Memory Operations. The attacker provides the list of memory regions that are readable and writable. For each memory operation, we enforce that the address is predictable, and falls within a valid region. Even if the value read (resp. written) is not used, this ensures that the operation will not interfere with the proper execution of the chain.

We group the pre-conditions per basic block, and we ensure that they are enforced only when the associated block may actually be executed. To do so, we introduce what we call *mayreach* predicates that encode whether a basic block may be reached: $\mathit{mayreach}_{\texttt{BBi}} \implies$ pre-conditions of instructions in **BBi**. Notably, *mayreach* predicates take into account the taints. It is sometimes impossible to know which exact path will be taken at runtime due to unpredictable values. Yet, we may still execute a gadget without knowing which exact path it takes. For instance, in Fig. 5, if $\texttt{zf}_0$ is unpredictable, we do not know whether **BB1** will be executed or not, so we consider that it may be reached. Otherwise, we know that **BB1** is reached only when $\texttt{zf}_0$ is null. Hence, the following *may-reach* predicate: $\mathit{mayreach}_{\texttt{BB1}} \triangleq \neg \texttt{zf}_0^{\bullet} \vee (\texttt{zf}_0 = 0)$.

6.3 Post-Conditions

Post-conditions encode the semantics of gadgets' instructions, and the propagation of taints from one state to another. Most arithmetic and bitwise operations of the IR have an equivalent operation in the bit-vector theory. Translating them in SMT formulas is straightforward. As for the taint, in most cases, we consider the result of an operation to be tainted only if all its operands are also tainted – see Appendix B for some exceptions. However, when it comes to memory operations and ϕ-nodes, the encoding is more complex:

Memory Operations. The theory of arrays includes two operations for modeling memory accesses: `select` and

`store`. For reading (resp. writing) n-bytes long values, we encode the memory access as n consecutive `select` (resp. `store`) operations – and concatenate (resp. split) the bytes depending on the endianness. We propagate the taints similarly. When writing to memory, we propagate the taint of the written value to each byte stored in the memory array. When reading memory, we consider the read value to be tainted only if all corresponding bytes in memory were also tainted.

ϕ-Nodes Assignments. Gating functions introduced in Section 6.1 enables to encode which value to assign in a ϕ-node. We simply translate gating functions using SMT *if-then-else* constructs. Regarding the taint, we distinguish two cases. For example, let us consider the ϕ-node in **BB2** of Fig. 5. If $\texttt{zf}_0$ is predictable, we propagate the taint of $\texttt{rcx}_0$ when $\texttt{zf}_0 = 1$, and the taint of $\texttt{rcx}_1$ when $\texttt{zf}_0 \neq 1$. However, if $\texttt{zf}_0$ is unpredictable, we cannot determine which path is taken at runtime to reach the ϕ-node. Nevertheless, if both $\texttt{rcx}_0$ and $\texttt{rcx}_1$ are predictable and equal to each other, then, whatever path is taken, the value of $\texttt{rcx}_2$ is always the same. Therefore, we encode the taint propagation as follows:

$$\texttt{rcx}_2^\bullet = \left(\texttt{rcx}_0^\bullet \wedge \texttt{rcx}_1^\bullet \wedge \texttt{rcx}_0 = \texttt{rcx}_1\right) \vee \left(\texttt{zf}_0^\bullet \wedge (\mathit{if}\ \texttt{zf}_0 = 1\ \mathit{then}\ \texttt{rcx}_0^\bullet\ \mathit{else}\ \texttt{rcx}_1^\bullet)\right)$$

7 Querying the SMT Solver

In this section, we provide an overview of how we query the solver and introduce some optimizations for better scalability.

7.1 Encoding & Incremental Solving

We encode Eq. (2) – see Section 5.3 – by associating a variable gid_i to each transition $s_i \rightarrow s_{i+1}$. Each gid_i represents the unique index of the gadget executed to go from s_i to s_{i+1}. We encode the transition from s_i to s_{i+1} as a big cascade of *if-then-else* formulas that compare the value of gid_i to each gadget index and apply the semantics of the corresponding gadget. Eventually, the solver's job is to attribute an index to each gid_i variable – i.e. choose which gadget is executed at each transition.

Yet, when trying to synthesize a chain, we generally want to search for a chain *up to* a certain length n. However, Eq. (2) denotes the problem of finding a chain of length n strictly. To tackle this issue, we use a feature of SMT solvers called *incremental solving*. Incremental solving allows to incrementally push new constraints to the solver, and check for satisfiability at each step – while keeping information computed during previous steps. Consequently, to synthesize a chain, we incrementally check the satisfiability step by step, by first checking for length 1, then 2... and up to length n. Refer to Appendix C for a commented algorithm.

7.2 Scalability

Gadget chain synthesis is a time and resource consuming problem. The search space grows exponentially with the number of available gadgets and the chain length. Since we cannot guide the underlying SMT solver as it acts like a black-box, we focused on selecting which gadgets to feed into the solver.

Gadget Library Sampling. Our incremental query scheme is similar to a *Breadth-First Search* (BFS), because we first query for chains of n gadgets before trying with $n+1$. Since the search space grows exponentially, this poses issues regarding the synthesis of long chains because reaching high depth becomes very costly in time and resources. Due to this, most existing approaches are based on *Depth-First Search* (DFS) instead [2, 12, 48]. Yet, DFS also has drawbacks – e.g. backtracking is costly if you start exploring the wrong path.

To mitigate the disadvantages of BFS, we randomly sample the gadget library into smaller subsets and launch multiple syntheses on different samples in parallel – we stop on the first success. By dividing the problem in a lot of smaller instances, it becomes less costly for each instance to search for long chains. However, doing so, we introduce similar drawbacks as DFS. That is, if an instance does not sample the good gadgets, it cannot find a chain. This is mitigated by running multiple instances in parallel, increasing the chance to succeed. We define the following parameters:

- `jobs`: Number of syntheses launched in parallel (job);
- `size`: Number of gadgets sampled by each job.

Memory Writes. Multiple other approaches had to tackle issues related to memory accesses [2, 8, 19, 48]. Especially, symbolic memory writes – whose target address is not constant – are well-known to be costly for solvers and thus, are also an issue for us. The solver must consider all possible overlapping read/write patterns, leading to an explosion in possible states and vast search spaces. Existing works proposed some heuristics-based solutions, including prohibiting symbolic memory writes [8], or forcing them to write to a precise restricted space [48].

In contrary, since we use the solver as a black-box, our approach allows symbolic memory writes. Yet, we impose limitations on the proportion of gadgets provided to the solver that contain memory writes. For each job, we introduce the `mem-write-ratio` parameter that defines the proportion of gadgets with memory writes fed to the solver. By default, we launch half the jobs with no memory writes, a quarter with 5% of memory writes and another quarter with 10%. This way, we keep the ability to use gadgets that write to memory, while amortizing the cost for the solver.

8 Experimental Evaluation

In this section, we evaluate the ability of our approach to synthesize chains fitting various situations and compare it to the

Table 1: Ability of ARCANIST and related works to exploit real-world vulnerabilities involving non-trivial exploitation layouts and diverse primitives. Time is given in seconds. (# gad. = # extracted gadgets, fcall*n* = arbitrary function call with *n* parameters).

Platform		Vulnerability			SGC, Angrop & Ropium	ARCANIST				
Arch.	OS	Reference	Target	Primitive		# gad.	`fcall3`		`fcall4`	
Arm	Linux	CVE-2022-46152 [39]	OPTEE	Arb. Decrement	✗	7199	✓	11s	✓	12s
		CVE-2019-1010298 [38]	OPTEE	Heap Overflow	✗	7199	✓	100s	✓	293s
		CVE-2023-30799 [4]	RouterOS	Arb. Jump	✗	537	✓	27s	✓	16s
x64	Linux	CVE-2016-10190 [26]	ffmpeg	Object Corruption	✗	17442	✓	224s	✓	113s
		Synacktiv PR4100 [21]	WD PR4100	Stack Overflow*	✗	906	✓	2464s	✗†	
	Windows	CVE-2021-35211 [31]	Serv-U FTP	Vtable Corruption	✗	46618	✓	314s	✓‡	837s
x86	Linux	CVE-2023-30799 [34]	RouterOS	Arb. Jump	✗	1241	✓	2s	✓	2s
	Windows	CVE-2018-9958 [51]	Foxit Reader	Use-after-Free	✗	67986	✓	2s	✓	2s
		CVE-2018-11529 [37]	VLC	Use-after-Free	✗	1185	✓	30s	✓	31s
		EDB-ID-47122 [17]	R	Stack Overflow*	✗	44278	✓	3s	✓	3s

† Not enough attacker-controlled data? ‡ Forced sampling of gadgets that set `r9`. ✓= success ✗= failure
* Stack-overflow but stack-pointer does not point to attacker-controlled data.

state-of-the-art. We answer a list of research questions explicitly denoted by the titles of Sections 8.1 to 8.6. Especially, we assess ARCANIST's real-world applicability on 10 vulnerabilities (CVEs) involving various non-trivial exploitation layouts and primitives. First, let us introduce our experimental setup:

Implementation. Our *proof-of-concept* implementation, named ARCANIST is written in Python and uses PySMT [20] to interface with the SMT solver. It uses `Ropper` [45] for the initial extraction of gadgets. ARCANIST leverages Binary Ninja's intermediate representation, called *LLIL* [1], to lift the gadgets. We used Boolector [41] as a backend SMT solver. ARCANIST supports `x86`, `x64` and `Arm` architectures, and is meant to be straightforwardly extensible to all architectures supported by Binary Ninja.

Hardware Setup. Our setup runs on an Intel® Xeon® Gold 5218 CPU 64 Cores @ 2.30 GHz, with 93 GB of RAM.

Competitors. We compared ARCANIST to Ropium [35], Angrop [2] and SGC [46]. Our evaluation aims to compare all four approaches in their entirety. That is, we let each tool use its own gadget extraction and own chaining technique. Note that Ropium and Angrop are single-threaded and take no additional parameters besides the goal to reach – both of them were run *as-is*. On the contrary, SGC and ARCANIST are both multi-threaded, configurable and randomly sample the gadget collections to run in parallel.

8.1 Can ARCANIST exploit real-world vulnerabilities with non-trivial layouts?

In this section, we evaluate the applicability of ARCANIST on 10 real-world cases of CVE exploitation that involve non-trivial layouts – not supported by competitors – and diverse primitives. We focused on fairly recent vulnerabilities known to be exploitable through code-reuse, targeting Windows and Linux applications, and all architectures natively supported by ARCANIST. For each case, we queried ARCANIST to generate chains that perform a function call with 3 and 4 arguments[1] – referred to as `fcall3` and `fcall4`. Function calls are, overall, universal goals for code-reuse attacks – e.g. `memmap` on Linux and `VirtualAlloc` on Windows. We asked for chains of maximum 9 gadgets, allocated a maximum of 56 jobs and a timeout of 3600s (1h). Our results are given in Table 1.

ARCANIST clearly outperforms its competitors on the evaluated vulnerabilities. While other approaches are unable to generate chains due to the non-trivial layouts involved, ARCANIST successfully generated a chain for each vulnerability. The diversity of attack primitives involved in the assessed vulnerabilities highlights the versatility of ARCANIST.

Table 1 also shows the ability of ARCANIST to synthesize chains in situations with both few gadgets – down to 537 for RouterOS on `Arm` – and lots of gadgets – up to 67986 for Foxit Reader on `x86`. Additionally, ARCANIST synthesized chains ranging from 1 – e.g. for EDB-ID-47122 – to 9 gadgets – for CVE-2021-35211. This further shows the versatility of ARCANIST, being able to generate rather long chains – see Sections 8.2 and 8.3 for examples.

Overall, ARCANIST responded quite quickly, and rarely in more than 5min. Yet, for the PR4100 case, it took up to 2464s (42min) to find a 6-gadget long chain for `fcall3`, and ARCANIST was unable to generate a `fcall4` chain. We believe this is correlated to the limited size of attacker-controlled data. In the case of PR4100, the attacker only controls `0x60` bytes. Considering the 6-gadget long chain (for `fcall3`), the addresses of gadgets alone already take `0x30` bytes, not leaving much room for more gadgets and data. Added to this, the small number of available gadgets (906) further reduces chaining possibilities.

[1] We stopped at 4 arguments since afterwards, arguments are generally loaded from the stack by the callee.

Listing 2: Chain found by ARCANIST for `fcall4` with CVE-2019-1010298. Grayed out instructions are irrelevant. Colors highlight the data-flow of some registers.

```
1  adds r0, #8;              r0 points to controlled data
2  ldr r3, [r0, #-0x4];
3  ldr r3, [r3, #0x54];  load the "JOP dispatcher" in r3
4  blx r3;                                        Gadget 1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
5  ldr r1, [r0];          load controlled address in r1
6  subs r2, r2, r3;
7  ldr r0, [r0, #4];      update r0
8  ldr r6, [r1, #0x10];                           Gadget 2
9  mov r1, r5;
10 blx r6;                                ("JOP dispatcher")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
11 mov r5, r2;
12 mov r7, r0;       make r7 points to controlled buffer
13 mov r6, r3;
14 blx r3;            re-chain the dispatcher      Gadget 3
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
15 ldr r1, [r0];
16 subs r2, r2, r3;
17 ldr r0, [r0, #4];
18 ldr r6, [r1, #0x10];                           Gadget 4
19 mov r1, r5;
20 blx r6;                                ("JOP dispatcher")
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
21 mov sp, r7;        stack-pivot to controlled buffer
22 pop {r4, r5, r6, r7, r8, sb, sl, pc};          Gadget 5
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
23 ldr r3, [sp, #0x18];  load arg3 from pivoted stack
24 ldr r2, [r5, #0x10];  load arg2, r5 from gadget 4
25 ldr r1, [r5, #0x1c];  load arg1, r5 from gadget 4
26 ldr r0, [sp, #0x14];  load arg0 from pivoted stack
27 blx r7;   function call, r7 from gadget 4      Gadget 6
```

Listing 3: Chain found by ARCANIST for `fcall4` with CVE-2021-35211. Grayed out instructions are irrelevant. Colors highlight the data-flow of some registers.

```
1  mov rsi, [r11 + 0x38];
2  mov rsp, r11;         stack-pivot to controlled buffer
3  pop r13;                          set r13 for gadget 4
4  pop rdi;
5  pop rbp;                          set rbp for gadget 5
6  ret;                                           Gadget 1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
7  pop rcx;                          set rcx for gadget 6
8  ret;                                           Gadget 2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
9  pop rdx;                           set value for arg1
10 ret;                                           Gadget 3
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
11 xchg r8, r13;                   arg2, r13 from gadget 1
12 ret;                                           Gadget 4
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
13 mov esi, [rsp + 0x40];
14 mov rax, rbp;                    set rax for gadget 6
15 mov rbp, [rsp + 0x38];
16 add rsp, 0x20;
17 pop rdi;
18 ret;                                           Gadget 5
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
19 mov r9, rcx;                     arg3, rcx from gadget 2
20 cmp dword ptr [rax + 0x2f8], 0; load controlled data
21 je 0x1a5359;                   ensure jump is not taken
22 xor eax, eax;
23 ret;                                           Gadget 6
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
24 pop rcx;                           set value for arg0
25 ret;                                           Gadget 7
```

Additionally, in the case of CVE-2021-35211, we had to force the inclusion of gadgets that set `r9` in the gadget samples. This was necessary because of the very low number of gadgets that modify `r9` among the high number of gadgets for this target (46618). The random sampling was struggling to select gadgets that set `r9`. We further discuss the limits of random sampling, in Section 8.6.

8.2 How complex are ARCANIST's chains?

ARCANIST is free from heuristics, it is not guided in the way it chains gadgets. Yet, without explicit requests from our side, ARCANIST was able to leverage complex chaining techniques on its own – and that competitors cannot leverage. In this section, we highlight different abilities of ARCANIST and, as examples, we detail two chains – in Listings 2 and 3 – for two CVEs of Table 1.

Unassisted Stack-Pivoting. When the stack-pointer does not point to attacker-controlled data, chaining gadgets becomes cumbersome because we cannot leverage return-ended gadgets. A solution is to perform a so-called *stack-pivot*, that is, find a way to overwrite the stack-pointer and make it point to attacker-controlled data. For all cases evaluated in Section 8.1, ARCANIST stack-pivoted on its own to gain access to more gadgets, and eventually reach its goal.

For example, in Listing 2, ARCANIST stack-pivots in gadget 5 at line 21. This allows gadget 6 to load parameters from the newly pivoted stack – in `r3` and `r0`. As for the chain in Listing 3, ARCANIST leverages the fact that `r11` points to attacker-controlled data. It directly stack-pivots in gadget 1 at line 2. This allows it to chain return-ended gadgets afterwards.

Autonomous Jump-Oriented Chaining. When stack-pivoting is not directly possible, one can leverage jump- or call-ended gadgets. This is commonly referred to as *Jump-Oriented Programming* (JOP) [5]. Chaining JOP gadgets can be complex since they are not linearly read from the stack. To simplify it, Bletsch et al. [5] introduced the concept of *dispatcher gadgets*, whose purpose is to load the next *functional* gadget to execute – that performs useful operations.

In the case of CVE-2019-1010298 (Listing 2), ARCANIST synthesized an almost fully-JOP chain. More interestingly, ARCANIST also made use of a dispatcher – gadget 2 and 4. First, gadget 1 sets up the dispatcher in `r3` (line 3). Then, the dispatcher chains gadget 3 by loading its address from memory using `r0` – pointing to controlled data. Then, gadget 3 loads a controlled address into `r7` (line 21), later used by gadget 5 to stack-pivot. Since gadget 3 does not update `r3`, the dispatcher is chained again at line 14. This time, the dispatcher (gadget 4) chains gadget 5. Finally, gadget 5 stack-pivots and gadget 6 performs the function call.

Utilizing Conditional Paths. Sometimes, the only way of controlling the value of a register is to execute a gadget with conditional jumps or predicated instructions. This was the case for CVE-2021-35211 (Listing 3). Due to the very small number of gadgets that modify `r9`, ARCANIST had to use a gadget with a conditional jump (gadget 6) at line 21. In order to reach the return at line 23, ARCANIST had to ensure that the jump is not taken. To do so, it first loaded a controlled value

into `rbp` at line 5 in gadget 1. Then, it moved `rbp` to `rax` in gadget 5 at line 14. Eventually, this led to `rax` being attacker-controlled in gadget 6. By making it point to a non-null value in memory, it avoided the jump at line 21.

Additionally, in other chains, ARCANIST made use of predicated `cmovne r9, ...;` instructions to set `r9`.

Leveraging Additional Buffers. Sometimes, one attacker-controlled buffer is not enough to reach the desired goal. For example, in our case-study CVE-2022-46152, there is very little attacker-controlled data. In this case, if an additional buffer is made available to ARCANIST, it can automatically use it to load data – e.g. the parameters for a function call. This is demonstrated in Section 8.3.

8.3 Case-Study: How does ARCANIST exploits CVE-2022-46152?

In this section, we focus on our case-study: CVE-2022-46152, targeting OP-TEE version 3.6 – see Section 4.3. Particularly, we analyze two chains synthesized by ARCANIST to reach the `fcall4` goal. The exploitation layout of CVE-2022-46152 fits layout type ③ making it hard to chain gadgets. In addition to the few attacker-controlled values pushed on the stack, OP-TEE allows the creation of shared buffers whose content is also attacker-controlled. Without us asking for it, ARCANIST made a wise use of this shared buffer to reach the requested goals.

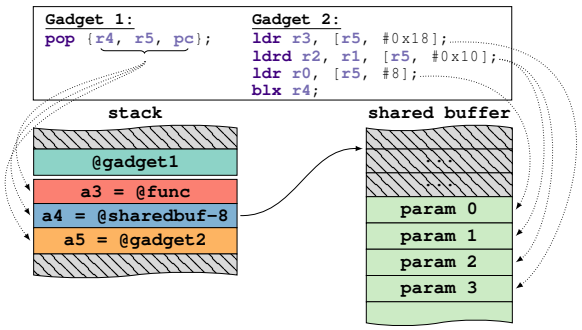


Figure 6: Non-pivoting gadget chain found by ARCANIST for a function call with CVE-2022-46152.

Due to the intricate exploitation layout, one could think that stack-pivoting to the shared buffer is needed to conduct an attack. Yet, ARCANIST found a two-gadgets long chain that did not stack-pivot. Fig. 6 illustrates how the chain works. Instead of stack-pivoting, ARCANIST figured out that it could use the shared buffer as a data section to store the function parameters. The choice of first gadget is limited – address must be less than `0x8DB028CF`, see Section 4.3. Yet, ARCANIST successfully found one that meets the requirements. This first gadget pops the address of the attacker-decided function to `r4`, and pops an address pointing near the shared buffer to `r5`. Then, the second gadget loads values into `r0`, `r1`, `r2` and `r3` from memory at addresses relative to `r5`. Since `r5` points near the shared buffer, it allows loading controlled parameters into the appropriate registers. Finally, it performs a function call using `r4`, which points to the desired function.

Listing 4: Stack-pivoting chain found by ARCANIST for `fcall4` with CVE-2022-46152. Grayed out instructions are irrelevant. Colors highlight the data-flow of some registers.

```
1  add sp, #0x10;
2  pop {r4, r5, r6, r7, r8, pc};                    Gadget 1
3  movs r2, #1;      set r2 for gadget 5
4  mov r1, r4;
5  mov r0, r4;
6  blx r8;           r8 popped by gadget 1          Gadget 2
7  it ne;
8  movne r0, #8;
9  pop {r3, pc};     set r3 for gadget 5            Gadget 3
10 mov r0, r4;
11 adds r7, #0x28;   r7 popped by gadget 1
12 mov sp, r7;       stack-pivot to shared buffer
13 pop {r4, r5, r6, r7, r8, sb, sl, pc};            Gadget 4
14 add r2, r3, r2, lsl #2; make r2 point to shared buf.
15 mov r1, sl;       arg1, sl from gadget 4
16 mov r3, r4;       arg3, r4 from gadget 4
17 ldr r2, [r2, #-0xc]; load arg2 from shared buffer
18 mov r0, r8;       arg0, r8 from gadget 4
19 blx r5;       function call, r5 from gadget 4 Gadget 5
```

ARCANIST was also able to find a longer stack-pivoting chain. We detail it in Listing 4. The first three gadgets are mainly here to set `r2` to 1 and load controlled data into `r7` and `r3`. These registers are later used by gadgets 4 and 5. Gadget 4 uses `r7` to stack-pivot to the shared buffer – at line 12. Then, it pops registers from the newly pivoted stack. Gadget 5 sets the value of `r0`, `r1` and `r3` by moving the registers previously popped by gadget 4. However, it is a bit more complicated for `r2`. Gadget 5 loads `r2` from memory at an address relative to `r2` – at line 17. This address is computed just before at line 14, using `r3` and `r2` as operands. ARCANIST took care of this by setting `r2` to 1 and loading an attacker-controlled value to `r3` – with gadgets 2 and 3. This results in an attacker-controlled address put into `r2` at line 14, allowing to load an attacker-controlled argument at line 17. Finally, it performs the function call using `r5` – popped by gadget 4.

8.4 How does ARCANIST perform in stack-overflow layouts (①)?

In Section 8.1 we have shown the ability of ARCANIST to outperform its competitors in real-world situations involving layouts ② and ③. In this section, we focus on layout ① to compare ARCANIST to related works. That is, we consider a generic stack-overflow situation giving full control of the stack to the attacker. Most state-of-the-art tools only support the `x64` architecture. Consequently, we ran our experiments on the following widely used `x64` binaries: `nginx`, `libc`, `httpd`, `dnsmasq`, `openssl` and `lighttpd`. We evaluated each tool's ability to achieve some goals commonly tar-

geted in code-reuse attacks: `fcall3/4` (function call with 3/4 arguments), `scall4` (system call with 4 arguments) and `pivot` (stack-pivot).

We configured SGC and ARCANIST with the same parameters: we asked for chains up to length 8, we allocated 24 CPU cores and the samples size was set to 600 gadgets. The timeout of all tools was set to 3600s (1h). Table 2 summarizes our results. We included the response time of each tool for comparison, especially SGC and ARCANIST. Refer to Section 8.6 for in-depth evaluations of ARCANIST's performance.

Table 2: Comparison of ARCANIST and related works ability to synthesize chains in layout ①. Response time is given in seconds. Chain lengths are given in parentheses.

Goals		Heuristics-Based		SMT-Based	
		Ropium	Angrop	SGC	ARCANIST
fcall3	nginx	✓ 2.48 (8)	✓ 89.4 (3)	✓ 405.1 (3)	✓ 150.5 (3)
	libc	✓ 0.01 (5)	✓ 1.80 (3)	✓ 41.66 (2)	✓ 45.64 (2)
	httpd	✓ 0.02 (8)	✓ 3179 (3)	✓ 273.2 (3)	✓ 97.56 (3)
	dnsmasq	✓ 0.01 (5)	✓ 2.89 (3)	✓ 247.0 (3)	✓ 141.3 (3)
	openssl	✗ 3.29	✓ 7.69 (3)	✓ 104.0 (3)	✓ 137.7 (3)
	lighttpd	✓ 0.01 (5)	✓ 1.07 (3)	✓ 337.1 (3)	✓ 127.7 (3)
fcall4	nginx	✓ 6.55 (9)	✓ 977.2 (4)	✓ 622.1 (3)	✓ 274.5 (4)
	libc	✓ 0.01 (6)	✓ 39.96 (4)	✓ 41.99 (2)	✓ 138.9 (3)
	httpd	✓ 0.02 (9)	✗ ∞	✓ 316.0 (3)	✓ 249.9 (4)
	dnsmasq	✓ 0.01 (7)	✓ 13.35 (4)	✓ 480.8 (4)	✓ 298.4 (4)
	openssl	✗ 7.35	✗ 649.6	✓ 614.0 (5)	✓ 407.0 (5)
	lighttpd	✓ 0.01 (6)	✓ 6.41 (3)	✓ 355.2 (3)	✓ 176.3 (3)
scall4	nginx	✗ 35.78	✗ ∞	✓ 1473 (4)	✓ 272.5 (4)
	libc	✗ 48.03	✗ 181.9	✓ 43.80 (2)	✓ 425.9 (4)
	httpd	✗ 60.08	✗ ∞	✓ 748.4 (4)	✓ 249.6 (4)
	dnsmasq	✗ 38.97	✗ 0.53	✓ 802.3 (4)	✓ 801.1 (5)
	openssl	✗ 21.07	✗ 0.48	✓ 758.4 (6)	✓ 1390 (6)
	lighttpd	✗ 44.51	✗ 1875	✓ 466.3 (3)	✓ 155.7 (3)
pivot	nginx	✗	✗	✗ 1122	✓ 5.20 (1)
	libc	✗	✗	✓ 993.1 (4)	✓ 5.08 (1)
	httpd	✗	✗	✗ 921.2	✓ 4.94 (1)
	dnsmasq	✗	✗	✗ 954.2	✓ 6.08 (1)
	openssl	✗	✗	✗ 1089	✓ 5.92 (1)
	lighttpd	✗	✗	✗ 1113	✓ 5.42 (1)

▨ = unsupported goal ∞ = timeout ✓= success ✗= failure

ARCANIST is the only one synthesizing chains for all goals and all binaries. Ropium was not able to synthesize chains for `openssl`, and Angrop failed to synthesize `fcall4` chains for both `httpd` and `openssl`. In contrast, only SGC and ARCANIST successfully generated attacks reaching the `scall4` goal. Finally, ARCANIST outperforms SGC on the `pivot` goal.

The versatility offered by SMT-based approaches comes with a cost in time. This is highlighted by the differences in time between Ropium and SMT-based tools. Yet, ARCANIST replies faster than SGC, showing the benefits of one-shot syntheses and incremental solving. Though, SGC was faster for `libc` due to the difference in chain lengths found by both tools. On instances where both ARCANIST and SGC succeeded, ARCANIST replied 39% faster on average. In total, ARCANIST was 61% faster than SGC overall.

Moreover, the robustness of SGC's attacks relies on the post-synthesis verification mentioned in Section 4. Through our experimental observations, we noted that this verification step dismissed around 30% of the generated attacks, forcing it to seek for alternatives. In this regard, ARCANIST outperforms SGC due to its correctness-by-design approach, ensuring one-shot syntheses of correct chains.

Regarding the chain length, SMT-oriented approaches perform the best. Yet, on average ARCANIST found chains 4.6% longer than SGC. We believe this is due to a statistical variation related to the random sampling.

In the end, ARCANIST proves to be much more flexible than its competitors as it is the only one synthesizing chains in all situations. This further highlights the versatility of ARCANIST, even on layouts of type ① – already supported by other tools.

8.5 How do memory writes affect the solver?

In this section, we assess the impact of the `mem-write-ratio` parameter on the performance of the underlying solver used by ARCANIST. Generally, ARCANIST spawns multiple instances in parallel – as introduced in Section 7.2. However, in this section, our focus lies on determining the average time taken for each *instance* to terminate – depending on the `mem-write-ratio`. We tested ARCANIST for the hardest goal in layout ①: `scall4`, with `mem-write-ratio` ranging from 0% to 25%. We launched syntheses targeting `libc`, each time with a *single solver*.

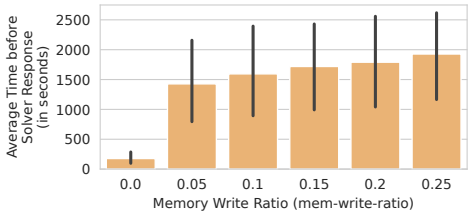


Figure 7: Average solver's response time on 20 syntheses, for several `mem-write-ratio` values. Goal is `scall4` for `libc`, with `size`=600, `max-length`=4. Timeout is set to 3600s.

Fig. 7 depicts our results. What is striking at first glance is the deep gap between the average response time in absence and in presence of memory write gadgets. With no memory writes, we observe very low response times, averaging at 176s. By just increasing the ratio to 5%, the average response time jumps to 1427s. Past this gap, the response time seems to increase linearly as we increase the `mem-write-ratio`, reaching an average of 1927s for a ratio of 25%. We believe that this gap is due to aggressive optimizations performed by the underlying SMT solver – only when no memory write is detected. We also observed that runs with no memory writes rarely consumed more than 2GB of RAM, whereas those with a ratio of 25% could consume up to 40GB. Additionally, we

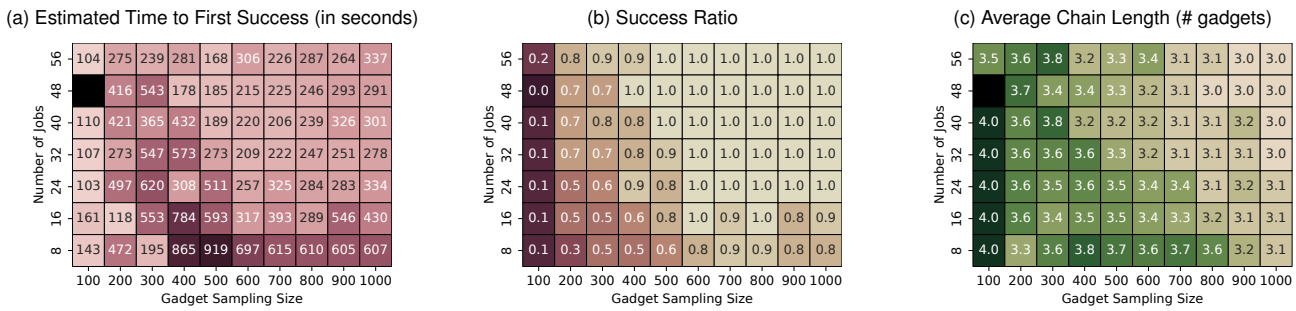


Figure 8: Estimated time to first success, success ratio and average chain length, when trying 10 successive syntheses of `scall4` for `libc` in layout ① (`max-length`=4). The brighter the colors, the better the results. Timeout is set to 1200s for each run.

tested syntheses with higher ratios, sometimes this resulted in crashing the underlying SAT solver used by Boolector, due to an excessive number of variables.

Removing memory write gadgets greatly improve the response time. However, it may also prevent us from achieving specific goals that are only reachable using memory writes – see the pattern in Appendix D. This highlights the benefits of launching concurrent syntheses both including and excluding memory writes – write-less instances may terminate early, while others may search for more intricate chains.

8.6 How does random sampling affect the performance of ARCANIST?

In this section, we assess the impact of random sampling on the performance of ARCANIST – the `jobs` and `size` parameters from Section 7.2. ARCANIST easily found chains for several real-world situations and in reasonable time – see Section 8.1. Though, for one of them we had to force the inclusion of gadgets that set a specific register – due to their rarity. There exist situations where very few and specific gadgets can be used to achieve the desired goal. In these cases, random sampling could struggle to pick the gadgets that are needed, leading to increased solving time or even failures. Consequently, we compare the results between situations with high and low diversity of usable gadgets.

8.6.1 Layout ①: High Diversity of Usable Gadgets

In this section, we consider a situation in layout ①. In such a situation, we can leverage *return-ended* gadgets and easily chain them together. That is, a lot of gadgets are usable to reach the desired goal. We launched syntheses of `scall4` for `libc`, with `max-length`=4, `jobs` ranging from 8 to 56, and `size` ranging from 100 to 1000. For each pair of `jobs` and `size`, we ran 10 successive syntheses. Fig. 8a displays the estimated time to first success, that is, the estimation of how much time it would take until the first success, if you were to run successively the 10 syntheses in a random order. Fig. 8b indicates the proportion of successes amongst the 10 syntheses. Fig. 8c gives the average length of generated chains. On all heatmaps, brighter colors mean better results.

Optimal performance, with estimated times of approximately 100 seconds, is hit when running syntheses with only 100 gadgets and more than 24 jobs. Despite the notably low success rate of around 0.1 when only providing 100 gadgets, failing syntheses terminate very quickly due to the reduced search space. This enables to quickly retry until a success, thereby explaining these low estimated times. Yet, due to the randomness involved in our strategy and the low success rate, all syntheses failed when running with 48 jobs and 100 gadgets – resulting in black squares in the heatmaps.

We notice a significant drop in estimated times between samples of size 300 and 600. With enough jobs, syntheses are overall twice faster with 500/600 gadgets than with 300/400 gadgets. We explain this by the increasing diversity of gadgets given to the solver, leading to higher success rates – see the correlation in Fig. 8b. The solver takes longer due to the increasing number of gadgets, but it succeeds more often, which ultimately reduces the time to first succeed. Additionally, more gadgets also mean more opportunities to quickly find shorter chains. This is highlighted by Fig. 8c.where drops in lengths correlate with drops in times in Fig. 8a.

8.6.2 Layout ②: Low Diversity of Usable Gadgets

In this section, we focus on layouts of type ②, where the attacker does not control the stack. Chaining gadgets without controlling the stack is harder since we cannot leverage return-ended gadgets. In these situations, a *stack pivot* is often useful – ARCANIST itself automatically pivoted for all the CVEs in Section 8.1. However, pivoting is not always easy as the number of gadgets usable for doing so is often limited. Consequently, in this section we focused on *synthetic* situations in layout ② that are far more constrained than real-world situations. Particularly, we consider that only one specific register is pointing to attacker-controlled data and that no other register is attacker-controlled – rarely the case in real-world

situations. That is, we build artificial situations that are intended to make the random sampling struggle and show its limits. We ran experiments on binaries of Section 8.4. We queried stack-pivots, and tried with different register pointing to controlled data: `rax`, `rsi`, `rdx`, `rdi`. Table 3 details our results.

ARCANIST only found chains for 46% of our queries. Note that a failure only means that ARCANIST did not find a chain, not that it failed when one existed. With no point of comparison, it is hard to assess how good is ARCANIST. However, for all failures, we also manually searched for chains and were unable to find some that ARCANIST would have missed.

Table 3: Ability of ARCANIST to pivot in our very constrained synthetic situations. Chain lengths are given in parentheses.

Target	Register pointing to attacker-controlled data			
	`rax`	`rsi`	`rdi`	`rdx`
`nginx`	✓(3)	✓(2)	✓(3)	✗
`libc`	✓(2)	✓(2)	✓(2)	✓(1)
`httpd`	✓(3)	✓(2)	✗	✗
`dnsmasq`	✗	✓(2)	✗	✗
`openssl`	✗	✓(2)	✗	✗
`lighttpd`	✗	✗	✗	✗

✓= success ✗= failure

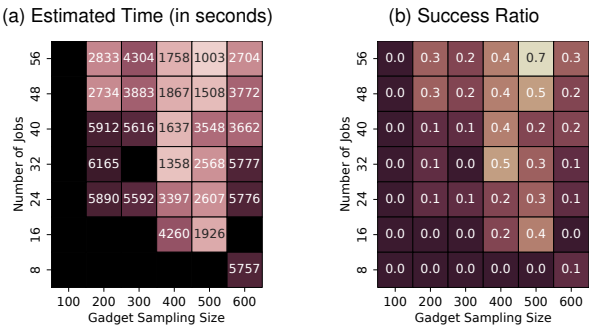


Figure 9: Estimated time to first success and success ratio, for 10 successive syntheses of stack-pivoting attacks via `rsi` for `dnsmasq` in layout ②. (Timeout = 1200s, `max-length`=4).

We observed higher solving time and much less overall efficiency than in Section 8.6.1. For instance, we depict in Fig. 9 the heatmaps associated to the results of `dnsmasq` – for the same parameters as in Section 8.6.1. Compared to the results given in Fig. 8, we observe a much lower overall success rate. Additionally, the estimated times before first success are overall one order of magnitude higher. These differences highlight the increased difficulty for ARCANIST to chain gadgets in these *synthetic* situations that are far more complicated than real-world CVEs – see Section 8.1. In the end, Fig. 9 shows the limits of our random sampling strategy. Only two or three specific gadgets chained together can reach the goal. As a result, we have much less chance to randomly pick these specific gadgets. For instance, chains of length 3 for `nginx` in Table 3 can take up to several days to be found by a single instance.

9 Discussion

In this section, we discuss the limits of our approach that could potentially impact its ability to synthesize an attack. We also suggest some ideas for improvement.

Limits of Random Sampling. In Section 7.2, we have introduced our random sampling strategy to enhance scalability. Our experiments in Section 8.6 have shown that using small samples could lead to great improvements in solving time. On the other hand, results from Section 8.6.2 have shown the limits of random sampling. In situations where a low diversity of gadgets can be leveraged to reach the attacker's goal, our random sampling strategy had more difficulties in picking the rare necessary gadgets. This results in increased solving times and a potential inability to find chains. We believe that our approach could greatly benefit from smarter assumption-aware and goal-aware sampling strategies. For instance, a pre-processing phase could easily dismiss gadgets that rely on the stack, when the stack is not under control.

Taint Propagation Completeness. Our approach suffers from the same disadvantages and inaccuracies of classical taint analyses. That is, sometimes it may classify values as unpredictable while they are predictable in practice – note that unpredictable values are never misclassified as predictable though, this ensures *soundness*. These inaccuracies can theoretically limit the completeness of our approach, but in practice, we did not observe any meaningful impact on our ability to find chains.

Control-Flow Integrity Mitigations. CFI aims at restricting the targets of branch instructions to ensure that the control flow is not deviated. As such, it prevents an attacker from chaining code-reuse gadgets. Yet, even if CFI is becoming more present, it is far from being a standard in real-world situations [31, 32, 52] – see the CVEs in Table 1.

Numerous approaches have tackled the problem of exploiting vulnerabilities with CFI enforced through so-called *Data-Only attacks* [27, 28, 43, 48, 59]. *Code-Reuse* and *Data-Only* attacks are somewhat orthogonal to each other. ARCANIST solves a *what-to-write* problem, that is, given a specific exploitation layout, it finds what the attacker must write to properly execute a gadget chain. On the other hand, exploiting *Data-Only* attacks is more a *where-to-write* problem. With CFI enforced, the attacker needs an arbitrary write-what-where primitive – i.e. have total control over all memory. In this case, the problem is more to figure out *where* to write to in memory – e.g. to gain control over the arguments of an otherwise legitimate system call – rather than *what* to write. No program synthesis is involved, as we cannot execute arbi-

trary gadgets. Overall, our approach tackles a fairly different problem than those involved in data-only attacks.

Yet, we can highlight similarities with the synthesis of *Block-Oriented Programming* (BOP) chains [27] – a data-only attack technique. BOP consists in finding legitimately chained basic blocks that fulfill some specific attack goals – thus bypassing CFI. We believe that techniques leveraged by ARCANIST could be adapted to find BOP snippets. Instead of code-reuse gadgets, one would extract and use legitimate basic blocks from the targeted program, and embed the constraints on their control-flow in the SMT formula. This would ensure that the generated *chain of basic blocks* would not break the integrity of the control-flow.

Furthermore, BOPC's [27] granularity is limited to single basic blocks. That is, it does not consider the semantics of combination of basic blocks. On the contrary, ARCANIST showed its ability to handle the semantics of multi basic blocks gadgets. Consequently, the same way ARCANIST can find more intricate code-reuse chains than its competitors, we believe that leveraging ARCANIST techniques might allow finding more intricate BOP chains.

Automated Exploit Generation. AEG [3] is still an active research topic [16, 25, 58, 60]. Tools like Mayhem [8] tackled the problem of generating end-to-end *shellcode only exploits* for stack-overflows. Yet, the multiplicity of exploitation primitives – e.g. *use-after-free, heap-overflow, format-string* – and the rise of mitigations – NX, stack canaries, CFI – makes generating end-to-end exploits very hard in the general case. ARCANIST tackles a challenging part within the development of end-to-end code-reuse exploits, that is chaining gadgets after the *code-pointer* is overwritten. Ultimately, ARCANIST could complement an AEG framework as a tool generating *code-reuse payloads*.

10 Related Works

Automating the synthesis of gadget chains has been explored by several works. ROPGadget [44] and Ropper [45] use pattern matching to build gadget chains, and are limited to building a few hard-coded ROP chains.

Q [47] was the first tool to integrate program verification techniques and SMT solving to categorize gadgets into semantic classes. However, verification is done only at the gadget level, not on the whole chain.

OptiROP [40] was meant to be a ROP assisting tool that leveraged SMT solving to search for single gadgets fitting a semantic query.

Angrop [2] and RiscyROP [12] leveraged symbolic execution. They symbolically explore candidate chains, driven by heuristics. The symbolic execution engine produces *path predicates* that, verified through SMT solving, ensure the validity of the chain. Yet, their application is limited to ROP in layout ①, and hard-coded goals.

SGC [46] delegates the chaining to an SMT solver. Yet, it needs a verification pass because its synthesis procedure is not sound. In a *Las Vegas* algorithm fashion, it generates candidates until one passes the verification. Finally, even though it relies on SMT solving, SGC is based on standard reachability (not robust) and it only supports ROP in layout ①.

More recent works started automating *Data-Only attacks* in order to bypass CFI. Data-only attacks are more constraining as they require a *write-what-where* exploitation primitive to be viable. Wollgast et al. [59] introduced a framework, based on the VEX IR and Z3 [14] SMT Solver, for finding CFI-resistant gadgets. Ispoglou et al. [27] introduced BOPC [27], meant to be a turing-complete data-only attack compiler. Similarly, Pewny et al. [43] introduced Steroids, a compiler for data-only attacks that also leverages SMT solving to solve constraints on path predicates. Limbo [48] leveraged concolic execution to symbolically explore reachable states and find exploitable execution paths. More recently, Johannesmeyer et al. [28] introduced an approach based on dynamic taint analysis to find data-only attacks. They keep track of attacker-controlled data to ensure it ends up in a desired location. Similarly, van der Veen et al. [56] used constraint-driven dynamic taint analysis to find function call gadgets in presence of CFI. In comparison, ARCANIST's use of taints is similar since it taints attacker-controlled data to ensures that they reach the goal state. However, while data-only approaches taint all memory – since attacker has total control over memory – ARCANIST only taints the specific memory location that are attacker-controlled, on a case-specific basis – depending on the attack primitive.

11 Conclusion

This paper has addressed the problem of synthesizing code-reuse gadget chains for arbitrary exploitation layouts. We proposed an approach to automatically synthesize gadget chains, based on robust reachability and component-based program synthesis. Unlike existing works, it has no requirements on the attacker capabilities and can adapt to any exploitation layout – whether it involves the heap, the stack, or any other region. It ensures the synthesis of correct-by-design chains that fit the precise memory layout given by the attacker. We implemented our approach in a tool named ARCANIST, leveraging SMT solving and taint propagation techniques. We evaluated the real-world applicability of ARCANIST on a set of 10 CVE exploitation cases involving non-trivial exploitation layouts. We showed that ARCANIST outperforms state-of-the-art tools by successfully synthesizing chain for each of the evaluated CVEs – when competitors were unable to. Finally, we showed that ARCANIST leverages complex chaining techniques on its own, without explicit requests from our side – e.g. autonomous stack-pivoting, jump-oriented chaining and use of conditional instructions.

Ethics Considerations

Disclosures. All vulnerabilities mentioned in the paper – e.g. CVE-2022-46152 – are already publicly known and patched.

Terms of Service & Law. We confirm that no terms of service nor any law was violated during our experiments.

Potential Negative Outcomes. Our work automatically synthesizes code-reuse gadget chains. As such, it could potentially help malicious actors. Yet, automating gadget chaining is nothing new, a lot of publicly available approaches already exist to do so.

Moreover, our work only focuses on synthesizing gadget chains, which is only part of the work needed to exploit a vulnerability. Our work does not tackle the problem of end-to-end attack generation. As such, building a complete exploit still requires a skilled-enough attacker to do so.

Additionally, a lot of mitigations have been proposed and are publicly available to counter code-reuse attacks. For example, fine-grained CFI and pointer authentication (PAC). In the end, if anyone is concerned about this type of attacks and willing to defend themselves, they already have several ways of doing so.

Open Science

In accordance with the Open Science Policy, we publicly share artifacts accessible through the following DOI: 10.5281/zenodo.14724514. The artifacts contain the following elements:

- the source-code of ARCANIST;
- all the binaries used for the experiments in Section 8;
- since ARCANIST relies on a non-free version of Binary Ninja [1] to lift gadgets, we share the gadget libraries already extracted and translated by ARCANIST for each binary, and ready to use as-is by the tool;
- containerized and ready-to-use scripts to reproduce the results of Table 1, Listing 2, Listing 3, Listing 4, Table 2, Fig. 7, Fig. 8 and Fig. 9.

References

[1] Vector 35. Binary Ninja, 2015. URL https://binary.ninja/.

[2] Angr Team. angrop. GitHub repository, March 2013. URL https://github.com/angr/angrop.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 57(2): 74–84, February 2014. ISSN 0001-0782. doi: 10.1145/2560217.2560219. URL https://doi.org/10.1145/2560217.2560219.

[4] Jacob Baines. Mikrotik FOISted revisited. VulnCheck blog, 2023. URL https://vulncheck.com/blog/mikrotik-foisted-revisited.

[5] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai 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)*, pages 30–40. ACM, March 2011.

[6] Erik Bosman and Herbert Bos. Framing signals – A return to portable shellcode. In *Proceeding of the IEEE Symposium on Security and Privacy, (SP)*, pages 243–258. IEEE Computer Society, May 2014.

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, pages 27–38. ACM, October 2008.

[8] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (SP)*, pages 380–394. IEEE Computer Society, May 2012.

[9] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, (CCS)*, pages 559–572. ACM, October 2010.

[10] Ping Chen, Xiao Xing, Bing Mao, Li Xie, Xiaobin Shen, and Xinchun Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 20–29. ACM, 2011.

[11] Brian Chess and Jacob West. Dynamic taint propagation: Finding vulnerabilities without attacking. *Inf. Secur. Tech. Rep.*, 13(1):33–39, 2008. doi: 10.1016/J.ISTR.2008.02.003. URL https://doi.org/10.1016/j.istr.2008.02.003.

[12] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. RiscyROP: Automated return-oriented programming attacks on RISC-V and

ARM64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 30–42. ACM, October 2022.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. ISSN 0164-0925.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 337–340. Springer, March 2008.

[15] OPTEE Development Team. OP-TEE-2022-0002. Available from OP-TEE security advisories on GitHub, OP-TEE-ID OP-TEE-2022-0002, 2022. URL https://github.com/OP-TEE/optee_os/security/advisories/GHSA-65w8-6mrg-52g7.

[16] Shruti Dixit, T K Geethna, Swaminathan Jayaraman, and Vipin Pavithran. Angerza: Automated exploit generation. In *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pages 1–6, 2021. doi: 10.1109/ICCCNT51525.2021.9579959.

[17] Exploit DB. EDB-ID-47122. Available on Exploit-DB, EDB-ID-47122, 2019. URL https://www.exploit-db.com/exploits/47122.

[18] Benjamin Farinier, Sébastien Bardin, Richard Bonichon, and Marie-Laure Potet. Model generation for quantified formulas: A taint-based approach. In *Proceedings of the 30th International Conference on Computer Aided Verification (CAV)*, volume 10982 of *LNCS*, pages 294–313. Springer, July 2018.

[19] Benjamin Farinier, Robin David, Sébastien Bardin, and Matthieu Lemerre. Arrays made simpler: An efficient, scalable and thorough preprocessing. In Gilles Barthe, Geoff Sutcliffe, and Margus Veanes, editors, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, volume 57 of *EPiC Series in Computing*, pages 363–380. EasyChair, 2018. doi: 10.29007/dc9b. URL https://doi.org/10.29007/dc9b.

[20] Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015.

[21] Lucas Georges, Julien Boutet, and Thomas Chauchefoin. Your vulnerability is in another OEM. Synacktiv blog, 2023. URL https://www.synacktiv.com/publications/your-vulnerability-is-in-another-oem.

[22] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not all bugs are created equal, but robust reachability can tell the difference. In *Proceedings of the 33rd International Conference on Computer Aided Verification (CAV)*, volume 12759 of *LNCS*, pages 669–693. Springer, 2021.

[23] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 62–73, June 2011.

[24] Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 768 of *LNCS*, pages 477–499. Springer, August 1993. ISBN 978-3-540-57659-4.

[25] Sean Heelan, Tom Melham, and Daniel Kroening. Automatic heap layout manipulation for exploitation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 763–779, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/heelan.

[26] J. Heng. CVE-2016-10190 Detailed Writeup. Nandy Narwhals CTF Team blog, 2017. URL https://nandynarwhals.org/cve-2016-10190/.

[27] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block oriented programming: Automating data-only attacks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, (CCS)*, pages 1868–1882. ACM, October 2018.

[28] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Data-Only attack generation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1401–1418, Philadelphia, PA, August 2024. USENIX Association. ISBN 978-1-939133-44-1. URL https://www.usenix.org/conference/usenixsecurity24/presentation/johannesmeyer.

[29] James C. King. Symbolic execution and program testing. *Journal of Communications of the ACM*, 19(7):385–394, 1976.

[30] Linaro Limited. Open portable trusted execution environment, 2024. URL https://op-tee.org.

[31] Carl Livitt and Alumnus. Creating an Exploit: Solarwinds Vulnerability CVE-2021-35211. Bishop Fox Blog, 2021. URL https://bishopfox.com/blog/exploit-for-cve-2021-35211.

[32] Carl Livitt and Jon Williams. A More Complete Exploit for Fortinet CVE-2022-42475. Bishop Fox Blog, 2023. URL https://bishopfox.com/blog/exploit-cve-2022-42475. [Accessed 13-08-2024].

[33] Hector Marco-Gisbert and Ismael Ripoll. Return-to-CSU: A New Method to Bypass 64-bit Linux ASLR. In *Black Hat Asia 2018*. Black Hat, March 2018.

[34] MarginResearch. FOISted. Available on GitHub, 2023. URL https://github.com/MarginResearch/FOISted.

[35] Boyan Milanov. Ropium. GitHub repository, February 2021. URL https://github.com/Boyan-MILANOV/ropium.

[36] Matt Miller. Trends and challenges in the vulnerability mitigation landscape. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, 2019.

[37] MITRE. CVE-2018-11529. Available from MITRE, CVE-ID CVE-2018-11529, 2018. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-11529.

[38] MITRE. CVE-2019-1010298. Available from MITRE, CVE-ID CVE-2019-1010298, 2019. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1010298.

[39] MITRE. CVE-2022-46152. Available from MITRE, CVE-ID CVE-2022-46152, 2022. URL http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-46152.

[40] Anh Quynh Nguyen. Optirop: Hunting for rop gadgets in style. Slideshow presented at BlackHat USA, 2013.

[41] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0. *Journal on Satisfiability, Boolean Modeling and Computation*, 9(1):53–58, 2014.

[42] Alexander Peslya. Getting around non-executable stack (and fix). Bugtraq Forum, August 1997. URL https://seclists.org/bugtraq/1997/Aug/63.

[43] Jannik Pewny, Philipp Koppe, and Thorsten Holz. Steroids for doped applications: A compiler for automated data-oriented programming. In *2019 IEEE European Symposium on Security and Privacy (EuroS& P)*, pages 111–126, 2019. doi: 10.1109/EuroSP.2019.00018.

[44] Jonathan Salwan and Alexey Vishnyakov. Ropgadget. GitHub repository, March 2023. URL https://github.com/JonathanSalwan/ROPgadget.

[45] Sascha Schirra. Ropper. GitHub repository, April 2023. URL https://github.com/sashs/Ropper.

[46] Moritz Schloegel, Tim Blazytko, Julius Basler, Fabian Hemmer, and Thorsten Holz. Towards automating code-reuse attacks using synthesized gadget chains. In *Proceedings of the 26th European Symposium on Research in Computer Security (ESORICS)*, pages 218–239. Springer, October 2021. ISBN 978-3-030-88417-8.

[47] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association, August 2011.

[48] Edward J. Schwartz, Cory F. Cohen, Jeffrey Gennari, and Stephanie M. Schwartz. A generic technique for automatically finding defense-aware code reuse attacks. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*, pages 1789–1801. ACM, November 2020.

[49] Hovav 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 (CCS)*, pages 552–561. ACM, October 2007.

[50] Carsten Sinz, Stephan Falke, and Florian Merz. A precise memory model for Low-Level bounded model checking. In *5th International Workshop on Systems Software Verification (SSV 10)*, Vancouver, BC, October 2010. USENIX Association. URL https://www.usenix.org/conference/ssv10/precise-memory-model-low-level-bounded-model-checking.

[51] Steven Steeley. Foxes Among Us: Foxit Reader Vulnerability Discovery and Exploitation. Available on Source Incite blog, 2018. URL https://srcincite.io/blog/2018/06/22/foxes-among-us-foxit-reader-vulnerability-discovery-and-exploitation.html.

[52] Bishop Fox Team. Building an Exploit for FortiGate Vulnerability CVE-2023-27997. Bishop Fox Blog, 2023. URL https://bishopfox.com/blog/building-exploit-fortigate-vulnerability-cve-2023-27997. [Accessed 13-08-2024].

[53] Cesare Tinelli and Clark Barrett. SMT-LIB: The satisfiability modulo theories library, 2010.

[54] Peng Tu and David Padua. Efficient building and placing of gating Functions. *Journal of ACM SIGPLAN Notices*, 30(6):47–55, June 1995. ISSN 0362-1340.

[55] Victor van der Veen, Nitish dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In Davide Balzarotti, Salvatore J. Stolfo, and Marco Cova, editors, *Research in Attacks, Intrusions, and Defenses - 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings*, volume 7462 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2012. doi: 10.1007/978-3-642-33338-5_5. URL https://doi.org/10.1007/978-3-642-33338-5_5.

[56] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrida. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1675–1689, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349468. doi: 10.1145/3133956.3134026. URL https://doi.org/10.1145/3133956.3134026.

[57] Alexey Vishnyakov and Alexey Nurmukhametov. Survey of methods for automated code-reuse exploit generation. *ACM Journal of Programming and Computing Software*, 47(4):271–297, 2021.

[58] Yan Wang, Chao Zhang, Zixuan Zhao, Bolun Zhang, Xiaorui Gong, and Wei Zou. MAZE: Towards automated heap feng shui. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1647–1664. USENIX Association, August 2021. ISBN 978-1-939133-24-3. URL https://www.usenix.org/conference/usenixsecurity21/presentation/wang-yan.

[59] Patrick Wollgast, Robert Gawlik, Behrad Garmany, Benjamin Kollenda, and Thorsten Holz. Automated multi-architectural discovery of CFI-resistant code gadgets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS)*, volume 9878 of *LNCS*, pages 602–620. Springer, September 2016.

[60] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel Use-After-Free vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 781–797, Baltimore, MD, August 2018. USENIX Association. ISBN 978-1-939133-04-5. URL https://www.usenix.org/conference/usenixsecurity18/presentation/wu-wei.

A Example: Assumptions and Specifications Encoding for CVE-2022-46152

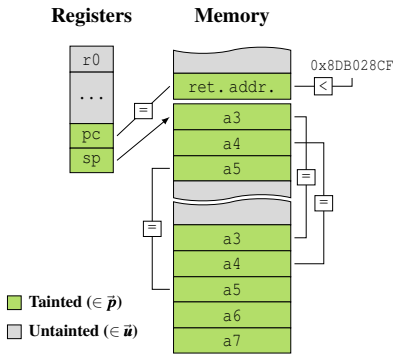


Figure 10: Assumptions encoding for CVE-2022-46152.

Fig. 10 illustrates the encoding of the assumptions for CVE-2022-46152 – see Section 4.3. All values controlled by the attacker are encoded as tainted in $\mathcal{A}^\bullet$. The rest of memory is indicated as untainted by default. Similarly, `pc` is also tainted since the overwritten return address is popped into it. Regarding the stack-pointer `sp`, it is not attacker-controlled, but it points to attacker-controlled data. Thus, it is predictable, hence tainted. Additionally, constraints on values are encoded in $\mathcal{A}$. We indicate that the overwritten address must be less than `0x8DF028CF`. We also encode that all locations where a same argument is stored must have the same value.

Regarding the specifications $\mathcal{S}$, let us consider that we want to execute a `memcpy` call. Appropriate registers must be set for the arguments – `r0`, `r1`, `r2` – and the *program-counter* must point to `memcpy`. In the end, we would set: $\mathcal{S}^\bullet \triangleq \texttt{r0}^\bullet \wedge \texttt{r1}^\bullet \wedge \texttt{r2}^\bullet \wedge \texttt{pc}^\bullet$ – i.e. eventually, `r0`, `r1`, `r2` and `pc` must be predictable. Regarding the values, we would have: $\mathcal{S} \triangleq (\texttt{r0} = \texttt{dst}) \wedge (\texttt{r1} = \texttt{src}) \wedge (\texttt{r2} = \texttt{len}) \wedge (\texttt{pc} = \texttt{@memcpy})$.

B Absorbing Elements

Table 4 gives a list of notable bit-vector operations, together with their absorbing element and taint propagation formula. We consider the bit-vectors to be of size k. We note 0_k (resp. 1_k) the bit-vector of size k, whose bits are all set to 0 (resp. 1).

C Incremental Synthesis Algorithm

Algorithm 1 shows how we incrementally search for chains up to n gadgets. ASSERT is the solver's function to add constraints. NEWSMTSTATE creates new SMT variables to represent a state. CHECKSATASSUMING is the solver's function to check the satisfiability of a formula against the constraints we have already asserted. Finally, GETMODEL retrieves the solution.

Table 4: List of absorbing elements and taint propagation formulas for various operation on bit-vectors of size k.

Operation	Operator	Absorbing Element	Taint Propagation Formula
$c \leftarrow a \,\&\, b$	*bit-wise and*	0_k	$c^\bullet \triangleq (a^\bullet \wedge b^\bullet) \vee (a^\bullet \wedge a^\star = 0_k) \vee (b^\bullet \wedge b^\star = 0_k)$
$c \leftarrow a \mid b$	*bit-wise or*	1_k	$c^\bullet \triangleq (a^\bullet \wedge b^\bullet) \vee (a^\bullet \wedge a^\star = 1_k) \vee (b^\bullet \wedge b^\star = 1_k)$
$c \leftarrow a * b$	*product*	0_k	$c^\bullet \triangleq (a^\bullet \wedge b^\bullet) \vee (a^\bullet \wedge a^\star = 0_k) \vee (b^\bullet \wedge b^\star = 0_k)$
$c \leftarrow a \ll b$	*logical shift left* of b bits	k	$c^\bullet \triangleq (a^\bullet \wedge b^\bullet) \vee (b^\bullet \wedge b^\star \geq k)$
$c \leftarrow a \gg b$	*logical shift right* of b bits		
$c \leftarrow a \leq b$	*unsigned comparison*	0_k for a, 1_k for b	$c^\bullet \triangleq (a^\bullet \wedge b^\bullet) \vee (a^\bullet \wedge a^\star = 0_k) \vee (b^\bullet \wedge b^\star = 1_k)$

Algorithm 1 Incremental synthesis of chains up to size n.

```
s_0 ← NEWSMTSTATE                          ▷ define the initial state
ASSERT(A(s_0))                             ▷ assert the assumptions A on s_0

for i = 1, ..., n do
    s_i ← NEWSMTSTATE                      ▷ define a new state
    ASSERT(s_{i-1} → s_i)                  ▷ assert s_{i-1} transitions to s_i
    result ← CHECKSATASSUMING(S(s_i))
    if result = SAT then                   ▷ check if s_i can fulfill S
        return GETMODEL                    ▷ solver found a chain
return ∅                                   ▷ no chain found
```

D Remarkable Pivot Pattern

When analyzing the generated chains in Section 8.6.2, we found out that most followed a common pattern. This pattern is composed of two gadgets relying on `rsi`, and present in almost all binaries. When `rsi` points to attacker-controlled areas, it takes control of `rsp` by pushing `rsi` and popping it:

```
1 push rsi; ...; jmp [rsi+...]    push rsi on stack
2 pop  rsp; ...; ret                pop it into rsp
```

For `nginx` and `httpd`, when `rsi` was not initially controlled, ARCANIST found additional gadgets that enabled taking control of `rsi`, and then leveraged the pattern introduced above. We believe that the presence of this pattern in generated chains is highly tied to the low diversity of gadgets allowing to overwrite `rsp`. We manually analyzed the gadget collections of each binary after our experiments. Most of the gadgets we found for overwriting `rsp` relied on this `pop rsp` instruction.