ɪTOP: Automating Counterfeit Object-Oriented Programming Attacks

Paul Muntean
Technical University of Munich
paul.muntean@sec.in.tum.de

Richard Viehoever
Technical University of Munich
richard.viehoever@tum.de

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

Gang Tan
Penn State University
gtan@psu.edu

Jens Grossklags
Technical University of Munich
jens.grossklags@in.tum.de

Claudia Eckert
Technical University of Munich
claudia.eckert@sec.in.tum.de

ABSTRACT

Exploiting a program requires a security analyst to manipulate data in program memory with the goal to obtain control over the program counter and to escalate privileges. However, this is a tedious and lengthy process as: (1) the analyst has to *massage* program data such that a logical reliable data passing chain can be established, and (2) depending on the attacker goal certain in-place fine-grained protection mechanisms need to be bypassed. Previous work has proposed various techniques to facilitate exploit development. Unfortunately, none of them can be easily used to address the given challenges. This is due to the fact that data in memory is difficult to be *massaged* by an analyst who does not know the peculiarities of the program as the attack specification is most of the time only textually available, and not automated at all.

In this paper, we present indirect transfer oriented programming (ɪTOP), a framework to automate the construction of control-flow hijacking attacks in the presence of strong protections including control flow integrity, data execution prevention, and stack canaries. Given a vulnerable program, ɪTOP automatically builds an exploit payload with a chain of viable gadgets with solved SMT-based memory constraints. One salient feature of ɪTOP is that it contains 13 attack primitives powered by a Turing complete payload specification language, ESL. It also combines virtual and non-virtual gadgets using COOP-like dispatchers. As such, when searching for gadget chains, ɪTOP can respect, for example, a previously enforced CFI policy, by using only legitimate control flow transfers. We have evaluated ɪTOP with a variety of programs and demonstrated that it can successfully generate working exploits with the developed attack primitives.

CCS CONCEPTS

• **Security and privacy** → **Systems security**; **Information flow control**; **Software security engineering**; **Software reverse engineering**.

KEYWORDS

Machine code, control flow integrity, Clang/LLVM, cyber attacks.

ACM Reference Format:
Paul Muntean, Richard Viehoever, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2021. ɪTOP: Automating Counterfeit Object-Oriented Programming Attacks. In *RAID '21: International Symposium on Research in Attacks, Intrusions and Defenses, Oct. 06–08, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3471621.3471847

1 INTRODUCTION

Large software (*e.g.,* Web browsers) is buggy due to its enormous complexity. Among all bugs, exploitable bugs (*i.e.,* bugs giving attackers an advantage) are the most dangerous ones. Being able to understand exploitability of bugs and to triage particularly dangerous ones is highly desirable. However, this is often a tedious and labor-intensive process. Major vendors (*e.g.,* Google [19], Apple [5], and Microsoft [28]) heavily rely on their bug bounty programs and highly specialized internal teams (*e.g.,* Google's Project Zero [20]). This situation is due to the facts that (1) given an exploitable vulnerability, the security analyst has to manually manipulate data in program memory in order to pass data around to achieve their goals; and (2) given the wide deployment of defenses such as control-flow integrity (CFI) *i.e.,* at binary [31] and source code [33] level, data execution prevention (DEP) [29], and stack canaries, the analysts needs to make sure that the exploit can bypass the defenses, *e.g.,* they are respecting the enforced control-flow graph (CFG) based policies.

While there are considerable efforts going on to automate exploit generation, every one fills a different niche and only partially addresses the needs of security analysts. Specifically, AEG [6] was the first tool to automatically search for an exploitable program vulnerability [34] and to generate a control-flow hijacking attack. Revery [49] is an extension of AEG that addressed additional challenges. However, it can only automatically create return-to-stack and return-to-libc exploits. Newton [48] is a runtime-based attack crafting tool which breaks forward-edge CFI during exploit generation, relying on run-time information such as values in registers and memory addresses to identify the appropriate gadgets and to stitch them together. Further, the attacker has to interact during attack creation within the attack specification language with program memory constraints, which significantly raises the entry bar for this tool's usage. BOPC [24] is a data-only attack generation tool, which searches the target program's basic blocks in the CFG for valid program traces fulfilling the attacker's goal. While it provides attack primitives, it does not offer programmable interfaces such as APIs for payload construction. Further, neither of the data-only nor control-flow hijacking tools show in their evaluations that these tools can evade state-or-the-art security defenses. In summary, there is a need to develop a tool based on static analysis (*i.e.,* without using runtime information) as most currently existing approaches are runtime-based. The tool should be programmable, and automatically generate exploits bypassing fine-grained forward-edge CFI (*i.e.,* context-insensitive CFI) to craft control-flow hijacking attacks.

In this paper, we present Indirect Transfer Oriented Programming (ɪTOP), a COOP [43] attack construction framework for automatic development of control-flow hijacking attacks. ɪTOP provides an extensible, Turing-complete language—in order to be able to encode all types of program control flows—called Exploit Specification Language (ESL). The novelty of ESL consists in the fact that it is not a monolithic language but rather relies on an extensible API which can be easily tailored for different types of attacks. ESL is motivated by the need for a light-weight and extensible language (its compact grammar will be introduced later), which lowers the hurdle for entry for the analyst; it has an extensible API and is based on ANTLR [3], enabling automated lexer and parser generation out of the box. The extensible API allows for extending (1) the gadget set by considering other target types (*e.g.,* basic blocks, non-virtual functions), and (2) the attack type by adding more attack crafting primitives.

Given a vulnerable program, an ESL attack specification, the source code of the application, and the modeled deployed CFI defense, ɪTOP is able to build control-flow hijacking attacks that bypass CFI. The final result of the ɪTOP analysis pipeline is a payload file, which if fed into the vulnerable program buffer, allows to perform the attack. The key novelty of this work is an end-to-end framework for crafting control-flow hijacking attacks under strong deployed CFI policies, improving the precision of prior analysis with a full-fledged symbolic execution. Also, unlike prior works, it has a novel gadget search algorithm that can be operated under fine-grained CFI defenses. At a high level, ɪTOP also requires the address of an attacker-controlled buffer as input, the Z3 [30] solver, ANGR [46], and symbolic execution to generate the attack payload. The generated payload can bypass state-of-the-art CFI defenses due to found usable calltargets which were previously identified. ɪTOP provides CFI policy modeling, integration, and attack construction guidance based on the currently selected CFI policy.

There are multiple challenges that are addressed by ɪTOP in order to automate the exploit generation. In particular, ɪTOP must resolve the abstract registers to match the correct registers of the underlying architecture. Meanwhile, the analysis has to determine which callsite gadgets are compatible to which calltarget gadgets, because of COOP attack requirements. Also, identifying whether a gadget is usable within a gadget chain using symbolic execution is time-consuming, and we have to improve its performance. In addition, ɪTOP has to construct the payload that leads to execution of the specified ESL program, solve gadget memory overlaps and resolve pointer destinations. Lastly, the attack has to bypass modern defenses such as, DEP [29], shadow stack [16], and deployed fine-grained CFI policies [32]. We have addressed these challenges and implemented ɪTOP.

To evaluate ɪTOP, we have conducted experiments with several real-world programs, and demonstrated that ɪTOP can generate code reuse attack payloads for a set of 13 test payload specifications. These payloads demonstrate the capabilities of ɪTOP and ESL (*e.g.,* loops, conditional branches, memory reads/writes) and the applicability to real-world attacks (*e.g.,* spawning a shell with attacker-controlled parameters). As opposed to common belief that high percentage values (above or around 90%) in the realm of automated attack construction are the key indicator for tool potential (or tool quality), we rather think that this is not the most essential indicator. As such, successful attack crafting depends primarily on the types of gadgets used, thus it is less significant to have many attack variations for a single target program when one attack is sufficient to compromise the whole system.

ɪTOP is, to our knowledge, the first automated static control-flow hijacking attack crafting tool that is aware of the deployed CFI defense. Based on a NodeJS use case, we show that ɪTOP can generate attacks even in the presence of strong CFI defenses. In contrast to other attack crafting tools (*e.g.,* Newton, BOPC, AEG, Revery), ɪTOP demonstrates that it can build attacks under strong real-world CFI defenses. Additionally, note that the applications of ɪTOP go beyond an attack framework. We envision ɪTOP as a tool for defenders to evaluate the residual attack surface of a program usable after a defense was deployed, enabling analysts to assess, for example, whether a certain CFI policy sufficiently protects the program, and which parts of a program require additional attention to stop them from enabling attacks.

In summary, we make the following contributions:

- **Gadget Search and Chain Building Framework.** We propose ɪTOP[1], a novel framework that generates gadget chains and validates their feasibility fully automatically. ɪTOP translates an ESL payload into a full-fledged control-flow hijacking attack for the target vulnerable binary and enables the discovery of viable gadgets and chains though an efficient search process.
- **Attack Specification Abstraction.** We propose ESL, a system independent attack payload specification language, which is based on a powerful and extensible library of predefined gadgets. The API is used to extend the ESL functionalities. ESL enables the necessary abstraction to scale to large programs and to be used by an analyst with no previous attack construction experience.
- **Attack Construction under Strong Defenses.** We propose an approach to illustrate that ɪTOP can run in the presence of strong defenses such as CFI, DEP, and Clang's shadow stack techniques [10]. We also show how much attack surface reduction is required to effectively defend against these attacks.
- **Experimental Results.** We evaluated ɪTOP by showing the generality of its techniques based on existing vulnerabilities where manual exploit construction may have been infeasible or very tedious. With the primitives provided by ɪTOP, we can successfully generate a payload to trigger a significant compromise (a) by first writing an exploit[2] for *all* target programs, and (b) second by spawning a system shell[3].

2 BACKGROUND

Control Flow Bending (CFB). CFB [9] attacks demonstrate that CRAs are still possible under strong CFI policies. By restricting

[1]ɪTOP's source code on GitHub. https://tinyurl.com/y8bnsk6w
[2]Video: exploit writing in ESL. https://tinyurl.com/y6cmbvyt
[3]Video: spawning a system shell exploit. https://tinyurl.com/y6a9gk7c

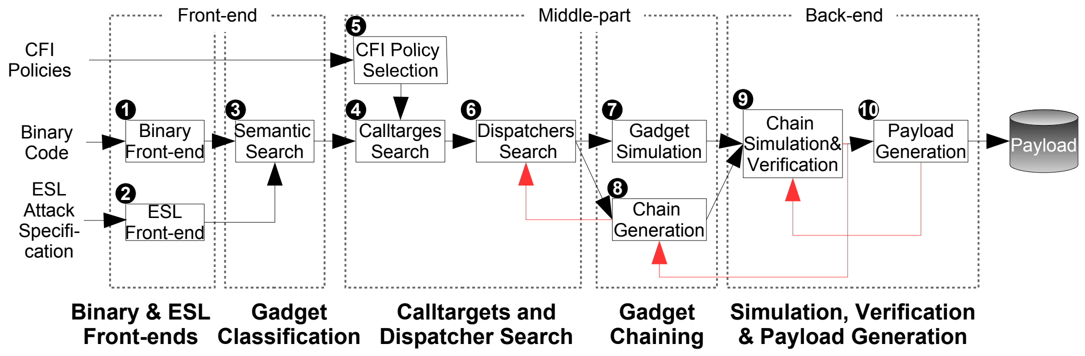

Figure 1: Attack generation pipeline. Black arrows represent data transfers while red arrows indicate analysis backtrack steps.

exploit execution traces within the legitimate CFG, control flow bending attacks avoid CFI defenses entirely. While the resulting exploit might not follow a valid non-exploit trace, every step follows legal control flow transfers. If an attacker modifies a function pointer to point to a different, yet still legal calltarget, he is performing a CFB attack. In a nutshell, CFB attacks demonstrate that an attacker, with precise knowledge of valid calltarget sets under specific CFI policies, can still construct an attack as long as the target set contains the necessary CRA gadgets.

Counterfeit Object-oriented Programming (COOP). COOP attacks [43] are a CRA technique which exploits C++ virtual function dispatches to generate an attack. COOP uses a main loop dispatcher gadget, which may be one of the following: (1) a function iterating over an array of objects and invoking a virtual function on each main loop dispatcher, (2) a function going through a linked list of objects invoking a virtual function for each LinkedList dispatcher, or (3) a virtual function invoking two virtual functions with two objects, which is leveraged by building a recursion-based dispatcher. This is done by having the second object be a pointer to the dispatcher itself (Recursion dispatcher). Using these three dispatcher types, virtual functions are chained together to create an attack. Lastly, COOP attacks respect backwards return edges (stack discipline), making it *immune* to shadow stack protection techniques.

3 THREAT MODEL AND ASSUMPTIONS

In this section, we present the defensive assumptions and attacker capabilities of our threat model (*e.g.,* STRIDE [27]).

Defensive Assumptions.

- *Writable ⊕ Executable Memory:* The target system ensures that memory can be either writable or executable, but never both at the same time through, *e.g.,* DEP [29]. This prevents attackers from injecting new code or modifying existing code.
- *Control Flow Protection:* We assume that the vulnerable application is protected by state-of-the-art control flow protection techniques and an efficient shadow stack implementation.
- *DEP, ASLR, Stack Canaries (DAS):* We assume that DAS [11] are deployed, correctly configured and active in the target system.

Attacker Capabilities.

- *System Configuration:* The attacker is aware of the target system configuration and applied defenses, and she has access to a binary of the target application that is not re-randomized at short intervals.
- *Vulnerability:* The target application suffers from a known memory corruption vulnerability, which can be leveraged by the attacker to write arbitrary values into writable memory.
- *Information Leak (IL):* To bypass ASLR, the attacker has access to an IL, which allows her to read arbitrary values. Note that the IL primitive is orthogonal to the program vulnerability requirement and necessary to bypass ASLR. Also note that the IL primitive can be achieved by using memory management side channels [21], or using hardware side channels [22, 25], *etc.*
- *Application Entry Point (AEP):* We assume that the attacker has access to an AEP, which is part of the vulnerability discovery phase. AEP is a point of failure that might enable a malicious actor (*i.e.,* attacker) to break the application and cause damage [40]. Note that the entry point is not the program `main()` function. The discovery of the AEP is orthogonal to our work and can be, for example, addressed with the plethora of available fuzzing tools or by using an already known and well-documented vulnerability tool in the style of Metasploit [26].
- *Read Write Operations:* We assume that one or more read/write operations are needed to modify the state of the target binary in order to initiate the attack. These operations are part of the attack payload, which together with the attack assembly operations facilitate successful payload execution. Note that a write operation does not necessarily need to be proceeded by a read operation.

4 SYSTEM OVERVIEW

Figure 1 provides an overview of ɪTOP's attack building pipeline. Numbers from one to ten denote the different components and operations of ɪTOP analysis, and three inputs: the CFI policies, the program binary, and the ESL attack specification. Further, the whole ɪTOP analysis is broken down into five stages for payload construction. These stages are: binary and ESL front-ends, gadget classification, calltargets and dispatchers gadget search, gadget chaining, as well as simulation, verification and payload generation.

Specifically, as depicted in Figure 1 from left to right, in ❶ the target program binary is passed into ɪTOP's binary front-end. Next, the address and size of the attacker-controlled buffer is indicated in the ESL attack specification. In ❷ the ESL payload specification is passed into iTOP's ESL front-end. A set of CFI policies is then provided. Each of the provided CFI policies is optional and is used to filter the results when analyzing the target binary during attack construction. Further, the target binary is first lifted into PyVEX intermediate representation (IR) [45] with the help of ANGR [46], while the ESL payload is translated into our own Python object-based IR. In ❸ based on the ESL specification a semantic search is performed. In ❹ iTOP searches for promising caltarget gadgets in order to build the callsite/calltarget matching pairs sets. In ❺ the CFI policies for the vulnerable program are modeled. Note that this step is an optional and iTOP still works without it.

Further, in ❻ iTOP searches for dispatcher gadgets for constructing the matching callsite/calltarget pairs. After this step, in ❼ the gadgets are simulated in order to verify their functionality. In parallel to this step, in ❽ the gadget chain is generated. After these two parallel steps, in ❾ the gadget chain is simulated and verified. Further, in ❿ the payload is generated if the previous two steps succeed. If ❾ or ❿ fails, iTOP backtracks (see red arrows) to the previous step and tries a different gadget chain or gadget combination. When both of these two steps succeed, the payload is generated and emitted as a distinct file. Lastly, note that the payload file needs to be written to the attacker-controlled buffer to initiate the attack.

5 DESIGN AND IMPLEMENTATION

We designed and implemented our automated attack construction approach inside the iTOP tool, which consists of 10 K Python LoC.

5.1 iTOP Front-ends

In this section, we present iTOP's front-ends and introduce several ESL based attack construction examples.

Binary Front-end. iTOP uses the ANGR [46] binary analysis framework to lift the target binary into an intermediate representation, in order to perform control flow analysis and symbolic execution. To access the binary's machine code, we use the Capstone [8] binary analysis framework. DWARF debug information, if present, is extracted from the binary using pyelftools [1]. Further, the binary front-end depicted in Figure 1 at step ❶ builds a set of candidate gadgets, C_G. Depending on the calltarget constraints, this set can contain all functions, all virtual functions, or all functions reachable from any dispatcher gadget under the provided CFI policy. If required, iTOP generates a mapping of files and line numbers to addresses, a mapping of human readable names to addresses, and an entry state of the target application.

ESL Front-end. The ESL front-end as depicted in Figure 1 at step ❷ creates an intermediate representation consisting of a tree of objects for the payload using the ANTLR [3] parser generator. For each statement, unique hash encoding preconditions, effects, and postconditions are calculated, enabling the search algorithm to later recognize repeat gadgets as well as gadgets that were already discovered in different chains. This information, IR_{py}, is then passed to the candidate gadget classification module.

Payload Specification in ESL. To ease the process of payload creation, we built ESL, as presented in Figure 1. ESL allows precise definition of exploits in a powerful language with fours features: (1)

```
<exploitlang> ::= <imports> <definitions> <main>?
<imports>     ::= ((<py_import> | <esl_import>) ';')*
<py_import>   ::= 'IMPORT' <gadget_id> '('  <types>? ')' 'RETURNS'
                                        (<type> | 'NONE') 'FROM' <str>
<expl_import> ::= 'IMPORT' <str>
<definitions> ::= <definition>*
<definition>  ::= 'DEF' <gadget_id> '(' <arg_ids> ')  RETURNS'
                                 (<type> | 'NONE') '{' <statements> '}'
<arg_ids>    ::= <arg_id> (',' <arg_id>)*
<main>       ::= <gadget_id> '{' <statements> '}'
<statements> ::= ((<statement> ';') | (<assert> ';')
                                        | (<jump> ';')
                                        | (<if_stmt> ';')
                                        | (<label> ':') )*
<statement>  ::= <gadget> | <assignment>
<assignment> ::= ( <type> <arg_id> '=' (<str> | <int>) )
                                   | ('reg' <reg_id> '=' <gadget>)
<gadget>     ::= <gadget_id> '(' <arguments>? ')'
<arguments>  ::= <argument> (',' <argument>)*
<argument>   ::= '&'? <arg_id> | <reg_id>
<assert>     ::= 'ASSERT' <condition>
<jump>       ::= 'GOTO' <label_id>
<if_stmt>    ::= 'IF' <condition> 'GOTO' <label_id>
<label>      ::= <label_id> ;
<condition>  ::= <reg_id> <cmp_op> (<int> | <reg_id>
                                          | <arg_id>
                                          | <str>)
<cmp_op>     ::= '<' | '>' | '==' | '>=' | '<='
<types>      ::= <type> (',' <type>)*
<type>       ::= 'string' | 'int' | 'reg'
<reg_id>     ::= '_r' [0-7]
<gadget_id>  ::= [A-Z_0-9]+
<arg_id>     ::= [a-z]+
<str>        ::= '"' ~('\r' | '\n' | '"')* '"'
<int>        ::= [0-9]+ | '0x'[0-9a-fA-F]+
<label_id>   ::= '_'[a-z]+
```

Listing 1: Extended Backus-Naur form of ESL.

Turing-completeness (control flow via loops and branches); (2) independence from target binary and architecture; (3) close mirroring of actual attack techniques; (4) a set of powerful and extensible Python-like API for attack construction using primitives and gadgets.

```
 1 /*import a part of the API functionality*/
 2 IMPORT MAINLOOP () RETURNS NONE FROM "mainloop.py";
 3 IMPORT LINKEDLIST () RETURNS NONE FROM "linkedlist.py";
 4 IMPORT RECURSION () RETURNS NONE FROM "loopless.py";
 5 IMPORT READ () RETURNS int FROM "read.py";
 6 IMPORT LOAD (int) RETURNS int FROM "load.py";
 7 IMPORT WRITE (int, int) RETURNS NONE FROM "write.py" ;
 8 IMPORT EXECUTE (int, int) RETURNS  NONE FROM "execute.py";
 9 IMPORT MANIPULATE (reg) RETURNS NONE FROM "manipulate.py";
10 IMPORT INC (reg) RETURNS NONE FROM "inc.py";
11 IMPORT IF_FN () RETURNS NONE FROM "if.py";
12 IMPORT STACKINC () RETURNS NONE FROM "stackinc.py";
13 IMPORT STACKDEC () RETURNS NONE FROM "stackdec.py";
14 /*import available dispatcher gadget types*/
15 IMPORT DISPATCHER () RETURNS NONE FROM "linkedlist.py";
16 IMPORT DISPATCHER () RETURNS NONE FROM "mainloop.py";
17 IMPORT DISPATCHER () RETURNS NONE FROM "loopless.py";
18 /*import read and execute functionality*/
19 IMPORT "esl_scripts/lib_read.esl";
20 IMPORT "esl_scripts/lib_execute.esl";
```

Listing 2: Extensible lib_coop.esl API.

To illustrate our extensible attack construction APIs[4], we provide an example as shown in Listing 2. Our ESL can be used to import different high-level Python functions which represent the main API capabilities. For example, in line 2 we import the MAINLOOP gadget features. Further, our API is based on our own domain specific language (DSL), dubbed ESL. It is easy to use as it is based on a simple DSL, which does not impose a high burden on the analyst. It is also flexible as the analyst can anytime go and define new functionality;

[4]ESL's extensible API. The names of the imported Python script files indicate the functionality of each of the imported API functions. https://tinyurl.com/ya9qzgmo

as, for example, in `read.py` depicted on line 5 in Listing 2. The `RETURN NONE FROM` is equivalent to the `return none` statement in Python. Note that this main API file will be imported by subsequent ESL attack construction scripts.

In order to craft an attack, ESL closely resembles how the actual attack might play out. A single statement corresponds to a single gadget, and the statements are linked together by a dispatcher gadget mechanism (*e.g.,* COOP). This ensures that an analyst can (1) precisely control the resulting payload's layout, (2) easily and intuitively craft new attacks, and (3) precisely evaluate how a countermeasure interacts with a certain attack layout. As mentioned, ESL offers a Python API containing functions and data structures that can be used to extend the set of gadgets and attack types. Within this work, we implemented 3 dispatch mechanisms and 9 gadget types, providing a rich toolbox that suffices to construct complex and realistic attacks.

```
1 IMPORT "lib_coop.esl";
2
3 LINKED_LIST_DISPATCH{
4 _loop:
5     reg _r1 = READ();
6     GOTO _loop;
7 }
8
```

```
1 IMPORT "lib_coop.esl";
2
3 LINKED_LIST_DISPATCH{
4  int printf = 0x7ffff784e390;
5  str text = "Hello World!\n";
6  IF _r1 != 0x1234 GOTO _end;
7  EXECUTE(printf, &text);
8 _end:
9 }
```

Listing 3: Left: ESL infinite loop; Right: ESL cond. branch.

To illustrate how complex and realistic attacks can be constructed, we introduce Listing 3 which shows how a loop and a conditional control flow payload can be expressed in ESL. First, the payloads import the gadget definitions for the COOP attack types (line 1, both listings). Then, the dispatch mechanism to be used is defined (line 3, both listings). The control flow is manipulated using jumps (line 6 left) and conditional statements (line 6, right side) to jump to labels (line 4 left and 8 right). In line 7 right, a gadget is invoked, leading to a call to `printf` with a pointer to the string `Hello World!` in the first argument. Line 5 on the left side invokes a READ gadget, reading a value into a register. This gadget is essentially a filler, as no empty loops are allowed.

```
1 IMPORT "lib_coop.esl";
2 DEF SYSTEM(arg) RETURNS NONE {
3  reg _r1 = READ();
4  MANIPULATE(_r1);
5  ASSERT _r1 == 0x7ffff784e390;
6  EXECUTE(_r1, arg);
7  }
8 DEF SYSTEM(arg) RETURNS NONE {
9  int system = 0x7ffff784e390;
10  EXECUTE(system, arg);
11  }
12 MAINLOOP_DISPATCH {
13  string shell = "/bin/sh\x00";
14  SYSTEM(&shell);
15  }
```

Listing 4: Spawning a system shell in ESL.

To demonstrate how to spawn a system shell, we introduce Listing 4 which shows an ESL-based `system` shell attack specification. Note that the address `0x7ffff784e390` of `system()` is a prerequisite and was found by the analyst with little manual effort. Note that in this example, the COOP gadget set is used (see line 1). This payload demonstrates another important feature, namely it defines multiple equivalent chains. The first chain (line 2 to 7) loads any value into the second (registers start at zero) argument register (mapped to a different register depending on target architecture), then manipulates that value using arithmetic operations to match the address of `system`. The `ASSERT` in line 5 adds a post-condition to the preceding `MANIPULATE` gadget: `_r1` has to be equal to `0x7ffff784e390` after `MANIPULATE` is executed for the manipulate to be considered valid. Note that in case an `ASSERT` condition is not met then the flow cannot continue. Also, note that `_r1` can be any register. The second chain (line 8 to 11), in turn, does not prepare any registers. Further, when generating the payload, ɪTOP first attempts to generate a payload using the second shorter chain, and if unsuccessful the first chain is used. Next, we describe ɪTOP's ESL front-end to understand how ESL specifications are used by ɪTOP.

Conditional Shell Spawning. For example, by using the conditional write from Listing 3 (right side) to target the destination address of the shell-spawning gadget from Listing 4, a payload to conditionally open a shell can be created. This works if the condition is satisfied; otherwise it will crash, because the calltarget of the `EXECUTE` gadget is overwritten with an invalid destination address by the conditional gadget. Thus, in total, the resulting payload contains 4 fake objects: (1) the dispatcher gadget, with an array that points to pointers, (2) the register initialization, (3) the conditional write, and (4) the call to the attacker-controlled target. Lastly, note that ɪTOP needs, in total, 43 minutes to generate the payload for this example.

5.2 Classifying Candidate Gadgets

In this section, we provide several examples on how to spawn a shell, initialize an argument register, and perform conditional memory write operations on real program binaries. Afterwards, we illustrate how to classify dispatcher gadgets as depicted in Figure 1 at step ❸, which consists in first obtaining an overview of the candidate calltarget gadgets as shown in Figure 1 at step ❹, and *CFI Policy Selection* depicted at step ❺. Note that using the fixed memory addresses as depicted in Listing 4 and Listing 5 is consistent with our threat model assumptions as we assume that ASLR was bypassed by using an available information leak. Similar approaches have been followed by [17, 43] as well.

Spawning a Shell Example. To demonstrate how a system shell is spawned at machine code level, we use Table 5 which shows a gadget that spawns a `system` shell [2] by using the `Libc` system library. This gadget is less complex than the dispatcher gadget. The function `system` opens a shell and it passes to it a char pointer in the first parameter. The register `rdi` starts off by pointing into attacker-controlled memory; as such, the attacker can control both the target address (line 6) and the value in the first argument register (line 8) when the target is called (line 9). ɪTOP creates a fake object containing the address of the function `system` in `Libc` at offset `0x38`, the address of the string `/bin/sh/` at offset `0x40`, and the vptr into the virtual table of `node::JSStream` at offset `0x0`. To build the payload, this fake object is combined with the fake object generated for the dispatcher.

```
; rdi contains the this pointer
; of the fake object
mov rbx, rdi
mov rax, qword ptr [rip+0x19fc9e0]
add rax, 0x10
mov qword ptr [rbx], rax
mov rax, qword ptr [rbx+0x38]
test rax, rax
je  0x7ffff6dc8f46
mov rdi, qword ptr [rbx+0x40]
callrax
```

Listing 5: Gadget `node::JSStream::~JSStream()`, calls arbitrary functions with an arbitrary argument in `rdi`.

```
1 ; rdi contains the this pointer
2 ; of the fake object
3 mov rcx, qword ptr [rdi + 0x10]
4 mov rax, qword ptr [rdi + 0x30]
5 sub rcx, qword ptr [rdi + 0x8]
6 sar rcx, 1
7 add rcx, qword ptr [rdi + 0x20]
8 mov rdx, rcx
9 sub rdx, rax
10 jb  0x7ffff69210e2
11 ...
12 mov qword ptr [rdi + 0x20], rcx
13 mov rax, qword ptr [rdi + 0x20]
14 mov qword ptr [rdi + 0x10], rax
15 mov qword ptr [rdi + 0x18], rax
16 xor eax, eax
17 ret
```

Listing 6: Initializing register `rdx` with an arbitrary value.

```
; rdi contains the this pointer
; of the fake object
mov rax, qword ptr [rdi + 0x10]
mov rcx, qword ptr [rdi + 0x18]
sub rcx, rax
cmp rcx, rsi
jae 0x7ffff6a949c3
xor eax, eax
ret
mov qword ptr [rdx], rax
add qword ptr [rdi + 0x10], rsi
mov al, 1
ret
```

Listing 7: Gadget used to write to an arbitrary memory address if register `rsi` is below an attacker-controlled value.

Initializing an Argument Register. In some cases, no gadget controlling both target address and argument value is available. Often, one of the two values has to be passed to the gadget via an argument register. The Linux/x86-64 ABI convention uses the registers `rdi`, `rsi`, `rdx`, `rcx`, `R8` and `R9` for this purpose. These are used for integer and memory addresses while the `XMM0-7` registers are used for floating point arguments. Being able to initialize these registers with arbitrary values makes more candidates usable for `LOAD`s, `WRITE`s, and `EXECUTE`s.

To illustrate how the `rdx` register can be initialized, we Listing 6 showing how the argument register `rdx` is initialized using the function `v8::internal::ExternalTwoByteString Utf16Character Stream::ReadBlock()`. Highlighted in green: register changed from unknown value to value under attacker control. Orange: register under attacker control. Red: Attacker loses control over register. The gadget depicted could be used to initialize `rdx`. iTOP generates a fake object set up in a way that the jump instruction at line 10 is always taken. The register `rdx` is initialized with an attacker-controlled value in line 7, and in line 8 an attacker-controlled value is subtracted. Using symbolic execution and the Z3 solver, iTOP finds appropriate values for the fake object at offsets `0x10`, `0x30` and `0x8`. Lastly, note that the gadget could also be used to initialize `rcx`.

Conditional Memory Write. To demonstrate how to write at an arbitrary address, we present Listing 7 containing a gadget used to write to an arbitrary address. For conditional memory writes, iTOP leverages, for example, the function `v8::internal::Value Deseria lizer :: ReadRawBytes(unsigned long, void const**)`. For this function to serve as a conditional memory write to an attacker-controlled target, register `rdx` has to contain the desired memory write destination. Using the method outlined in Section 5.2, `rdx` can be initialized to hold the desired value. The dispatcher gadget does not modify `rdx` or `rsi`, and thus the resulting payload leads to a memory write only if `rsi` had the desired value before executing the exploit.

Candidate Gadget Classification. ESL payloads are independent from the target binary or architecture. Thus, ESL statements have to be mapped to functions in the actual program. This consists of the following two steps. First, a pre-filtering step based on matching instructions is performed in order to make the set of candidates more manageable and to enable callsite analysis. Secondly, a precise, symbolic execution-based classification step, as described in Section 5.5, is performed.

For the first step, iTOP considers all functions in the target binary, filtering them and assigning potential gadget categories to them using a semantic approach. While this first analysis is rather permissive, frequently marking potential gadgets as usable that cannot really be used, this step is needed: (1) to allow for elimination of dispatcher gadgets with incomplete calltarget sets, and (2) to considerably reduce the time needed for simulation and chaining: 50 gadgets/sec can be checked using semantics vs. 1 gadget/sec using symbolic execution. Next, we illustrate some candidate gadget examples.

Table 1 presents how functions are filtered and grouped into gadget categories. The function's machine instructions are compared to a set of *Semantic Filter Patterns*. An intermediate representation of the function that closely resembles the machine instructions is generated using the Capstone engine. The function has to match at least one of the filters to be considered a potentially usable candidate gadget. To eliminate functions containing unwanted instructions such as `call` or unconstrained `jump` instructions, we use a general blocklist. While the blocklist can be modified for individual gadget categories, we use the same list for all categories not relying on unconstrained calls or jumps. Further, for each type of gadget, iTOP builds a set of candidate functions, F_g. A single function can be a candidate for multiple gadget types. In larger programs, there can be hundreds of thousands of functions that have to be checked; as a result, performance in this step is more important than precision. Lastly, note that whether a gadget is actually usable or not is determined in Section 5.5.

5.3 Searching for Dispatcher Gadgets

In this section, we show how to construct an attack by *tricking* the target program into executing the gadgets defined by the attacker in a specific order. The gadget searching process is depicted in Figure 1 at step ❻. To achieve this, iTOP uses a COOP type dispatch mechanism, such as: (1) a loop calling functions from an array of function pointers, (2) a loop dispatching virtual functions for attacker-controlled objects, or (3) overwriting return addresses on the stack. While we also implemented a proof-of-concept dispatch mechanism

Table 1: Mapping ESL statements to machine code gadgets. Statement: goal description; ESL Representation: ESL specification; Semantic Filter: examples of how gadgets are pre-filtered by matching assembler instructions. Constraints: post-conditions applied to the simulated state of the symbolic execution engine after a gadget has been completely stepped through. reg_α: Target register, C: Constant value, A: Address, R: Register; Example: machine code matching the filter and constraints. The memory address from where data is read, loaded or written into is depicted in the adjacent Machine Code column examples.

Statement	ESL Representation	Semantic Filter (ex.)	Constraints	Machine Code
Register Assignment	LOAD	mov reg_α, C mov reg_α, $[A \cup R]$	reg_α == target	`movzx rsi, 7h`
	READ	mov reg_α, [A]	reg_α == mem[target]	`mov rsi, DWORD PTR [rdi+8]`
Register Modification	MANIPULATE	inc reg_α add reg_α, $C \cup R$	$reg_{\alpha before}$!= reg_α	`inc rsi`
Memory Write	WRITE	mov [A], $C \cup R$	mem[A] == target	`mov [rdi+8] rsi`
Call	EXECUTE	call $C \cup R$ call $[A \cup R]$	regs.rip == target	`call DWORD PTR [rdi]`

Conditional Jump	IF (*cond*) GOTO	test $R \cup C$, $R \cup C$	if cond: regs.rip == target	`test rsi, r8`
				`jnz addr`

based on function pointer arrays, the focus in this work is on COOP and COOP-like attacks (*i.e.,* type (2) above). Further, keep in mind that within a COOP attack, a dispatcher gadget either iterates over an array or linked list of objects and calls a virtual function on each one, or the dispatcher gadget first calls a virtual function and virtually dispatches itself, calling one virtual function per recursion step.

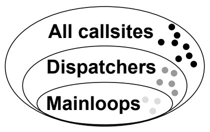

Figure 2: Different types of dispatcher gadgets.

Thus, CFI techniques can limit the set of calltargets a dispatcher gadget can reach.To understand the different types of existing dispatcher gadgets, we use Figure 2 to show the inclusion relationship between different dispatcher gadget types. Note that in general mainloops are a subset of dispatcher gadgets, and dispatcher gadgets are a subset of all callsites. As such, mainloops are a subset of all dispatcher gadgets, fulfilling conditions (1)-(3), while dispatcher gadgets only fulfill conditions (2)-(3). The set of all callsites is even less strict; only condition (2) applies. While identifying the set of all callsites is a simple task, extracting the set of useful dispatcher gadgets requires extensive analysis. The more specific the requirements for a dispatcher gadget are, the less probable it is that a compatible dispatcher gadget exists. For each dispatcher gadget, the set of available calltargets is compared against the set of gadgets generated in the previous step. As a consequence, if the intersection between the two sets is empty for any gadget required for the attack, the dispatcher gadget is eliminated.

To illustrate how not only the number of dispatcher gadgets but also the number of calltargets is influenced by selecting a specific CFI policy we use Figure 3. It shows a virtual table hierarchy and how different CFI policies can limit the set of calltargets available to a dispatcher mainloop gadget that calls the `C2::function2()`. Note how virtual tables: `C1`-`C3`, and `D1` (right side) map to virtual table hierarchy nodes (see tree nodes in the left side marked with `C1`-`C3`, and `D1`) depicted in the left side. Black shaded dots represent classes,

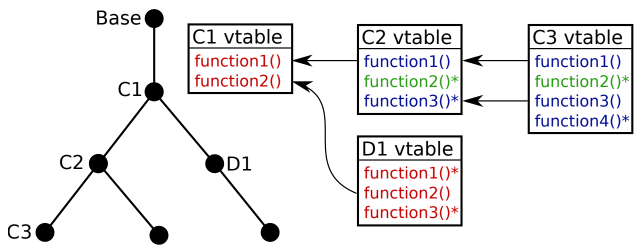

Figure 3: Left: virtual table hierarchy; Right: virtual tables.

left edges between nodes mean inheritance. Bottom children classes inherit from top father classes. Functions marked with an asterisk (right) override the parent class' function, while functions without are inherited (arrows). For example, when no CFI policy is enforced, all functions depicted in Figure 3, and other functions including non-virtual functions, can be reached and called during an attack. Further, if the virtual table hierarchy/island (*Marx* CFI policy [39]) is used, all virtual functions depicted in Figure 3 would be valid calltargets. In constrast, a strong policy such as *ShrinkWrap*'s [23] CFI policy allows only to target the functions marked in green by using fake objects to be invoked by the dispatcher gadget. Further, note that not every dispatcher gadget is suitable, since it has to match the specifications required by the attack type. For COOP's mainloop dispatcher gadgets, the requirements include: (1) having a loop containing a (2) virtual function dispatch contained within the loop, with the loop iterating over (3) an attacker-controlled array of fake objects that are invoked by the dispatcher gadget. Lastly, also note that not all calltargets are usable during attack construction.

To demonstrate which calltargets are usable during attack construction for different deployed CFI policies, we use Figure 4, which shows, from top to bottom, the process of gadget filtering based on the target set of virtual table entries which in turn is based on the deployed CFI policy. An arrow depicts virtual table inheritance direction. Each circle represents a virtual table which has at least one virtual table entry (a virtual function). Note that the number of entries grows from top to bottom due to virtual table inheritance. The top node contains a dispatcher gadget. The other nodes are

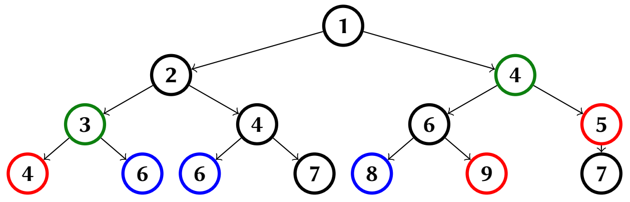

Figure 4: From root node (*i.e.,* 1) to bottom nodes the attack gadget filtering process is described. Nodes can have the same number (*i.e.,* 4) inside. A number in the node represents the total number functions that can be called from that particular class. More specifically, the number represents the sum of virtual and non-virtual table entries (*i.e.,* functions), and the functions inherited along a single inheritance path from the root node down to a particular leaf node.

other types of gadgets. A black circle denotes that it contains at least one virtual table entry which can serve as gadget. A red circle indicates there is no virtual table entry which can serve as gadget. And a blue circle shows a virtual table denoting the end of a virtual table inheritance path; it may or may not contain a useful gadget. A green circle shows there is a virtual table gadget which can be used as gadget but this is not reachable due to a CFI policy in-place; thus, everything what follows after this node is not reachable, too.

For example, path: $1-2-4-7$ represents a valid gadget chain if the functionality contained in this path is sufficient for an attack to be performed. This path is a complete gadget chain. The difference between a complete and partial gadget chain is that the first constitutes a complete attack while the second is not a complete end-to-end attack. Further, path: $1-2-3-4$ is not valid as the virtual table 4 is not usable and also everything which follows in this particular inheritance path after node 3 is not reachable, too. Path: $1-4-6-8$ is a valid chain that reaches the end of a virtual table inheritance tree path, but everything which follows after node 4 is not reachable. Lastly, in the path $1-4-5-7$ everything which follows after virtual table 4 is not reachable; thus it cannot be used during attack chain construction, even though node 7 is useful and node 5 is not useful.

Building Dispatcher Gadget Matching Sets. To understand the notation used within this paper, we use Table 2 which depicts the

Table 2: Used symbol descriptions.

Symbol	Description
IR_{py}	payload representation in Python
IR_{angr}	binary representation in ANGR
F_g	set of all working gadgets for all categories
Src	source code of target binary
CFI	static CFI policies to apply
D_s	set of potential dispatcher gadgets
C_s	set of potential calltargets per callsite
δ_s	delta set of callsites and reachable gadgets
Ch	gadget chain
Obj	a list of object layouts that implement the payload

used notation along their descriptions. As such, note that only after the candidate set has been built, usable dispatcher gadgets can be identified. The set of calltargets for the dispatcher gadget has to be compatible with the candidate sets F_g. Further, iTOP identifies calltargets for each callsite by analyzing the source code Src of the target binary. This data is then translated into a dispatcher gadget set D_s, and legal calltarget sets for each dispatcher gadget C_s. The intersection $C_s \cap F_g$ between the candidate gadgets and the available gadgets is called the *delta set* δ_s. The delta set contains the candidates for every gadget type that are still reachable from the dispatcher gadget with CFI policies in place. If the delta set is empty for any gadget required to build the attack, the dispatcher gadget is immediately discarded. A potential dispatcher gadget's machine code is then further analyzed to identify whether it is compatible with the attack type.

Further, we describe the steps needed to find a main loop gadget. First, a virtual function from the virtual table is called. The virtual function's offset is saved and later used during payload generation. Second, the virtual table address is returned from an array. Third, the function iterates over the array. Fourth, all the previous instructions are enclosed in a loop. Fifth, functions with compatible delta sets that fulfill these conditions are considered usable dispatcher gadgets. Lastly, in case none are found, the algorithm halts.

5.4 Assembling a Gadget Chain

In this section, we show how a combination of gadgets fulfilling the attacker's goals that are available to the dispatcher gadget is determined. The gadget assembly phase is depicted in Figure 1 and consists of analysis step ❼ and step ❽. ESL allows to define alternatives for blocks of gadgets, which is leveraged to create multiple equivalent Ch alternatives, thus increasing the probability of discovering usable attacks.

Searching for Optimal Gadget Chains. Equation 1 presents iTOP's chain ranking formula (S = $score$). The chain ranking is based on difficulty (Diff.) (*i.e.,* a rating between 0 and 1 that an experienced analyst can use to estimate the probability that a random function is actually usable as a gadget of a certain required type), average time to check a gadget (*i.e.,* obtained from our gadget checking analysis), and total availability of a gadget (*i.e.,* how often that gadget type is found within the analyzed program). Given the delta set δ_s and the intermediate representation of the payload IR_{angr}, chains of gadgets that encode the payload have to be generated.

$$S(Chain) = \min_{Gadget \in Chain} \frac{|Gadget|}{\text{Diff.}(Gadget) \times \text{Time}(Gadget)} \quad (1)$$

iTOP leverages the *alternative chain* definition feature of ESL to generate chain variants, and then orders those by likelihood of success. Metrics used to rank chains can include length, mean number of candidate gadgets, minimum number of candidate gadgets, and gadget difficulty. Depending on the analyst's goals, several rankings can make sense. First, a ranking that tries shortest chains will produce the smallest payload. Second, a ranking based on difficulty might yield faster results. iTOP leverages information obtained in the simulation step (see Section 5.5 for more details). In case iTOP attempts to build a chain and fails, the information obtained is used to eliminate chains that will fail for similar reasons, and statistics about usable gadget discovery rates can be used to reorder the chain ranking.

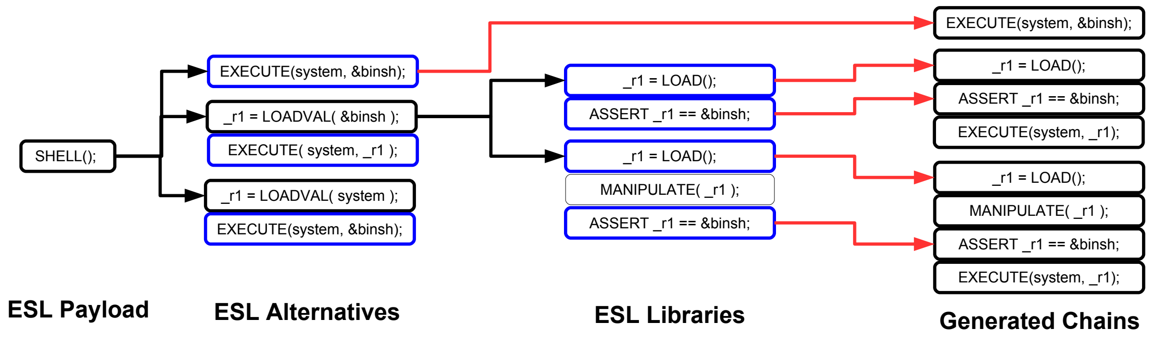

Figure 5: ιTOP's gadget chain alternative building. Black box: gadgets for this statement exist. Blue box: no further gadgets exist. After a blue box follows always a red arrow indicating that the analysis cannot continue. For *e.g.*, for `SHELL()`, 3 alternatives (count arrows) are specified in the payload definition. For `LOADVAL(&binsh)`, 2 alternatives are predefined in the ESL library.

To further understand how attack chains are specified by an analyst, we use Figure 5 which shows how, for the goal of spawning a shell, 3 different alternatives (count number of arrows starting from `SHELL()`) can be specified by the analyst. The last 2 chains, in turn, use the `LOADVAL(...)` function, for which 2 alternatives are already predefined in the ESL libraries. Thus, note that the ESL standard library provides a wide array of alternatives for register initialization and execution of arbitrary functions. As such, different chains for spawning a shell can be generated. Further, Figure 5 depicts 2 alternatives for `SHELL()` which use the same function: `_r1 = LOADVAL(system);`. Since this block is fully independent from the surrounding blocks, valid subchains or the information that no such chain exists can be reused and only has to be calculated once. This way, even when hundreds of chains are possible, the search space and thus runtime stays manageable.

5.5 Simulation and Payload Generation

In this section, we show how each gadget chain is simulated using ANGR [46] in order to check whether it fulfills all constraints required by the gadget definition and whether it is compatible with the gadgets contained so far in the chain. The gadget simulation and payload generation phases are depicted in Figure 1 and consist of analysis step ❾ and step ❿.

Figure 5 depicts this process. More specifically, dispatcher gadget `EXECUTE(system, &binsh), the first gadget in category one (ESL Alternatives)`, cannot be linked (*i.e.*, one red arrow goes out) to any gadget in category 2 (ESL Libraries), and is thus unusable. This can happen, for example, when gadgets can only initialize registers with certain ranges of values, with the second gadget requiring the value to be outside the range. In contrast, the second candidate for gadget one, `_r1 = LOADVAL(&binsh)`, can be linked (*i.e.*, two black arrows go out), and a valid chain is found. If there are no suitable gadget combinations for a chain, ιTOP backtracks and and a different gadget chain will be used instead.

Gadget Chain Simulation. To understand how each gadget chain is constructed and simulated, we use Algorithm 1 which depicts, based on ANGR's symbolic execution, how a chain of gadgets and its dispatcher gadget are combined and simulated. For example, after a gadget has been simulated (line 6), constraints are added to

Algorithm 1: Extend gadget chain (rec_chain)

```
chain      : list of gadget types in the order they should be called in
candidates : mapping of gadget types to candidate gadgets
state      : prog. state just before the virt. func. dispatch
Result: a chain of compatible gadgets
1  begin
2      gadget ⟵ pop first element of chain
3      targets ⟵ candidates.lookup(gadget)
4      for function ∈ targets do
5          state: call function
6          state: simulate until return to dispatcher gadget
7          state: add gadget constaints
8          if state is feasible then
9              state: simulate until callsite is reached again
10             if chain = ∅ then
11                 return function
12             else
13                 tmp ⟵ rec_chain(rest_of_chain, candidates, state)
14                 if tmp ≠ ∅ then
15                     return function + tmp
16     return ∅
```

the simulation as described in the constraints column of Table 1 (line 7). If, at any point, the simulation state becomes unsatisfiable, the simulation backtracks and tries a different gadget candidate.

Further, in case ιTOP discovers no satisfiable chain which can be built, it backtracks and generates a different chain. If, however, the simulation state is still satisfiable after all gadgets have been called, a compatible exploit has been discovered and a payload is generated. Note that during simulation, ιTOP generates a set of memory constraints C_M for the buffer.

Generating the Attack Payload. The steps to find a valid gadget chain are as follows. After the simulation phase, a valid chain is found; next a payload can be generated. Since payload generation depends on details such as attack type (*e.g.*, COOP requires additional logic for object overlapping) and payload specifics (loops and conditionals require additional logic), the precise approach can differ depending on the payload, but the overall approach stays the same. For example, ɪTOP leverages the set of constraints C_M as well as the combination of gadgets in the valid chain to generate a memory layout for the target buffer that will lead to the execution flow as defined in the ESL definition. Thus, when implementing a new attack type using ESL's Python-based API, the payload generation logic can be modified to target the new attack.

The valid memory layout (*i.e.,* list of object layouts *Obj*) is then saved into a file for further usage by the analyst. Optionally, ɪTOP can continue the search to discover more payloads. The total number of payloads found can help in evaluating the quality of a defense mechanism, and alternative payloads could help in circumventing blocklists and pattern-based defense systems.

6 EVALUATION

In this section, we address the following research questions (RQs).

- **RQ1:** How can ɪTOP be used to *construct control-flow hijacking attacks*? (Section 6.1)
- **RQ2:** How much *protection do static CFI policies offer* after deployment? (Section 6.2)
- **RQ3:** How much *attack surface reduction* is needed to effectively protect from control-flow hijacking attacks? (Section 6.3)
- **RQ4:** How *vulnerable are programs with no CFI protection in-place* to control-flow hijacking attacks? (Section 6.4)

Target Programs. For our detailed evaluation, we focus on NodeJS (v. 8.9.1, `C/C++` code, compiled as library) [37]. For our broader evaluation, we target Nginx [35] (v. 1.13.7, `C` code), Apache Httpd [4] (v. 2.2.24, `C` code), LibTorrent [14] (v. 1.1.0, `C` code), Redis [41] (v. 2.6.14, `C` code), Firefox [13] (v. 36.0a1, `C/C++` code) and Chrome [18] (v.33 `C/C++` code).

Table 3: The CFI policies used in this work do not impose any runtime write constrains and are context-insensitive.

CFI Policy	Solutions	Description
IFCC/MCFI	[36, 47]	function parameter source types
Safe IFCC/MCFI	[36, 47]	function parameter safe source types
ShrinkWrap/IVT	[23]	strict program sub-hierarchy
VTV	[7, 47]	program sub-hierarchy
VTint	[52]	all program virtual tables
Marx/VCI	[12, 39]	impose the virtual table hierarchy

Experimental Setup. The evaluations were performed on a system with Intel i5-2500k CPU (3.30 GHz), 16 GB RAM, and running the Linux Mint 18.3 operating system. The CFI policies used in these experiments are presented in Table 3. Note that the created exploits were executed by enabling a CFI policy, which consist in removing the calltargets that would be not available after such a CFI defense is deployed. Lastly, all programs were compiled with Clang/LLVM `-O2` compiler optimization flag.

Gadget Search Timeout. Note that we used a timeout of 6 hours (1/4 of a day) in our gadget search algorithm. Once this time limit was reached the search was terminated. While exploit generation is a *1-time* process and from a practical point of view, an approach that requires even multiple days of computation could still be useful (since manually generating a complex exploit for a real-world CFI-hardened binary can also take a significant amount of time). In this work, we opted to keep a common and plausible timeout level for comparison purposes, which can be extended in future work.

Table 4: The ESL payloads. Each payload produces chains containing N statements. Symbol meaning: ✓: payload contains loop/conditional statements, ✗= not available gadget.

Payload	Description	N	Loops	cond
`regset`	Load argument register with constants	1-2	✗	✗
`memrd`	Read value from address to `register`	1	✗	✗
`memwrt`	Write value from register to address	1	✗	✗
`regadd`	Add values from two registers	3	✗	✗
`printf`	Write to stdout	1-5	✗	✗
`shell`	Spawn a shell	1-5	✗	✗
`iloop`	Cause an infinite loop	1	✓	✗
`cond`	Conditionally write to address	2	✗	✓
`for`	Loop with exit conditions	4-6	✓	✓
`cshell`	Spawn a shell only if value at address	3-11	✗	✓
`count`	Print all integers from 0 - 100	5-8	✓	✓
`mprt`	Call mprotect with user controlled args	1-9	✗	✗
`env`	Read an environment variable	1-5	✗	✗

To understand ESL's 13 attack primitives, we use Table 4, which depicts the 13 ESL payload scenarios for which different payloads were generated. We developed these 13 primitives by studying related work and by investigating the most used and useful minimal set of program primitiveness needed in order to be able to express all kinds of program behaviors. The goal of ESL primitives was to include payloads which: (1) demonstrate basic features of ɪTOP, (2) have complex control flow, and (3) are realistic payloads usable for real-world attacks.

6.1 Case Study: NodeJS

In this section, we analyze ɪTOP's payload building capabilities on NodeJS. We chose NodeJS because it is a popular, widely used application containing both `C` and `C++` code that is frequently used as a library in other applications. We focus on three representative payloads to demonstrate the core features of ɪTOP: spawning a shell, controlling argument registers, and finally conditionally writing a value into memory. We then combine these payloads to a complex payload to manipulate the control flow.

Table 5: Performance results for payload generation.

Payload	Time	Chains	Payloads
`shell`	10:53	31	10+
`iloop`	02:29	1	10+
`cond`	21:56	1	10+

To highlight the results obtained for different payloads when using NodeJS, we use Table 5 which shows the time needed to perform the analysis, number of chains and number of payloads generated. Time indicates the time needed to generate the first payload in minutes and seconds (mm:ss) format, Chains: number of candidate chains, Payloads: number of different payloads (capped at 10). We recorded two demo videos: (1)[5] spawning a system shell with no CFI

[5]Video: spawning a system shell with no CFI policy used: https://tinyurl.com/yyvxncqj

policy in-place, and (2)[6] spawning a system shell based on NodeJS under the VTint CFI policy in-place. These demonstrate ɪTOP's attack building capabilities with and without CFI protection active.

6.2 Assessing CFI Policies against NodeJs

In this section, we show under which CFI policies ɪTOP is still able to generate exploits. Note that static CFI policies can considerably reduce the attack surface of a binary, significantly complicating attack generation. Using callsite/calltarget mappings, we can evaluate which CFI policies are effective attack deterrents.

Table 6: Payload generation under six CFI policies.

Payload	No CFI	IFCC	IFCC-safe	VTint	VTV	Marx	SW
`regset`	✓ 00:11:50	✓ 00:34:36	✓ 00:40:22	✓ 00:29:31	✓ 02:51:19	✓ 03:35:42	✓ 04:24:22
`memrd`	✓ 00:41:22	✓ 01:11:21	✓ 01:22:04	✗	✓ 02:44:18	✓ 02:31:07	✗
`memwrt`	✓ 00:03:47	✓ 00:37:52	✓ 00:41:12	✗	✓ 01:10:08	✓ 02:12:03	✓ 01:07:05
`regadd`	✓ 00:20:24	✓ 00:40:12	✓ 00:43:44	✓ 00:05:18	✗	✗	✗
`printf`	✓ 00:11:32	✓ 01:35:56	✓ 01:46:12	✓ 00:31:29	✓ 00:37:10	✓ 00:46:12	✓ 01:06:04
`shell`	✓ 00:10:53	✓ 01:45:54	✓ 02:00:11	✓ 00:30:10	✓ 02:21:44	✗	✗
`iloop`	✓ 00:02:29	✓ 00:37:43	✓ 00:41:48	✓ 00:01:35	✗	✗	✓ 00:43:47
`cond`	✓ 00:21:56	✗	✗	✗	✓ 03:25:39	✓ 02:49:04	✓ 05:43:14
`for`	✓ 00:33:30	🕒	🕒	✗	✓ 04:25:04	✓ 03:03:10	✓ 04:45:27
`cshell`	✓ 00:43:14	🕒	🕒	✗	✓ 00:54:19	✓ 00:48:44	✓ 00:53:41
`count`	✓ 00:27:47	✓ 00:55:23	🕒	✗	✗	✓ 02:12:05	✓ 03:23:11
`mprt`	✓ 00:37:00	✓ 00:57:32	✓ 01:25:23	🕒	✗	✓ 01:15:35	✓ 00:55:23
`env`	✓ 00:11:20	✓ 01:30:25	✓ 01:44:35	✓ 00:33:47	✓ 01:32:28	✓ 01:49:27	✓ 02:55:23

To highlight the capabilities of ɪTOP w.r.t. payload generation under fine-grained CFI policies, we employ Table 6 which depicts the payload generation results for NodeJS by running ɪTOP with each of the 6 CFI defenses. We used the payloads presented in Table 4 with different CFI policies in-place. SW: ShrinkWrap, ✓ denotes one or more payloads were successfully generated, while a ✗ indicates that no valid payload was found. Behind the ✓ symbol the total time needed for the generation of the payload is given (hh:mm:ss format). The reasons for failure: 🕒= timeout; or ✗ = required gadget is not available. We observed that there is an overlap of gadgets that are usable depending on the used defense. In the future, we want to investigate if these gadgets exhibit specific characteristics that make them particularly difficult to harden in order to gain insight into how to harden/improve existing defenses. Note that with these 6 CFI policies we still can successfully generate exploits in 77% (60 out a total of 78 of the cases whereas with no CFI policy deployed the exploit construction succeeds in 100% of the cases (see the second column of Table 6).

A higher success rate is, of course, possible by using a larger timeout. We support this statement with our experimental observations, which confirm that the payload generation success rate improves by increasing the imposed analysis timeout. Further, without applying any policy, *all* payloads could be generated. In case no payload was generated, this was mainly due to the fact that the required gadgets were not available. Further note that, in case ɪTOP times out, a valid payload might still exist, while, when a required gadget is not available, no payload exists.

Important to note is that *IFCC* and *IFCC-safe* (introduced by Tice *et al.* [47]) are policies enforcing correct parameter counts for calltargets. We found that most payloads could still be built under these CFI policies, since both arbitrary memory reads/writes and calling of arbitrary functions are still possible. *VTint* allows only calltargets in any virtual table, eliminating all non-virtual calltargets. While *VTint* offers more protection than *IFCC*, some realistic payloads could not be eliminated, and spawning a shell remains possible. The *VTV*, *Marx* and *ShrinkWrap* policies enforce class hierarchies, allowing almost exclusively intended calltargets to be reached from a dispatcher. ɪTOP could not generate a payload under *VTV* in 4 cases, *Marx* in 3 cases and *ShrinkWrap* in 3 cases given the preset time limit as these are very strong CFI policies. Note that this is not a strong limitation of our approach and we think that on other programs and by increasing the timeout new attacks can be generated.

[6]Video: spawning a system shell, VTint [52] in-place: https://tinyurl.com/yyrso75k

6.3 Stopping Attacks: NodeJS Case Study

In this section, we investigate attack probabilities under various calltarget reduction and gadget usability scenarios.

$$P_A(f_{red}) = 1 - (1 - P_G)^{(|F| * f_{red})} \quad (2)$$

To understand how we calculated the attack probability, we use Equation 2 to calculate the probability P_A of whether an attack exists under the assumption of independence. This is a reasonable assumption as all CFI defenses target calltargets that can be only functions. The attack surface reduction factor f_{red} refers to the reduction of calltargets for the dispatcher with the highest remaining calltarget count. Using the values from the analysis of NodeJS and gadget usability probability of $P_G = 0.001$, we determine that a 85% attack surface reduction has a 50% probability of stopping attacks, and 99.98% reduction resulting in a less than 1% probability of an attack still existing. This does not prove that the defenses are bullet-proof but rather that the attack crafting likelihood is greatly reduced. This estimation matches our evaluation results: for *IFCC*, a policy providing 85% attack surface reduction, two payload generation timed out, while for *ShrinkWrap*, a policy providing 99.4% reduction, in 3 cases no payloads were generated.

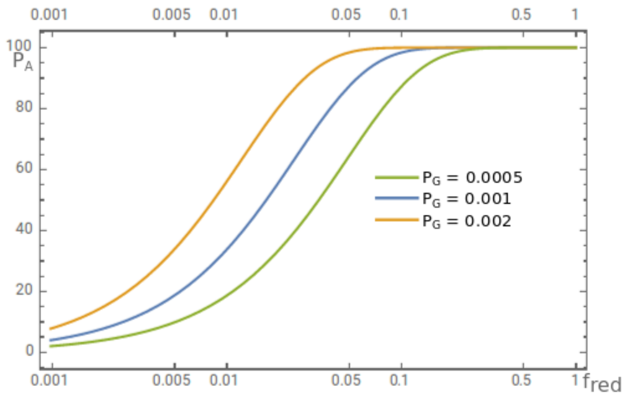

Figure 6: Probability of attack P_A with attack surface f_{red}.

To show how the attack probabilities vary depending on the probability of gadget usability we introduce Figure 6 which shows the estimated probability of an attack still being possible w.r.t. attack surface reduction for different values of P_G. We can estimate how

Table 7: Results for several vulnerable programs. Time: Time needed to load the binary and to analyze all its dispatchers. Dispatchers: Number of usable dispatchers found per category. Gadgets: Number of candidate gadgets found for each category; ML: Mainloop dispatcher gadget, LS: LinkedList dispatcher gadget, REC: Recursive dispatcher gadget.

Vulnerable Application			*Time*	*COOP Gadget*			*Gadgets*			
Program	**Vulnerability**	**Functions**	**(m:s)**	**ML**	**LS**	**REC**	**READ**	**LOAD**	**WRITE**	**EXECUTE**
Nginx	CVE-2013-2028	1,192	00:13	0	3	3	256	176	107	743
Apache Httpd	CVE-2006-3747	1,917	00:08	0	0	3	399	210	66	730
LibTorrent	library	2,002	00:11	0	0	27	340	272	55	1,123
Redis	CVE-2018-11218	1,996	00:12	0	0	2	647	266	137	1,511
NodeJS	CVE-2014-5256	41,189	03:27	7	24	1,025	11,249	6,730	3,804	23,309
Chromium	CVE-2017-7000	145,206	11:53	25	67	1,137	33,515	17,009	16,100	89,956
Firefox	CVE-2018-5150	201,741	10:36	1	59	1,653	120,702	57,936	43,649	118,764

Table 8: ESL payloads applied to a variety of programs. ✓: iTOP generated one or more payloads; no payload was be generated, due to (a) 🕒: timeout or (b) ✗: gadget was not available. Note that within this experiment no CFI policy was used, as we wanted to find out how iTOP performs under real-world scenarios where most of the time programs are not protected by CFI policies.

Program	`regset`	`memrd`	`memwrt`	`regadd`	`printf`	`shell`	`iloop`	`cond`	`for`	`cshell`	`count`	`mprt`	`env`
Nginx	✓	✓	✗	✓	✓	✓	✓	✗	✗	✗	✗	🕒	✓
Apache Httpd	✓	✓	✗	✗	✓	✓	✗	✗	✗	✗	✗	🕒	✓
LibTorrent	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	✓	✓
Redis	✓	✗	✗	✗	✓	✓	✗	✗	✗	✗	✗	🕒	✓
NodeJS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Chromium	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Firefox	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

much attack surface reduction a CFI policy has to provide to effectively prevent CRA. The data gathered by iTOP indicates that, with no CFI policy applied, 25 functions out of the 41,189 targets were usable to call arbitrary functions with arbitrary values in the first argument register, without depending on any prior register initialization. With, for example, the *IFCC* CFI policy deployed, 7 out of the 5,588 were usable out of the box, while with the *VTint* policy 4 out of 5,852 functions were usable. While these figures fluctuate and strongly depend on the applied policy, the ratio of usable gadgets to all calltargets usually ends up between 0.2% and 0.05%. Reducing the number of usable gadgets to zero stops all attacks on the target application, as no valid gadget chain can be built without any gadget available.

Lastly, to reproduce how we derived Equation 2 used in this section note that by using the previously mentioned information, a probability of $0.0005 \leq P_G \leq 0.002$ of a usable gadget existing is estimated, which, combined with the number of calltargets, $|F| = 41,189$, and the attack surface reduction factor, f_{red}.

6.4 General Results

In this section, rather then looking for successful exploitation under different CFI policies, we want to find out the time and number of gadgets needed to generate different payloads when using iTOP as these are the first steps towards successful exploit generation. Note that the aim of this section is not to confirm/show that exploits can be built for known vulnerable programs but rather to present a fully new and automated way to generate new exploits based on existing documented vulnerable programs. To achieve this goal, we evaluated iTOP by targeting six widely used programs. Most of the evaluated programs contain a CVE corresponding to an arbitrary memory write, fulfilling the requirements specified in our threat model. Lastly, note that we had access to a program information leakage which provided us with an arbitrary read as required in our threat model.

To illustrate how many COOP gadgets and ESL primitives were found for different vulnerable programs we use Table 7 which depicts the results for each of the six target applications. Further, Table 7 includes the total number of potential target functions the application contains and detailed statistics on the availability of each gadget and dispatcher type. Note that the `LibTorrent` program is the only program without a `CVE` listed as it is a library used in a variety of other potentially vulnerable applications. Further, note that generating the candidate dispatcher and gadget sets is a time-consuming process, but the results can be reused to generate multiple attacks. iTOP caches these results to a file and reloads them for further searches, eliminating the startup overhead listed in Table 7.

In order to show how successful iTOP is in generating exploits based on the ESL's 13 attack primitives we use Table 8 which depicts the attempts of iTOP to generate an exploits for the payloads listed in Table 4 for each vulnerable application. Thus, iTOP was able, for example, to generate a payload in 60/91 cases (66% success rate, 60/91). Note that the most interesting payload to an attacker, *shell*, could be generated in seven out of seven cases. When excluding all payloads which rely on control flow manipulation, iTOP generated payloads in 44 out of 56 cases (79%). The main reasons for the failing cases are either timeouts or the specific gadget is not available.

Along the successful and unsuccessfully attempts to generate exploits for the ESL's 13 primitives, Table 8 also shows that applications with larger target-sets yield significantly higher success rates, because the variety of functions is much higher and thus the probability to find a function exactly matching the constraints is higher. Payloads including conditionals and loops require a `LINKEDLIST`-type dispatcher to function, because the control flow is modified by rewriting the next item fields in linked lists. While other approaches to dynamically alter exploit control flow are possible, we focus on this approach because it can easily generate Turing complete exploits. Further, most of the `C` programs did not contain any usable `LINKEDLIST` dispatchers, while all of the `C++` programs did. In some cases, ιTOP was unable to find a payload. Generally, there are several possible reasons for this. First, there are no valid dispatcher blocks. Second, gadgets required to build the payload are not available to the dispatcher. Lastly, the constraints on the payload are unsatisfiable.

Lastly, in order to better understand the failing attempts Table 8 lists the exact reasons for failure. As such, in most cases, a required gadget was missing, such as an arbitrary memory write gadget. In other cases, when generating the complex `mprt` payload, the search timed out as in column 13—from left to right—in Table 8 for the programs Nginx, Apache Httpd, and Redis.

7 DISCUSSION

Symbolic Execution. Symbolic execution is a powerful approach for payload generation as it allows for automated exploration of the program control flow graph, precisely program state reasoning, and automatic generation of the payload. However, it is greatly limited by the size of the target binary. Further, by for example using basic block constraints summaries, similarly to BOPC [24], this could enable a better analysis of combinations of gadgets, at the cost of even more runtime performance increase, making them infeasible for large binaries. However, an alternative approach based on pattern recognition instead of symbolic execution would lead to significant speedup, sacrificing the capabilities to identify complex, branching, but still usable gadgets.

ιTOP's Analysis Generality. In this work, we focus on COOP-like attacks, similarly to [48], but we envisage ιTOP to be used for crafting other types of attacks as well. In contrast to Newton, for example, ιTOP is currently limited to static CFI defenses but can be extended to work with dynamic CFI defenses. In this work, we decided to mainly address the expressiveness of our attack specification language (ESL) and thus focusing only on static CFI defenses. Further, in order for ιTOP to craft other attack types, the calltargets (*i.e.,* gadgets) used within an attack have to be reconsidered such that consecutive instructions (*i.e.,* gadgets) can be targeted by the indirect control flow transfers used to assemble the gadget chain. In order to achieve this, the first step is to use an analysis that first classifies these gadgets and then creates a map with all their locations in memory.

Tool Potential. As opposed to a common belief that high percentages in the realm of automated attack construction is an indicator for tool potential we rather think that this is not the most essential indicator for tool potential, as attack success depends on the types of gadgets used and it is less significant to have for example 1000 possible attack variations for a single target program but rather one attack is sufficient to perform the attack. Further, in real attack creation scenarios, the situation is more simplified as all available tools are used to craft the attack. Note that only one of these tools has to be successful.

Gadget Evaluation. Due to intrinsic limitations of symbolic execution: reads from, writes to, and jumps to unconstrained addresses cannot reliably be evaluated, as these would lead to state explosion issues. We addressed this within ιTOP by constraining reads and writes to point to predefined addresses. This solution is far from being perfect, and some usable gadgets might be missed. Further, reads and writes to symbolic file descriptors and I/O in general is also hard to model using symbolic execution, as there is no way to predict which files are present and what these contain during the actual execution of the target program. ιTOP addresses this by skipping all functions containing I/O, possibly missing gadgets. Also large gadgets (size > 1k bytes), gadgets containing many calls to other functions, and gadgets that would require symbolic objects larger than 128 bytes to be simulated are currently skipped to avoid increased runtime overhead.

COOP Attacks. ιTOP generates COOP and COOP-like attacks, which do not violate the program stack discipline, thus eliminating the need to bypass shadow stack techniques. Further, we showed that COOP attacks are flexible enough to work around CFI constraints, and powerful enough to implement complex control flows. Lastly, as long as no fully precise class hierarchy based CFI policy is enforced our attacks are feasible in a majority of cases.

8 LIMITATIONS AND FUTURE WORK

Gadget Discovery Time. For large binaries such as web browsers, ιTOP needs a large amount of time to be spent in order to generate a payload due to the number of gadgets that have to be evaluated. In future work, by developing a more precise gadget discovery framework for COOP, COOP-like or arbitrary attacks that can pre-calculate the set of usable gadgets for a target binary it could provide significant help for attack building frameworks, reducing their domain purely to finding gadget chains.

Gadget Search Granularity. ιTOP analyzes one function at a time and maps one ESL statement to one gadget. In some cases, multiple ESL statements can be mapped using only one gadget, or multiple gadgets could be combined to fulfill the requirements defined by one ESL statement. While both of these issues can be mitigated by our gadget chaining algorithm, an analyst has to manually specify all alternatives required, which requires precise knowledge of ιTOP's limitations. In future work, by making ιTOP inherently aware of such shortcuts, we could improve its gadget generation and analysis capabilities as more attacks could be constructed.

Attack Probabilities Formula. To carry out the attack we need a suitable gadget of each type. These types of gadgets do not have necessarily the same frequencies. It might be that, say, 5% of the functions is suitable as a dispatcher, 20% to write %rdi, and 1% to execute a function. Now we need one of each for a particular attack. W.r.t. this point Equation 2 might be too simple, as it does not consider the differences in probabilities. Moreover, a CFI defense does not remove the different gadget types at the same rate. A target of an indirect call would be relatively likely to be a callback or virtual function, and we cannot assume that this is independent of the gadget types. In particular, it is plausible these would be simpler on average than functions that are not indirect call targets. In future work, we can improve Equation 2 by modeling the frequency distribution of gadgets. Further in the updated formula the gadget probabilities will be longer averaged as in the current version. Thus, more accurate results can be obtained.

Different Attack Types. ESL provides a Python like API which can be used to craft COOP-like attacks. In future work, by extending our DSL-based ESL to implement new attacks, other attack types can be specified. As such, ιTOP's API can make the implementation of other types of attacks relatively easy. Thus, by implementing other types of attacks (*e.g.,* ROP), this would considerably enhance ιTOP's target binary analysis of the assessed CFI policy.

9 RELATED WORK

Automatic Exploit Generation (AEG) [6] automatically searches for a vulnerability in the target binary and generates an exploit. Assuming no defenses are in-place, AEG [6] is an end-to-end attack crafting tools which first discovers a vulnerability and then tries to generate exploits, if possible, for both source code and binary programs, respectively. In contrast to ιTOP, AEG finds first the vulnerability and is not designed to operate under strong defenses. Opposed to ιTOP, AEG has no attack specification language. ιTOP is not dependent on the provided vulnerability type, while AEG focuses only on stack overflows and format string vulnerabilities.

Revery [49] is a dynamic attack crafting tool that analyzes a vulnerable program and collects runtime information on the crashing path as for example taint attributes of variables. Revery is an extension of AEG but goes beyond by focusing on other challenges. In contrast to ιTOP, Revery analyzes crashing paths and as such a vulnerability has not be provided. Revery fails in certain cases to generate an attack due to complicated defense mechanisms of which the tool is not aware. Further, in some cases, Revery does not generate exploits due to dynamic decisions that have to be made during exploitation. In contrast to ιTOP, Revery does not come with an attack specification language.

Newton [48] is a runtime attack crafting tool whereas ιTOP is a static attack crafting tool that both provide an attack specification language. In contrast to ιTOP which focuses on COOP-like attacks, Newton focus on other types of attacks as well. Further Newton, ιTOP uses a static specification for constructing the payload while Newton uses an attack specification language which is used to interact during runtime with the monitored program. Further, Newton uses a more general black-box approach based on dynamic taint analysis and is not limited to COOP attacks. In contrast, ιTOP covers both black-box (*i.e.,* a CFI policy is not deployed) and white-box (*i.e.,* a CFI policy is deployed) during attack construction.

BOPC [24] is a framework for automatic static building of data-only attacks which do not violate the control flow of the program. BOPC can assess whether an attacker can perform arbitrary code execution attacks within a binary which was previously hardened with CFI and/or shadow stack defenses. In contrast to BOPC, ιTOP does not address data-only attacks as BOPC does, but rather control-flow violating attacks. BOPC searches inside the legitimate CFG of the program for machine code basic blocks to used for an attack, in contrast ιTOP searches inside and outside of the program CFG for targets which may or may not be protected by a CFI policy which then can be used as gadgets for constructing COOP-like attacks.

Other tools such as PSHAPE [15], Kepler [51], ropc [38], ROPGadget [42], Q [44], and work by Wollgast *et al.* [50] seek to automate the full attack construction process. In contrast to ιTOP, these tools are limited, as these rely on finding hard-coded sequences of instructions to identify gadgets and can only build pre-determined gadget chains.

10 CONCLUSION

We have presented ιTOP, a framework for fully automated construction of control-flow hijacking attacks, which can bypass state-of-the-art deployed CFI defenses and shadow stack defenses. ιTOP automates the analysis of the target binary, the identification of useful gadgets and gadget dispatch mechanisms, and can build payloads under consideration of state-of-the-art CFI policies. We have evaluated ιTOP by testing it on seven real-world programs, successfully creating payloads ranging from spawning a shell to loops and conditionals when using our attack generation primitives. These payloads have spawned a shell (no CFI policy deployed) for *all* evaluated binaries and demonstrated that many state-of-the-art CFI policies are too permissive, allowing an attacker with in-depth knowledge of the vulnerable program to construct attacks that bypass these deployed state-of-the-art fine-grained CFI defenses.

ACKNOWLEDGMENTS

We are grateful to the anonymous reviewers, and our shepherd Erik van der Kouwe, VU Amsterdam, Netherlands, for their help in improving this work. We also would like to thank Chao Zhang, Tsinghua University, P. R. China and Haohuang Wen, The Ohio State University, USA, for providing insightful feedback on earlier versions of this paper.

REFERENCES

[1] 2018. pyelftools. (2018). https://github.com/eliben/pyelftools.
[2] 2018. System – Execute a shell command in Linux man pages. (2018). https://linux.die.net/man/3/system.
[3] ANTLR. 2020. ANother Tool for Language Recognition. (2020). https://www.antlr.org/.
[4] Apache. 2017. Apache Httpd. (2017). https://httpd.apache.org/.
[5] Apple. 2020. Apple Security Bounty. (2020). https://developer.apple.com/security-bounty/.
[6] T. Avgerinos, S. K. Cha, A. Rebert, E. Schwartz, M. Woo, and D. Brumley. 2015. Automatic exploit generation. In *Communications of the ACM*. ACM.
[7] D. Bounov, R. G. Kici, and S. Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. ISOC.
[8] Capstone. 2019. Capstone Disassembly Framework. (2019). http://www.capstone-engine.org/.
[9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of USENIX Security Symposium (USENIX Security)*. USENIX.
[10] Clang. [n. d.]. Clang SafeStack. ([n. d.]). https://clang.llvm.org/docs/SafeStack.html.
[11] T. H. Y. Dang, P. Maniatis, and D. Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM.
[12] M. Elsabagh, D. Fleck, and A. Stavrou. 2017. Strict Virtual Call Integrity Checking for C ++ Binaries. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM.
[13] Firefox. 2019. Firefox. (2019). https://www.mozilla.org/en-US/firefox.
[14] Firefox. 2019. LibTorrent. (2019). https://www.libtorrent.org/.

[15] A. Follner, A. Bartel, H. Peng, Y. C. Chang, K. Ispoglou, M. Payer, and E. Bodden. 2016. PSHAPE: Automatically Combining Gadgets for Arbitrary Method Execution. In *Proceedings of the International Workshop on Security and Trust Management (STM)*. ACM.

[16] GCC. 2016. shadow stack proposal. (2016). https://gcc.gnu.org/ml/gcc/2016-04/msg00083.html.

[17] E. Goektas, E. Athanasopoulos, H. Bos, and G. Portokalidis. 2014. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE.

[18] Google. 2017. Google Chrome. (2017). https://www.chromium.org/.

[19] Google. 2020. Google Application Security. (2020). https://www.google.ch/about/appsecurity/reward-program/.

[20] Google. 2020. Google's Project Zero . (2020). https://googleprojectzero.blogspot.com/.

[21] B. Gras, K. Razavi, E. Bosman, H. Herbert, and C. Giuffrida. 2017. ASLR on the Line: Practical Cache Attacks on the MMU. *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (2017).

[22] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM.

[23] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. 2015. ShrinkWrap: VTable Protection Without Loose Ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM.

[24] K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. 2018. Block Oriented Programming: Automating Data-Only Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM.

[25] J. Koschel, C. Giuffrida, H. Bos, and K. Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P)*. IEEE.

[26] Metasploit. 2019. Metasploit Framework. (2019). https://github.com/rapid7/metasploit-framework.

[27] Microsoft. 2009. The STRIDE Threat Model. (2009). https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN.

[28] Microsoft. 2020. Microsoft Bug Bounty Program. (2020). https://www.microsoft.com/en-us/msrc/bounty.

[29] Changes to Functionality in Microsoft Windows XP Service Pack 2. Microsoft. 2003. (2003). https://technet.microsoft.com/en-us/library/bb457151.aspx.

[30] L. d. Moura and N. Bjorner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*. Springer.

[31] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. 2018. τFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer.

[32] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. 2019. Analyzing Control Flow Integrity with LLVM-CFI . In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM.

[33] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. 2020. ρFEM: Efficient Backward-edge Protection Using Reversed Forward-edge Mappings. In *Annual Computer Security Applications Conference (ACSAC)*. ACM.

[34] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert. 2018. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer.

[35] Nginx. 2017. Nginx web server. (2017). https://nginx.org/en/.

[36] B. Niu and G. Tan. 2014. Modular Control-Flow Integrity. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*. ACM.

[37] NodeJS. 2017. Open-source, cross-platform JavaScript run-time environment. (2017). https://nodejs.org/en/.

[38] Pakt. 2013. ropc: A turing complete ROP compiler. (2013). https://github.com/pakt/ropc.

[39] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. 2017. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC.

[40] Application Entry Point. 2021. InfoSec Institute. (2021). https://resources.infosecinstitute.com/topic/discovering-entry-points/.

[41] Redis. 2017. Redis in-memory database. (2017). https://redis.io/.

[42] J. Salwan. 2011. ROPgadget - Gadgets Finder and Auto-roper. (2011). http://shell-storm.org/project/ROPgadget/.

[43] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE.

[44] E. J. Schwartz, T. Avgerinos, and D. Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX.

[45] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*. ISOC.

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

[47] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *Proceedings of USENIX Security Symposium (USENIX Security)*. USENIX.

[48] V. van der. Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. 2017. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM.

[49] Y. Wang, C. Zhang, X. Xiang, Z. Zhao, W. Li, X. Gong, B. Liu, K. Chen, and W. Zou. 2018. Revery: From Proof-of-Concept to Exploitable. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM.

[50] P. Wollgast, R. Gawlik, B. Garmany, B. Kollenda, and T. Holz. 2016. Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer.

[51] W. Wu, Y Chen, X Xing, and W. Zou. 2019. Kepler: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities. In *Proceedings of USENIX Security Symposium (USENIX Security)*. USENIX.

[52] C. Zhang, C. Song, K. Chen Zhijie, Z. Chen, and D. Song. 2015. vTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC.