On the Effectiveness of Type-based Control Flow Integrity

Reza Mirzazade farkhani
Northeastern University
reza699@ccs.neu.edu

Saman Jafari
Northeastern University
jafari1149@ccs.neu.edu

Sajjad Arshad
Northeastern University
arshad@ccs.neu.edu

William Robertson
Northeastern University
wkr@ccs.neu.edu

Engin Kirda
Northeastern University
ek@ccs.neu.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

ABSTRACT

Control flow integrity (CFI) has received significant attention in the community to combat control hijacking attacks in the presence of memory corruption vulnerabilities. The challenges in creating a practical CFI has resulted in the development of a new type of CFI based on runtime type checking (RTC). RTC-based CFI has been implemented in a number of recent practical efforts such as GRSecurity Reuse Attack Protector (RAP) and LLVM-CFI. While there has been a number of previous efforts that studied the strengths and limitations of other types of CFI techniques, little has been done to evaluate the RTC-based CFI. In this work, we study the effectiveness of RTC from the security and practicality aspects. From the security perspective, we observe that type collisions are abundant in sufficiently large code bases but exploiting them to build a functional attack is not straightforward. Then we show how an attacker can successfully bypass RTC techniques using a variant of ROP attacks that respect type checking (called *TROP*) and also built two proof-of-concept exploits, one against Nginx web server and the other against Exim mail server. We also discuss practical challenges of implementing RTC. Our findings suggest that while RTC is more practical for applying CFI to large code bases, its policy is not strong enough when facing a motivated attacker.

ACM Reference Format:
Reza Mirzazade farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. 2018. On the Effectiveness of Type-based Control Flow Integrity. In *2018 Annual Computer Security Applications Conference (ACSAC '18), December 3–7, 2018, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3274694.3274739

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

1 INTRODUCTION

Memory corruption attacks continue to pose a major threat to computer systems. Over the past decades, the sophistication of such attacks has risen from simple code injection [28] to various forms of code-reuse attacks (a.k.a. return-oriented programming – ROP) [4, 8, 37, 38] as a result of the widespread adoption of defenses such as $W \oplus X$ [29].

Preventing memory corruption attacks in legacy, memory unsafe languages such as C/C++ is challenging. Complete memory safety techniques that guarantee spatial and temporal pointer safety often incur large runtime overhead [24, 25]. As a result, lighter-weight defenses have been proposed that enforce weaker policies, but incur lower performance overhead. One class of such defenses randomizes or diversifies code at compile-time, load-time, or runtime [18] to create non-determinism for an attacker. However, code randomization and diversification techniques are shown to be vulnerable to various forms of direct [40], indirect [11], and side-channel [36] information leakage attacks. Even leakage-resilient variants of such defenses are shown to be vulnerable to code inference [39] and indirect profiling attacks [33].

A class of memory defenses that aims to provide a balance between security and performance is Control Flow Integrity (CFI) [6]. CFI aims to prevent control hijacking memory corruption attacks by checking the control flow transfers at runtime. While the policy enforced by CFI does not prevent non-control hijacking attacks (*e.g.,* data-only attacks [17]), the relatively low overhead incurred by CFI and its resilience to information leakage attacks make it one of the desirable classes of defenses. CFI has even been called "one of the most promising ways to stop advanced code-reuse attacks" [49].

One of the distinguishing factors among various CFI techniques is how the control flow graph (CFG) is generated. Three major classes of CFI defenses are: 1) those that generate the CFG statically using points-to analysis [1, 2, 5, 26, 50], 2) those that generate the CFG dynamically at runtime [13, 27, 30], and 3) those that generate the CFG based on type information [21, 23, 44, 46, 49]. We call the third class Runtime Type Checking (RTC)-based CFI (or simply RTC in the rest of this paper). Since points-to analysis is often very imprecise, difficult to modularize, and hard when only the binary is available, many recent CFI techniques are designed based on RTC [21, 23, 44, 46, 49].

In RTC, for forward edge protection, the type of function pointer and the target are checked at each forward edge control transfer. A weaker subclass of RTC techniques only checks the *arity* (argument count) of forward edge transfers, and not the precise type [47, 49]. For backward edge protection (*i.e.,* return address protection), the type of callee is checked during the function epilogue. RAP [46], TypeArmor [49], Kernel CFI (KCFI) [23], MCFI [44], IFCC [47], and LLVM-CFI [21] are some of the examples of RTC techniques. While extensive work has been done on the effectiveness of CFI based on points-to analysis (e.g., [7, 12, 14, 16, 20]), the strength of RTC has not been studied.

To the best of our knowledge, three implementations of RTC are available that protect both forward and backward edges with type checking: KCFI [23], RAP [46], and MCFI [44]. Other approaches such as IFCC [47], LLVM-CFI [21], and TypeArmor [49] only protect the forward edge. While an implementation of KCFI is not available, an open source version of RAP and LLVM-CFI are available. RAP and LLVM-CFI also provide the most stable implementations of RTC as they are targeting production environments, and are not research prototypes (RAP has even been applied to the Linux distribution, Subgraph OS [41]). In addition, RAP provides a more accurate CFG than LLVM-CFI because it removes static functions from the target set, unlike LLVM-CFI. Furthermore, neither RAP nor LLVM-CFI limit the target set to address-taken functions. For these reasons, we focus our analysis on RAP in this paper. In addition, we focus only on C programs because RAP C++ is not free[1].

In this paper, we provide the first study on the security and practicality of RTC. From the security perspective, we illustrate that type collisions exist, and are, in fact, very common in sufficiently large applications. While, at first glance, it may appear that such collisions should be straightforward to exploit (similar to attacks that leverage the imprecisions in points-to analysis-based CFI), we show that practical exploits against RTC face a major challenge: it is unlikely that a corruptible pointer has the exact collision with a desirable function for an attack (*e.g.,* a system call). Indeed, we show that while collisions are abundant, collisions with sensitive targets such as system calls are, in fact, rare. We use a layered invocation method against RTC in which a sensitive function is called indirectly through multiple layers of other calls that eventually end in a call that has a collision with a corruptible function pointer. In other words, a sensitive function is called from a corruptible pointer through various layers of other functions. We call the sensitive function, the function that collides with a corruptible pointer, and the other layers Execution-Gadget (E-GADGET), Collision-Gadget (C-GADGET), and Linker-Gadget (L-GADGET), respectively. Since this form of ROP attack respects the type checking (thus bypassing RTC), we call it *Typed ROP (TROP)* .

In order to illustrate the practicality of TROP, we build two proof-of-concept exploits, one against Nginx and the other against Exim, that successfully hijack control in the presence of RTC. Our exploits successfully bypass the open-source version of RAP [46]. Furthermore, we perform an analysis of exploitable conditions in many popular applications and servers. Our findings indicate that collisions are abundantly found in real-world applications, and that the gadgets necessary for a TROP attack (i.e., C-GADGET, L-GADGET, and E-GADGET) are prevalent in popular servers. Our results suggest that, while RTC techniques complicate successful attacks and are, in many cases, more practical than points-to analysis-based CFI, on their own, they are not sufficient to prevent control hijacking in the face of motivated attackers.

In summary, our contributions are as follows:

- We provide a first in-depth analysis of the effectiveness of RTC techniques.
- We illustrate a code reuse attack, TROP, that can successfully bypass precise RTC even in the absence of collisions with sensitive functions.
- We build two proof-of-concept exploits against Nginx and Exim to show the practicality of TROP.
- We analyze many popular applications and servers, and show that the conditions necessary for a successful attack are abundantly found in the real-world.
- We discuss the practical challenges of adopting RTC techniques in large programs.

2 BACKGROUND AND PROBLEM DEFINITION

Lack of memory management in unsafe programming languages, such as C/C++, has been introducing significant threats since 1988 when the first Internet worm exploited a buffer overflow vulnerability in Fingerd [35]. As a result, there has been a continuous arms race between the development of attacks and defenses.

Defenses in the memory corruption domain can be broadly categorized into enforcement-based and randomization-based techniques. While randomization-based techniques [18] are vulnerable to various forms of information leakage (e.g., [4, 36, 40]) attacks, enforcement-based techniques [1, 24, 25] are resilient to such attacks. Full memory safety techniques that enforce spatial [25] and temporal [24] safety on pointers are examples of enforcement-based defenses. Lighter-weight defenses in the enforcement-based category impose more relaxed policies on code execution at runtime, but provide better performance. Control Flow Integrity (CFI) [6] is an example of such a defense that has received significant attention in the community over the past years, and has even been deployed in real-world systems [31, 45].

2.1 Control Flow Integrity (CFI)

CFI checks the indirect control transfers at runtime to prevent control hijacking attacks [6]. It checks forward-edges (*e.g.,* indirect jumps and calls) and/or backward-edges (*e.g.,* function returns) to prevent the corruption of indirect control transfers via memory bugs. While this policy is weaker than full memory safety (for example, it does not prevent data-only attacks [17]), CFI aims to prevent the most pernicious types of memory corruption attacks at a relatively low performance cost.

CFI techniques can be categorized into three broad classes based on how they generate their control flow graph (CFG). Perhaps the most widely studied class of CFI is points-to analysis-based CFI as it was originally proposed by Abadi *et al.* [1]. In this technique, the CFG is constructed statically using points-to analysis [1, 2, 5, 26, 50]. Another class of CFI techniques construct their CFG dynamically [13, 27, 30]. Dynamic CFIs need additional computations to identify and add new edges to the CFG during execution. In this paper, we do not study them.

[1]We tried to obtain the commercial version of RAP, but unfortunately were not able to do so because, based on our exchanges with the staff at GRSecurity, procuring the commercial version actually requires contracting GRSecurity's security service, and is not as simple as purchasing a software package for a fee.

The effectiveness of points-to analysis-based CFI crucially depends on the ability to construct an accurate CFG. However, a sound and precise CFG is hard to construct in the general case due to the undecidability of points-to analysis [19, 32]. Coarse-grained points-to analysis-based CFI techniques [53, 54] tackle this problem by grouping many branch targets together; however, such over-approximation is shown to be too relaxed to prevent control hijacking attacks [12, 16]. Even fine-grained points-to analysis-based CFI techniques are shown to be too permissive to prevent all forms of control hijacking attacks, either because of the imprecisions of static analysis [14], or because of the versatility of functions like `printf()` [7]. These challenges along with the practical difficulties of generating and handling CFGs in a modular way (e.g., dynamic loading) have motivated the development of a third class of CFI based on Runtime Type Checking (RTC).

2.2 Runtime Type Checking

RTC matches the type signature of each indirect control transfer with its target. For forward-edge protection, RTC checks the type signature of a function pointer and its target prior to each indirect control transfer. For backward-edge protection, the type signature of the callee is stored before the call site. Then this signature is checked during the epilogue of the callee to make sure that such a type signature exists in the call site. In other words, RTC limits the control flow of a program to respect the type signatures. It, thus, implements a form of CFI that relies on types. The type matching can be enforced using label-based annotations, in which labels are 'hashes' of function signatures. What constitutes the type signature requires careful considerations, and is further discussed in Section 2.4.

2.3 Arity Checking

Arity refers to the number of arguments of a function. Arity checking is a strictly weaker form of RTC, in which only the number of arguments of call sites and target functions are matched. In other words, arity checking is a form of RTC in which only one type exists for the argument. Hence, at runtime only the *number* of arguments are compared. Arity checking has been implemented both at the source code (IFCC) [47] and binary levels (TypeArmor) [49]. IFCC assigns each function to a set based on its number of arguments, and only allows indirect calls to the set with the correct number of arguments. TypeArmor, in addition to the number of arguments, considers the return type as well. TypeArmor has been shown to prevent advanced code reuse attacks such as COOP [34].

2.4 Reuse Attack Protector (RAP)

RAP is a code-reuse attack protection that implements RTC. Figure 1 shows an example of type parts for a function pointer and a function that are used in RAP. In this case, which shows a function and a function pointer, the return type (`void`) and the argument types (`int` and `long`) are parts of the type signature, while the names are not. RAP generates two hashes for each function. One of them is used when the function is the target of a function pointer. The other one is used during backward-edge checking.

RAP calculates a hash for each function pointer and instruments the call site, as shown in the line 4 of Figure 2. This line compares

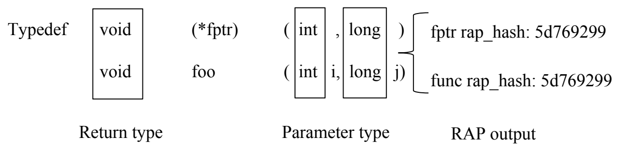


Figure 1: Type parts in RAP.

```
1   dq 0x11223344
2   func:
3   ...
4   cmpq $0x11223344,-8(%rax)
5   jne .error
6   call *%rax
```

Figure 2: Forward-edge checking in RAP.

```
1   jmp label
2   dq 0xffffffffaabbccdd
3   label:
4   call func
5
6   func()
7   .
8   .
9   mov %(rsp),%rcx
10  cmpq $0xaabbccdd,2(%rcx)
11  jne .error
12  retn
```

Figure 3: Backward-edge checking in RAP.

the expected hash with the function's type hash that is located before the actual function's memory address. If the hashes do not match, the execution jumps to an error; otherwise, the indirect call is taken as shown in line 6. Similarly, for backward-edges, the hash value that is located before the call site and the one in epilogue are compared and if they match, the program continues to return as shown in Figure 3. Previous researches have shown that backward-edge protection without enough sensitivity leads to the flexibility to return to different call sites [7, 16]. To address this issue, RAP encrypts the return address with a key which is stored in a register. This technique makes a context sensitive version of RAP. However, this feature is not available in open source version of RAP. Note that RTC without a shadow stack or an equivalent technique such as encrypted return address is vulnerable to backward-edge attacks that have been studied comprehensively by other researchers [7, 16]. Hence, for the rest of this paper, we only focus on forward-edges.

2.5 Type Collisions

An astute reader might suspect that type collisions should exist in sufficiently large code bases. This is indeed correct; however, we show that type collisions do not immediately indicate the feasibility of a practical attack against RTC. To clarify and explain, let us start with an example.

Figure 4 shows a sample source code with intentional vulnerabilities in lines 36 and 37 for leaking and overwriting. This code is compiled by the RAP plugin that protects the code with RTC. Figure 5 shows its CFG. Although type collisions exist in this code (between *invalid_target()* and *corruptible_fptr*), according to RTC,

```c
1  typedef void (*FunctionPointer)(void);
2
3  int flag = 0;
4  char *cmd;
5
6  void valid_target1(void){
7      printf("Valid Target 1\n");
8  }
9
10 void valid_target2(void){
11     printf("Valid Target 2\n");
12 }
13
14 int final_target(char *cmd){
15     system(cmd);
16 }
17
18 int linker_func(void){
19     if (flag ==1)
20       final_target(cmd);
21 }
22
23 void invalid_target(void){
24     linker_func();
25 }
26
27 void vulnerable(char * input){
28     FunctionPointer corruptible_fptr;
29     char buf[20];
30
31     if (strcmp(input, "1") == 0)
32         corruptible_fptr = &valid_target1;
33     else
34         corruptible_fptr = &valid_target2;
35
36     printf(input);
37     strcpy(buf, input);
38
39     corruptible_fptr();
40 }
```

Figure 4: A sample vulnerable program.

it is not allowable to call functions such as *linker_func()* or *final_target()* (which may be interesting targets for an attacker because of their ability to spawn a malicious shell) with the *corruptible_fptr* function pointer. Such a call is prohibited because the type of the functions and the *corruptible_fptr* function pointer are different. As a result, while many type collisions might exist in large code bases, their usefulness for a practical attack is questionable. There may not exist any sensitive target that has type collision with a corruptible function pointer. Indeed in our analysis of real-world code bases, we rarely found a package in which a sensitive function (*e.g.,* a system call) collides with a corruptible function pointer. Thus, the questions regarding the effectiveness of RTC are not trivial to answer.

However, we make an observation in this sample source code that provides an insight into how an attack can be built. We observe that while the sensitive functions cannot directly be called by the *corruptible_fptr* function pointer, it is still possible to invoke these functions through other functions. In this case, the *invalid_target()* function has the same type as the *corruptible_fptr* function pointer, so it is feasible to call this function, and as it can be seen, there is a path from this function to the *final_target()* function. At least in theory, it looks like that it should be feasible to reach the *final_target()* function indirectly from the *corruptible_fptr* function pointer. To do so; however, one must consider the constraints in the execution path such as the *if* condition in line 19. In this case, the constraint is satisfiable because the *if* condition checks a global variable which can be overwritten by the attacker.

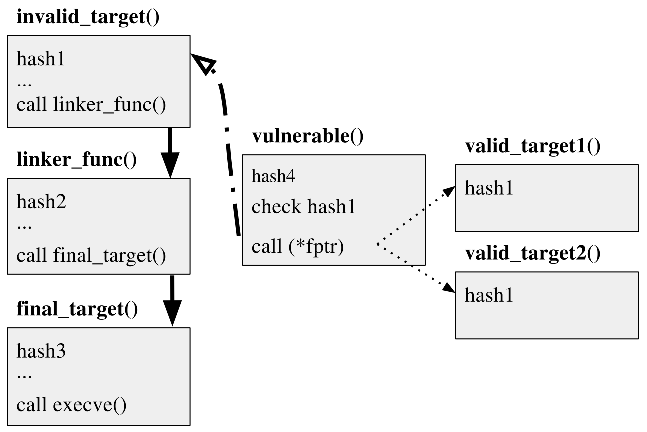


Figure 5: Control flow graph of the sample vulnerable program. This figure illustrates how type collision leads to over-approximation.

2.6 Research Questions

The above example and the observations made about possible collisions and indirect invocations of sensitive functions raise the following research questions:

- Can RTC be practically bypassed using type collisions?
- Are there enough intermediate functions with satisfiable constraints in real-world applications that allow an attacker to hijack control to a sensitive function (*e.g.,* system calls) in the presence of RTC?
- How prevalent are these constructs in real-world applications?

In the rest of the paper, we provide answers to these questions in order to evaluate the effectiveness of the RTC from the security perspective in Sections 3, 4, and 5. We discuss the practicality considerations in Section 6.

3 ATTACK OVERVIEW

In order to evaluate the effectiveness of the RTC, we show how an attacker can exploit type collisions to build real attacks. Although RTC significantly reduces the number of valid targets, we show that the scope and prevalence of collisions make it possible to attack many real-world applications.

3.1 Threat Model

Our threat model is inline with the larger body of literature in the area of memory corruption. Since data execution prevention (a.k.a. DEP or W $\oplus$ X) and address space layout randomization (ASLR) are widely deployed in modern operating systems, we assume that they are enabled on the target. Moreover, we assume that RAP is also deployed on the target. Any control hijacking attempt that violates type checking is properly detected and stopped. Moreover, since we are interested in studying the effectiveness of the RTC paradigm as a whole, and not the quality of any particular implementation, we consider *implementation* bugs or flaws out-of-scope for our attacks. In other word, we do not target any implementation weakness.

On the attacker side, we assume the target application contains one strong or multiple limited memory corruption vulnerabilities that allow an attacker to write arbitrary values to arbitrary memory locations including stack and heap and also leak some information by arbitrary read primitive. A strong vulnerability similar to CVE-2017-7184 (exploited in Pwn2Own 2017), CVE-2017-0143 (exploited in WannaCry), CVE-2016-4117, or CVE-2015-0057 simultaneously provides arbitrary read and write primitives, and can be exploited multiple times to overwrite multiple values in memory [15]. A more limited set of vulnerabilities can alternatively be used to first read memory, and then write to it maliciously. For instance, array out-of-bound access (CVE-2018-5008), non-terminated strings (CVE-2017-7790) and format string can be used for arbitrary read primitive. Other vulnerabilities such as double free (CVE-2018-4990), type confusion (CVE-2015-1641) and format string can be used for arbitrary write primitive. The long history of memory corruption vulnerabilities has demonstrated that assuming their existence even in the most tested code bases (*e.g.,* Linux kernel and Windows Services) is a reasonable and often valid proposition. The assumptions in this work are thus realistic and inline with the defenses [3, 9, 10, 22, 49] and attacks [4, 11, 33, 34, 38] presented in the literature.

3.2 Attack Preliminaries

In order to perform a successful Typed ROP (TROP) attack against RTC, an adversary needs to follow some steps to execute an arbitrary code. First, during the offline preparation phase, the attacker searches for an interesting function that allows arbitrary execution, for example *execve()* or any other function that has an equivalent behavior. To do so, the arguments of an intended function must be controllable by the attacker. For example, if the arguments reside on heap or global memory, they can be overwritten at anytime by an attacker. Next, the attacker needs to change a function pointer to hijack control. It is imperative to note that the type of the function pointer and the function should be the same; otherwise, RTC will prevent the control transfer. This is what distinguishes a TROP attack from a traditional ROP attack; *i.e.,* a TROP attack is a special form of ROP in which the hijacked control respects the type checking.

Although this attack looks powerful in that it allows arbitrary code execution, as mentioned earlier, it is very unlikely to find such a type collision in real-world applications. This is due to the fact that the attacker needs to have access to a corruptible pointer that has the same type as the sensitive function of interest for hijacking (in this case *execve()*). In order to address this challenge, we use a technique based on layered invocations and type collisions, which gives the attacker the opportunity to call functions even with differing types from a corruptible function pointer. In a nutshell, a target function of interest (*i.e.,* a sensitive function) is invoked indirectly through other intermediary functions in such a way that: 1) the first invoked function has the same type signature as the corrupted function pointer, 2) each function in the chain contains a valid invocation of the next function in its code, and 3) the last function in the chain is the target function of interest.

To facilitate the description and analysis of TROP, we introduce three types of gadgets that make it possible. We call the first function that has a type collision with a corruptible function pointer a Collision gadget (C-GADGET), the target function that the attacker intends to invoke maliciously an Execution gadget (E-GADGET), and the intermediary functions that are called in a nested fashion (from the collision gadget to the execution gadget) Linker gadgets (L-GADGET).

Figure 6 illustrates the high-level view of the location of the gadgets. Each node is a representation of a gadget. The attacker first corrupts a function pointer to redirect control to the C-GADGET, which, by definition, respects the type checking. The C-GADGET then calls other L-GADGETS that, in turn, eventually call the E-GADGET.

3.3 Finding Gadgets

In this section, we describe the method to find and chain proper gadgets to perform a TROP attack. In our terminology, we define the whole function as a gadget. Finding gadgets starts with a corruptible function pointer. For example, any function pointer on the stack or heap adjacent to an overflow-able buffer (*e.g.,* CVE-2016-9679), or those that have known or leakable addresses (*e.g.,* CVE-2017-7219) are potential targets for corruption. The ideal scenario is to point this pointer to the final target, but due to RTC, this is not possible in most cases, so instead, the control is transferred to a C-GADGET. Any function that has the same type signature as the corruptible function pointer is a candidate for a C-GADGET. The next step is to find an appropriate E-GADGET. For E-GADGET, we chose to focus on functions that spawn a shell (*e.g.,* `execve` system call or their wrappers) since that provides an attacker with a wide range of malicious capabilities sought after in the payloads. However, an attacker may choose more limited functions (*e.g.,* one that disables DEP) if those are sufficient for the ultimate purpose of the attack. After selecting the C-GADGET and E-GADGET, we need to find proper L-GADGETS to chain them together. With the help of the program's call graph, both direct and indirect calls, L-GADGETS can be found by traversing all the candidate paths starting from a C-GADGET and ending in an E-GADGET.

We have developed a semi-automated tool that receives as input the set of all C-GADGETS and E-GADGETS as well as the program's call graph, and finds all candidate L-GADGETS in a given code base. The tool uses RAP's output during compilation which is in verbose mode. The call sites and target sets are identifiable in this output. The simplified algorithm inside our tool for finding all candidate paths is listed in Algorithm 1. In the algorithm:

(1) G is the adjacency matrix of the program's direct and indirect call graph
(2) N is the set of all nodes (functions) in the graph
(3) CG is the set of all C-GADGETS
(4) EG is the set of all E-GADGETS
(5) CP is the set of all candidate paths
(6) cg is the C-GADGET
(7) eg is the E-GADGET
(8) V is the set of already visited nodes (functions) in the graph in order to prevent loops through recursive path discovery algorithm (DISCOVERPATHS)

(9) P is an ordered list of functions in the call chain

Algorithm 1 Finding candidate paths.

```
function FindCandidatePaths($G$, $N$, $CG$, $EG$)
    $CP \leftarrow \emptyset$

    for $cg \in CG$ do
        for $eg \in EG$ do
            DiscoverPaths($CP$, $cg$, $eg$, $\emptyset$, $\emptyset$)
        end for
    end for

    return $CP$
end function

function DiscoverPaths($CP$, $cg$, $eg$, $P$, $V$)
    $P \leftarrow P \cup \{cg\}$, $V \leftarrow V \cup \{cg\}$

    if $cg == eg$ then
        $CP \leftarrow CP \cup \{P\}$
    else
        for $g \in N$ do
            if $g \notin V$ and $G[cg][g] = 1$ then
                DiscoverPaths($CP$, $g$, $eg$, $P$, $V$)
            end if
        end for
    end if

    $P \leftarrow P - \{cg\}$, $V \leftarrow V - \{cg\}$
end function
```

As it can be seen in Algorithm 1, the FindCandidatePaths function iterates over all C-gadgets and E-gadgets, and by using DiscoverPaths, it finds all the possible paths (*candidate* paths) between every combination of C-gadget and E-gadget. Finding proper L-gadgets is more challenging though. We define an ideal L-gadget as one that satisfies these conditions:

(1) The constraints inside the gadgets are controllable from outside of it. For example, the constraints are based on global variables that could be maliciously modified from outside of the gadget.
(2) There is no constraint inside of the L-gadgets that affects the flow to the E-gadget.

As it can be seen in Figure 6, there might be multiple paths from a C-gadget to an E-gadget. Some paths are useful while others are not. Empty nodes depict L-gadgets that do not have at least one of the two attributes mentioned above, so they are eliminated. The shaded nodes depict good candidates to reach the target. There are also some cases where there is a pointer that can be used to switch between different paths (in this example, nodes 1 and 2). This provides more flexibility to the attacker.

We also note that it is possible to create a loop using these gadgets. Creating a loop gives the opportunity to the attacker to trigger a vulnerability multiple times. This is depicted in Figure 6 by edge #3.

3.4 Constraint Solving

As we consider the whole function as a gadget, there might be some conditions in L-gadgets which influence the control flow of the program. If an L-gadget contains a condition that prevents reaching the intended E-gadget, then it needs to be resolved. For instance, if there is an *if* condition in the L-gadget that returns from the function, and halts the flow to the invocation point of

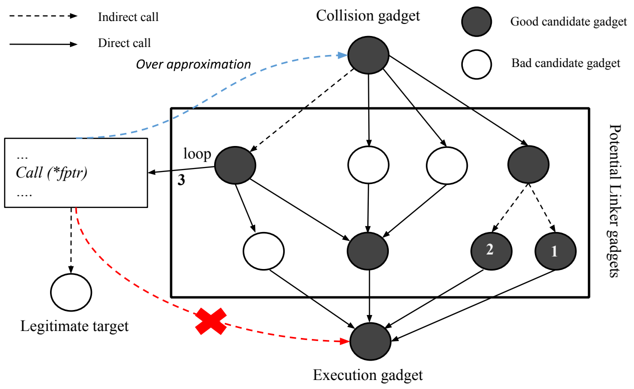


Figure 6: The location of gadgets in the CFG of the program. Due to the type mismatching, the `fptr` is not able to point to the execution gadget directly; hence it utilizes over approximation edge.

the E-gadget, or if there is a null data structure that is needed to be filled with proper data before the L-gadget can be executed, additional constraints need to be satisfied. Therefore, the list of constraints between a C-gadget and an E-gadget is every point in the program that changes the flow between these two nodes. To see if the constraints before that point are satisfiable, the following conditions should be in place:

(1) The constraint can be solved by overwriting data in memory.
(2) The data should be also accessible globally in memory outside the function.

The first condition ensures that the constraint can actually be controlled by a malicious memory overwrite, while the second condition ensures that the window of opportunity for overwriting memory is not too short. For example, while local variables may become corruptible during limited windows of time, their sometimes short longevity makes them inappropriate targets for constraint solving. In the last step, to get the concrete values for the constraints, depending on the number of constraints and complexity of them, either manual solving or symbolic execution are applicable. For our PoC exploits, we solved the constraints manually.

4 PROOF-OF-CONCEPT EXPLOITS

As a proof-of-concept, we show how the Nginx web server and the Exim mail server protected by RTC can be exploited by TROP attacks. In both cases, we manage to achieve arbitrary execution while the server is protected by open source version of RAP. We show how an attacker is able to redirect control to a function that calls a function from the *exec* family. This enables us to execute arbitrary commands via the spawned shell, which is a powerful attack. To do so, we compiled Nginx-1.10.1 and Exim-4.89 with the RAP GCC plugin. As we assumed in our threat model that we have arbitrary read and write access to the memory, gdb is used to mimic these primitives.

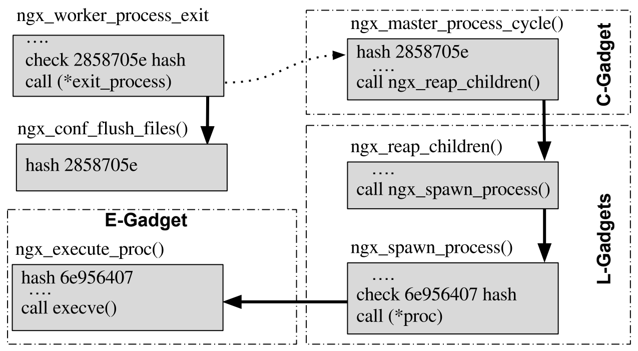


Figure 7: The flow of the exploit in Nginx and the corresponding gadgets.

4.1 Nginx Exploit

In the best case scenario, an attacker needs at least two types of gadgets, C-GADGET and E-GADGET. However, our analysis of Nginx using our tool indicates that there is no direct collision between a potentially corruptible function pointer and a sensitive function (*e.g.,* a system call). We thus need to find enough L-GADGETS to link corruptible function pointers with a sensitive function such as *execve()*. Searching through Nginx's code using the algorithms described in the previous section, we can indeed find enough L-GADGETS with satisfiable constraints.

The overview of our Nginx attack is illustrated in Figure 7. The attack starts by corrupting the *exit_process* function pointer. Both the *exit_process* function pointer and the *ngx_master_process_cycle()* function have the same type signature, as it can be seen in Figure 8. Therefore, *ngx_master_process_cycle()* function can be used as our C-GADGET. Due to the matching type signatures, such a malicious redirection is not detected by RTC. Even though *ngx_conf_flush_files()* function is the the valid target of the *exit_process* function pointer, due to the type collision, we can change it to point to the *ngx_master_process_cycle()* function instead. Our analysis indicates that the *ngx_execute_proc()* function, which calls *execve()* directly, can be a great candidate for our E-GADGET, so we use it in our attack. Figure 9 shows that the parameters of the *execve()* function are also controllable in a global data structure. Now that we have our C-GADGET and E-GADGET, we have to chain them with enough L-GADGETS. Our analysis based on the call graph of Nginx indicates that two good candidates for L-GADGETS are the *ngx_reap_children()* and the *ngx_spawn_process()* functions. Figure 10 shows there is the *proc* function pointer in our L-GADGET which provides more flexibility for the attacker to change the flow.

Constraint Solving. There are some constraints in our C-GADGET and L-GADGETS that should be solved. For example, there is an *if* condition in the *ngx_master_process_cycle()* function on *ngx_reap* variable that might prevent execution from reaching *ngx_reap_children()*. Moreover, there are multiple *if* conditions in the *ngx_reap_children()* function, our first L-GADGET, that have to be satisfied to call the *ngx_spawn_process()* function, our second L-GADGET. Our second L-GADGET (*ngx_spawn_process*) always calls our E-GADGET without any interruption, so in this case, there is no constraint to be satisfied.

```
// Type definition of exit_process pointer
void (*exit_process)(ngx_cycle_t *cycle)
// Type definition of ngx_master_process_cycle
void ngx_master_process_cycle(ngx_cycle_t *cycle)

void ngx_master_process_cycle(ngx_cycle_t *cycle) {
  ...
  // This function helps to create a loop. It
       calls (*exit_process) in the following
  ngx_start_worker_processes(cycle,
       ccf->worker_processes, NGX_PROCESS_RESPAWN);
  ...
  // By setting this condition to true, the
       attacker can reach to the next gadget which
       is ngx_reap_children()
  if (ngx_reap) {
      ngx_reap = 0;
      ngx_log_debug0(NGX_LOG_DEBUG_EVENT,
           cycle->log, 0, "reap children");
      live = ngx_reap_children(cycle);}
```

Figure 8: ngx_master_process_cycle is a C-GADGET

```
ngx_execute_proc(ngx_cycle_t *cycle, void *data)
{
    ngx_exec_ctx_t  *ctx = data;
    if (execve(ctx->path, ctx->argv, ctx->envp) ==
         -1) {
        ngx_log_error(
        NGX_LOG_ALERT,
        cycle->log,
        ngx_errno,
        "execve() failed while executing %s
             \"%s\"",
        ctx->name, ctx->path);
    }
    exit(1);
}
```

Figure 9: ngx_execute_proc is an E-GADGET. The parameters of execve function are controllable globally.

In order to satisfy the constraints listed above, we observe that they are all controllable by overwriting global variables or heap objects. Consequently, in our exploit, we first use our arbitrary write vulnerability to set the value of *ngx_reap* that resides in global memory to 1. This allows our C-GADGET (*ngx_master_process_cycle()*) to call our first L-GADGET (*ngx_reap_children()*). We then use the same vulnerability to write the desired values to *ngx_processes[i].respawn*, *ngx_processes[i].exiting*, *ngx_terminate*, and *ngx_quit* on the heap that allows our first L-GADGET (*ngx_reap_children()*) to call our second L-GADGET. Then we overwrite the function pointer (*exit_process*) to point to our C-GADGET. After this overwrite, the execution passes from C-GADGET to the L-GADGETS to the E-GADGET, at which point a malicious shell is spawned under our control.

4.2 Exim Exploit

For the sake of brevity and due to its similarity to Nginx exploit, we skip the details of the steps necessary to build Exim exploit. We find that Exim's source code is indeed large enough that it contains proper gadgets, and a large number of corruptible memory read and write operations. A previous example of such a corruption happened with CVE-2016-9963 that allowed a remote attacker to read the private DKIM signing key, and write it to a log file. In our

```
ngx_spawn_process(
  ngx_cycle_t *cycle,
  ngx_spawn_proc_pt proc,
  void *data,
  char *name,
  ngx_int_t respawn) {
...
  proc(cycle, data); // The proc function pointer invokes ngx_execute_proc function in this case
}
```

Figure 10: `ngx_spawn_process` is an L-GADGET

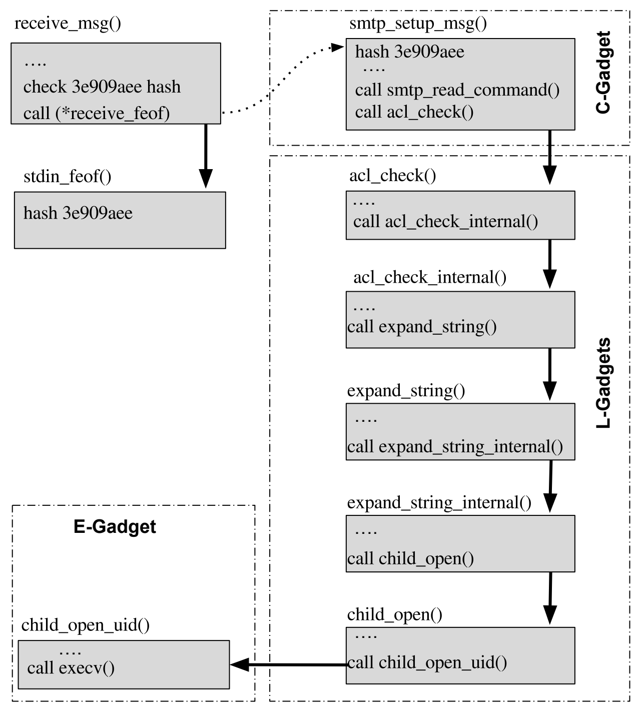


Figure 11: The flow of the exploit in Exim and the corresponding gadgets.

exploit, we again focus on spawning a malicious shell because of its generality and strength as a remote attack.

Like our Nginx exploit, we use our gadget finder tool to find the proper gadgets and paths in the program that can be used to reach an E-GADGET, in Exim's case, the *child_open_uid()* function. We maliciously overwrite the *receive_feof* function pointer inside the *receive_msg()* function and redirect execution to the *smtp_setup_msg()* function, which serves as our C-GADGET. The functions *acl_check()*, *acl_check_internal()*, *expand_string()*, *expand_string_internal*, and *child_open()* serve as our L-GADGETS with easily satisfiable constraints. Note that while the chain of gadgets is much longer in the case of Exim, the exploit is indeed simpler because there are fewer constraints in the L-GADGETS. The flow of the exploit and the position of gadgets are shown in Figure 11. In order to bypass some constraints of the L-GADGETS, we used `${run{/bin/bash}}$\\` as the payload. This complex payload helps us execute the shell command and solve the constraints.

4.3 Summary

These two exploits indicate that although RTC does complicate a successful exploit and places a number of limitations on how such an exploit can be built, motivated attackers may still find feasible and realistic opportunities to build damaging TROP attacks using type collisions.

5 EVALUATION

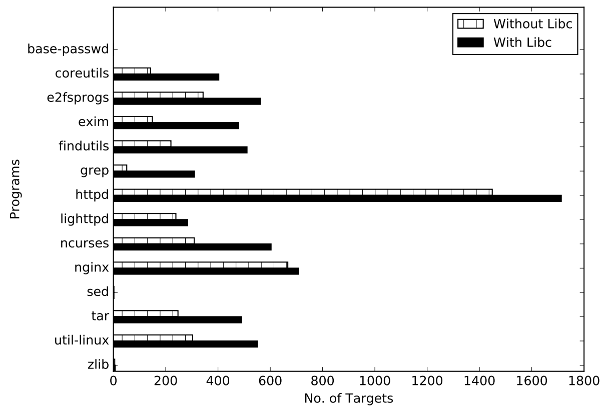
In this section, we evaluate the prevalence of collisions and gadgets necessary for launching a TROP attack. For our evaluation, we chose the top 10 C packages in the Ubuntu repository [48] as well as three widely used web servers (Httpd, Nginx, and Lighttpd) and a mail server (Exim). The list of the applications are shown in Table 1. For each of these applications, we compiled it with the RAP GCC plugin. We then used our gadget finder tool that parses the RAP's output and generates a JSON file containing the function pointers as well as their call sites in the program. In addition, our tool extracts the target functions for each of these call sites. In all of our evaluation, the default modules of the analyzed applications were used. In addition, the open source version of RAP was deployed with default options and compilation flags.

5.1 Type Collisions

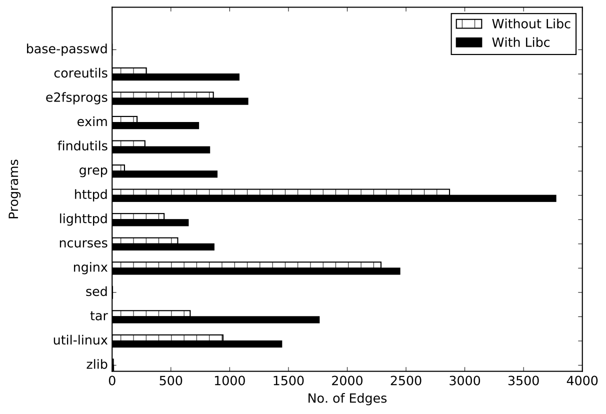
In order to establish a baseline for the accuracy of type signatures, we have to distinguish valid and invalid targets. Because of the imprecision of points-to analysis, no automated analysis can perform sound and precise determination of valid and invalid targets, so we first manually analyze the JSON output and label targets as valid or invalid using careful inspection of the source code. This establishes the baseline to which we can compare RAP's type signatures. Note that the manual labeling is not because of the limitations in the analysis of this effort; rather, it is a fundamental limitation of points-to analysis. Because no automated analysis can establish sound and precise points-to analysis, comparing RAP's targets with any other automated analysis would be meaningless; there would be no basis to believe one as the ground truth. Moreover, if such a precise automated analysis existed, points-to analysis-based CFI could have been implemented precisely. However, we know that points-to analysis is imprecise, and in fact, the imprecisions have been shown in the previous work [14]. That is why we manually label targets. The rest of our analysis is automated.

Table 1 illustrates the statistics for the analyzed applications, among which the number of function pointer signatures defined by the program and the number of all locations from which these function pointers are being called (call sites). For example, observe that applications such as `httpd`, heavily use function pointers due to their modular design.

Furthermore, we extract all functions defined in the programs, as well as the functions for which RAP generates a type signature (hash). Interestingly, RAP generates no hash for 34.6% of the functions, on average, across all programs. One reason is the fact that RAP does not generate a hash for `static` functions which cannot be invoked by external callers and are not called indirectly locally. Our analysis indicates that a large fraction of functions are *not* called indirectly. Specifically, only 15.2% of functions are also valid targets on average across all programs. In other words, about 50.2%



(a) Targets



(b) Edges

Figure 12: Increase in the number of targets and edges when linked with `glibc`.

Table 1: Over-approximantion of target functions and indirect calls because of type collisions.

App	Version	Function Pointer	Call Sites	Functions	Functions w/ Hash	Function Targets		Indirect Calls	
						All	Invalid	All	Invalid
base-passwd	3.5.39	6	6	45	45 (100.0%)	0	0 (0.0%)	0	0 (0.0%)
coreutils	8.2	42	80	1,789	682 (38.1%)	116	43 (37.1%)	416	110 (26.4%)
e2fsprogs	1.42.13	97	264	1,964	1,243 (63.3%)	251	176 (70.1%)	1,383	400 (28.9%)
exim	4.89	43	93	968	607 (62.7%)	88	121 (137.5%)	359	165 (46.0%)
findutils	4.6.0	28	52	821	554 (67.5%)	200	89 (44.5%)	326	65 (19.9%)
grep	2.25	19	28	460	264 (57.4%)	38	19 (50.0%)	113	52 (46.0%)
httpd	2.4.25	248	546	2,800	2,338 (83.5%)	1,332	483 (36.3%)	3,915	794 (20.3%)
lighttpd	1.4.45	27	108	899	524 (58.3%)	228	40 (17.5%)	830	221 (26.6%)
ncurses	6.0	46	77	1,835	1,045 (56.9%)	156	273 (175.0%)	969	397 (41.0%)
nginx	1.10.1	84	290	1,299	977 (75.2%)	610	319 (52.3%)	5,977	3,512 (58.8%)
sed	4.2.2	1	1	213	140 (65.7%)	2	0 (0.0%)	2	0 (0.0%)
tar	1.28	46	86	1,166	730 (62.6%)	141	166 (117.7%)	1,008	754 (74.8%)
util-linux	2.27.1	53	75	3,143	1,681 (53.5%)	211	177 (83.9%)	1,060	643 (60.7%)
zlib	1.2.8	5	14	152	108 (71.1%)	5	0 (0.0%)	13	0 (0.0%)

of functions on average have a hash generated for them, while they are not called indirectly at all. This, consequently, increases the hash collisions and creates further opportunities for an attacker to call these functions using irrelevant function pointers with the same signature. As mentioned before, it is not uncommon for practical implementations to include non-address taken functions into the target sets. We observed this behavior in both RAP and LLVM-CFI implementations.

The important factor for estimating the impact of type collision is determining the number of functions that are not valid targets of function pointers but can be indirectly called through one of the function pointers in the program. The eighth and tenth columns of Table 1 show the number of invalid target functions and invalid indirect calls possible in the program because of type collisions.

These numbers underline the scope of the weakness created by type collisions. In Nginx alone, for example, 3,512 function pointers can invalidly point to 319 functions which are never intended to be indirect targets just because they happen to share the same type signature. Note that in applications that rarely use function pointers, such as `base-passwd` and `sed`, the number of possible corruptible indirect calls because of type collision is zero, but these applications are the exceptions rather than the rule. In our analysis, any application that contains more than tens of function pointers present abundant opportunities for function pointer corruptions that respect the RTC.

Table 2: Gadget distributions.

App	Version	C-gadget	L-gadget	E-gadget
nginx	1.10.1	8	6	1
httpd	2.4.25	40	19	5
lighttpd	1.4.45	8	29	6
exim	4.90	16	32	7

5.2 Gadget Distribution

Now that we know type collisions can result in numerous opportunities for control redirection, we shift our focus to counting the gadgets necessary for an exploit. We use the algorithms described in Section 3. Table 2 shows the number of gadgets in four popular web and mail servers. Any combination of these gadgets could be a new invocation chain as described in section 3.3 and 3.4. However, more inter-procedural analysis might be needed to determine controllable L-GADGETS. For these results, we limited our analysis only to the programs themselves and not the linked libraries because this provides a more accurate result, and avoids double counting gadgets in overlapping sets of linked libraries. Moreover, as described earlier, the E-GADGETS can be many different targets depending on the exact goal of the payload (*e.g.*, modifying target configurations, disabling W ⊕ X, running a shell script, *etc.*). For this analysis, we chose a general, yet powerful type of E-GADGETS, namely those that allow arbitrary execution via spawning a malicious shell (*e.g.*, *exec* family or *system*). We also chose C-GADGETS based on the path to the E-GADGETS from the list of invalid function targets.

As can be observed from the table, there are many gadgets available in these applications. In fact, Nginx has the lowest number of gadgets among the four servers analyzed, but as demonstrated, we could successfully hijack its control and launch a malicious shell. In our exploit, we used four gadgets from the 15 available in Nginx.

Linked libraries provide numerous other opportunities for malicious control hijacking in the face of RTC. For the sake of simplicity and accuracy, our gadget counts in Table 2 do not include the gadgets from linked libraries, but for the sake of completeness, we now evaluate the impact of linked libraries, primarily the ubiquitous library in Linux applications and servers, Libc.

5.3 Libc

For complete protection, the linked libraries must also be protected with RTC. However, counter-intuitively, here we show that protecting linked libraries with RTC significantly increases the number of opportunities for the attacker.

To evaluate the impact of linked libraries, we compile the applications listed in Table 1 this time with `glibc`[2], and recount the number of invalid indirect targets and invalid edges (indirect calls), introduced by the additional collisions. Figure 12a shows the number of targets in the analyzed applications with and without `glibc`. We observe that linking `glibc` significantly increases the number of targets that have collisions with the function pointers in the applications, thus magnifying the opportunities for an attack. The new collisions open new paths for attackers to transfer control from the function pointers in the application to the functions inside `glibc` while respecting RTC. For example, in `coreutil`, there are 297 new target functions to which the execution can be transferred, and in `grep`, 291 target functions are added to the existing gadgets.

More important than potential targets, however, are the additional edges. Figure 12b illustrates the additional edges (indirect calls) allowed in the applications as a result of collisions introduced by `glibc`. We observe that the number of the new edges is much higher than the number of new target functions in `glibc`. This is because different function pointers can call all of the colliding target functions, thus growing the number of possible edges multiplicatively.

[2]https://www.gnu.org/s/libc/

Table 3: Invalid indirect calls added to programs because of type collisions and imprecise points-to analysis.

App	Base	Type Checking		Points-to Analysis	
		Total	Invalid	Total	Invalid
base-passwd	0	0	0 (0.0%)	0	0 (0.0%)
coreutils	213	291	78 (26.8%)	308	198 (64.3%)
e2fsprogs	557	861	304 (35.3%)	42	15 (35.7%)
exim	107	212	105 (49.5%)	169	99 (58.6%)
findutils	237	279	42 (15.1%)	448	231 (51.6%)
grep	54	105	51 (48.6%)	108	60 (55.6%)
httpd	2,126	2,870	744 (25.9%)	-	-
lighttpd	327	442	115 (26.0%)	1,096	938 (85.6%)
ncurses	291	558	267 (47.8%)	507	238 (46.9%)
nginx	1,276	2,287	1,011 (44.2%)	-	-
sed	2	2	0 (0.0%)	2	0 (0.0%)
tar	208	664	456 (68.7%)	360	167 (46.4%)
util-linux	311	943	632 (67.0%)	596	465 (78.0%)
zlib	10	10	0 (0.0%)	10	4 (40.0%)

5.4 Type Checking vs. Points-to Analysis

Previous attacks such as Control Jujutsu [14] describe the importance of precise CFG. Even though our focus is evaluating RTC-based CFI variants, we posit that it can be useful to compare the over-approximation in RTC with points-to analysis-based CFI. Recall that the over-approximation in points-to analysis-based CFI is because of imprecise points-to analysis, while in RTC-based CFI, it is because of type collisions.

In order to compare the two, we calculate the number of invalid edges allowed because of their imprecisions. As a baseline for points-to analysis-based CFI, we use SVF [42, 52], which is the state-of-the-art in flow sensitive points-to analysis. We again used manual labeling and source code inspection to identify valid and invalid edges. Table 3 shows the number of total and invalid edges in both type checking and points-to analysis. The base column shows the ground truth (manual labels). The results provide no clear advantage for one or the other approach in terms of the number of targets. In 4 of the 14 programs, RTC is less accurate than points-to analysis, while in 8 of them, it is the opposite. However, we observe that flow sensitive points-to analysis is not always possible. For example, for Nginx and Httpd, SVF was not able to finish the analysis process, and in fact crashed after 5 hours (showed by a dash in the table). On the other hand, RTC can be applied in large code bases more easily. This experiment suggests that RTC is a more practical solution which offers almost the same security guarantees for large real-world programs.

6 DISCUSSION

There are further practical challenges of implementing RTC for real-world applications. For the sake of completeness, we review them briefly in this section.

6.1 Type Diversification

As it was shown in our study, the main source of problem in RTC is type collision. RAP proposes a type diversification technique in order to generate unique types. Diversification of colliding types,

```
typedef ngx_int_t (*ngx_output_chain_filter_pt)
  (void *ctx, ngx_chain_t *in);

static ngx_int_t ngx_http_charset_body_filter
  (ngx_http_request_t *r, ngx_chain_t *in)
```

Figure 13: Type mismatch between the `ngx_output_chain_filter_pt` function pointer and its target `ngx_http_charset_body_filter`.

in such a way that each function pointer only shares type signature with its true target would reduce the type collision problem. By this technique, the attacker will not be able to find C-GADGET and therefore the attacks like TROP could be mitigated from the beginning. However, such a diversification, unlike generic RTC, requires a precise points-to analysis to establish the exact target of each function pointer. As discussed earlier, precise points-to analysis is shown to be hard, and the imprecisions are shown to be exploitable [14]. In other words, type diversification re-introduces the challenge faced by points-to analysis-based CFI, namely the imprecision of points-to analysis, to RTC.

6.2 Separate Compilation

Large code bases are often compiled in separate compilation units at different times. This allows easier collaboration and debugging in large projects. However, separate compilation further complicates type diversification. Since not all source files are available during each compilation step, the colliding types to diversify are not known to RTC. If each compilation unit is diversified independent of the other units, false positives will be introduced and execution will halt whenever function pointer from a compilation unit benignly calls a function in a different compilation unit (because their diversified types are very likely different). Consequently, proper type diversification in the presence of separate compilation units requires additional meta data and tracking mechanisms to avoid breaking functionality.

6.3 Mismatch types

The main assumption in RTC is that the types of caller and callee are the same. However, this assumption is not always true. For example in real-world applications there are cases that some part of the function pointers are *void** and these function pointers are able to point to any other pointer types such as *int** and *long**. According to RTC, this is a mismatch. However, in fact, this feature makes the programs more flexible. Figure 13 shows an example of this type mismatch between the function pointer with its valid function target in Nginx.

In order to address this challenge, there are two possible approaches. The first approach is to cast all the *void** to the proper types in order to prevent any mismatch. This approach, used by RAP, requires large modifications in the source code, which makes it time-consuming. The second approach is to generalize *void** to other types and allow matching of *void** to any pointer type. Although this method does not require source code modifications, it makes the RTC more relaxed and creates more opportunity for attacks.

6.4 Support for Assembly Code

In many low-level libraries, there are portions of the code that are written in in-line assembly. Prominently, `glibc` implements system calls in assembly. Automated annotation of assembly with type information is hard. A permissive policy for assembly code could create additional exploitation opportunities, while a restrictive one crashes benign applications. Further work is necessary in this area to safely handle assembly code.

7 RELATED WORK

The related work in the area of memory attacks and defenses is vast. For broader treatment of the related work, we refer the reader to surveys and systematization of knowledge papers in this area [6, 18, 43]. Here we limit our discussions to closely related efforts. Control Jujutsu attack [14] bypasses points-to analysis-based CFI using the additional edges introduced to a CFG because of the imprecision of points-to analysis. Control Jujutsu further demonstrates that certain coding practices such as recursions and modular design exacerbate the imprecision of context-sensitive and flow-sensitive points-to analysis. Control-Flow Bending [7] also attacks CFI. It assumes a fully-precise CFG, but uses versatile functions such as `printf()` as its mechanism. Our attack is not reliant on any specific function like that. Furthermore, a CFB attack is prevented by the modern `Fortify Source` option in libc that stops "%n" format string attacks [51]. Counterfeit Object Oriented-Programming (COOP) [34] is another attack on CFI. COOP exclusively relies on C++ virtual functions for its exploitation technique.

8 CONCLUSION

In this work, we evaluated the effectiveness of CFI techniques based on Runtime Type Checking (RTC). We examined RTC from security and practicality perspectives. We showed that while direct type collisions between corruptible forward edges and target functions are rare, type collisions with other functions can be exploited in a nested fashion to implement an attack (TROP). We further evaluated the prevalence of opportunities for such attacks and showed that both the type collisions and the gadgets necessary for TROP attacks are abundantly found in many real-world applications. We also compared the imprecisions of RTC and points-to analysis techniques and found that their strength is heavily dependent on the code base. Our findings indicate that, while RTC is a practical defense that can complicate exploitation, on its own, it is not sufficient to prevent control-hijacking memory corruption attacks.

ACKNOWLEDGMENTS

We thank our shepherd, Sangho Lee, and the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 1409738. This research was also sponsored by the U.S. Department of the Navy, Office of Naval Research, under Grant No. N00014-15-1-2948.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS).*
[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing memory error exploits with WIT. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on.* IEEE, 263–277.

[3] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *ACM Conference on Computer and Communications Security (CCS)*.

[4] Andrea Bittau, Adam Belay, Ali José Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *IEEE Symposium on Security and Privacy (S&P)*.

[5] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM, 353–362.

[6] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2016. Control-Flow Integrity: Precision, Security, and Performance. https://arxiv.org/abs/1602.04056. In *arXiv*.

[7] Nicolas Carlini, Antonio Barresi, Mathias Payer, and David Wagner. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*.

[8] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav ShachamâĂă, and Marcel Winandy. 2010. Return-Oriented Programming Without Returns. In *ACM Conference on Computer and Communications Security (CCS)*.

[9] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. 2015. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *IEEE Symposium on Security and Privacy (S&P)*.

[10] Stephen Crane, Stijn Volckaert, Felix Schuster, Christopher Liebchen, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Thorsten Holz, Bjorn De Sutter, and Michael Franz. 2015. It's A TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[11] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. 2015. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Network and Distributed System Security Symposium (NDSS)*.

[12] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium*.

[13] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient protection of path-sensitive control security. In *26th USENIX Security Symposium (USENIX Security 17). Vancouver, BC: USENIX Association*. 131–148.

[14] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*.

[15] gera and riq. 2002. Advances in Format String Exploitation. http://phrack.org/issues/59/7.html. (2002).

[16] Enes GÃűktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*.

[17] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *IEEE Symposium on Security and Privacy (S&P)*.

[18] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *IEEE Symposium on Security and Privacy (S&P)*.

[19] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[20] Christopher Liebchen, Marco Negro, Per Larsen, Lucas Davi, Ahmad-Reza Sadeghi, Stephen Crane, Mohaned Qunaibit, Michael Franz, and Mauro Conti. 2015. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[21] LLVM Developer Group. 2018. LLVM CFI. https://clang.llvm.org/docs/ControlFlowIntegrity.html. (2018).

[22] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P. Chung, Taesoo Kim, and Wenke Lee. 2015. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *ACM Conference on Computer and Communications Security (CCS)*.

[23] João Moreira, Sandro Rigo, Michalis Polychronakis, and Vasileios Kemerlis. 2017. DROP THE ROP: Fine-grained Control-Flow Integrity for the Linux Kernel. (2017).

[24] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety For C. In *International Symposium on Memory Management (ISMM)*.

[25] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[26] Ben Niu and Gang Tan. 2013. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 199–210.

[27] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 914–926.

[28] Aleph One. 1996. Smashing The Stack For Fun And Profit. http://phrack.org/issues/49/14.html. (1996).

[29] PaX Team. 2003. Non-Executable Pages Design. https://pax.grsecurity.net/docs/pax.txt. (2003).

[30] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2015. Fine-grained control-flow integrity through binary hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 144–164.

[31] The Chromium Projects. 2015. Control Flow Integrity in Chromium. https://www.chromium.org/developers/testing/control-flow-integrity. (2015).

[32] Ganesan Ramalingam. 1994. The Undecidability of Aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1467–1471.

[33] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address-Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity. In *Network and Distributed System Security Symposium (NDSS)*.

[34] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*.

[35] SecurityFocus. 1988. BSD Fingerd Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/2/info. (1988).

[36] Jeff Seibert, Hamed Okhravi, and Eric Söderström. 2014. Information Leaks Without Memory Disclosures: Remote Side Channel Attacks on Diversified Code. In *ACM Conference on Computer and Communications Security (CCS)*.

[37] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *ACM Conference on Computer and Communications Security (CCS)*.

[38] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy (S&P)*.

[39] Kevin Z. Snow, Roman Rogowski, Fabian Monrose, Jan Werner, Hyungjoon Koo, and Michalis Polychronakis. 2016. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *IEEE Symposium on Security and Privacy (S&P)*.

[40] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. 2009. Breaking The Memory Secrecy Assumption. In *European Workshop on System Security (EUROSEC)*.

[41] Subgraph Team. 2014. Subgraph OS. https://subgraph.com/. (2014).

[42] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *International Conference on Compiler Construction (CC)*.

[43] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy (S&P)*.

[44] Ben Niu Gang Tan. 2014. Modular Control-Flow Integrity. In *Programming Language Design and Implementation (PLDI)*.

[45] Jack Tang. 2015. Exploring Control Flow Guard in Windows 10. http://blog.trendmicro.com/trendlabs-security-intelligence/exploring-control-flow-guard-in-windows-10. (2015).

[46] Pax Team. 2015. RAP: RIP ROP. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf. (2015).

[47] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, ÃŽlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium*.

[48] Ubuntu. 2017. Ubuntu Popularity Contest. http://popcon.ubuntu.com/by_inst. (2017).

[49] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawlowski, Xi ChenâĂă, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE Symposium on Security and Privacy (S&P)*.

[50] Zhi Wang and Xuxian Jiang. 2010. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 380–395.

[51] Fedora Wiki. 2018. Security Features Matrix. (2018). https://fedoraproject.org/wiki/Security_Features_Matrix

[52] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-Based Selective Flow-Sensitive Pointer Analysis. In *International Static Analysis Symposium*.

[53] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (S&P)*.

[54] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *USENIX Security Symposium*.