# Type Based Control Flow Integrity Typed Based Return Oriented Attacks and Mechanisms for Prevention

Michael Gonzalez, SeungHwan Chung, Yonga Lhamo
CSCI 400-01
Professor Dietrich
December 10, 2018

## ABSTRACT

Memory corruption attacks are a widespread threat, and it ranges from code injections to more complicated code-reuse attacks. Control Flow Integrity (CFI) is therefore a significant countermeasure for such attacks and minimizing the vulnerabilities of these attacks. There are various types of CFI that will be be elaborated upon, but the focus of this paper is to analyze the security and the practical usage of RAP (Reuse Attack Protector) and how it performs when attacks by Typed Based Return-Oriented Programming attacks (TROP). RAP is a free open source implementation of a runtime type checking (RTC) CFI, that protects both the forward and backward edges with type checking.

## INTRODUCTION

There are numerous ways to guard against memory corruption, therefore in order to choose a method we weigh in the aspects of security and practicality of the methods. In order to achieve complete spatial and temporal pointer safety, methods often incur large overhead time thereby making it impractical to implement them. An example of such a defense method is code-diversification. This defense mechanism randomizes code at run time in order to make it difficult for an attacker but is susceptible to direct attacks and side channel information leakage attacks. A defense that protects against such attacks is the Control Flow Integrity (CFI). CFI techniques prevents memory corruption by checking the control flow at run time. CFI is distinctly based on the type of graph it's protocol is based on and this is known as the Control Flow Graph (CFG). There are three methods of generating the CFG and that determines the type of protocol that CFI implements and they are the point-to analysis, here the CFG is generated statically, the second CFG is generated dynamically at run-time and the last that we will delve into further, is the Run-time type checking (RTC).

In this context, control flow hijacking refers to the manipulation of function calls and memory spaces for malicious intent and that can lead to execution of a defined payload. Visualizations of these function calls can be done through program disassembly engines, these are often referred to as Control Flow Graphs (CFGs). CFGs provide blocks of memory space with directed-edges that correspond to non-sequential instruction sets. [1] Control flow Graphs will be used extensively within this project in order to visualize assembly code instructions.

In comparing a CFI that is point-to analysis versus the type checking (that is the focus of this report), The RTC or the type checking CFI is more durable with less crashes while dealing with large code bases, all the while providing the same level of protection for the code as point-to analysis.

The RTC based CFI has is own vulnerabilities and we will therefore attempt to first, analyze how it is successful in its defenses and then scrutinize it's vulnerabilities by testing it against a code reuse attack (TROP). After analyzing the security aspect of RTC we will discuss the practical challenges of implementing the technique and its applications and condition requirements in a real world environment.

---

[1] SPARKS, Sherri & EMBLETON, Shawn & CUNNINGHAM, Ryan & ZOU, Cliff *Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting.* University of Central Florida. 2007.

### 1.1 Understanding Control Flow Integrity

Control Flow Integrity are the schemes in which prevent malicious and undefined behavior within a program. Often the occurrence of control flow manipulation (hijacking) is when an attacker uses this redirection of control flow in order to gain access to a system or to execute arbitrary code. It's implementation is often associated found in packages which deal with code sanitization. Some common violations of control flow can occur in overflows of a program, the most common being buffer overflows, integer overflows, and format string types.

### 1.4 Runtime Type Checking CFI

CFI provides protection for forward edge and backward edges. There are RTC techniques that sometimes provide either one such as the IFCC and LLVM-CFI (provide forward edges)and other that provide protection in both the edges such as the KCFI and RAP. To further elaborate, the forward-edge protection, The RTC checks the type of signature of a function pointer before every control transfer that is indirect. While for the backward edge protection, it stores the signature of the calee before the call site and this signature is verified before execution.

### 1.5 Reuse Attack Protector (RAP)

RAP is a type of Run Type Checking CFI that protects the forward and the backward edges. RAP in forward edge checking calculates the hash before the function's memory address and verifies, and spits an error if it does not match, otherwise the call is taken. And in terms of backward edges, the hash is compared at the call site and then the execution is determined or if it does not match, an error is given.

## 1. Findings

The high availability of return oriented programming flaws within operating systems, in conjunction with improper handling of user submitted data inputs can result into large scale memory corruption attacks, stack based buffer overflows, and even arbitrary code execution.

### 1.1 C Run Type Checking

Within our confined testing of Ubuntu version 16.04 with GCC 5.4.0 we found several underlying issues with control flow hijacking occuring. As a unix system we found that, improperly programmed C programs executed within the user level were able to functionally execute kernel space and bypass any address space randomization and control flow integrity checks present on the machine.

### 1.2 LLVM and Sandboxing

The main issue stems from a now patched flaw within the Ubuntu subsystem, effectively the Lower Level Virtual Machine mechanism for sandboxing within Ubuntu fails to check function parameters types during run time. This enables memory corruption attacks on the stack, and creates a large attack surface due to the amount of C programs needed to run the Linux operating system. The idea of the LLVM is that within the compiler, when memory is accessed, functions are entered and exited, and other forms of memory allocation will log these forms of memory accessing and act as a medium between the kernel and the user [2]. Within the C language this instrumentation is crucial in order to distinguish the kernel space, and prevent access to it by a malicious function such as the same functions found in TROP attacks.

An extension to the LLVM framework is another mechanism for protecting against various forms of memory corruption attacks, within the Unix system called Clang is used to act as a built-in sandbox for apps/languages running in the environment. Many of Clangs features include control flow integrity, and methods to protect against memory attacks. Natively it comes with CFI schemes pre installed but they do need to be enabled. “Clang includes an implementation of a number of control flow integrity schemes, that are designed to terminate the program on the detection certain forms of behavior that can potentially allow attackers to modify or corrupt the program’s control flow. Within Clang CFI is optimized for performance [3].
To enable the Clang-CFI schemes a flag will need to be specified, -fsanitize=cfi. In enabling CFI, clang will now be able to use a variety of schemes and the user is able to specify arguments for interpreting the control flow within Clang. Many developers opt out of CFI protection mechanisms in fear of creating a larger overhead, poor secure code writing, or in some cases complete lack of knowledge of the dangers of Control Flow Hijacking through memory corruption attacks on specific functions.

### 1.3 GCC 5.4 Issues

GCC 5.4 [4] presents another issue with CFI, within this version, at compile time, function pointers are susceptible to memory corruption attacks such as Jump to Libc attacks, Typed Based Return Oriented attacks (TROP), Code Reuse attacks, and the list goes on. We found that this issue affects both 32 bit and 64 bit variants of Ubuntu 16.04 due to their native inclusion of GCC 5.4.

## Other Attacks Vectors Exploiting CFI Protections

## 2. Protection Mechanisms

### 2.1 Control Flow Graph Demonstration

---

[2] Miucin, Svetozar & Brady, Conor & Fedorova, Alexandra. (2016). DINAMITE: A modern approach to memory performance profiling.
[3] https://clang.llvm.org/docs/ControlFlowIntegrity.html
[4] https://gcc.gnu.org/gcc-5/

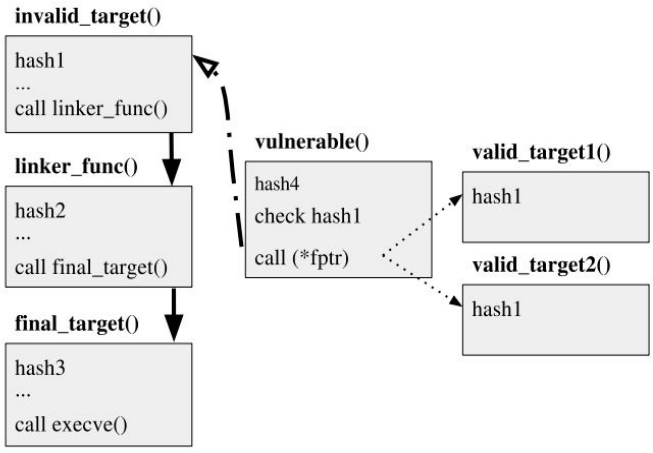


Figure 3. A Control Flow graph demonstrating the flaw in the code. [5]

This Control Flow Graph demonstrates the vulnerable C code functions within the code before patching with the RAP plugin. Essentially what occurs is a vulnerable function will have an extra pointer value passed to it calling a function of the same type with hash4 inside. This will effectively bypass any restrictions on calling functions because within this C code this Call(*fptr) executes due to the same data types on both functions.

This is an example of a typed checking Control Flow Graph. Within the compiler at runtime, the types of functions are checked to see if a function calls to another function with a different data type, if this occurs then the program will abort. Workarounds for this include pointer references to a function of the same type, which leads to the chain of executions down the graph from the vulnerable() function down to the final_target() function.

## 2.2 Code Analysis

During the process of the experiment, we were able to receive a demonstration vulnerable C code, RAP plug-in, and makefile from one of the member of the author, Reza Farkhani, from the *On the Effectiveness of Type-based Control Flow Integrity* paper. Before we look in to the vulnerable.c code for a moment, we have to know how buffer has been declared and it's relationship with the input. Once we understand the concept of the buffer overflow, we can relate that concept to the code:

```
void vulnfunc(char * input){
fptr fIndirectCall;
char buf[20];

if (strcmp(input, "1") == 0)
        fIndirectCall = &foo1;
else
```

[5] JAFARI, Saman, et al. On the Effectiveness of Type-based Control Flow Integrity. ACSAC '18, December 3–7, 2018, San Juan, PR, USA © Association for Computing Machinery.

```
        fIndirectCall = &foo2;
printf(input);
strcpy(buf, input);
fIndirectCall();
}
```

Figure 2. Vulnerable_code.c provided by the authors.

In this code, we have to carefully look at the fptr fIndirectCall, the buffer size, if statement, and strcpy(buf,input) with fIndirectCall(). Before getting to details of the code, we have to check the output of the code first to know the process of the code. When we compile just the vulnerable_code.c and executed with input random string called “hi”, the output shows as hiIndirect call of foo2. Once we know the output of the code, now we dive deeper part of the code.

First, fptr fIndirectCall is written down and the function pointer is defined as a typedef void(*fptr) (void) in the code. The function pointer contains the parameter of the void and make sure the fIndirectCall function is points at the void.

Second, the buffer size of the character is 20. However the actual characters that can be stored into array is 19 bytes because the last character is null character.

Third, the if statement is where determines the memory location of both foo1 and foo2 depends on the input.

Fourth, the strcpy(buf, input) is where the input stores into the buffer array and calls the fIndirectCall when the process is over. This is where it the exploitation happens as known as buffer overflow attack.

The Buffer itself is setted into size of 20, but the reality of the size that can store the input characters are 19 and the last character is known as null character. So when we just input hi, it will regularly gives us fine output: hiIndirect call of foo2. By changing the length of the input into this program will corrupt the stack of the buffer and becomes the segmentation fault (core dumped). The meaning of the segmentation fault is that the function pointer is redirecting the pointer to the memory that the adversor, a user, is not authorized to access. The program may be crashed during the process of the buffer overflow, but it leaves the chance for the adversor to gain the root permission by using this attack.

The part of the experiment was done with the specific defined character with outdated gcc compiler, but simple buffer overflow attack may bypass the security provided by the linux machine and potentially lead to a disastrous situation.

### 2.3 Reuse Attack Protector

Throughout the experiment, the main method that we discovered to protect the type-based Return Oriented Attacks, mainly the buffer overflow attacks, is patching the any c vulnerable code with Reuse Attack Protector plug-ins. Basically explain of the function of the Reuse Attack Protector, it designed to protect against all forms of indirect calls. As we learned the issue from the code analysis part, we learned that the 19 characters were supposed to be save in the buffer size, but the rest of after the 19 characters are indirectly points to the other parts of the memory, which is not authorized to be accessed, and be saved. The Reuse Attack Protector will recognize the unwanted accessing pointers and it will terminate the return instruction so that it doesn't touch any of the important part of the memory.

With the vulnerable.c code, we decided to patch the code using RAP_plugins, provided by the authors, and make the file successfully. Unfortunately we cannot provide the rap plugin code in detail, but it uses -typecheck=call and report to the plugin if it recognize using specific parts called "func, fptr, abs". Once we get the executable file, we decide to input more than 19 characters to see what will happen. Surprisingly, the program has been formally terminated with stating the condition of the program"stack smashing detected". Unlike unpatched vulnerable code, the patch finds out the reused indirecting pointers and terminate the instruction with terminating the whole program itself.

In order to understand more detail how the RAP works, we decided to input the patched program into the guigdb[6], GNU debugger for graphic user interface version, and see the process of the generating output.

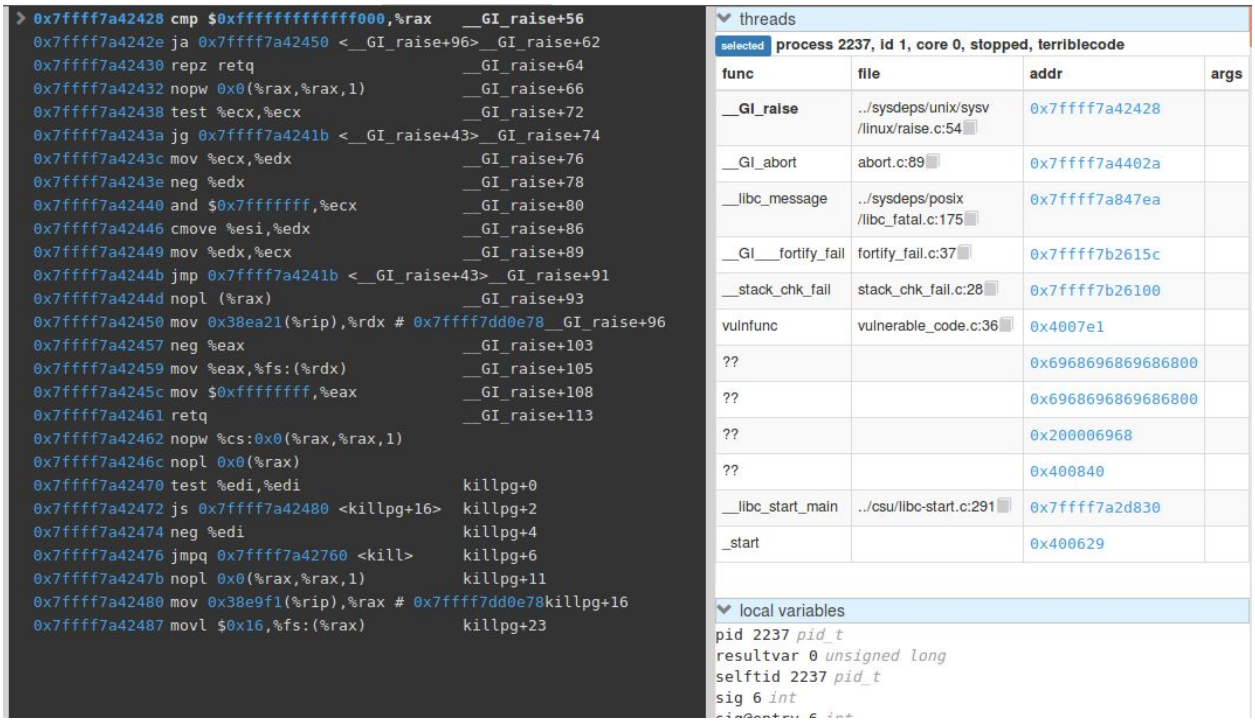


Figure 4. GNU Debugger Disassembly Reuse Attack Protector Patch

[6] https://gdbgui.com/

Without inserting any breakpoints into the program, we immediately realized that the plugins are raising the flag by recognize the abnormal usage of the pointers and force to terminate the program. There are multiple functions that were provided from the threads and one of the function is called stack_chk_fail.

This is where the IDA Pro debugger comes out to see the functions import lists from the RAP Plugins.

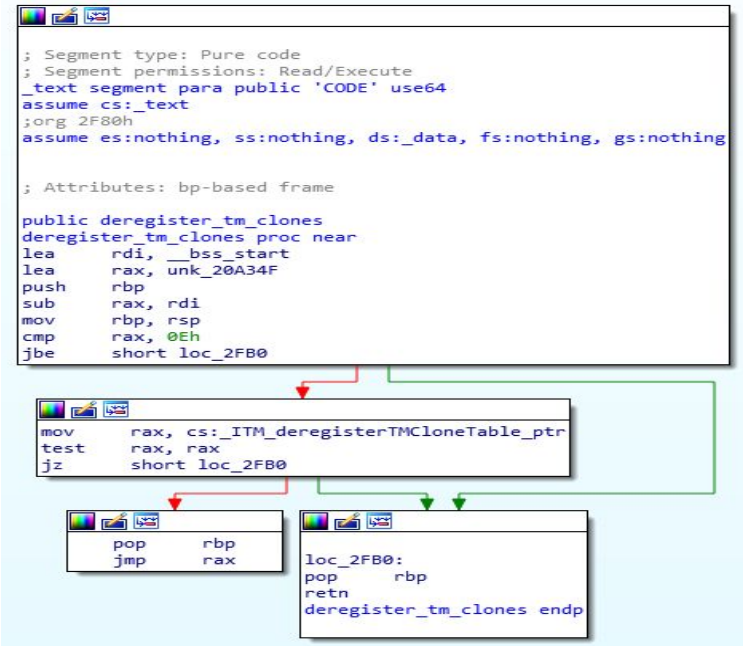


Figure 5. Ida Pro Disassembly Reuse Attack Protector Patch

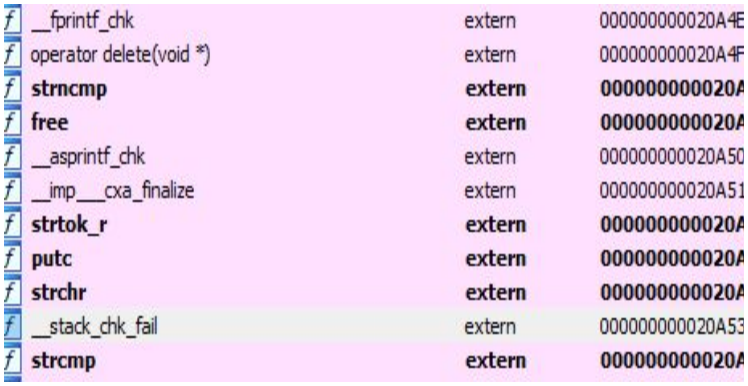


Figure 6. Ida Pro Disassembly Reuse Attack Protector Patch.

Understand the Images from the Figure 2 and Figure 3, we discovered that the RAP plugins are successfully patched the vulnerable code and secure the abnormal usage of the pointers. As we recognize the functions imported by the RAP, the RAP plug-in is strictly searching for stack failure problem, strncmp call and checks the pointers, and fptr status check.

Therefore, Reuse Attack Protector Patch is necessary for the vulnerable code that uses multiple functions with potential weakness to the fix-size of the buffer. By patching the code with RAP, it

will completely terminate the reusing pointers instructions and terminate the entire program as well to prevent the further chances to the adversary.

## 3. Flaws in Protection Mechanisms

There are multiple ways to patch the vulnerable C code which prevents the Typed Based Return Oriented Attacks. Reuse Attack Protector does scan the functions that involves inside the vulnerable C code and limits the reuse of the pointers and corrupting the stack. However, the small size of the Reuse Attack Protector, a demonstration version that we received from the authors, doesn't contain not enough scanning through the code. What if the malfunctioning code wasn't targeting the buffer or simply targeting multiple buffer arrays to make it complicated? Or what if it doesn't use a stack or using different type of variables? (In terms of older version of GCC compiler). This is where the flaws in the protection mechanisms can offer. We saw through the Ida Pro that the RAP is specifically scans the specific functions such as function pointer fptr with fIndirectcall or stack and terminates the reusing the pointer instruction by raising the flags. However, if the protector scans the code and couldn't find the specific target, then it wouldn't raise any of the flags and let the adversary take in charge of another memory location. In addition, some RAP protectors are not compilable with newer version of GCC compiler. In order to achieve some of these mechanisms, we can use Clang which contains additional libraries.

### 3.1 Scope of Control Flow based Hijacking

Return Oriented Programming

ROP is a generalization of return-into-libc attacks where an attacker causes the program to return to arbitrary points in the program's code. It reuses (executable) kernel code [4]

Shadow Stack Attacks

Shadow Stack attacks are a form of attacks which use the shadow stack data structure, in order to subvert control flow, execute arbitrary code, etc. It works via creating copies of return addresses, however this does not offer a complete solution to protecting code. In fact, local variables and forms of overflows can be used to overwrite function pointer addresses, thus rendering the hidden shadow stack useless as an attacker can focus on function local variables vs. global ones.[7]

Integer Overflows

Another form of popular overflow type vulnerabilities are integer overflows. These forms of overflow occur when dealing with the data type of integers. In the case of an overflow, the byte size of the integer due to an operation is larger than the value expected to fit into the traditional size of an integer. These bugs can prove nasty fast, resulting in buffer overflows and other unexpected behaviors. Integer Operations prove to be a worrisome factor for security of an application due the ease of corrupting/manipulating a data field to overflow or underflow an input. [8]

---

[7] https://cs.unc.edu/~fabian/courses/CS600.624/slides/exploits.pdf

[8] http://projects.webappsec.org/w/page/13246946/Integer%20Overflows

Code Reuse
A code reuse attack can be seen in typed based return oriented programming where an attacker simply latches onto a function with a matching type as the first function in the chain, creating a nasty situation which allows for further traveling down function calls, effectively reusing the underlying mechanisms made for legitimate function calls in order to subvert the control flow of an application.

Multiple Vector Attacks
Simply put, multiple vector attacks in Control Flow can occur by a multitude of exploits within both buffer overflows and memory corruption alike. In fact, if an attacker wanted to, they could use all of the aforementioned components to create a multi leveled vector attack on a specific application. An attacker can also attack a specific set of hardware/software which in conjunction would lead to an attack exclusive to that version/chipset. Although the setup would be difficult, this approach may help subvert current mechanisms employed to patch an application including, Clang, LLVM, and sandboxing environments.

Improperly Configured Compilers
In our research efforts using GCC 5.4, we found GCC to be obsolete in the preservation of Control Flow Integrity. This is partly due to a circumvention in function call logic but also due to the age of GCC 5.4, the mechanisms to protect against TROP are not employed and do not exist. As this occurs even in everyday practices, security practitioners must be ensure to update and configure their compilers according to the specifications of their company. However, security must also be considered, a good option for most cases is Clang which provides various frameworks, plugins, and flags to protect against a multitude of Control Flow exploits current and outdated alike.

## 3.2 RTC Performance against TROP attacks

TROP or Typed Return-oriented program is an attack where the attacker redirects the code to arbitrary points. This attack does not require any additional code to be injected but still enables an attacker to gain access and perform malicious activities.A general ROP attack consists of sequence of gadgets that carry out specific functions. During an attack , the attacker finds gadgets in the original code, and edits the conditions to perform other than what was intended and manages to redirect the flow of the program. Attackers make the codes perform indirect-jumps and chains the gadget together with the target function as the destination.

Here we will test the effectiveness of Run Type Checking CFI by attacking it and analyzing how much of the threat is minimized. With RTC CFI, inside the execution of the program, there should be verification of the data entered at run time is what will be returned by the time of termination of the program.

### 3.3 Type Return-oriented Programing Model

We start off the model by first assuming that RAP is implemented in the Operating System, therefore any hijacking attempts made that violates type checking will be detected. The target

system can then be infested with memory corruption vulnerabilities(array out of bound access, format strings, etc) that can prove to be a testbed for an attack.

In order to set up for a Typed ROP attack, we will follow the following phases:

1. Enable ourselves to be able to alter the argument of a function that allows arbitrary execution.
2. Change the function pointer to hijack control.
3. Ensure that the function and the pointer are of the same type since RTC implements type checking and will not allow the control transfer.

Although this attack may succeed in real life it is unlikely to find such a type collision therefore we proceed with the following:

To deploy an attack technique that is based on layered invocations and type collision, this gives the attacker the leeway to call functions even with a different types from a corruptible function pointer so that will be the same type as the sensitive function of interest. This process of attack is broken into three types of gadgets [9]:

- C-Gadget: Collision gadget is the first function that has a type collision with the corruptible function pointer.
- E-Gadget : Execution gadget is the malicious function that the attacker plans to deploy.
- L-Gadget: Linker gadget as the name states are the functions that link from the first function to the execution function.

*In order to execute a TROP attack* :

We will attempt to properly understand the gadgets and how to design them in unity and how to house it for the attack.

In choosing the right gadgets, at first we will attempt to find a suitable C gadget and a proper candidate for that would be any function that has the same type signature as the corruptible function pointer.

A suitable candidate for E-gadget would be functions that spawn a shell (ex.execve system call) because that gives the attacker a wide variation of malicious capabilities that are in payloads. A differing candidate for this gadget would also be to choose a limited function if it suffices the needs for the final purpose of the attack.

Having selected the two above gadgets, we will need an appropriate L-gadget to chain them all together. The call graphs helps to find the L-gadget by routing all possible paths beginning at C-gadget to ending with the E-Gadget.

---

[9] JAFARI, Saman, et al. On the Effectiveness of Type-based Control Flow Integrity. ACSAC ’18, December 3–7, 2018, San Juan, PR, USA © Association for Computing Machinery.

Gadgets are global variables and therefore can be maliciously modified to align with the motive and method of the attack.

The algorithm for the above mentioned sequence of attack is the follow:

```
function FindCandidatePaths(G, N, CG, EG)
    CP ← ∅

    for cg ∈ CG do
        for eg ∈ EG do
            DiscoverPaths(CP, cg, eg, ∅, ∅)
        end for
    end for

    return CP
end function

function DiscoverPaths(CP, cg, eg, P, V)
    P ← P ∪ {cg}, V ← V ∪ {cg}

    if cg == eg then
        CP ← CP ∪ {P}
    else
        for g ∈ N do
            if g ∉ V and G[cg][g] = 1 then
                DiscoverPaths(CP, g, eg, P, V)
            end if
        end for
    end if

    P ← P − {cg}, V ← V − {cg}
end function
```

Algorithm 1. Finding Candidate Path

This algorithm travels through all the possible path between the Collision gadget and the execution gadget. It is broken down into the functions FindCandidatePaths and the function DiscoverPaths, where the first identifies all of the possible c gadget and e gadget.The second function lists all the possible routes between the two.

Not all paths between the C gadget and E gadgets are useful to the attack.In order to pick the L gadget it has to satisfy two requirements: One such that the given constraints elaborated in the gadget is modifiable from outside of it, that is that the constraints should be a global variable so it can be modified and fulfilled. And the second being that there should be no constraints in the L-gadget that would change the flow of the E-gadget. If the two aforementioned conditions are not met then the plausible path between the C-gadget and the E-gadget are abandoned , thus zoning in on the remaining of the plausible paths.
Once the path for the L-gadget is narrowed down it is important to resolve the conditions of the L-gadget. To determine the path, we can see which of the possible L-path conditions can be solvable , and there are two ways to do that, the first is to check if the condition can be solved by

rewriting data in the memory and the second being, if the data , along with the contraints should be accessible globally in the memory outside of the function. After these condition are met, we solve the constraints and determine the path of the attack .

### 3.4 Type Based Collisions

Type collisions are one of our findings while running the vulnerable C code. Essentially, there are multiple functions from the vulnerable C code which contain invalid targets/references of function pointers and these pointers are indirectly called within the function parameters. The reason why this is hard to prevent, is due to the usage of the gadgets mentioned earlier, which can be implemented in a variety of ways. Due to the way that the C language interprets functions, pointers, and data types, if we pass it a type collision with a malicious function pointer we can thus trick the program to run anywhere in memory and have the now overloaded function point to our malicious gadget. Through this process, we ensure complete control over system instructions by injecting our malicious function early in the program and having a link to it from the start of runtime all the way until the end of the programs execution.

Type collisions in C are particularly dangerous due to function type ambiguity, leading to unexpected outcomes (segmentation faults, memory leaks, arbitrary code execution, etc). The reason for these dangerous bugs are due to the ways in which C handles three types at run time for CFI, static typed analysis, dynamic typed analysis, and typed based analysis.

### 3.5 Analysis Types in Detail

Static analysis control flow graphs are created in a non-runtime environment, they are determined static because the data type determined at compile time is considered as the final value and cannot be updated/changed at run time. Static analysis is performed in a non-runtime environment and its purpose is to inspect program code for all possible runtime behaviors and seek out coding flaws, back doors, and potentially malicious code. However, control flow integrity is not demonstrated within static analysis schemes, and instead under static analysis the program will just terminate and leaves no trace unlike typed based CFI.

Dynamic analysis control flow graphs are created during run time. They are considered dynamic because during run time data types can change and be updated, depending on the context of the program.

Lastly, typed based analysis of control flow graphs refer to a loose structure of control flow graphs which refers to the type being referred to the function type which makes a call to it.

### Challenges with RTC-RAP

-There are numerous challenges regarding the implementation of RTC, One such being that although it is helpful against selected few attacks , as explained above it is not completely bulletproof against attacks such as TROP  that is crafted according to its target goal thus being able to bypass the RAP defenses. Although it takes a lot of effort in coordinating the gadgets it is still very much feasible.

-Another such challenge to the implementation of RTC is Type Diversification. RAP in order to protect against type collision in other RTC methods have developed numerous type such that the each function pointer shares a type with its specific target, thereby attempting to reduce attacks such as TROP. But this has over time proven futile, due to the lack of precision in point-to-analysis.
-Separate Compilation is another factor that proves to be a challenge for the real life implementation of RTC. Separate compilation complicates the type diversification by breaking functionality. If each unit of compilation is different and thus compiled separately, there will be false positive and the process is halted when a function calls another function from a different type unit and compilation unit. Therefore in order to avoid this it requires additional effort to be able to function smoothly without any interruptions.
-The fourth factor that affects the usability of RTC in real life scenarios is the mismatched types. Since the default belief of the RTC is that the types of the calling function has to match the type of the called function, it proves to be a challenge when take for instance a void function , which in general should be permitted to call a function of another type , is restricted from doing so. It makes the program inflexible and a hassle. This issue can be combatted only by modifying void to the specific type of the target file , this requires us to change the source code and is a waste of time and energy. Another method to combat this issue would be to allow void the access to point to any type function but this in exchange makes the program more malleable and also end up proving to be an opportunity for easy attacks.
-Last factor that makes the use of RTC impractical is the support for assembly code. Just like the above issues if the permissions on the assembly code is relaxed it fosters in a lot of attacks but if it is restricted then it is inflexible and crashed a lot of application .

## 4. Conclusion

In this work, we went over what is Control Flow Integrity, and how it can protect us against memory corruption attacks. We discussed over how the Control flow Graph determines the target function of each CFI. Went into further discussion over the functionality of Run Type Checking CFI and dove into the demonstration of Reuse Attack Protector which is a type of the RTC CFI. We went on to understand that though RTC is formidable against certain attacks it is still vulnerable to other attacks such as the Typed Return Oriented Programing attacks .
In regards to TROP, We understood the functionality of the attack, the necessary gadgets that are required for its execution, namely the Collision gadget, Link gadget and the execution gadget. Having understood the process and the requirements of each gadget, we went on to acknowledge the abundant availability of the resources required to execute an attack such a TROP.
After the discussion of the TROP attack , we geared back to RTC, and understood its weaknesses and how in terms of real life application it is still a challenge to implement due to reasons such as advanced attacks,separate compilation, mismatched types,type diversification etc.
Through our findings it is easy to come to a conclusion that though RTC-RAP is successful defense against certain memory corruption attacks , it is still not adequate enough to protect completely against memory corruption attacks thus leaving us still vulnerable to them.

As security practitioners, vigilance must be placed on memory corruption attacks as a whole. In fact, out of the newest security patches for a variety of devices and chipsets, overflow and

memory corruption attacks are extremely prevalent. Google recently patched in Kernel based Control Flow Integrity checks for their Android smartphones, a feature which did not exist since the release of the Android Operating System, which uses a subsystem of Linux. Android 9 (Pie) released in August, 2018 was the first implementation of Kernel based CFI (KCFI). [10] The CFI mechanisms located in the Android system are backed by LLVM structure with the Clang toolchain present and enabled by default. As time goes on, memory attacks need to be at the top of the list in priority, as a multitude of devices can fall victim to attacks of ranging skill levels to perform.

[10] https://source.android.com/devices/tech/debug/cfi

References

JAFARI, Saman, et al. *On the Effectiveness of Type-based Control Flow Integrity.* ACSAC ’18, December 3–7, 2018, San Juan, PR, USA © Association for Computing Machinery.

MIUCIN, Svetozar & BRADY, Conor & FEDORVA, Alexandra. DINAMITE: *A Modern Approach to Memory Performance Profiling.* 2016.

MUNTEAN, Paul, et al. CFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, Cham, 2018. p. 423-444.

SPARKS, Sherri & EMBLETON, Shawn & CUNNINGHAM, Ryan & ZOU, Cliff *Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting.* University of Central Florida. 2007.