# Characterization of Forward-edge Control-flow Integrity Targets in LLVM-compiled Linux

Jonathan Vexler
*Brown University*

## Abstract

In this paper, we analyze the Linux Kernel compiled with LLVM and Control Flow Integrity (CFI) instrumentation enabled. Our analysis involves identifying functions called through indirect calls and characterizing functions sharing the same type signature. Insights into the composition of function type groups are vital for understanding the CFI protections that LLVM can provide.

## 1 Introduction

Exploits involving stack corruption have been mitigated by advancements in the field, and attackers have shifted their focus to target exploits that corrupt the heap and overwrite function pointers [1]. Many attempts have been made to counteract these exploitations by constraining the allowable values of function pointers.

One control flow integrity technique that constrains function pointers by type has been integrated into the LLVM toolchain. This technique analyzes the types of all functions during compilation, and puts all the functions of the same type into their own groups. Then inline instructions are added to constrain each indirect function call to one function types. For function calls across DSO's, a hash of the mangled function type is compared to the expected function type pre-computed in the same manner.

While the described protections limit many of the potential targets for a corrupted function pointer, attacks can be made against type-based CFI protection if the victim and target function both have the same type signature. To understand the scope of this type of exploit, it is important to understand the size and composition of these function type groups.

In this paper, we characterize the target groups in the Linux Kernel. We compile the Linux 5.4-rc8 kernel using LLVM 10. To identify the target groups, we locate protected indirect calls by matching instruction patterns. We then extract the expected function groups.

## 2 Background

### 2.1 Indirect Function Calls

While many modern programming languages have built-in features for object oriented programming, C was not designed to be object oriented. However, by using function pointers, C programmers can implement object oriented design patterns and reduce code reuse. A function pointer in a struct can be reassigned just like a child class can override a method of a parent class.

Function pointers are a powerful tool, but are extremely vulnerable to exploitation. When a function is called directly, its address will be hard-coded into the instruction. The instructions are in read-only memory. When an indirect function (function pointer) is called, the address of the function will be read from a register. If an attacker can manipulate the address in the register, they will be able to manipulate the flow of the program. The risk of these attacks are high in programming languages such as C where there is no memory safety.

### 2.2 Control Flow Integrity

Control Flow Integrity is the class of techniques that attempt to ensure that indirect function calls go to their intended targets. Control Flow Integrity techniques involve generating a control flow graph of the program either statically or dynamically or by analyzing function types [4]. Additional techniques involve identifying suspicious behavior, such as number of instructions between indirect calls or number of consecutive indirect calls [3].

Techniques that consult the control flow graph often carry a significant performance overhead [3]. The constant monitoring of the program needs to include checks whenever instruction flow is nonlinear. However, in a memory unsafe language, Control Flow Integrity techniques must preform checks during run-time. This must be the case given that memory corruption will be caused due to user input. Be-

cause these checks must occur at run-time, it is imperative that the protections have minimal overhead at run-time.

If sufficient setup occurs during compilation, Type-based Control Flow Integrity has minimal overhead during execution. The compiler will identify functions with the same type and create groups containing all the functions of each type. Indirect calls are only allowed if they are calling a function with the type definition that the compiler believes is intended for that function. An attacker will have a very small number of possible targets when corrupting a function pointer. The challenges with this technique are identifying the indented type for a function pointer and segregating all functions with the same type. This type of Control Flow Integrity protection is called forward edge protection because it protects the jumps or calls into functions (as opposed to backward edge, which protects function returns).

### 2.3 CFI in LLVM

LLVM has built-in CFI protection for forward edges that is implemented using type checking. During compilation, functions are grouped with others sharing the same type definition and a jump-table is created for each group. This table contains jump instructions, with `INT3` instructions used as padding. Without CFI protection, when a function pointer is assigned, the value would be the address of the function. With the LLVM CFI protection enabled, the value assigned to function pointers is now an address in a jump table. The address in the jump table corresponds to an instruction that jumps to the start of the function that the user intended to create a pointer for. This only adds one additional instruction for each indirect call, but offers a significant advantage: jump table pointers can be positioned independently from function pointers.

LLVM organizes functions in jump tables so that few instructions will be necessary to compute a CFI type check. A jump table is created for each function type. Now, to check if an indirect call is to a function with a specific type, a simple check can be computed to ensure that the indirect function address is in the correct jump table and byte aligned. This computation only requires the address of the jump table and the size of the table. With a couple of instructions inserted before each indirect function call, this technique will enforce type based CFI with minimal overhead during execution [6].

Recent advancements in development have allowed for the compilation of the Linux kernel with LLVM instead of GCC. We are therefore able to compile Linux with LLVM and gain the type-based CFI protection. However, due to the use of DSO's (dynamic shared objects) in the Linux Kernel, it is necessary to discuss how LLVM handles indirect function calls across DSO boundaries. The type-grouped jump-tables will not contain functions from a DSO. Therefore, if the address of an indirect call is not in the correct jump table, a call to `__cfi_slowpath()` will be made [5]. The slowpath function will use a specialized CFI check for the function. To find this specialized function, a mapping of memory pages to CFI check functions is created as indirect function pointers are used for the first time. Our analysis only characterizes functions in jump tables, and does not inspect the indirect calls across DSO boundaries.

## 3 Implementation

To compile Linux 5.4-rc8 with LLVM and CFI instrumentation enabled, we used a branch from kees [2]. Our goal was to identify and categorize the subsets of allowable functions from indirect calls. To do this, we needed to be able to consistently identify protected calls. A code pattern was found:

```
8104dc77: mov $0x81a106b0, %rcx
8104dc7e: sub %rcx,        %rax
8104dc81: ror $0x3,        %rax
8104dc85: cmp $0xe,        %rax
```

In this pattern, a register was set as the base address of a jump table. This address was subtracted from the indirect call address. Then this difference was rotated 3 bits because the entries in the jump table are byte aligned. A rotate was used instead of a shift so that non-byte aligned addresses are not allowed. Finally the rotated difference was compared with an immediate value. This value is one less than the number of entries in the jump table. After this code pattern there will be a jump and then an indirect function call. If target register, in this case `%rax`, is a valid address in the target jump-table, then the indirect function call will be executed. Else, the address in `%rax` is not acceptable, and the jump will lead to code that handles a corrupted jump.

The instruction ror is not widely used outside of cryptography and networking, so this was the initial identifier of a suspected protected call. We found 7072 instances of rotate instructions. After identifying a rotate, we checked for proceeding subtraction instruction and a subsequent comparison instruction. Of the 7072 instances of the rotate instruction 4660 of them matched the CFI protection pattern. Finally we extracted the base address of the jump table. Unlike the example pattern, the move instruction did not always proceed the subtraction, and the register of the base address was not always `%rcx`. To identify the base address, we extracted the register from the subtraction and searched the proceeding instructions in the function for a move of an immediate into that register. These jump-tables can be called more than once, and from these 4660 indirect calls, there were 772 unique function types. Using those jump tables, we identified functions and extracted their type definitions.

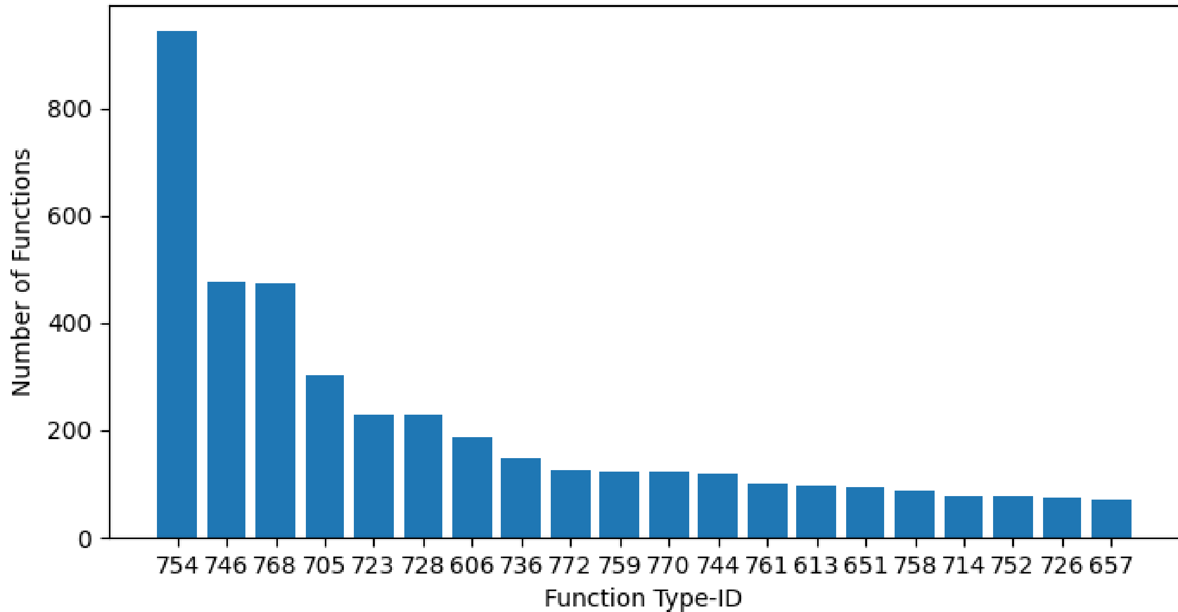


Figure 1: Top-20 CFI-protected function types in the Linux Kernel

## 4 Results

| Function Type-ID | Type Definition |
|---|---|
| 754 | (int)(int, int, size_t, unsigned int) |
| 746 | (int)(void) |
| 768 | (ssize_t)(struct device *, struct device_attribute *, char *) |
| 705 | (void)(void) |
| 723 | (int)(char *) |
| 728 | (int)(struct inode *, struct file *) |
| 606 | (int)(struct seq_file *, void *) |
| 736 | (void)(struct work_struct *) |
| 772 | (ssize_t)(struct device *, struct device_attribute *, char *, size_t) |
| 759 | (int)(struct device *) |

`754` corresponds to system calls, `746` corresponds to initialization functions, `768` corresponds uncore/system agent functions, `705` corresponds to functions without inputs or return that are primarily used to lock and unlock data, `723` corresponds to additional initialization functions, `728` corresponds to functions in the file-system for opening and releasing, `606` corresponds to file system information functions, `736` corresponds to deferrable functions, `772` and `759` correspond to functions that manage connected devices

## 5 Conclusion

The characterization of Control Flow Integrity targets show that there exist a group of function types that are most vulnerable after LLVM's CFI protection has been enabled when compiling the Linux kernel.

We at least 183 function types that have at least 10 control flow targets. Additionally there are 5 types that have at least 230 targets. If there is a vulnerability that involves one of these type groups, the CFI protection will not be very effective because there is such a large possibility of target functions.

## 6 Future Work

The next step in our work of the analysis of CFI targets in the Linux Kernel would be to expand this analysis to the cross-DSO function calls that are handled with `__cfi_slowpath()`. This will give a more accurate and complete picture of the most vulnerable types.

This research can be additionally be used for development of the Linux Kernel. Our data shows the most vulnerable types if a function pointer of that type can be corrupted. Therefore, it is more vital to identify targets for corruption involving those types.

Finally, this research can be used for the development of CFI protections, namely type-based implementations. The Linux Kernel is one of the most used software, and thus one

of the most important to protect from control flow hijacking. If some of the type-groups of functions are deemed to be "too large" our research could show that a type-based CFI protection scheme is not acceptable to provide protection.

## Acknowledgments

I would like to thank my advisor, Professor Vasileios Kemerlis, for his guidance and support throughout the year.

## References

[1] Peter Collingbourne Stephen Checkoway Úlfar Erlingsson Luis Lozano Geoff Pike Caroline Tice, Tom Roeder. Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium*, 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf.

[2] Kees Cook. experimenting with clang cfi on upstream linux, 2019. https://outflux.net/blog/archives/2019/11/20/experimenting-with-clang-cfi-on-upstream-linux/.

[3] Daniel Lehmann Fabian Monrose Lucas Davi, Ahmad-Reza Sadeghi. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-davi.pdf.

[4] Sajjad Arshad William Robertson Engin Kirda Reza Mirzazade farkhani, Saman Jafari and Hamed Okhravi. On the effectiveness of type-based control flow integrity. In *2018 Annual Computer Security Applications Conference*, 2018. https://sajjadium.github.io/files/acsac2018typecfi_paper.pdf.

[5] Chris Rohlf. Cross dso cfi - llvm and android, 2019. https://struct.github.io/cross_dso_cfi.html.

[6] The Clang Team. Control flow integrity design documentation, 2020. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html.