Practical Protection Design of Forward-Edge Control-Flow Integrity for Linux Kernel

Jinmeng Zhou
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
jinmengzhou@zju.edu.cn

Ziyue Pan
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
ziyuepan@zju.edu.cn

Xun Xie
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
xiexun@zju.edu.cn

Wenbo Shen*
College of Computer Science and Technology
Zhejiang University
Hangzhou, China
shenwenbo@zju.edu.cn

***Abstract*—The operating system kernel is the security foundation for the entire system. Yet control flow hijacking is a prevalent attack method that continually threatens its security. Control-Flow Integrity (CFI) defends against these attacks by enforcing execution compliance with a pre-computed Control-Flow Graph (CFG). The kernel is very sensitive to performance overhead, challenging the CFI scheme design. The existing Clang-CFI is software-based, making it the general solution widely deployed as the de facto CFI. However, it is coarse-grained, and its CFI scheme design cannot be easily applied to fine-grained CFGs.**

To provide CFI for fine-grained CFG, we propose a flexible protection design, FLEX-CFI. This software-based approach enhances generality by eliminating hardware dependencies. Our CFI design is practical and applicable to various fine-grained CFGs with negligible overhead. We implemented a prototype of FLEX-CFI based on Clang/LLVM and evaluated it on the Android ARM64 Linux kernel using a real-world hardware device. The results show that FLEX-CFI effectively secures 92% of all indirect call-sites in the `allyesconfig` kernel configuration while reducing target functions by 93.5% and imposing close-to-zero performance overhead.

***Index Terms*—Control Flow Integrity, Practical Protection Design, Linux kernel**

I. Introduction

Control-flow integrity (CFI) has been an active research area since its inception, from early proposals to recent state-of-the-art works [1], [2], [3], [4], [5], [6], [7]. CFI prevents control-flow hijacking by ensuring that program execution follows a pre-computed control-flow graph (CFG). CFI includes an important part: forward-edge CFI, which enforces that function pointers of indirect calls only jump to valid targets.

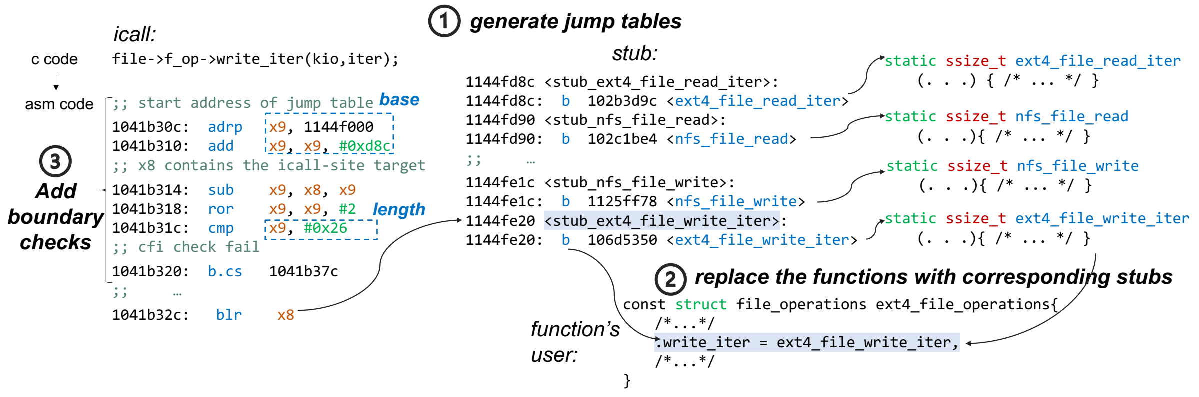
Current forward-edge CFI falls into two categories. First, dynamic CFI analyzes call graphs by tracking indirect calls and their valid targets at runtime [8], [9], [10], [11], [12], typically leveraging hardware features like Intel Process Tracer (PT) and Last Branch Record (LBR). However, preserving extensive runtime context information impacts practicality and performance, especially for large-scale programs like the Linux kernel that are performance-sensitive. Second, static CFI, which builds call graphs through static analysis and instruments indirect calls to prevent control flow hijacking, remains promising [6] with several recent advances [13], [14], [15], [16], [6], [7]. Commercial compilers [17] and major software vendors like Microsoft and Google [18], [17], [19] have adopted this approach.

Static CFI solutions use either software or hardware implementations [20]. Hardware-based approaches achieve precise CFI with acceptable overhead [13], [14], [16], [21], [22], [6], [7] using features like ARM PA, Intel CET, and Intel MPK. However, these features limit deployability across platforms. For example, PA on ARM cannot run on x86, while some features, e.g., Intel MPX used in OS-CFI [14], have been deprecated. Software-based static CFI is platform-independent, but the current widely-supported mechanism is coarse-grained: Clang-CFI [17] matches function signatures of defined functions with indirect call targets. Due to its low overhead and simple implementation, Clang-CFI is integrated into Linux kernels. However, its CFI design is limited to coarse-grained call graphs and can not directly apply to finer-grained ones. Existing work [23] has already proved that coarse-grained CFI is ineffective in stopping some real-world exploits.

Recent work proposes fine-grained CFGs, but practical CFI enforcement with acceptable overhead remains elusive. Approaches like KIRIN [24], MLTA [25], and TFA [26] generate fine-grained CFGs but lack efficient enforcement mechanisms. FINE-CFI [15] incorporates struct-based analysis but omits propagation between jump-target groups, resulting in low precision (i.e., only protects 4,074 indirect calls/jumps with 13.14 average targets). Additionally, it leverages indexed hooks to protect indirect calls/jumps in the Linux kernel, which creates a jump table for each indirect call and increases the number of compiled instructions.

To bridge the gap between granularity and performance

*Wenbo Shen is the corresponding author



```
file->f_op->write_iter(kio,iter);

;; start address of jump table
1041b30c:  adrp   x9, 1144f000
1041b310:  add    x9, x9, #0xd8c
;; x8 contains the icall-site target
1041b314:  sub    x9, x8, x9
1041b318:  ror    x9, x9, #2
1041b31c:  cmp    x9, #0x26
;; cfi check fail
1041b320:  b.cs   1041b37c
;;     …
1041b32c:   blr    x8
```

```
1144fd8c <stub_ext4_file_read_iter>:
1144fd8c:  b  102b3d9c <ext4_file_read_iter>
1144fd90 <stub_nfs_file_read>:
1144fd90:  b  102c1be4 <nfs_file_read>
;;     …
1144fe1c <stub_nfs_file_write>:
1144fe1c:  b  1125ff78 <nfs_file_write>
1144fe20 <stub_ext4_file_write_iter>:
1144fe20:  b  106d5350 <ext4_file_write_iter>
```

```
static ssize_t ext4_file_read_iter
    (. . .) { /* ... */ }
static ssize_t nfs_file_read
    (. . .){ /* ... */ }
static ssize_t nfs_file_write
    (. . .){ /* ... */ }
static ssize_t ext4_file_write_iter
    (. . .){ /* ... */ }
```

```
const struct file_operations ext4_file_operations{
   /*...*/
   .write_iter = ext4_file_write_iter,
   /*...*/
}
```

Fig. 1: Clang-CFI protection and its large jump table.

overhead, we propose a practical and flexible CFI design, FLEX-CFI. FLEX-CFI enforces a smaller target of indirect calls with a close-to-zero performance overhead. This design is flexible and can be applied to any fine-grained CFG. According to the CFG, target functions are distinguished into different small groups (flow-in analysis), and an indirect call is forced to jump to a group (flow-out analysis). To achieve higher precision, FLEX-CFI also integrates group-to-group analysis. As a result, FLEX-CFI can create a jump table for each group instead of each indirect call, i.e., different indirect calls can share the same jump table to reduce the number of compiled instructions. As an implementation example, it leverages the fine-grained CFG by matching struct-fields [25], [24]. The evaluation results show that FLEX-CFI protects 92% of all indirect call-sites while reducing 93.5% of possible jump targets and introducing close-to-zero performance overhead.

In summary, this paper has the following contributions:

- **New techniques:** We propose the flow-in, flow-out, and group-to-group analysis to analyze the complex data flow of target function groups.
- **Prototype implementation:** We have implemented a prototype of FLEX-CFI based on a Clang/LLVM LTO pass, which contains 3,000 lines of C++ code.
- **Practical evaluation:** We have evaluated FLEX-CFI on Android ARM64 Linux kernel with a real hardware board. The evaluation results show that FLEX-CFI can protect 92% of all indirect call-sites for allyesconfig kernel, while reducing 93.5% of possible jump targets, and introducing close-to-zero performance overhead.

II. Background: Clang-CFI

Clang-CFI confines targets of every **indirect call (icall for short)** by matching the type of an icall with the dereferenced value. A function type (often known as function signature) is defined by the types of its parameters and return value. When resolving an icall, Clang-CFI retrieves the function type of its target and only allows the pointer to point to the functions with the same type. This approach has been practically applied to harden C/C++ programs and Linux kernels. In the following of this section, we use an example to illustrate how it works.

Figure 1 shows an icall protected by Clang-CFI in ARM64 kernel. An icall is typically implemented as a `blr` instruction, its c code is `file->f_op->write_iter` in this example. Clang-CFI checks if icall target is in the valid target set before allowing the indirect jump. To check the target function(s) in constant time, Clang-CFI creates a jump table for a set of functions that share the same type. Each jump table has its base address and length. Therefore, Clang-CFI can determine whether the jump target is good or not by checking whether the called address is inbound. To accommodate different function lengths, Clang-CFI creates **a stub for each jump table entry** with fixed 4 bytes. Stub, in turn, jumps to the actual function, so calling a stub is equivalent to calling a function.

The CFI enforcement in Clang-CFI can be divided into 3 steps: (1) Generating a jump table for the same type of functions. For example, in Figure 1, the jump table for the function type `ssize_t (struct kiocb *iocb, struct iov_iter *to)`. (2) Replacing functions with a stub at the user, then the stub propagates in the kernel. In Figure 1, there is a user of `ext4_file_write_iter` where the address of `stub_ext4_file_write_iter` is stored to `write_iter`. (3) Adding boundary check for each icall. Clang-CFI retrieves the function type of the icall target to obtain its corresponding jump table. Then Clang-CFI takes out the stub that icall calls to. If the address of the stub is in the jump table by comparing the base address and length of the jump table, this icall is valid and passes the CFI check. Otherwise, there will be a CFI check failure. As shown in Figure 1, the base address of the jump table is `0xffff80001144fd8c`, and there are 0x26 stubs in the jump table. If the check is successful, it will go to `blr` and jump to the target `ext4_file_write_iter` finally.

III. System Design

We propose FLEX-CFI, which enhances CFI security by narrowing valid target scopes while maintaining performance efficiency. FLEX-CFI enforces an icall to a small-sized group of target functions, requiring data-flow analysis for proper group usage. The more fine-grained the groups, the more complex the data flow between them.

FLEX-CFI separates all address-taken functions into smaller groups, offering fine-grained protection. Using the same exam-

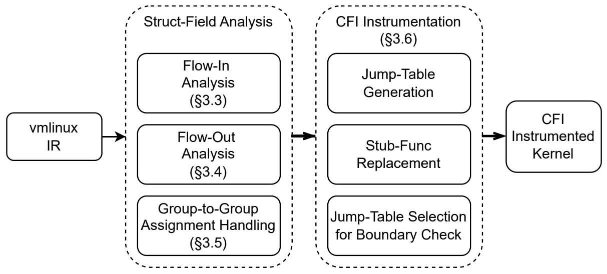


Fig. 2: Overview of instrumenting kernel using FLEX-CFI.

ple shown in Figure 1, FLEX-CFI builds a jump table for each field in struct `file_operations`. The separate jump tables for `read_iter` and `write_iter` are shown in Figure 1, containing read and write interfaces, respectively. To demonstrate our flexible CFI design, we select the struct-field matching method as an implementation case to generate the fine-grained CFG. The following will use struct-field to represent a group. As illustrated in Figure 2, FLEX-CFI takes kernel IR file as input and outputs the CFI-instrumented kernel.

A. *Flow-in Analysis*

In flow-in analysis, FLEX-CFI analyzes all functions that flow into a group, i.e., struct-field. The stubs of these functions compose the jump table for this group, which guides the jump table generation in §III-D1. A function address can enter a struct-field in two ways: (1) directly through initialization or assignment, or (2) indirectly via an intermediate variable V. For untraceable cases, we fall back to the type-based approach, placing them in signature-matched jump tables.

1) Direct Flow-in: First, FLEX-CFI analyzes all users of the address-taken function and finds the struct-fields that are directly assigned by these functions. Initialization or assignment can store a function in a certain struct-field. However, the representations of the two storage methods are different in IR: a struct type variable with initial value(s) becomes a global variable in IR even if it is a local variable in the source code. The assignment of a function pointer to a struct-field in source code converts to a `store` instruction in IR. Based on these above observations, the two cases are treated separately.

Initialization. FLEX-CFI traverses all global variable definitions that are struct variables with at least one function pointer field initialized. FLEX-CFI records the struct-field, the initialized function, and its global variable for each initialization.

Assignment. FLEX-CFI collects users of address-taken function to filter and analyze store instructions. If a function stores to a field of struct-field, FLEX-CFI records the struct-field, the function, and the user (store instruction).

2) Indirect Flow-in: Starting from a struct-field, FLEX-CFI backward analyzes all variables V assigned to it. After that, FLEX-CFI uses an intra-procedural analysis to backtrace data flow of V, i.e., traces the value flow into V, which is divided into two cases: (1) When V is assigned by a function address, this function belongs to the group of this struct-field. As the assignment of the function's user, serving as the

```
 1 struct musb_io {
 2   /* ... */
 3   u32 (*busctl_offset)(u8 epnum, u16 offset);
 4 };
 5
 6 struct musb_platform_ops {
 7   /* ... */
 8   u32 (*busctl_offset)(u8 epnum, u16 offset);
 9 };
10
11 struct musb {
12   /* ... */
13   struct musb_io io;
14   const struct musb_platform_ops *ops;
15   /* ... */
16 };
17
18 static int musb_init_controller(struct device *dev, int
   ↪  nIrq, void __iomem *ctrl) {
19   /* ... */
20   struct musb *musb;
21   /* ... */
22   if (musb->ops->busctl_offset)
23     musb->io.busctl_offset = musb->ops->busctl_offset;
24   /* ... */
25 }
```

Fig. 3: An example of group-to-group assignment.

propagation source, FLEX-CFI records it to help locate the users that should be replaced by the stub in this struct-field's jump table. (2) When V comes from another struct-field, it is an indirect propagation between different struct-fields, which will be discussed in §III-C.

B. *Flow-out Analysis*

In the flow-out analysis, FLEX-CFI analyzes all data flows that flow out from a group, i.e., struct-field. As a result, it separates different icalls based on the groups. So it helps determine the corresponding jump table at the icall's boundary checking. Similarly, flow-out analysis divides icalls into two categories based on which struct-field the function pointers are retrieved: (1) The function pointers directly come from struct-fields. (2) The function pointers are standalone variables V, and V come from struct-fields. Due to space constraints, we will not go into detail. Note that if the target of an icall is not related to any groups within our intra-procedural analysis, we will fall back to the type-based approach for this icall.

C. *Group-to-Group Assignment Handling*

After stub replacement, assignments between groups are effectively assignments of stubs in different jump tables. FLEX-CFI ensures each stub's address remains within its jump table for proper CFI checks. Assignments within the same group (i.e., struct-field) require no special handling; only cross-group assignments need to be managed.

If one struct-field is stored to another struct-field directly or indirectly, FLEX-CFI merges their jump table and treats these two struct-fields as the same one. When stub replacement is done, this assignment assigns a stub in one jump table to another struct-field. Such as the struct-field assignment at line 23 in Figure 3: `musb->io.busctl_offset =musb->ops->busctl_offset`. As the struct-fields hold the stub address, the assignment between struct-field leads to the propagation of stubs. `stub_ops` represents a stub for

TABLE I: Average jump target number comparison.

Config	iCalls	Ave.Clang	Ave.FLEX	Reduce ratio
allyesconfig	165.7k	186.9	12.08	93.5%
defconfig	19.8k	34.8	6.93	80.1%
boardconfig	12k	11.4	5.49	51.8%

TABLE II: Statistics of different categories of indirect calls.

Category		allyesconfig	defconfig	boardconfig
From non-struct		7.1k (4%)	2.5k (10%)	2.5k (15%)
From struct	Leaked	7.2k (4%)	2.3k (9%)	1.7k (11%)
	Has Target	152.7k (85%)	16.5k (67%)	7.3k (45%)
	No Target	13k (7%)	3.3k (14%)	4.7k (29%)
	FLEX-CFI	165.7k (92%)	19.8k (81%)	12k (74%)
All icalls		180k	24.6k	16.2k

`musb_platform_ops->busctl_offset`, `stub_io` represents the other. This assignment leads to the `stub_io` being replaced by `stub_ops`. When `musb->io.busctl_offset` is called, icall target has been replaced by `stub_ops`, causing a CFI failure. Therefore, FLEX-CFI handles this assignment by merging the set of target functions of them struct-fields, and recording equivalence sets to store the merged struct-fields.

D. *CFI Instrumentation*

The CFI instrumentation in FLEX-CFI is roughly the same as in Clang-CFI, involving three steps.

1) Jump Table Generation: FLEX-CFI takes out the target functions of each group, i.e., struct-field. As all functions are already stored in this map in §III-A, FLEX-CFI uses the results to generate jump tables directly. For example in Figure 1, FLEX-CFI builds jump tables for `read_iter` and `write_iter` of struct `file_operations` separately. The length of these two tables is smaller than the Clang-CFI type-based jump table.

2) Stub Replacement: In flow-in analysis (§III-A), function users are recorded when a function flows into a struct field, allowing FLEX-CFI to determine the appropriate stub replacement. If a user is already recorded, the function is replaced with the corresponding stub in the struct-field jump table. For example, in Figure 1, one user of `ext4_file_write_iter` is initialized to the field `write_iter` of struct `file_operations`. The function will be replaced by `stub_ext4_file_write_iter` in the jump table of this struct-field.

3) Jump Table Selection for Boundary Check: At each icall, FLEX-CFI needs to instrument boundary checks in three steps (Figure 1): (1) obtain the target address and jump table start address, (2) compute the offset by subtracting the start address from the target, and (3) divide the offset by the stub size to compare with the jump table size. If the result is within bounds, the jump is allowed; otherwise, FLEX-CFI triggers CFI failure handling.

IV. Evaluation

We evaluate FLEX-CFI in two aspects: (1) **Effectiveness.** A key measure of CFI's effectiveness is its ability to minimize the number of indirect call targets. Our evaluation quantifies the average reduction in jump target numbers achieved by FLEX-CFI in §IV-A. (2) **Performance Overhead.** We assess the performance overhead of FLEX-CFI by conducting tests on real hardware, with findings presented in §IV-B.

Experimental Setup. FLEX-CFI compiles the kernel using three configurations: defconfig, allyesconfig and configuration for DragonBoard 845c [27] (boardconfig for short). We only change the configurations by enabling CFI and disabling ThinLTO. Given that FLEX-CFI does not support virtual dynamic shared objects, we disable the slow path in CFI as well. The allyesconfig is chosen because it enables as many modules as possible, which stress-tests FLEX-CFI. We compile the kernel with the configuration of DragonBoard 845c to test FLEX-CFI on a real board. The default optimization level for the kernel is O2. We instrument the kernel using FLEX-CFI on the machine with Debian 10, x86_64 (Linux kernel 4.19.118-2), Intel Core i7-8700 3.20GHz (6 cores), and 32GB DRAM. DragonBoard 845c consists of a custom 64-bit ARM v8-compliant octa-core CPU, 4GB DRAM, and 64GB storage.

A. *Effectiveness*

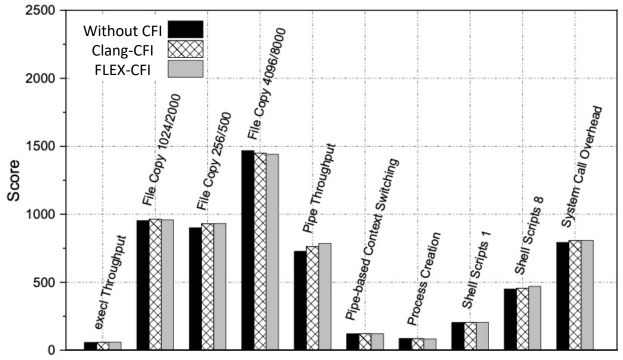
The main goal of FLEX-CFI is to shrink the jump table to reduce potential jump targets based on a precise call graph that has been computed. We assess the security effectiveness of our tool by comparing the indirect calls enforced by security measures with the outcomes derived from the computed call graph [25], [24]. The results show that we can protect the call graph almost as precisely as the one provided.

Average jump target numbers. Table I shows the average jump target number. Column *iCalls for FLEX-CFI* denotes how many icalls FLEX-CFI can protect. Clang-CFI (i.e., the one-layer type analysis in [25]) matches function signatures. Its average target number is 186.9 for allyesconfig. The number of FLEX-CFI is 12.08, reducing 93.5% of possible jump targets. FLEX-CFI reduces 80.1% for the defconfig and reduces 51.8% for the boardconfig.

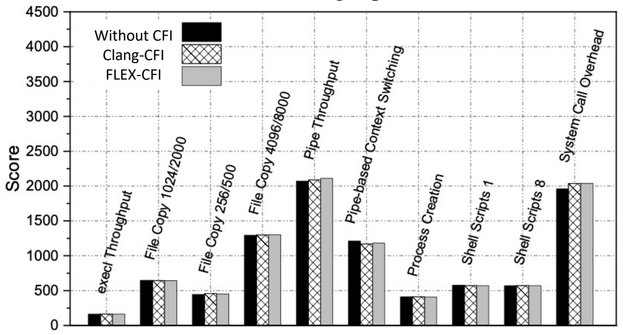
Ratio of protected icalls. The ratios of icalls protected by FLEX-CFI are shown in Table II. Row *From struct* denotes the number of icalls whose targets come from struct-fields (groups). Sub-row *Leaked* denotes the number of unprotected calls (i.e., the untraceable stubs that uncontrollably flow out of struct-field). For the allyesconfig, 180k icalls in the whole kernel among which 173k icall targets come from struct-fields. Among them, FLEX-CFI identifies that 152.7k icalls have targets, while 13k icalls have no target. Therefore, FLEX-CFI can protect 165.7k icalls in total with a ratio of 92%. For these calls without targets, we manually checked 200 of them and found that they were not assigned/initialized with functions indeed. For defconfig and boardconfig, FLEX-CFI can protect 81% and 74% icalls, respectively.

B. *Performance Overhead*

The experiments are conducted using UnixBench. We compile three kernel settings: without CFI, with Clang-CFI, and with FLEX-CFI. We replace the kernel with different kernel settings and run UnixBench on Android 12. In order to keep the experimental environment as consistent as possible, all boards are placed in the same place and all experiments are conducted at the same time. We run tests for 10 rounds and



(a) Result of single process.



(b) Result of multiple processes.

Fig. 4: Result of UnixBench. The x-axis represents different test items, and the y-axis represents the score of the test item (a higher score means better performance).

cool down the board for 10 minutes after each round. The final scores are the average of 10 rounds the results.

The result is shown in Figure 4. FLEX-CFI shows similar performance to Clang-CFI on both single process and multiple processes tests in most cases. The impact of CFI is minimal because of optimizations, which is consistent with the Clang-CFI performance results from Google [28], [19]. The average performance overhead of FLEX-CFI ranges from -3.5% to 1.8% for single process test and from -3.95% to 2.6% for multiple processes test. Overall, FLEX-CFI has a close-to-zero performance overhead.

Comparison with existing works. We compare FLEX-CFI with two kinds of software-based CFI protection methods, and non-general hardware-based CFI methods are out of scope. First, TyPro [29] uses switch/case to traverse all target candidates, and its performance overhead is proportional to the number of indirect call targets. However, FLEX-CFI protects each indirect call with the same performance overhead (because of a fixed number of CFI instructions), regardless of the number of target functions. Second, FINE-CFI [15] uses indexed hooks, which have close-to-zero overhead about indirect call protection, but it has lower precision due to the lack of propagation analysis between groups.

V. Limitation

Unprotected Cases. FLEX-CFI skips protection for untraceable groups, such as when function targets are stored in globals or function parameters. This analysis requires interprocedural points-to analysis, which is too expensive to perform on large codebases like Linux kernel. Similar to MLTA [25], FLEX-CFI is unable to protect two kinds of icalls. First, the type of an icall is a generic pointer type (e.g., void *). Second, the icall's target that is ever used for arithmetical computation.

False Positive. FLEX-CFI merges the jump tables of groups when there is a flow between the groups, which may lead to false positives (e.g., false targets). We believe the combination of indexed hooks [30] can mitigate this problem. It creates a stub copy in each group to avoid merging when dealing with too many positives. This keeps runtime performance overhead low but may increase the compiled binary size.

VI. Related Work

Coarse-grained CFI [31], [32], [33], [34] computes a relaxed CFG, thus allowing more legal jump targets, which leads to false positives. However, researchers [23] have already shown that coarse-grained CFI is ineffective in stopping some real-world exploits. Fine-grained CFI [35], [36], [17], [37], [38] computes a more accurate Control-Flow Graph. With a restricted CFG, fine-grained CFI is able to reduce false positives. Although several works realize fine-grained CFI for userspace programs [14], [11], they cannot be deployed on kernel because of heavy overhead or specific requirements.

The complexity of the kernel CFG makes the kernel hard to analyze and implement CFI policies. Therefore, some works [39], [15], [25] focus on obtaining the CFG of the kernel efficiently. Ge and Talele et al. [39] propose a fine-grained control flow integrity for the kernel. FINE-CFI[15] reduces the number of average jump targets of a limited set of indirect calls to 13.14. This work aims to provide a flexible CFI scheme, which is applicable to protect a larger set of indirect calls with fewer average jump targets.

VII. Conclusion

In this paper, we presented forward-edge FLEX-CFI to protect the indirect calls in the kernel. FLEX-CFI is a flexible design to implement CFI given a fine-grained CFG. It analyzes the data flow for groups to build complete jump targets for indirect calls. The evaluation results show that FLEX-CFI is able to protect 92% of all indirect call-sites for kernel `allyesconfig`, while reducing 93.5% of possible jump targets, and introducing close-to-zero performance overhead.

VIII. Acknowledgements

We would like to thank all reviewers for their insightful comments that greatly helped improve our paper. This work is partially supported by the National Key R&D Program of China (No. 2022YFE0113200).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[2] ——, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "{Control-Flow} bending: On the effectiveness of {Control-Flow} integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[4] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 38–49.

[5] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[6] H. Xiang, Z. Cheng, J. Li, J. Ma, and K. Lu, "Boosting practical control-flow integrity with complete field sensitivity and origin awareness," in *Proceedings of the 31st ACM Conference on Computer and Communications Security*, 2024.

[7] L. Maar, P. Nasahl, and S. Mangard, "Beyond the edges of kernel control-flow hijacking protection with hek-cfi," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1214–1230.

[8] V. Van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 927–940.

[9] B. Niu and G. Tan, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 914–926.

[10] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 131–148.

[11] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.

[12] V. Duta, C. Giuffrida, H. Bos, and E. Van Der Kouwe, "Pibe: practical kernel control-flow hardening with profile-guided indirect branch elimination," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 743–757.

[13] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive call-site sensitive control flow integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019, pp. 95–110.

[14] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 195–211.

[15] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-cfi: Fine-grained control-flow integrity for operating system kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, 2018.

[16] A. J. Gaidis, J. Moreira, K. Sun, A. Milburn, V. Atlidakis, and V. P. Kemerlis, "Fineibt: Fine-grain control-flow enforcement with indirect branch tracking," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 527–546.

[17] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 941–955.

[18] "Control flow guard for platform security - win32 apps — microsoft learn," https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard, (Accessed on 10/06/2024).

[19] Google, "Control flow integrity," 2020, https://source.android.com/devices/tech/debug/cfi.

[20] L. Becker, M. Hollick, and J. Classen, "{SoK}: On the effectiveness of {Control-Flow} integrity in practice," in *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, 2024, pp. 189–209.

[21] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "{PAC} it up: Towards pointer integrity using {ARM} pointer authentication," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 177–194.

[22] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "{In-Kernel}{Control-Flow} integrity on commodity {OSes} using {ARM} pointer authentication," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 89–106.

[23] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 401–416.

[24] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "Pex: A permission check analysis framework for linux kernel," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1205–1220.

[25] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.

[26] D. Liu, S. Ji, K. Lu, and Q. He, "Improving Indirect-Call analysis in LLVM with type and Data-Flow Co-Analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 5895–5912. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-dinghao-improving

[27] Qualcomm, "Dragonboard 845c," 2020, https://www.96boards.org/documentation/consumer/dragonboard/dragonboard845c.

[28] Google, "Control flow integrity in the android kernel," 2020, https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html.

[29] M. Bauer, I. Grishchenko, and C. Rossow, "Typro: Forward cfi for c-style indirect function calls using type propagation," in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 346–360. [Online]. Available: https://doi.org/10.1145/3564625.3564627

[30] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang, "Comprehensive and efficient protection of kernel control data," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 4, pp. 1404–1417, 2011.

[31] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," *Apr*, vol. 1, pp. 1–2, 2012.

[32] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/ropecker-generic-and-practical-approach-defending-against-rop-attacks

[33] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," Technical report, Tech. Rep., 2012.

[34] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 337–352.

[35] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 144–164.

[36] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2014, pp. 1–6.

[37] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh, "Cryptographically enforced control flow integrity," *arXiv preprint arXiv:1408.1451*, 2014.

[38] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS*, vol. 26, 2015, pp. 27–30.

[39] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 179–194.