Go With the FLOW: Fine-Grained Control-Flow Integrity for the Kernel

João Moreira[1], **Sandro Rigo**[1]

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 - Cidade Universitária, Campinas - SP, 13083-852

joao.moreira@lsc.ic.unicamp.br, srigo@ic.unicamp.br

Abstract. *This paper describes FLOW: a fine-grained control-flow integrity (CFI) implementation that focuses on protecting the Linux kernel. By combining source-code and binary analysis, FLOW maps valid execution paths into a fine-grained control-flow graph, which is later used to instrument the kernel with label-based CFI checks that prevent control-flow hijacking attacks. FLOW induces an average overhead of 17% on system call latency and 5% on I/O throughput, while its impact on real-world applications is* $\approx$ *1%.*

1. Introduction

Control-flow attacks target the modification of regular instruction sequences of a computer system, leading to unexpected behaviors. These attacks usually take advantage of algorithmically misplaced memory operations to overwrite control data, such as function pointers or return addresses. By modifying these data, attackers redirect control-flow to force a malicious program logic, built to obtain privileges on the target system.

Different protection mechanisms have been proposed since Stack Smashing [One 1996], the first big control-flow attack, was disclosed. These defenses, one after another, were proven inefficient as more attack methodologies were discovered in the wild or exposed by security researchers. Amongst the proposed solutions, Control-Flow Integrity (CFI) [Abadi et al. 2005] stands out as a practical and simple alternative, compatible with most existent software and hardware, capable of preventing powerful attacks to which other solutions are inefficient, such as return-oriented programming (ROP) attacks that can perform Turing-complete computation by arbitrarily chaining branch terminated instruction sequences that exist in the attacked code [Checkoway et al. 2010].

This paper presents FLOW, a practical fine-grained CFI solution focused on the Linux kernel that is capable of preventing control-flow hijacking attacks. To the best of our knowledge, this is the first fully fine-grained CFI mechanism supporting the Linux kernel. FLOW overheads are small, presenting averages of 17% on micro-benchmarks focused on stressing kernel code and 0.9% on user-land benchmarks running over the protected kernel. The contributions of this paper are: (i) we describe the implementation of FLOW, a tag based fine-grained CFI mechanism for the Linux kernel; (ii) we describe a methodology for building a specialized Control-Flow Graph used to build CFI policies; (iii) we present a study on tail call optimizations and how this feature influences on CFI mechanisms and (iv) we show, through a study of protected and unprotected code, how to solve incompatibilities due to these kinds of binary interactions.

The paper is organized as follows: Section 2 reviews related work; Section 3 brings background on CFI mechanisms that are crucial to understanding FLOW; Section

4 describes how FLOW was implemented and the challenges involved; Section 5 evaluates FLOW regarding overheads, protection efficiency and code coverage; Section 6 brings a final discussion on protections against ROP; Section 7 draws our conclusions.

2. Related Work

Control-Flow Integrity (CFI) [Abadi et al. 2005] is a technique originally proposed to ensure the correctness of control-transfer instructions on software. The general idea behind CFI consists of instrumenting the program with instructions (guards) that will validate the targets of indirect control-flow transfers right before its execution takes place. By doing that, these mechanisms are capable of identifying and deflecting sophisticated attacks such as ROP and variants [Shacham 2007, Checkoway et al. 2010, Bletsch et al. 2011]. Different ways of validating a target exist, but the most generic one consists of dereferencing the pointer used in the indirect control transfer and verifying if it is actually pointing towards pre-defined values (tags). In this sense, enhancing software with CFI consists of correctly calculating the set of valid targets for every indirect control-transfer instruction, instrumenting these instructions with guards and the destinations with the respective tags.

CFI is said to be coarse-grained [Zhang et al. 2013, Mingwei and Sekar 2013] if its branch target validation is loose, allowing returns to any site after a *call* instruction and indirect calls to the first instruction of any function. These mechanisms were proven vulnerable as valid malicious flows could be traced inside the protected applications even though the CFI restrictions were applied [Göktas et al. 2014, Carlini and Wagner 2014, Göktaş et al. 2014, Davi et al. 2014]. Motivated by that, fine-grained CFI mechanisms were developed [Tice et al. 2014, Mashtizadeh et al. 2015], tightening the control-flow target validation through only allowing returns to sites right after a *call* to the function which is returning and indirect calls to the first instruction of functions that follow a predefined rule, such as having the same prototype as the code pointer used in the call.

Despite all the previously mentioned efforts, the problem remains open as, for different reasons, these systems were not largely adopted and new control-flow hijacking attacks are constantly discovered. Also, few implementations focus specific domains, such as the operating system (OS) kernel. In this direction, while HyperSafe [Wang and Jiang 2010] implements CFI for indirect calls on hypervisors, kCoFI [Criswell et al. 2014] protects the OS Kernel in FreeBSD through the use of coarse-grained policies. The first fine-grained CFI mechanism for OSes was proposed by Ge et al. [Ge et al. 2016], but with restrictions on language features and execution implications that led this system to only support FreeBSD and MINIX.

3. Background

The first challenge in implementing an efficient fine-grained CFI scheme for a program consists in building a CFI-specific graph from where the sets of valid targets for a control-transfer will be extracted. For CFI-specific we mean a specialized callgraph that brings information about valid targets for indirect control-transfers between functions from different modules. We call this graph an *indirect control-flow graph (ICFG)*, to distinguish it from the control-flow graph in the traditional compiler theory, which is limited to mapping direct transitions between basic blocks.

Identifying valid targets for indirect transfers is not a trivial task, what raises the

bar of building an ICFG. Besides that, programming language features and compiler optimizations, such as symbol aliasing, weak functions, inlining, tail call and link-time optimizations, imply on different ICFG transformations that happen in different parts of the compilation pipeline. While some of these constraints can be tackled with more efficiency in the early compilation stages, others are better understood by analyzing the final binary, requiring a combination of analysis strategies and algorithms to achieve the tightest possible ICFG. The goal is to obtain an ICFG as consistent as possible to the control-transfers present in the binary, i.e., containing the smallest, but whole, number of valid paths to avoid unnecessary permissiveness.

In the ICFG, edges represent indirect control-transfers usually seen in code that is written in C. These transfers come from regular return instructions, indirect calls through function pointers, and indirect jumps resulted from tail call optimizations on indirect calls. Nodes in the ICFG can represent a function or a cluster of functions. This way, an edge can represent a return between two clearly defined functions, or an indirect call from a function to a cluster which represents all functions that can be invoked through that pointer. From now on, we will refer to the control-transfers due to return instructions as *back-edges* and to the other control-transfers as *forward-edges*.

While dynamic libraries are not a concern in a CFI mechanism for the Kernel, dynamically loadable modules require support. It is also important to notice that the majority of the kernel code is written in C, with a significant part written in Assembly, and that frequent interactions between code written in both languages exist. This is especially important because the underlying infrastructure used to implement FLOW has limitations on Assembly instrumentation. Details on these matters are discussed in Section 4.3.1.

4. FLOW

FLOW is a fine-grained CFI mechanism for the Linux kernel. Its implementation was mostly written as compilation passes inside the LLVM compiler [Lattner and Adve 2004], making use of its capabilities and Intermediate Representation (IR) to instrument kernel code during compilation time. FLOW supports dynamically loadable kernel modules, as long as the whole system is compiled and instrumented altogether. Due to differences between the Kernel source-code and LLVM syntax standards, the kernel code was patched with llvm-linux project [Linux Foundation] patches. Despite its focus on the Linux kernel, the principles behind FLOW are general enough to be applied on any other OS.

4.1. Design

As most of the existing CFI mechanisms, FLOW relies on two primitives, which are *tags* and *guards*. Tags mark special spots in the binary. Guards ensure that a certain pointer targets a specific spot, marked by a pre-defined tag, prior to its use. Theses primitives consist of sequences of assembly instructions exemplified in Figure 1.

Figure 1(a) shows the standard tag used for marking code throughout the whole kernel binary. The tag, which is a unique value highlighted in the figure, is encoded as the operand being copied by the `movl` instruction. The destination address of this instruction is a global variable reserved by the compiler for this purpose. As the value at this address is never read, the whole scheme benefits from write-back cache strategies.

Figure 1(b) shows a forward-edge guard example. In this snippet, the highlighted instruction `callq *%rcx` was protected by the instructions above it. In line 1, the value pointed to by `rcx` is verified through the `cmpl` instruction. This instruction will dereference the value pointed to by `rcx` plus an offset (`0x7`) and compare it to the expected tag. The offset exists because, as explained before, the tag is encoded as an operand of `movl`, whose first bytes represent the instruction opcode. The following instruction, `je`, will branch the code to line 6 if the previous comparison matches. If the comparison does not match, `je` does not branch and the code continues to line 3, that will push the attempted target address into the stack for analysis by the violation handler function, which is called in the next line. Once invoked, the violation handler reports a control-flow mismatch and returns to the next instruction, which will restore the stack and execute the original call. In some cases, an extra memory dereference may be needed before the guard if the function pointer is in memory, and not loaded into a register as in the example.

Figure 1(c) is a back-edge guard example where the highlighted `ret` instruction is protected. As a `ret` implicitly pops its target address, the return address is not yet loaded into any register, requiring it to be loaded by dereferencing of `rsp`, as seen on line 1. The following steps are analogous to the forward-edge guard: the tag is verified and if it does not match, the violation handler is invoked before executing the `ret` instruction.

The violation handler is a function written in C which is compiled and linked along with other kernel files. It can be easily adjusted to any system's needs, being suitable to report incidents, raise system crash or even reestablish valid flows whenever possible.

The careful reader may argue that this scheme is not efficient in terms of branch prediction or code size. On a legit program flow, it forces a branch to skip the violation handler invocation every time an indirect call happens. Also, by using a single basic block to invoke the violation handler, instead of multiple similar basic blocks for every indirect call, it is possible to save space. Although this is true, using just one basic block, positioned anywhere else in the code that is not between the comparison and the indirect call, would limit the capabilities of the violation handler for two reasons: (i) there would be no reliable way of pushing the target of the instruction that triggered the violation into the stack because indirect calls may use different registers and memory locations to hold its pointers and (ii) as the violation handler is called right before the control-transfer, the address pushed into the stack when it is called can be used as a reference to identify the instruction responsible for the violation, reducing the complexity on reporting the violation. Experiments with both approaches on the SPEC 2006 [Henning 2006] benchmark showed that the adopted strategy only degrades performance by an average of 0.9%, which is an acceptable overhead given its benefits.

The original CFI scheme [Abadi et al. 2005] also considered the use of a shadow stack to validate return addresses before using them. This enhancement would harden return address protection by ensuring that only one amongst many different call sites to the same function is considered a valid return target. Although slightly more loose, FLOW does not make use of shadow stacks for two reasons: (i) Shadow Stacks are too costly in terms of efficiency and incur significant overheads that are prohibitive, especially in the context of kernel code; (ii) A memory corruption bug in the kernel context allows not only overwriting code pointers but also the corruption of values inside the shadow stack - it is hard to protect the shadow stack against privileged exploits in kernel-land.

(a) tag

```
1 movl $0x3a66ac66,0xffffffff8585e000
```

(b) forward-edge guard

```
1 cmpl  $0x3a66ac66,0x7(%rcx)
2 je    <6>
3 push  %rcx
4 callq <violation_handler>
5 pop   %rcx
6 callq *%rcx
```

(c) back-edge guard

```
1 mov   (%rsp),%rdx
2 cmpl  $0x3a66ac66,0x7(%rdx)
3 je    7
4 push  %rdx
5 callq <violation_handler>
6 pop   %rdx
7 retq
```

Figure 1. Example Tags and Guards

4.2. FLOW Pipeline

FLOW takes advantage of information available during compilation time, such as pointer types and function attributes, to build an ICFG of valid control transfers. Part of the toolset consists of binary analysis tools, ICFG construction enhancement, and optimizations. The tools are organized as a compilation pipeline composed of three different stages:

- **Stage 1: Exploratory Compilation:** In this stage the whole kernel is compiled and information regarding control transfers is extracted. Besides compiling the code, this stage also builds a data structure to describe kernel functions, indirect transfers, and groups of functions which are suitable as destinations for the same indirect transfers for having similar prototypes. This stage also instruments every function with a unique identifier and generates a map of all aliases and aliasees present in the source code, both used for solving ICFG mismatches and source code ambiguities due to weak symbols, aliasing, and inlining.
- **Stage 2: ICFG Closure:** This stage consists of a series of analysis made over the data structure and the binary generated in the previous stage. It is composed of four different tools: (i) **Dump Shrinker**, which removes information that does not concern control-flow from the first-stage generated kernel dump, making it easier and faster to parse; (ii) **CFI merge**, that merges information generated by multiple compilation threads into a single file; (iii) **Assembly Node Mapper**, that creates ICFG nodes for the Assembly functions; and (iv) **Direct Edge Mapper**, that builds ICFG information regarding direct edges between functions, task that cannot be performed during compilation time due to linking restrictions.
- **Stage 3: Final Compilation:** Instruments the kernel with CFI guards and tags.

After these stages, the generated kernel image is ready to be installed and run.

4.3. ICFG Construction and CFI Binding

FLOW incrementally builds a data structure that holds information about the ICFG. For every module compiled, the functions are parsed and stashed as an ICFG node. Indirect calls through function pointers are stashed as an ICFG edge. Function pointer prototypes are used to create an ICFG cluster, which is a special kind of node that represents many functions, for every respective prototype. Figure 2(a) shows an example source code which is written in a given *example.c* file. If this code is processed throughout *Exploratory Compilation* and *ICFG Closure* (FLOW's pipeline stages 1 and 2), it will generate a data structure with the contents shown in Figure 2(b).

The indirect call in function bar (line 8 in Figure 2(a)) is represented by the indirect call edge (*icall*) in the ICFG Edges table (Figure 2(b)). The *From* field contains the id of the node representing the function `i32 bar(i32)` (`7d63f629`) and the *To* field contains the `i32 (i32)` cluster id (`6a8597ea`), because every function with this respective prototype is represented by this cluster, thus is considered valid. This relationship can be seen in Figure 2(c), where the gray area represents the cluster which holds both nodes `i32 foo(i32)` and `i32 bar(i32)`. In this figure, the indirect call is represented by the dashed edge, which points to the respective cluster of allowed targets.

Figures 2(b) and 2(c) also represent the direct call from `i32 bar(i32)` to `void dummy(i32)`. On FLOW's pipeline, this edge is only discovered on Stage 2. Such information is not extracted during compilation since it is easier to solve function name ambiguities through binary analysis once all the modules were already compiled and linked.

(a) source-code

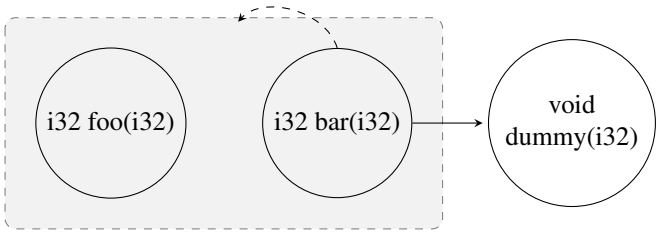
```
1  int foo(int x){
2    return x*x;
3  }
4  int bar(int y){
5    int(*f)(int);
6    f = &foo;
7    dummy(30);
8    return 7 * f(y);
9  }
10 void dummy(int z){
11   while(1){ };
12 }
```

(b) resulting ICFG data structure

ICFG Nodes				
Name	**Prototype**	**Module**	**ID**	**ret TAG**
foo	i32 (i32)	example.c	290f2fd5	1dc2aaf0
bar	i32 (i32)	example.c	7d63f629	6e28b9d1
dummy	void (i32)	example.c	6ba8458b	164e44a8

ICFG Clusters			
Prototype	**ID**	**call TAG**	**ret TAG**
i32 (i32)	6a8597ea	69e1b040	46068a5c

ICFG Edges			
Type	**ID**	**From**	**To**
icall	7dcdc019	7d63f629	6a8597ea
dcall	7728cc01	290f2fd5	6ba8458b



(c) resulting ICFG

Figure 2. ICFG Construction

On what concerns matching guards and tags, the algorithm for building guards works as follows. (i) Forward-edges: the algorithm checks the pointer type and acquires the respective cluster from the ICFG data structure, depicted in Figure 2(b). Then, the `cmp` instruction from the guard is generated with the *call TAG* for this cluster, as exemplified on line 1 of Figure 1(b). (ii) Back-edges: the algorithm first searches a cluster with a prototype that matches the function being instrumented. If such cluster exists, the `cmp` instruction is generated with the *ret TAG* for this cluster. If the cluster does not exist, this means that this function will never be called through a pointer and it is safe to use an exclusive *ret TAG* for it, so the `cmp` instruction is generated with the node's *ret TAG*. Tags and guards for back-edges are shown in Figure 1(c).

Similarly to what is done for placing the guards, the algorithm that places the tags after each `call` first checks if the transfer is direct or indirect. If it is an indirect transfer, the algorithm uses the *ret TAG* of the cluster. If it is a direct call, the algorithm checks if there is a cluster that matches the prototype of the called function. If such cluster exists, then its *ret TAG* is used. If no cluster matches the directly called function, then the *ret TAG* placed is the one respective to the node. Only functions that can be called indirectly must have a *call TAG*. Again, the algorithm processing the function verifies if it has a correspondent cluster and, if it does, places the *call TAG* on the function's prologue.

Both *ret* and *call TAGS* are mutually unique to avoid unexpected edges.

4.3.1. Assembly Nodes

A meaningful part of FLOW's code was implemented as optimization passes in the LLVM compiler infrastructure. LLVM translates source-code into an IR to which it applies optimizations and then translates into machine code. Assembly code is not translated into IR by LLVM, being directly translated into machine code. Because of that, the passes responsible for source-code analysis and CFI instrumentation are unable to process Assembly functions, leaving part of the code apart from FLOW.

The first problem generated by Assembly functions concerns the ICFG construction. By the end of FLOW's *Exploratory Compilation*, the node list only includes C functions and not Assembly functions. To fix that, FLOW runs a tool called **Assembly Node Mapper**, that parses the binary generated by this stage and maps which functions were not instrumented with unique identifiers, meaning that these are Assembly functions. These functions are added to the node list with a special Assembly flag.

After mapping all the Assembly nodes, it is possible to run the tool **Direct Edge Mapper** that identifies all edges. If this tool was executed previously, some edges would have dangling targets as some nodes would not have been identified yet.

The second problem regarding Assembly code arises from the interaction between Assembly and C code. While there is no problem in directly calling Assembly functions from C code, it is not straightforward to call C code from Assembly functions. This happens because return instructions originated from C code are instrumented with back-edge guards and will check for a tag which is not present in the Assembly code since it was not instrumented. Similarly, whenever C code indirectly calls Assembly functions, the forward-edge guard will search for a tag which is again not present, raising another violation. Such false-positives were fixed through a white-list embedded into the violation handler function. The white-list was built upon information extracted from the binary which allowed the identification of interaction sites between Assembly and C code.

The last problem due to this constraint is the difficulty of protecting edges that originate from Assembly code. As FLOW is unable to add guards to returns and indirect calls in such functions, these edges are left unprotected. Protection coverage and exploitation feasibility due to this limitation are discussed in detail in Section 5.3.

4.3.2. Tail Call Optimizations

Compilers apply Tail Call Optimizations (TCO) by replacing a `call` instruction positioned at the end of a function by a `jmp` instruction. Whenever applied, this optimization will make the callee reuse the stack of the caller function and, because of that, whenever the callee returns, it will return directly to the function underneath the caller.

This optimization poses a special challenge to CFI mechanisms because the callee's return guard will check for the callee's tag in the return address, which is actually instrumented with the caller's tag, as depicted in Figure 3. In this example, when the optimization is not applied (Figure 3(a)) the tags and the guards match. When the optimization is applied (Figure 3(b)), the function *anubis_crypt* will check for its tag before returning, but the instruction right after the address to which it will return contains the tag for the function *anubis_encrypt*.

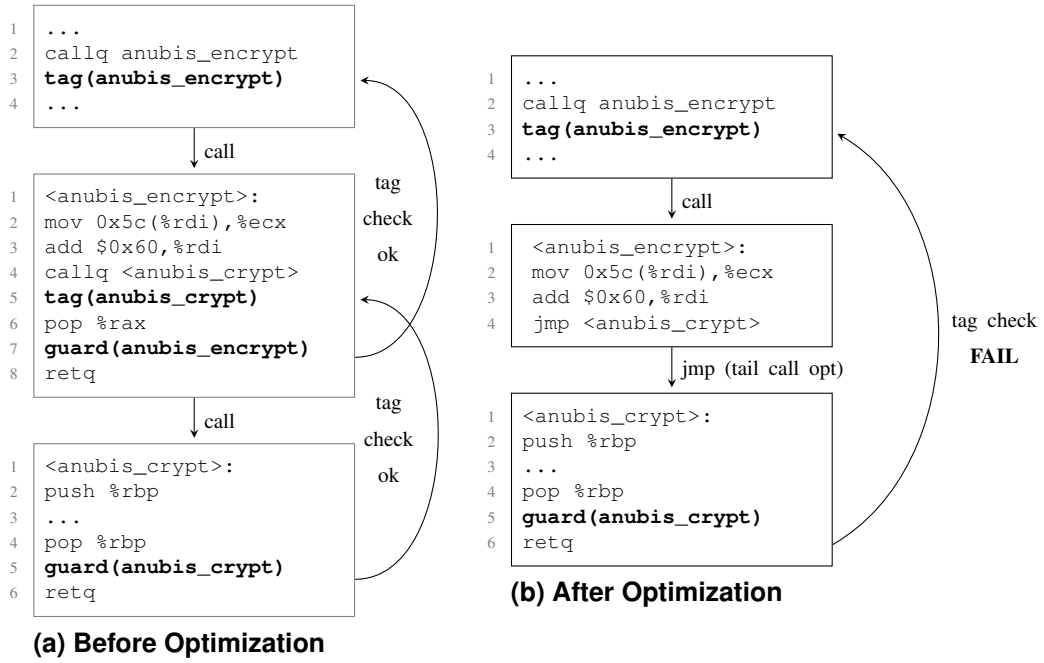


Figure 3. Assembly Before/After Tail Call Optimization

Tackling this issue by validation inside the violation handler is not worthy. A violation handler invocation requires extra `push`/`pop` instructions, plus at least one `call` and one `ret`. Even if the total cost of the computation executed inside the violation handler is ignored, these extra instructions already overweight TCO benefits.

A second solution would be collapsing the ICFG Nodes that represent the callee and the caller, so they would have similar tags. Although this solution would work, it would imply in a less-fine-grained ICFG and would only be justifiable if the benefits from applying TCO to the Kernel were significantly meaningful.

To evaluate TCO impact on kernel performance, the vanilla version of the Linux kernel was compiled with and without TCO. We used the kernel specific micro-benchmark LMBench [McVoy and Staelin 1996] in this experiment. An average performance overhead of 5% was generated by disabling TCO. We considered this as a low

impact, especially considering a kernel-specific benchmark, what implies in even smaller overheads for user-land software. For this reason, we incorporated this overhead to the overhead generated by FLOW in our evaluation and proceeded its development without TCO support. Alternatives for incorporating TCO stands as future work.

5. Evaluation

We evaluate FLOW from two different perspectives: performance and security. First, we assess the performance overhead implied by using FLOW with two well-known benchmarks, LMBench [McVoy and Staelin 1996] and SPEC2006 [Henning 2006]. Second, we evaluate FLOW coverage and its capabilities as a security enhancer through verifying its efficiency on defeating control-flow hijacking attacks.

The system used for measurement on all tests was an Intel(R) Core(TM) i7-6700 8 core CPU @ 3.40GHz, with 32GB RAM memory, running Debian Linux with GNU kernel v3.19.0 on VMware Workstation 12 Player. Every program unit inside each benchmark was executed 10 times, and the results are averages of the observed numbers. Also, it is important to highlight that the protection was only applied to the system's kernel code, thus, all benchmarks were compiled with GCC v4.9.2 without CFI instrumentation.

To clearly separate FLOW's overhead from the overhead generated by disabling TCO, we also tested an unprotected kernel without TCO, allowing the performance comparison of three different subjects: (i) a regular compilation of the Linux vanilla kernel in minimal configuration (Vreg), (ii) the same kernel compiled without TCO (VnoTCO) and (iii) the same kernel instrumented with FLOW and also without TCO (Vflow).

5.1. LMBench

To assess FLOW effects on kernel performance, we used the LMBench micro-benchmark [McVoy and Staelin 1996] . LMBench stresses the OS functionalities, allowing the observation of overheads on tasks such as performing system calls or handling page-faults.

While testing LMBench, we focused on the assessment of latencies and communication bandwidth overheads. Specifically, we evaluated the latencies of OS capabilities for executing: null syscalls; I/O critical syscalls (which are read/write, fstat and select); open/close syscalls; the installation of a signal handler; process creation followed by exit, execve and /bin/sh; context switching between processes; select syscall on 100 file descriptors; page fault handling and inter-process communication with socket and pipe. The bandwidth was measured while communicating through pipe, unix sockets (AF_UNIX) and TCP sockets. The results of these tests can be seen in Figure 4, which shows latency and bandwidth overheads while comparing VnoTCO and Vflow to our baseline, Vreg.

We measured an average latency overhead of 17.75% while comparing Vflow against Vreg. When comparing VnoTCO against Vreg, the overhead was 6.33%, making it clear that there is room for improving a future version of FLOW that can benefit from being TCO-compatible. Regarding Vflow, the smallest overheads appeared in *null syscall*, which did not show discernible overheads, and in *fork+exec*, which had a 9%overhead. On the other hand, a *select* operation on 100 file descriptors (*select 100 fd's*) presented the biggest overhead, which is 42%. The high overhead for *select* is due to a bigger than average number of CFI checks performed by the invoked syscall.

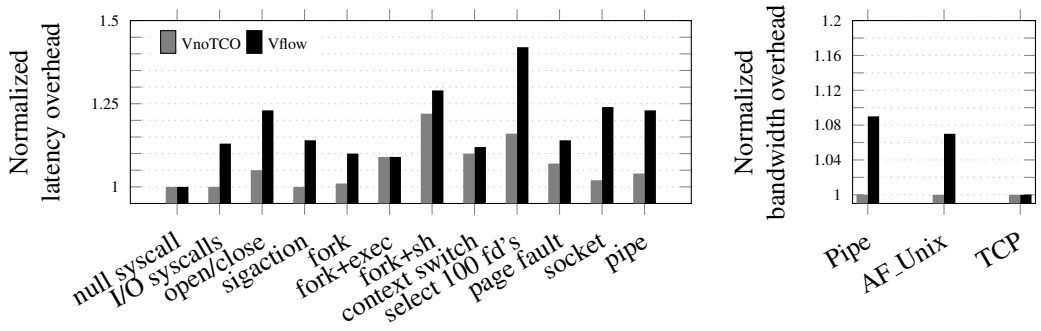


Figure 4. FLOW performance on LMBench

SPEC INT 2006 Vflow x Vreg overheads					
perlbench	0.5%	**bzip2**	0.9%	**gcc**	1.1%
mcf	0.4%	**gobmk**	1.2%	**hmmer**	1.5%
sjeng	1.4%	**libquantum**	1.3%	**h264ref**	0.8%
omnetpp	0.8%	**astar**	0.3%	**xalancbmk**	0.6%
				average	0.9

Figure 5. SPEC INT 2006 overheads

The bandwidth overheads describe how less efficient the communication was, meaning that less data was transported in the same period of time. Vflow presented an average bandwidth overhead of 5.33%, while VnoTCO did not present any discernible overhead at all. The biggest overhead (9%) was verified on *pipe*, while *AF_Unix* presented an overhead of 7% and *TCP* did not present any significative overhead.

As shown, FLOW performs well on the execution of LMBench, presenting low overhead on latencies for the execution of system calls and not harming significantly the communication bandwidth. When a comparison is possible, FLOW performs better or similarly to other existing CFI mechanisms for OSes: kCoFI [Criswell et al. 2014] presented overheads for LMBench applications that ranged from 100% to 250%.

5.2. SPEC 2006

Although LMBench shows meaningful numbers, computer systems have most of their useful work done by user-land software. We used the SPEC 2006[Henning 2006] benchmark to assess FLOW in this scenario. SPEC is composed of user-land CPU-intensive applications, therefore, the amount of execution time spent on the kernel code is reduced. All programs in the SPEC-INT 2006 suite were executed using the reference input.

When comparing the Vreg against VnoTCO, the average overhead was of 0.3%, with a maximum of 0.7% for *omnetpp*. Applications *perlbench*, *gcc*, and *mcf* presented overheads below 0.1%, what was considered indiscernible. The full comparison between Vreg and Vflow is presented in Figure 5. As shown, Vflow induced an average overhead of 0.9%, with a maximum of 1.5% for *hmmer*, confirming the low overhead expectancy.

5.3. Security and Coverage

FLOW improves previously existent coarse-grained CFI mechanisms. For being a fine-grained, FLOW has a smaller set of valid execution paths, imposing harder exploitation

barriers. If the same tested kernel was instrumented with both approaches, the function pointers with the prototype `void ()` would have 1130 valid target functions when instrumented with FLOW, while on a coarse-grained system this permissiveness would be $\approx$ 2525% higher, allowing it to target the beginning of any function in the kernel, a total of 28537 valid targets. On the compiled system, the function prototype `void ()` is the most common one, thus 1130 functions is the biggest allowed target functions set on the system, what makes this proportion even larger when comparing other prototypes. The average number of valid targets for a function pointer on FLOW is $\approx$10.6.

As explained before, Assembly is not translated into IR by LLVM and, thus, is not instrumented by FLOW. Statistics on how much of the kernel was left unprotected were grabbed through binary analysis. The kernel binary has 28537 functions from which 351 are Assembly functions, representing $\approx$ 1.2% of the code. Throughout the binary, we had 28865 return instructions from which 85 were not protected, representing $\approx$ 0.3% of the total `ret` instructions. This proportion is even smaller for indirect calls as 8 unprotected indirect calls out of 6053 were found, representing less than 0.2% of the total.

To evaluate the effectiveness of FLOW against real-world control-flow hijacking attacks, we used the ROP exploit for CVE-2013-2094 [Kemerlis et al. 2014], which targets Linux v3.8 (x86-64). We first verified that the exploit was successful on the respective kernel, and then tested it on the same kernel protected with FLOW. As expected, the code-reuse attack failed; the ROP payload used by the exploit relied on pre-computed gadget addresses, none of which corresponded to a valid control-flow transfer under FLOW.

These results show that a very significant part of the code is covered by our mechanism, imposing significant difficulties to attackers intended to compromise a system with such protection. Overcoming FLOW requires an attacker to exploit these specific targets, what is unlikely to be possible as attacks require very specific contexts to be deployable and these are now heavily limited by such a small window of unprotected instructions.

5.4. Code-size

Disabling TCO and instrumenting the binary added an overhead of 15.9% on code-size between Vflow to Vreg. No discernible overheads were seen between VnoTCO to Vreg.

6. Discussion

Recently, it was shown that fine-grained CFI may be vulnerable to attacks if very specific circumstances are met. Control-Flow Bending (CFB) [Carlini et al. 2015] is a control-flow attack built on top of a data-only attack. This attack uses a technique called *printf-oriented programming*, which is strongly based on an old attack called *format strings* [Scut and Teso 2001] to perform Turing-complete computation. CFB also relies on the creation of execution loops, which is performed by the corruption of return pointers towards valid targets that will later take the execution towards the exploited function. Control Jujutsu [Evans et al. 2015] demonstrates how indirect calls are still vulnerable even under fine-grained CFI. This work describes a hypothetical fine-grained CFI whose valid indirect-call targets were gathered through the DSA [Lattner et al. 2007] algorithm. Later, the paper shows a specific pointer corruption that enables system exploitation.

In spite of these attacks towards CFI, we argue that our scheme remains relevant as (i) the requirements for both attacks are very specific and tight, hardly met in real-world

scenarios and unlikely to be found in the kernel context; (ii) if extended with a shadow stack mechanism, FLOW would be able to deflect CFB, so it stands as an important first step in this direction; and (iii) for using a different algorithm to build its ICFG, FLOW does not allow the edges used in the Control Jujutsu attack, proving that using extensible and modular techniques to build an ICFG is a good strategy to tackle the problem.

Branch frequency-based mechanisms were also proposed as a solution against ROP [Pappas et al. 2013, Cheng et al. 2014]. These defenses monitor a ratio of instructions executed in between branches. As ROP gadgets are small sequences of instructions followed by a control-transfer and launching ROP attacks consists in chaining the execution of these gadgets, it is possible to monitor the branching ratio, define acceptable thresholds and identify the anomalous behaviors. Even thought optimizations have been proposed to these schemes [Rubens et al. 2015], the problem remains open as they (i) present prohibitive overheads, much higher than compiler-based solutions, (ii) are vulnerable to gadget chaining obfuscation attacks [Göktas et al. 2014, Carlini and Wagner 2014, Davi et al. 2014, Göktaş et al. 2014] and (iii) do not add extra defenses and are also vulnerable to the two attacks previously described against CFI.

We also would like to add that FLOW differs from the fine-grained CFI implementation proposed by Ge [Ge et al. 2016]. This work covers FreeBSD and MINIX and relies on techniques such as the conversion of indirect calls into direct calls. Its ICFG is built through taint analyses that impose restrictions on the use of language features such as operations on function pointers. FLOW also brings fine-grained CFI for the kernel but differs from Ge's work by (i) supporting the Linux kernel, which is a larger and more diverse code base; (ii) not relying on code modifications, such as pointer conversion, which can harm system design, and (iii) using a different clustering method, not dependent of analyses that restricts the use of language features and that was designed to be easily extended by methods such as static control-flow analysis [Lattner et al. 2007].

7. Conclusions

In this paper, we presented FLOW, a CFI solution for the Linux kernel. To develop a CFI mechanism capable of protecting an OS, specific challenges were faced, such as defining a control-flow graph that describes valid flows without much relaxation and is compatible to circumstances imposed by the domain, such as interactions between C and Assembly code, symbol aliasing, weak linking, inlining and other compiler optimizations.

FLOW proved to be an efficient solution against control-flow attacks and presented low overheads, showing an average of 17% on kernel micro-benchmarks. The paper also shows that this overhead is significantly amortized for user-land applications, reaching an average of 0.9%, when measured in the SPEC 2006 benchmark.

References

Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. (2005). Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, CCS '05, New York, NY, USA. ACM.

Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. (2011). Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on*

Information, Computer and Communications Security, ASIACCS '11, pages 30–40, New York, NY, USA. ACM.

Carlini, N., Barresi, A., Payer, M., Wagner, D., and Gross, T. R. (2015). Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C. USENIX Association.

Carlini, N. and Wagner, D. (2014). Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA. USENIX Association.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, New York, NY, USA. ACM.

Cheng, Y., Zhou, Z., Yu, M., Ding, X., and Deng, R. H. (2014). Ropecker: A generic and practical approach for defending against rop attacks. In *NDSS*. The Internet Society.

Criswell, J., Dautenhahn, N., and Adve, V. (2014). Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*.

Davi, L., Sadeghi, A.-R., Lehmann, D., and Monrose, F. (2014). Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA. USENIX Association.

Evans, I., Long, F., Otgonbaatar, U., Shrobe, H., Rinard, M., Okhravi, H., and Sidiroglou-Douskos, S. (2015). Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, New York, NY, USA. ACM.

Ge, X., Talele, N., Payer, M., and Jaeger, T. (2016). Fine-grained control-flow integrity for kernel software. In *IEEE European Symposium on Security and Privacy 2016*, Euro S&P, Washington, USA. IEEE Computer Society.

Göktas, E., Athanasopoulos, E., Bos, H., and Portokalidis, G. (2014). Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, Washington, DC, USA. IEEE Computer Society.

Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., and Portokalidis, G. (2014). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA. USENIX Association.

Henning, J. L. (2006). SPECCPU 2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4).

Kemerlis, V. P., Polychronakis, M., and Keromytis, A. D. (2014). ret2dir: Rethinking Kernel Isolation. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA. USENIX Association.

Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code*

Generation and Optimization: Feedback-directed and Runtime Optimization, CGO '04, Washington, USA. IEEE Computer Society.

Lattner, C., Lenharth, A., and Adve, V. (2007). Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California.

Linux Foundation. LLVMLinux. `http://llvm.linuxfoundation.org/`. Accessed 2016-05-22.

Mashtizadeh, A. J., Bittau, A., Boneh, D., and Mazières, D. (2015). Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, New York, NY, USA. ACM.

McVoy, L. and Staelin, C. (1996). Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, Berkeley, CA, USA. USENIX Association.

Mingwei, Z. and Sekar, R. (2013). Control flow integrity for cots binaries. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C. USENIX.

One, A. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2013). Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, Washington, D.C. USENIX.

Rubens, E., Tymburibá, M., and Pereira, F. (2015). Inferência estática da frequência máxima de instruções de retorno para detecção de ataques rop. In *Simpsósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, SBSEG XV.

Scut and Teso, T. (2001). Exploiting format string vulnerabilities. `http://julianor.tripod.com/bc/formatstring-1.2.pdf`. Accessed 2016-05-22.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, New York, NY, USA. ACM.

Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L., and Pike, G. (2014). Enforcing forward-edge control-flow integrity in gcc & llvm. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA. USENIX Association.

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. (2013). Practical control flow integrity and randomization for binary executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, Washington, DC, USA. IEEE Computer Society.

