# Transparent and Efficient CFI Enforcement with Intel Processor Trace

Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen*, Binyu Zang, Haibing Guan
Shanghai Key Laboratory for Scalable Computing and Systems
Shanghai Jiao Tong University

## ABSTRACT

Current control flow integrity (CFI) enforcement approaches either require instrumenting application executables and even shared libraries, or are unable to defend against sophisticated attacks due to relaxed security policies, or both; many of them also incur high runtime overhead. This paper observes that the main obstacle of providing transparent and strong defense against sophisticated adversaries is the lack of sufficient runtime control flow information. To this end, this paper describes FlowGuard, a lightweight, transparent CFI enforcement approach by a novel reuse of Intel Processor Trace (IPT), a recent hardware feature that efficiently captures the entire runtime control flow. The main challenge is that IPT is designed for offline performance analysis and software debugging such that decoding collected control flow traces is prohibitively slow on the fly. FlowGuard addresses this challenge by reconstructing applications' conservative control flow graphs (CFG) to be compatible with the compressed encoding format of IPT, and labeling the CFG edges with credits in the help of fuzzing-like dynamic training. At runtime, FlowGuard separates fast and slow paths such that the fast path compares the labeled CFGs with the IPT traces for fast filtering, while the slow path decodes necessary IPT traces for strong security. We have implemented and evaluated FlowGuard on a commodity Intel Skylake machine with IPT support. Evaluation results show that FlowGuard is effective in enforcing CFI for several applications, while introducing only small performance overhead. We also show that, with minor hardware extensions, the performance overhead can be further reduced.

## 1. INTRODUCTION

The race between advanced attacking means and sophisticated defenses persists. With novel defenses like data execution prevention (DEP) or no-execute (NX) that prevent code injection, attackers have resorted to using sophisticated means that reuses existing code. For example, return oriented programming [1] and its variants, as the major advancement along the line of evolution of code reuse attacks, have emerged as the major attacking means for arbitrary code execution even under DEP. Control flow integrity (CFI) [2], a general property that thwarts attacks manipulating execution flow, has gained considerable interests over the last decade.

**Enforcing CFI: challenges.** While CFI is a very intriguing idea, efficiently and securely enforcing CFI in a practical way is challenging. Despite sustained interests in CFI enforcement, prior approaches usually have to strike a balance among the following requirements, which we believe are keys to practical CFI enforcement: 1). *Precision*, by which the enforcement must be fine-grained enough to eliminate as much attack surface as possible; 2). *Efficiency*, by which the imposed runtime overhead should be small; 3). *Transparency*, i.e., the approach should be transparent to applications, easily deployable on existing OS and hardware and are generally compatible with existing security mechanisms.

More specifically, while approaches using binary rewriting [2, 3, 4, 5, 6, 7] maintains reasonable transparency, they inevitably break the binary code integrity that leads to incompatibility with some existing security mechanisms (e.g., Window 7 system library protection and remote attestation). Besides, most classic binary-based approaches [3, 4] usually relax constraints on the set of legal targets of branches and thus sacrifice fine-grained CFI enforcement at the risk of exposing intensive attack surfaces. Many compiler-based approaches [8, 9, 10, 11, 12] implement finer-grained CFI protection by generating more precise backward (e.g., shadow stack) and forward (e.g., indirect call checks) control edges, which, however, are unfriendly to shared libraries, and not applicable to deployed COTS applications. Architectural approaches [13, 14, 15, 16] require changes to processor architecture and/or ISA and thus are not readily deployable in commodity platforms. While there have been approaches leveraging existing hardware supports for CFI enforcement [17, 18, 19, 20, 21], they either impose high overhead [17], or are imprecise and thus vulnerable to sophisticated attacks [19, 18]. PathArmor [20] uses context-sensitive protection, however it suffers from the problem of LBR pollution, thus has to resort to instrumenting libraries.

**Lightweight and Transparent Protection with IPT.** In this paper, we present a novel CFI enforcement approach that we believe simultaneously meets the three requirements. Our approach, namely FlowGuard, is the first to make a novel reuse of Intel Processor Trace (IPT) to efficiently collect runtime control flow traces, which are compared with a statically derived control flow graph to detect abnormal control flow. FlowGuard is precise since IPT has all program's control flow traces; it is efficient thanks to the low-overhead runtime tracing capability of IPT; finally it is transparent such that it requires no binary instrumentation, and can be readily deployable on commodity hardware.

However, as an offline tracing mechanism, IPT poses a significant challenge for being used as an online detection mechanism: the runtime traces are highly compressed for

*Corresponding author: haibochen@sjtu.edu.cn efficiency and decoding the traces is usually orders of magnitude slower than tracing, which prevents IPT from being used as an online detection mechanism. FlowGuard addresses this challenge in two ways. First, instead of dynamically decoding the traces for CFI checking, FlowGuard constructs a control flow graph offline in a form that can be directly compared with IPT traces. Specifically, FlowGuard reconstructs a conservatively generated CFG to an indirect targets connected one (ITC-CFG) which conforms to IPT's trace packets, and then uses a coverage-oriented fuzzing to label the edges with credits and branch taking information. Second, FlowGuard adopts a hybrid flow checking mechanism that separates the fast and slow paths: most of the runtime flow traces can be checked in a fast path by directly searching on the reconstructed ITC-CFG with credits, while only some rare abnormal flow traces are passed to the slow path for precise CFI checking and enforcement. Such a separation of fast and slow paths makes FlowGuard embrace both runtime efficiency and CFI detection precision.

To demonstrate the effectiveness and efficiency of FlowGuard, we have implemented FlowGuard on an Intel Skylake machine with IPT support. Like prior approaches, FlowGuard uses critical system calls as endpoints for CFI checking. We have applied FlowGuard to defend against ROP-like control flow hijacking attacks in a transparent way. Our evaluation shows that FlowGuard is effective in significantly reducing the average indirect targets allowed (AIA) [22]. Performance evaluation shows that FlowGuard incurs only small performance slowdown. Our investigation also shows that with minor hardware extensions, e.g., hardware-assisted fast decoder, etc., FlowGuard can further reduce the overhead of reusing IPT for CFI enforcement.

**Contributions.** In summary, this paper makes the following contributions:

- *Novel reuse of offline IPT mechanism for online CFI enforcement.* We take the first step of leveraging IPT to trace control flow and detect violation of CFI. We analyze the feasibility in such an approach, and address the challenges using IPT in a novel way (§3).

- *Hybrid flow checking that embraces efficiency and precision.* We present FlowGuard, a fully transparent approach to detecting CFI violation. FlowGuard uses static binary analysis to generate the complete CFG and reconstruct it adapting to the IPT output pattern, and dynamically trains it with a fuzzing-like approach, to label edges with credits and branch taking information(§4), which is important in implementing a hybrid flow checking mechanism with fast and slow paths at runtime (§5).

- *Implementation and evaluation.* We implemented and evaluated FlowGuard on a commodity Intel machine using popular vulnerable server applications like Nginx, and Linux utilities. The evaluation confirms its effectiveness in detecting control flow violation and reducing possible legal target sets, as well as its efficiency in introducing small performance overhead (§7). Our experience of using IPT also gives effective suggestions on improving hardware for a better CFI enabler(§6).

## 2. EXECUTION TRACING IN HARDWARE

Currently, there are several control flow tracing mechanisms in hardware, each representing a different set of trade-offs (as shown in Table 1). *Branch Trace Store* (BTS) can capture all control transfer events (e.g., call, return and all types of jumps) to a memory-resident BTS buffer. Each record contains the addresses of source and target of the branch instruction, thus there is no need to decode it. However, BTS introduces very high overhead during tracing and is inflexible due to the lack of event filtering mechanisms. While *Last Brach Record* (LBR) has some support of event filtering (e.g., filtering out conditional branches), it can only record 16 or 32 most recent branch pairs (source and target) into a register stack. Though it incurs very low tracing overhead, it can hardly provide precise protection.

**Table 1: A comparison of hardware control flow tracing mechanisms. Tracing overhead is the geometric mean tested on SPECCPU 2006. Other non CPU-intensive applications have lower overhead.**

| | **Precise** | **Tracing overhead** | **Decoding overhead** | **Filtering mechanisms** |
|---|---|---|---|---|
| **BTS** | Full | High (50X) | None | None |
| **LBR** | Low | Very Low (<1%) | None | CPL, CoFI type |
| **IPT** | Full | Low (3%) | High | CPL, CR3, IP |

Due to the capability of dynamically tracing control flow, BTS and LBR have been exploited to defend against ROP-like attacks, which, however, either incur high overhead (those using BTS [17]) or sacrifice security due to imprecise tracing (LBR [18, 19, 20, 21]).

**Intel Processor Trace.** IPT is introduced in Intel Core M and 5th generation Intel Core processors. Each CPU core has its own IPT hardware that generates trace information of running programs in the form of *packets*. IPT configuration can only be done by the privileged agents (e.g., OS) using certain model-specific registers (MSRs). The traced packets are written to the pre-configured memory buffer in a compressed form to minimize the output bandwidth and reduce the tracing overhead. The software decoder can decode the traced packets based on pre-defined format, with the extra information like the program binaries, as well as some runtime data provided by the control agent, to precisely reconstruct the program flow. Thanks to the aggressive compression of traces, it can collect more control flow tracing information including control flow, execution modes, and timings than BTS, yet incurring much less tracing overhead compared to BTS. This, however, also incurs orders of magnitude slower decoding speed than tracing.

**Table 2: An example of how IPT traces execution**

| **No.** | **Execution Flow** | **Traced Packets** |
|---|---|---|
| **1** | 0x8fa jg 0x8fe // taken | TNT(1) |
| **2** | 0x8fe jmpq *%rax // %rax = 0x905 | TIP(0x905) |
| **3** | 0x905 callq fun1 | |
| **4** | 0x90a mov -0x18(%rbp),%rax | |
| | **fun1**: | |
| **5** | 0x940 ... | |
| **6** | 0x970 cmp %rax, %rax | |
| **7** | 0x974 je 0x983 // not-taken | TNT(0) |
| **8** | 0x979 jmpq 0xe10 | |
| **9** | 0xe10 leaveq; retq | TIP(0x90a) |

Table 2 explicates an example of how IPT traces execution flow. Each packet is generated only for non-statically known control flow changes, i.e., unconditional direct branches are not logged (e.g., no output for No.3 and No.8 branches). Each conditional branch is compressed to a single bit to imply taken or non-taken (e.g., TNT packets for No.1 and No.7 branch); other control flow will generate the target addresses of indirect branches, exceptions and interrupts (e.g., TIP packets for No.2 and No.9 instructions), or the source addresses for asynchronous events (e.g., FUP packets, not shown in Table 2). Table 3 lists all of the change of flow instructions (CoFI), as well as their respective output packets traced by IPT. Using this degree of compression, there is less than 1 bit information recorded for each retired instruction on average. Further, IPT supports powerful event filtering based on current privilege level (CPL), CR3 value which represents the page directory base register, or certain instruction pointer (IP) ranges, which can be leveraged to filter out unnecessary packets.

**Table 3: CoFI and associated IPT output**

| CoFI type | Scenarios | Output |
|---|---|---|
| Unconditional Direct Branch | JMP and CALL (direct) | No output |
| Conditional Branch | Jcc, J*CXZ, LOOP | TNT |
| Indirect Branch | JMP and CALL (indirect) | TIP |
| Near Ret | RET | TIP |
| Far Transfers | Interrupts, traps, etc. | TIP \| FUP |

Undoubtedly there is a price to this fast tracing mechanism: the performance overhead is shifted from the tracing to the decoding. Since the traced information is compressed and incomplete, the decoder must associate the traced packets with the binaries, to precisely reconstruct the program flow. For example, in order to reconstruct the execution flow, the Intel's reference implementation of its IPT decoder library uses the instruction flow layer of abstraction, which parses the program binary instruction by instruction, and combines the traced packets for the entire decoding. We conduct a simple evaluation to provide an intuition about how slow the decoding could be. We run SPECCPU 2006 benchmarks and trace their execution flow using IPT, whenever the traced buffer is filled, we pause the execution and decode the packets by associating corresponding binaries. The geometric mean of the overhead is about 230X, and 8 out of 12 benchmarks incur more than 500X overhead.

# 3. REUSING IPT FOR CFI

## 3.1 Challenges

While IPT provides several useful features like precise tracing, low tracing overhead and event filtering that are promising for runtime CFI checking, there is also a major challenge due to incomplete trace packet and slow decoding. This is because the initial purpose of IPT is for offline analysis such as performance profiling, tuning and software debugging, it aggressively trades slow decoding for fast tracing and relies on offline reconstruction to derive the complete control flow information. One particularly observation is that with traditional CFG, where basic blocks are connected with each other using (in)direct branch edges, is unfriendly to the IPT recorded flow checking. For example, as known control flow changes like unconditional direct branches as well as program counter information are incomplete, it is impossible to derive complete control flow traces for runtime control flow checking without slow full decoding. Worse even, with the IPT tracing format, there are generally only 2 types of packets (TNT and TIP) available for control flow checking. An analyzer has no idea about the type (e.g., call, return, or jump) of the related instruction for each packet.

## 3.2 FlowGuard Overview

Like prior approaches [17, 18, 19, 20, 21], FlowGuard combines offline CFG construction and online control flow checking to enforce CFI. To address the challenge of slow decoding of IPT, FlowGuard instead resorts to enhancing the offline CFG construction to generate a CFG that conforms to IPT's packet format for efficient runtime checking.

**CFG generation and reconstruction.** FlowGuard constructs an IPT-compatible CFG by eliminating the direct branch edges and connecting the target basic blocks of indirect branches. For the convenience's sake, we call it as indirect targets connected CFG (ITC-CFG). Though ITC-CFG lacks information like direct call/jmp and conditional jmp, it is guaranteed not to introduce false positive. The IPT traced packets can be directly and efficiently searched on the ITC-CFG to minimize runtime checking overhead.

**Refining CFG via a best-effort dynamic approach.** Since the static CFG is generated conservatively, for each indirect branch, its legal target set may unfortunately be too large that it provides more degrees of freedom for the attackers. FlowGuard uses a novel refinement of the ITC-CFG with a fuzzing-like dynamic training phase. In a nutshell, FlowGuard generates inputs for the protected application using coverage-oriented fuzzing and then enhances the IPT-compatible CFG with some strongly credible edges and additional branch taken information. Since the training phase is best-effort without full coverage, the rest of edges should not be discarded, but labeled with less credits. The credits of the edges will be considered to see if there is a need for further checking during the runtime CFI enforcement phase.

**Hybrid control flow checking.** FlowGuard adopts a hybrid checking scheme to separate fast and slow paths based on the labeled credits. Starting from the triggering point, FlowGuard checks a specific number of TIP packets traced by IPT. The checking fails upon any violation of the CFG. Such a checking is fast and incurs no false positive. If all checked edges are with high credits, the checking passes, otherwise, FlowGuard resorts to a slow but precise flow checking with the help of more runtime context and binaries. Since entering into the slow path is rare thanks to the high-coverage training and relatively fixed boundaries of protected endpoints in normal cases, FlowGuard can enjoy good performance without sacrificing security.

**Architecture overview.** The FlowGuard overview is shown in Figure 1. The protected executable and libraries are parsed by the static analysis module (step ①) to generate an ITC-CFG, whose edges are then labeled with credits using the fuzzing training module (step ②): the trained edges are labeled with high credits, and associated with TNT information, while the others are with low credits. Given the reconstructed and weighted CFG, the CFI of the protected processes can be checked during runtime. FlowGuard relies on a kernel module to do the flow checking. The kernel module configures the CPU cores to start and keep tracing the protected process filtered by specific *CR3* values (step ③), the traced packets are recorded to the pre-configured memory buffer. During runtime, FlowGuard intercepts the security-sensitive system calls, if any of them is issued by the protected process (step ④), the flow checking is triggered.

During the flow checking phase (step ⑤), the fast path takes precedence over the slow one, the former can rapidly check out if the flow is out of the ITC-CFG thus malicious, or it is suspicious without all edges with high credits. The latter will trigger a full checking in the slow path.

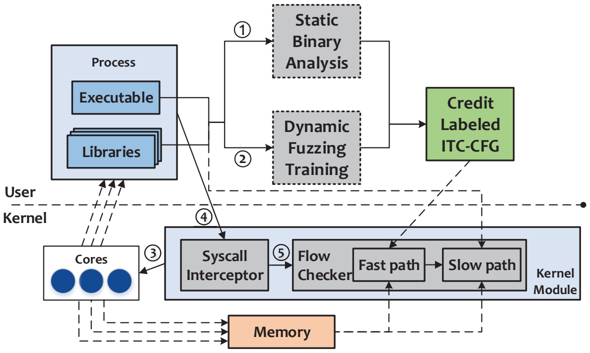


**Figure 1: The architecture of FlowGuard**

## 3.3 Threat Model and Assumptions

We consider a remote adversary targeting at a user-level application and on purpose of launching code reuse attacks by elaborately constructing input. The sophisticated adversary knows everything about the application, so that she can leverage the vulnerabilities within it and utilize the possible gadgets inside the executables and shared libraries. Nonetheless, the kernel is trusted thus its services (e.g., MMU protection) cannot be subverted. Furthermore, FlowGuard assumes that mechanisms (e.g., DEP, NX, etc.) are provided by the OS, and the code pages are read-only. FlowGuard makes no assumption of the Address Space Layout Randomization (ASLR) mechanism in the system, it does not rely on but can be adapted to it. Finally, before the distribution of the protected software, the static CFG generation and dynamic training are securely conducted, the binaries are not created with malicious purpose and there is no self-modification logic in them. Thus the generated CFG is well-intentioned and trusted.

# 4. CONTROL FLOW GRAPH

FlowGuard constructs an IPT-compatible CFG from the protected binaries during the offline analysis phase. As shown in Figure 2, the construction can be divided into three steps: static binary analysis to generate a conservative CFG, CFG parsing to construct the IPT-compatible ITC-CFG, and dynamic fuzzing training to label the ITC-CFG edges with credits and TNT information.

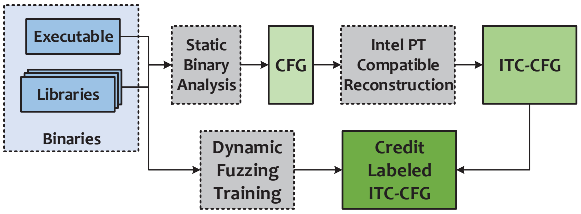


**Figure 2: The process of final CFG construction**

## 4.1 Conservative CFG Generation

Without relying on source code, FlowGuard uses existing binary-based approaches [4, 23] to disassembling and analyzing a binary executable and its dependent shared libraries. As prior approaches, the CFG is conservative such that the possible outgoing edges for indeterminable indirect branches may be more than necessary, to avoid false positives under which FlowGuard rejects legitimate flow during runtime checking.

FlowGuard first analyzes different modules such as executable and libraries independently and constructs the intra-module CFGs. It then constructs the inter-module edges by using the procedure linkage table (PLT) exposed by the dynamic linking mechanism. Different modules can only be connected by the indirect jumps in the PLT as well as the corresponding return instructions from callees to callers. FlowGuard adds edges among these particular basic blocks accordingly. To handle the global symbol interpose problem where one symbol may exist in different modules, FlowGuard likewise uses the information (e.g., DT_NEEDED) fields in the binaries to find the prior library and binds a symbol to the specific address. Another type of inter-module branches is due to the virtual dynamically-linked shared object (VDSO) mechanism, which is used to accelerate syscall invocation. For instance, the *gettimeofday()* usually results in VDSO function call instead of library call. The functions within the VDSO segments take precedence over libraries.

To generate the intra-module CFG, each direct call/jump instruction has one exact outgoing target, and each conditional branch has two possible targets. For indirect calls, FlowGuard restricts the targets using the TypeArmor's [7] use-def and liveness analysis, and connects the return instructions to the valid return addresses right after the call sites in a manner like call/return matching, and finally resorts to the underlying binary analysis framework to conservatively resolve the indirect jumps.

Another issue is handling tail-call optimization. A tail call is normally issued in the final part of a function (e.g., *fun_b*). The tail call reuses the current stack frame and uses *jump* instead of *call* to the target function (*fun_c*), pretending that it is being called by the caller (*fun_a*) of the current function (*fun_b*). In this case, the return instruction of the *fun_c* should go back to *fun_a*, even though there is no call from *fun_a* to *fun_c*. FlowGuard adapts a prior approach [22] to detect and handle such tail calls. Specifically, at each call site, FlowGuard emulates the execution of the target function and sequentially follows the branch instructions, collects any inter-procedure jump instructions targeted at function entries, and connects the return instructions of those functions to the return address right after the very beginning call site, until encountering specific stop conditions.

An example of the generated CFG is shown in part (a) of Figure 3, which uses only 10 basic blocks for clarity. After generating the conservative CFG, the basic blocks are connected with each other by either direct or indirect branches as edges. One such edge associates the exit address of the source basic block to the entry address of the target one.

## 4.2 IPT-compatible CFG Construction

The generated CFG above is, however, incompatible with the traces collected by IPT. FlowGuard further refines the above CFG to generate an IPT-compatible CFG that is connected using indirect branches only. As elaborated in

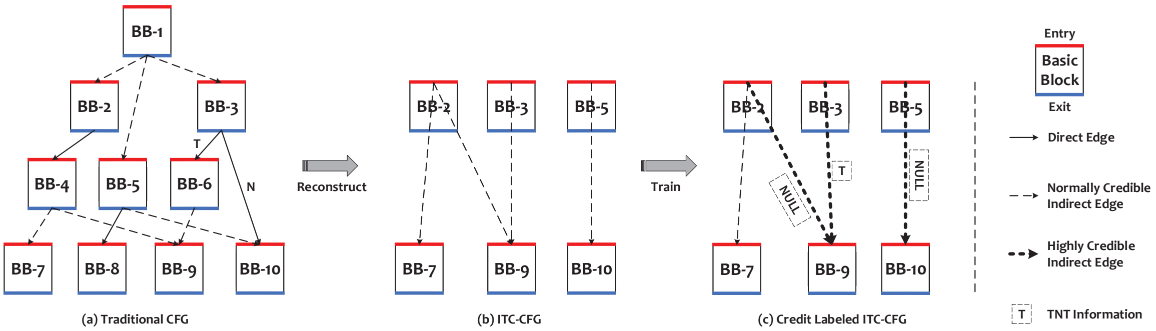


**Figure 3: An example of reconstructing O-CFG (a) to ITC-CFG (b) and credit labeled ITC-CFG (c)**

§2, IPT generates respective packets for specific CoFI type. We observe that within these packets, only two types are supportive for CFI checking: the TNT packet indicating whether the conditional branch is taken or not, and the TIP packet recording the target address of the indirect branch.

To support fast online searching over an ITC-CFG, FlowGuard only considers TIP packets to avoid path explosion. Specifically, FlowGuard collapses all edges with direct types such that the target basic blocks of the indirect branches are connected to each other, and each edge associates the entry address of one indirect target basic block to the entry address of another indirect target basic block.

An example of the conversion of the CFG in part (a) is shown in part (b) of Figure 3. For simplicity, in the following paragraphs, we name the original CFG as O-CFG, the reconstructed one as ITC-CFG, and BB-n is short for the $n$th basic block in Figure 3. In the O-CFG, edges are divided into direct and indirect categories, FlowGuard only considers the basic blocks which are targets of at least one indirect edges, e.g., BB-2 is the target basic block of indirect edge incoming from BB-1, while BB-4 does not belong to any incoming indirect edges. Thus the basic blocks of number (2, 3, 5, 7, 9, 10) are left in the ITC-CFG, we call them the indirect target basic blocks (IT-BB). For the edge reconnection, each edge is re-associated from one IT-BB to its nearest IT-BBs if there is one indirect edge in the path in O-CFG. For instance, in O-CFG from BB-3 to BB-9, there is one indirect edge from BB-6 to BB-9, thus in ITC-CFG, there is one edge connected from BB-3 to BB-9. While there is no indirect edge in the path from BB-3 to BB-10 in O-CFG, BB-3 is not connected to BB-10 in ITC-CFG. This makes sense that, if there is no indirect branch from one IT-BB to another, then no TIP packet will be generated by IPT during runtime, thus there should be no connection between them even if they are both IT-BBs and connected by direct edges. Furthermore, since the TIP packets reveal the target memory addresses of indirect branches, the edges of ITC-CFG connect the entry addresses of the IT-BBs to each other, instead of bridging the exit of one with the entry of another, as shown in Figure 3. For example, there are 6 IT-BBs out of 10 basic blocks left in the ITC-CFG, and the edges are reconnected so that the TIP packet flow generated by IPT can be directly searched on the ITC-CFG.

For the correctness of this constructed ITC-CFG, suppose at a randomly selected time the entry address of BB-x is recorded, then BB-x must be one of the IT-BBs, otherwise there is no indirect branch targeting at it, thus no TIP packet should be traced according to the tracing scheme of IPT. Now suppose the next immediate time another entry address of BB-y is recorded, then we should prove that there is one edge connecting from BB-x to BB-y in the ITC-CFG. By reduction, if there is no edge from BB-x to BB-y in the ITC-CFG, which means there is no edge of indirect branch in the path from BB-x to BB-y in the O-CFG. Then, BB-y should not be executed or traced. Therefore, for any two consecutive TIP packets traced by IPT, there must be an edge in ITC-CFG to represent this control flow, otherwise some anomalies have happened.

## 4.3 Fuzzing Training for Edges Labeling

While the ITC-CFG allows direct searching of IPT traces, there are still two security issues.

**Coarse-grained CFI.** There are still large false negatives during runtime flow checking due to coarse-grained CFI. As demonstrated in recent sophisticated attacks [24, 25, 26], coarse-grained CFI from conservative CFG generation can lead to superfluous legal targets for each branch, resulting in bypassed protection. Though IPT provides a terrific opportunity to provide better precision during runtime by combining online decoding and offline CFG, this usually comes with unacceptable overhead.

**Precision loss.** Worse even, the constructed ITC-CFG may weaken the security provided by the O-CFG due to CFG coarsening. We illustrate it by using one metric called Average Indirect targets Allowed (AIA) proposed by [22]:

$$AIA = \frac{1}{n}\sum_{i=1}^{n}|T_i|$$

where n is the number of indirect branch instructions, $T_i$ is the set of allowed targets for the $i$th indirect branch instruction. Intuitively, a smaller AIA represents more precise CFG. It was argued that AIA is one of the most proper metrics for measuring CFI strength [22], especially when the protected software involves large code base.

Considering the CFG reconstruction shown in Figure 4, in O-CFG, the AIA is 2, while in ITC-CFG, the AIA is 3, which means the ITC-CFG is less precise than the original one. Specifically in this example, the number of allowed targets for BB-2 and BB-3 changes from 2 to 3 after the reconstruction. This precision derogation is due to the absence of direct branches information that may fork the control flow. We observe that the only possible direct branches being able to fork the execution flow is the conditional branches, the taken or non-taken (TNT) branches from BB-1 to BB-2 or BB-3 in this specific example. ITC-CFG removes these edges to avoid path explosion. Fortunately this TNT information is traced by IPT, which can be used to strengthen the ITC-CFG during the training phase.

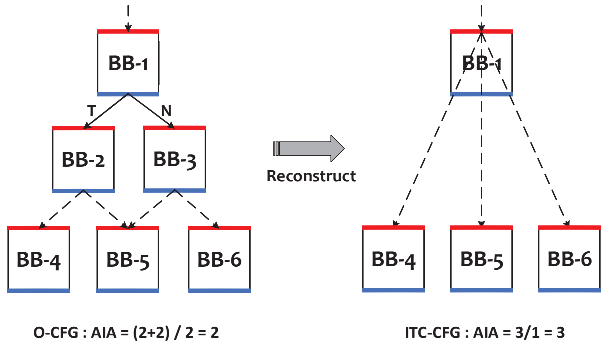


**Figure 4: An example of AIA derogation caused by ITC-CFG construction**

*Coverage-oriented fuzzing training.* FlowGuard uses a training phase to label selective edges with credits and additional TNT information. The decision of whether labeling an edge with high credit depends on whether this edge exists during the training phase. In consideration of achieving high coverage to minimize the requirement of heavy context-sensitive analysis, FlowGuard adopts a coverage-oriented fuzzing-like inputs generation [27] for dynamic training.

Fuzzing [28] is one of the most effective approaches that have been leveraged to identify vulnerabilities in real-world software. It involves in generating as many randomly selected data as possible, and uses them as inputs for the tested software to brute-force the possible paths and seek to trigger security issues. We choose fuzzing instead of other dynamic techniques (e.g., symbolic execution) because it is simple and practical to deploy. Briefly, the overall process of this training phase consists of three steps:

In step 1, the trained application runs in QEMU with the instrumentation logics implemented on top of it in user emulation mode. The instrumentation is responsible for discovering any new state transition caused by the input. In step 2, some initial test cases are provided and stored into one queue. The test cases in the queue are fetched one by one, and mutated to generate new test cases using a balanced and well-researched variety of traditional fuzzing strategies. These test cases are sequentially fed to the software running in QEMU. If any mutated test case results in a new state transition as observed by the QEMU, it will be added to the queue for subsequent mutation. This phase will generate a bunch of test cases that may triggers different path explorations. In step 3, FlowGuard collects the test cases generated in step 2, uses them as inputs to feed the trained application running on the real hardware, leverages IPT to trace its execution flow, and finally labels the edges in ITC-CFG with high credits based on these traced data.

The policy of labeling edges with credits can be varied for different needs. For instance, one can use more than two levels of credit values to label the edges, based on their number of occurrences during the training phase. For simplicity, we choose the binary-based labeling such that each edge is either with a high credit or a low one. The part (c) of Figure 3 presents one possible result of training the ITC-CFG. In this case, all edges except the one from BB-2 to BB-7 in the ITC-CFG are labeled with high credits, and the sequences of TNT packets obtained during the training phase are associated with corresponding edges. This TNT information is important, that with the direct forking information, FlowGuard can prevent attackers from derogating the AIA (e.g., with TNT information within the edge from BB-3 to BB-9 in Figure 3, the attacker cannot flow to BB-9 through BB-10, which is allowed using the ITC-CFG). Thus, the labeling resolves the precision loss problem.

It should be noted that the security of FlowGuard does not rely on the path coverage, though a higher coverage usually leads to better performance (§7).

## 5. RUNTIME PROTECTION

FlowGuard relies on a kernel module to protect the user-level processes. As illustrated in Figure 1, the kernel module is generally responsible for three parts: (i) configuring IPT to trace the execution flow of the target process, (ii) intercepting specific security-sensitive system calls to trigger subsequent operations, and (iii) checking the traced execution flow in fast and/or slow paths.

### 5.1 Execution Flow Tracing

IPT is configured by the kernel module to start and keep tracing control flow of the protected process. It briefly consists of two basic steps:

**Enabling controls for packet generation.** There are a variety of controls to determine whether the branch packets can be generated, these controls are enabled and configured by a collection of IA32_RTIT_* family model-specific registers (MSRs), among which the most important one is the IA32_RTIT_CTL MSR. It is the primary enable and control MSR for trace packet generation. FlowGuard's kernel module sets the *TraceEn* and *BranchEn* bits to enable CoFI-based packets tracing. It clears the *OS* bit and sets the *User* bit in order to exclusively trace user-level control flow. Meanwhile, it sets the *CR3Filter* bit to enable CR3 filtering mechanism, and configures the IA32_RTIT_CR3_MATCH MSR to the CR3 of the protected process. Finally it clears the *FabricEn* bit to directly send the trace output to the memory subsystem, and sets the *ToPA* bit to enable ToPA output scheme which will be explained latter. While other bits of the IA32_RTIT_CTL MSR are left as default.

**Configuring memory regions for trace output.** The trace output of IPT can be configured by one of two output schemes, A single contiguous region of physical address space, or a collection of variable-sized regions of physical memory which are linked together by tables of pointers. FlowGuard opts for the latter one, which is referred to as Table of Physical Addresses (ToPA), and stores the trace output into one ToPA with two regions.

### 5.2 System Call Interception

Selecting appropriate trigger points of flow checking is one of the important criteria of effective protection. It determines the timeliness and performance of the protection. One intuitive approach is triggering upon PMI and checking all of the packets in the interrupted region. This approach can ensure all of the execution flow of the protected process being checked, however it may introduce significant overhead. Instead, FlowGuard uses a similar approach as prior work [18, 20] by performing flow checking at specified security-sensitive endpoints. While such endpoints are configurable, FlowGuard pre-defines some default ones to provide reasonable security guarantee.

The pre-defined endpoints mainly consists of the specified security-sensitive syscalls, e.g., execve, mmap, mprotect.

For the sake of simplicity, we select the same sets of syscalls as PathArmor [20] since they represent the major threats that can be utilized by the attackers. FlowGuard chooses to intercept these security-sensitive syscalls by temporarily modifying the syscall table and installing one alternative syscall handler for each of them. Whenever such a syscall is invoked, the newly installed handler first checks whether it is called by the protected process through the information like CR3, process name, or process ID. If the answer is yes, the flow checking is issued, otherwise, it simply forwards the execution to the corresponding original syscall handler.

It is noted that the security-sensitive syscalls may be invoked from either shared libraries or the executable. Therefore, FlowGuard is responsible for checking the execution flow of both executable and shared libraries. If the flow checking failed, the kernel module sends SIGKILL signal to the process and reports the detection of control flow violation to the administrators or users.

## 5.3 Flow Checking

The general process of flow checking is as follows: the fast path matches the generated flow to ITC-CFG labeled with credits and TNT information, a security alarm is raised upon an edge mismatch. Otherwise, if any matched edge is with low credit or different TNT information, the slow path is launched.

**Fast path.** The fast path logic justifies the traced flow based on the pre-generated credit-labeled ITC-CFG. It starts from fast decoding the traced packet stored in the ToPA memory region. It is noted that at this stage, it only parses the packets based on the IPT formats and extracts out the TIP and TNT packets, without referring to the binaries with the instruction flow layer of abstraction. Meanwhile, with the help of packet stream boundary (PSB) packets, which are served as sync points for the decoder, this process can be done in parallel to further accelerate the decoding. On the other hand, it is not required to decode the whole ToPA buffer, FlowGuard only checks a specified number of TIP packets. And in consideration of guarding against attacks which hack in one module and invoke syscall in another, FlowGuard is ensured to decode TIP packets striding across more than one modules' memory regions, and at least one of them is within the executable.

With the runtime collected indirect target addresses, FlowGuard matches them to the credit-labeled ITC-CFG. Specifically, FlowGuard maintains an array of data structures for source nodes in the ITC-CFG, each source node has a *count* field indicating the number of its outgoing edges, and a pointer pointing to the start of the array of its target addresses. All of the arrays are sorted according to the addresses, so that binary search is used to reduce the time complexity. To further accelerate the matching process, FlowGuard preserves separate memory to store the source nodes and their targets connected by edges with high credits and TNT information, and use it as the cache for fast matching. During the fast path checking, for each address recorded in the TIP packet, FlowGuard first checks it by searching on the array of source nodes, then checks that its successor address can be found in the array of its target addresses. If any of these two checks is unsatisfied, the whole flow checking failed, and the positive result (which means attack detected) is returned back to the syscall handler.

Till the whole flow of indirect targets is confirmed to be compliance to the ITC-CFG, FlowGuard starts to evaluate the credibility of this fast path check. As explained in §4, the conservative CFG grants attackers superfluous legal targets to utilize, during the fuzzing training phase, each edge of the ITC-CFG is labeled with credit and TNT information. Hence at this stage, FlowGuard needs to assess the credibility of the fast path check. If the assessed credibility is higher than the specified threshold, FlowGuard reports the result as negative (no attack), otherwise, the flow checking is forwarded to the slow path engine.

**Slow path.** Besides the TIP and TNT packets used in the fast path logic, the inputs for the slow path engine also include the binaries being protected. The analysis performed in slow path is referred to the Intel's reference implementation of its IPT decoder library, which uses instruction flow layer of abstraction, parses the binaries instruction by instruction, and combines the traced packets for the entire decoding. Whenever the slow path checking is triggered, FlowGuard issues an upcall to the waiting user-level process to finish this task.

The policies enforced in the slow path can be very precise, as it can capture the whole execution flow and further perform a context-sensitive analysis. At a very basic level, FlowGuard is responsible for guaranteeing that the traced flow conforms to the O-CFG with the fine-grained forward-edge analysis [7]. In addition, for backward-edges, shadow stack is maintained using the instruction flow layer of abstraction, and compared with the traced packets to enforce single-target policy for the return branches.

# 6. REFLECTION ON HARDWARE

Hardware-assisted approaches are becoming big trends for enforcing CFI, to minimize performance impact, and retain transparency for software. For instance, Intel recently released a new specification called Control-flow Enforcement Technology (CET), which proposes a shadow stack exclusively used for control transfer operations, and defines a new *ENDBRANCH* instruction to mark legal targets for indirect branches. Though Intel CET seems like a killer for ROP attacks, its coarse-grained protection for forward edges makes it still problematic for other code reuse attacks, e.g., JOP [29], COOP [30], CFB [31], etc.

Our novel reuse of IPT resolves its main problem of slow decoding, and shows that it is a potentially complementary approach to CET for complete CFI protection. Nonetheless our experience of using IPT gives possible hardware suggestions for a better and more efficient CFI enabler: 1). *Hardware for fast decoding*, without sacrificing precise and fast tracing for IPT, we believe the only possible way to fundamentally resolve the relatively slow decoding is adopting a dedicated hardware for regular patterns of packets decoding, this hardware decoder can be very simple that it only requires a pattern-matching engine to process the buffer according to patterns with two 8-bits words, and route corresponding packets to specific memory location; 2). *More CFI friendly filtering mechanisms*, e.g., for applications with multiple processes, one CR3 related MSR is not enough, the configurations should be more flexible for filtering policies; 3). *Configurable hardware logics for simple CFI policies enforcement*, the hardware can arm with simple logics, e.g., identifying exact patterns of execution, etc., so that heuristic or simple CFG policies can be defined for non endpoints runtime traces to improve security; 4). *More triggering mechanisms*, besides buffer-filled PMI interrupt, more configurable event handlers can be added, e.g., when certain system events happen, etc., so that the time to check is more controllable and flexible. The first two are beneficial for performance (§7), while the latters create an opportunity for a full CFI checker.

# 7. EVALUATION

We have implemented a working prototype of FlowGuard on a commodity Intel Skylake machine with IPT support. All experiments are done on that machine with 8 Intel i7-6700K cores running at 4.0 GHz and 16 GB RAM. The underlying OS is Debian 8 with Linux kernel 4.3.0. The static CFG generation component is implemented as a plugin for the Dyninst [32] binary analysis framework. The fuzzing-based test cases discovery component is based on the open-source, coverage-assisted fuzzer AFL [27] with Qemu running unmodified binaries and discovering new state transitions. Since the AFL cannot directly handle network input, for the network based software (e.g., nginx), FlowGuard uses the *desock* module in the preeny project [33] to channel socket communication to the console. The runtime protection engine is implemented as a kernel module that can be enabled by a user-level software. The slow path logic is based on the Intel's reference implementation of the decoder library [34], which runs as a user-level process which can be triggered using an upcall from the kernel module. In total, FlowGuard adds 1829 lines of C code to the kernel module, the Dyninst plugin consists of 3736 lines of C++ code and 153 lines of script code, and the slow path logic has 856 lines of C code except for the decoder library.

## 7.1 Security Test and Analysis

### 7.1.1 Decisions on Parameters

There are two parameters that may affect the security level of the whole mechanism. The number of TIP packets (*pkt_count*) to be checked when the specified endpoints are triggered, and the ratio of edges with high credits (*cred_ratio*) during the checking. These two parameters mainly balance the tradeoff between performance and security.

*pkt_count* is important to defeat against attacks that leverage legal control flow to flush the illegal one and bypass security check. One intuitive example is library pollution that attackers invoke lib-calls instead of sys-calls to trigger security-sensitive endpoints (e.g., return-to-lib). Another history flushing attack is shown in [35], which combines the short flushing gadgets and long termination gadget to bypass heuristic checks [18, 19]. In FlowGuard, we choose 30 as the lower-bound of *pkt_count* such that at least 30 TIP packets are checked. Meanwhile, it is ensured to check packets striding across more than one modules, and at least one of them is within the executable. Based on this strategy, the control flow violation before library calls will not be omitted; thus attacks using return-to-lib can be prevented, On the latter case, the only possible way to flush the history of FlowGuard is to craft a valid path of more than 30 NOP-like gadgets conforming to the high credits labeled ITC-CFG, which is significantly more difficult than chaining arbitrary and CFG-agnostic gadgets. Hence, FlowGuard significantly raises the bar of attacks.

On the other hand, *cred_ratio* is the parameter to control the granularity of CFI enforcement. As the *cred_ratio* increases, the AIA of the checked indirect branches decreases roughly in a manner of the following formula:

$$AIA_{ratio} = ratio * AIA_{fine} + (1 - ratio) * AIA_{itc}$$

where $AIA_{fine}$ is the AIA of fine-granularity analysis in the slow path, $AIA_{itc}$ is the AIA of ITC-CFG without TNT information. We find in our evaluation that when *cred_ratio* exceeds 70%, the AIA of all benchmarks can be better than the O-CFG protection. To achieve the best security guarantee and prevent attacks combining a few low-credit edges with many high-credit edges to bypass slow path checking, we set *cred_ratio* to 1 so that any high-credit CFG edge violation leads to slow path. Thanks to the high-coverage training and relatively fixed boundaries of syscall endpoints in normal cases, the violation happens rarely, and the negative (no attack) results of slow path checking are cached for the subsequent fast path checking, thus makes the performance better and better.

### 7.1.2 Security Analysis

**AIA optimization.** Table 4 shows the statistics collected during the evolving steps of final CFG generation across different server applications that FlowGuard is deployed to protect. The second column lists the number of dependent libraries for each application, the third and fourth groups of columns present the number of basic blocks and edges for executables and libraries in the CFG which is used by the traditional CFI and the slow path of FlowGuard. The AIA statistics are shown in the last three groups of columns. Meanwhile, it also presents the number of basic blocks ($|V|$) and edges ($|E|$) in the ITC-CFG. The average AIA is reduced from 72 to 20.

**Real attacks prevention.** To show the abilities of FlowGuard in stopping real-world control flow hijacking attacks, we artificially implant an obvious vulnerability in nginx code and conduct one traditional ROP attack and another SROP [36] attack on it. These two attacks have different attack routes, while both end up with writing arbitrary data into a specified file. With FlowGuard protection, the control flow violation is detected during *write* syscall for the traditional ROP and *sigreturn* syscall for the SROP.

**Fine-grained protection.** Some attacks [37, 35] use techniques (e.g., history flushing, evasion attacks, etc.) to bypass heuristic protections [18, 19]. As showing above, the default setting of FlowGuard significantly raises the bar for them. Other more sophisticated attacks [24, 25] unearth attack friendly gadgets to bypass coarse-grained binary-based CFI mechanisms [3, 4]. FlowGuard is not vulnerable to them, as these attacks in FlowGuard is equivalent to the attackers crafting more than 30 gadgets chained by high-credit edges, or bypassing the context-sensitive analysis in the slow path, which is hardly possible. We can also make the fast path more context-sensitive by matching the high-credit paths, each of which consisting of multiple consecutive high-credit edges. This can significantly strengthen the security of fast path, however, it may introduce larger number of slow path checking; we leave this as our future work.

**Endpoints bypassing.** As prior work [18, 20], FlowGuard relies on an assumption that attacks will finally trigger some specified security-sensitive endpoints to achieve their goals. Hence, it is vulnerable to the endpoint-pruning attack, where an attacker leverages alternative approaches

**Table 4: Statistics about CFG nodes ($|V|$) and edges ($|E|$), as well as CFI metrics like AIA**

| Application | Library | Basic Block # | | Edge # | | O-CFG | ITC-CFG | | | FlowGuard |
|---|---|---|---|---|---|---|---|---|---|---|
| | # | EXEC | LIB | EXEC | LIB | AIA | $\|V\|$ | $\|E\|$ | AIA (w/ tnt) | AIA |
| **nginx** | 8 | 29660 | 124749 | 297333 | 768745 | 76.77 | 34183 | 6164493 | 192.86 (76.77) | 26.65 |
| **vsftpd** | 3 | 6832 | 68679 | 10456 | 606575 | 110.60 | 13495 | 4135272 | 351.79 (110.6) | 25.98 |
| **openssh** | 21 | 14166 | 165942 | 23729 | 796010 | 44.54 | 41893 | 5439232 | 173.69 (44.54) | 11.31 |
| **exim** | 18 | 9996 | 237702 | 14864 | 1174948 | 54.81 | 52010 | 6368066 | 128.07 (54.81) | 16.92 |

without triggering the predefined endpoints. Based on lessons from prior solution [20], FlowGuard provides an interface for users to specify their own endpoints in case our default settings cannot satisfy their requirements. In the worst case, FlowGuard can rely on periodic performance monitoring interrupts (PMIs) generated when the trace buffer is full as endpoints. Meanwhile, since we assume the kernel and hardware, which are isolated from user-level apps, are trusted, FlowGuard can hardly be hacked under this model to give erroneous reports.

**False positives and false negatives.** FlowGuard introduces no false positive as it uses the conservatively generated CFG. For false negatives, with the slow path checking, FlowGuard can largely trim the AIA of backward edges with the shadow stack policy. For attacks conducted in control jujutsu [26] and control-flow bending [31] that modify function pointers to violate forward edge CFI, FlowGuard can stop them using the CFG generation mechanism like that in TypeArmor [7], while share the same false negatives due to the limitation of static analysis. It is worth noting that even the state-of-the-art compiler-based approaches with fine-grained static analysis cannot completely stop those attacks abusing functions with valid signatures but invalid intention. These categories of data-only attacks may be defended by other approaches like control-data isolation [12, 38], which are orthogonal to FlowGuard.

## 7.2 Performance Evaluation

We focus our performance evaluation on four aspects: 1). Macro benchmarks to give an overall picture about the runtime performance of real-world software running on FlowGuard; 2). Micro benchmarks to shed light on the factors introducing this overhead; 3). A discussion about the fuzzing-like dynamic training; 4). The breakdown of overhead to show the benefits of our hardware suggestions.

### *7.2.1 Macro Benchmarks*

We evaluate three categories of applications: servers, Linux utilities, and CPU intensive benchmarks like SPECCPU 2006. The protections are all binary-based, and these applications are configured with their default settings. We set the lower bound of *pkt_count* to 30, and ensure to check packets in executable and libraries. The *cred_ratio* is strictly set to 100%, that any violation of high-credit edge triggers slow path checking. The experiments for all of the software are conducted about 20 times, and the geometric means of the incurred overhead are reported. With the above configuration, we find that the slow path happens rarely (less than 1%) thanks to the fuzzing training.

**Server software.** Considering FlowGuard represents an approach to monitoring, server daemons are the most suitable scenarios for it to protect, as they keep running on the system and are the most pervasive targets for attackers. We choose 4 servers including web server nginx-1.6.3, FTP server vsftpd-3.0.3, SSH server OpenSSH-6.7p1 and email server exim4. For nginx, we rely on the Apache benchmarks (ab) to simulate 10 concurrent clients constantly sending 20K requests, each of them asks for one 20B-sized file. For FTP server, we use pyftpbench to download 10MB-sized files. For OpenSSH and exim, we write homegrown scripts to constantly login the server and execute common commands, or send emails respectively. Part (a) of Figure 5 shows their normalized overhead compared to the ones without FlowGuard protection. The geometric mean is about 4%.

**Linux utilities.** We also evaluate the category of applications that simply execute once and instantly exit, like most of the Linux utilities. We select four representative ones including *tar*, *dd*, *make*, *scp* to show the overhead caused by FlowGuard mechanism. We write a program that fork a child process to execute these utilities. Before the *execv* syscall, the child process first calls ptrace with the flag PTRACE_TRACEME on, so that the parent process can get the CR3 of the child process before it runs. With CR3 filtering, FlowGuard can selectively trace corresponding process and do the protection. Part (b) of Figure 5 shows the normalized overhead of these Linux utilities. It implies that the overhead for these utilities is negligible (geometric mean is 0.82), especially for *dd*, which has small number of branch instructions and seldomly invokes system calls.

**SPECCPU 2006.** We also measure the time of running all the C programs in the SPECCPU 2006 benchmarks, the results are shown in part (c) of Figure 5. It implies that the geometric mean is 3.79%, and most benchmarks introduce less than 10% overhead, except 1 anomaly: *h264ref*. We found that the core logic of *h264ref* is a loop with many indirect calls, and it generated much more traces (90%) than other benchmarks at runtime.

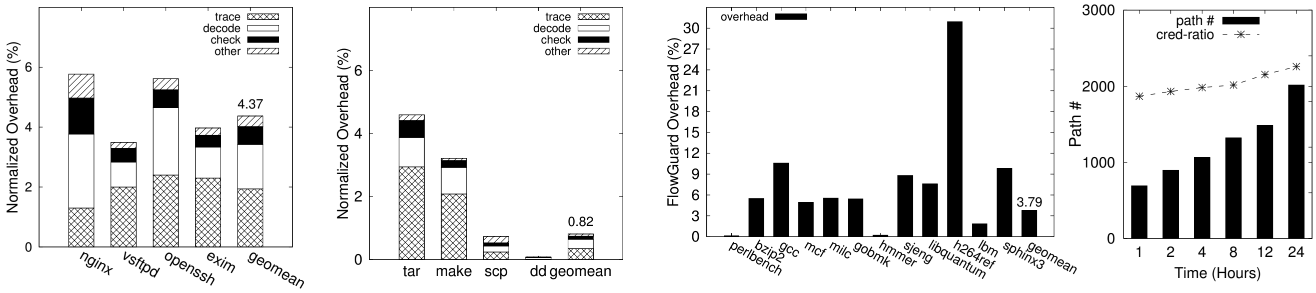
### *7.2.2 Micro Benchmarks*

**Checking time.** §2 gives an overall intuition about the slow decoding compared to fast tracing. With the ITC-CFG, fast decoding is enabled, and the checking takes little time by virtue of the efficient search algorithm. Therefore the fast path of flow checking can be very rapid. However, since the security still relies on the slow path, thus we also draw a picture about how much time it may take to have a context-sensitive analysis during the slow path for ranges of memory containing 100 TIP packets. The results show that the context-sensitive analysis takes about 0.23ms, which is averagely 60 times slower than the fast path.

**Table 5: Memory usage and CFG generation time**

| | **nginx** | **vsftpd** | **openssh** | **exim** |
|---|---|---|---|---|
| **Memory usage (MB)** | 54.51 | 35.84 | 43.96 | 52.67 |
| **CFG generation time (s)** | 443 | 362 | 462 | 490 |

**Memory usage.** Additional kernel memory are used for control flow tracing and ITC-CFG maintaining. Our default configuration of ToPA buffer consumes about 16KB-sized buffer for each core running the application, and the memory consumption of ITC-CFG depends on the complexity of



(a) Server (b) Linux utility (c) SPECCPU 2006 (d) Fuzzing training

**Figure 5: Performance evaluation: (a) to (c) show the slowdown of using FlowGuard to protect three categories of applications; (d) presents the performance benefits brought by fuzzing training**

the protected software. Table 5 shows the size of ITC-CFG for server applications, which can be optimized using more compact data structures.

**CFG generation time.** We also measure the time of ITC-CFG generation for different applications. The results are also shown in Table 5. The average CFG generation time is about seven minutes, which is acceptable since it is a one-time effort. Meanwhile, we found that more than 90% of time spends on analyzing libraries (e.g., libc), which are shared among all applications, thus it is reasonable to cache the resulted CFGs of them and reuse them accordingly to optimize the CFG generation time.

### *7.2.3 Fuzzing Training*

Though the security of FlowGuard does not rely on the training process, it plays an important role in improving performance. A thorough fuzzing test is time consuming, nonetheless since the training results are for performance consideration, we can relax the expected path coverage. On the other hand, there may be time ranging from weeks to months before software deployment, thus the fuzz test is still one of the most potential approaches to generating enough inputs for high coverage of control flow paths. Due to space limit, we simply provide an intuition about the benefits brought by the fuzzing training phase.

We use nginx protection as the example. During the fuzzing training of nginx, it may discover ascending number of inputs that can result in different paths as time goes on. As illustrated in part (d) of Figure 5, we feed these inputs generated during different training time to the running nginx server, and combine the traced packets with ITC-CFG to label edges with high credits. And then run the *ab* benchmarks multiple times to see the average ratio of edges with high credits during the checking phase. It shows that the number of newly probed paths is growing, and the ratio of highly credible edges can be more than 97%. We believe that the result can be further improved with longer training time and larger number of inputs.

### *7.2.4 Benefits from Minor Hardware Extensions*

We provided several suggestions on improving hardware for a better CFI enabler (§6). To show its benefits, we break down the overhead of macro benchmarks into 4 phases: tracing, decoding, checking and others, as shown in Figure 5. We can see that in most situations, decoding contributes to a large fraction of the overhead (more than 30% for server applications). Thus, a dedicated hardware decoder can significantly reduce such overhead. The tracing overhead depends on the applications: single-process applications (e.g., nginx) outperforms multi-processes ones due to the single CR3 filtering mechanism. Therefore, more CFI-friendly filtering mechanisms (e.g., using configurable numbers to filter CR3s) are valuable for efficiency. Moreover, we can see that the overall tracing overhead is small. Hence, with hardware logics to enforce simple CFI policies as well as more configurable triggering mechanisms, FlowGuard can provide complete CFI checking at low overhead.

# 8. RELATED WORK

## 8.1 Software-based Control-flow Protection

To defend against a variant of code-reuse attacks [1, 29, 39], researchers mainly focus on two techniques: randomization and enforcement. Randomization-based approaches try to conceal the runtime execution flow from attackers. Software diversity [40], address space randomization [41] are all possible solutions. Enforcement approaches instead try to preserve pre-defined policies at runtime. Control flow integrity (CFI) [2], and its variants, when properly implemented, can be used to prevent attackers from executing control flow edges outside a statically generated CFG.

**Compiler-based CFI:** With the availability of source code, a compiler can generate more fine-grained CFG with the help of detailed side information. For instance, both Modular CFI [8] and IFCC [9] proposed to use the type information as call signatures for the forward-edge CFG generation. CCFI [10] cryptographically computed and verified the MAC for every control-flow object so that it could hardly be abused. $\pi$CFI [11] differentiated the static CFG (SCFG) and the enforced CFG (ECFG), and lazily added edges to the ECFG for the concrete input. CDI [12] even eliminated all of the *ret* and indirect *call/jmp* instructions, and replaced them with a sled of conditional branch or direct *jmp* pairs. KCoFI [42] enforced the kernel CFI by using secure virtual architecture (SVA) based approach, and the fine-grained kernel CFI [22] approach leveraged LLVM to protect the CFI of FreeBSD and MINIX kernels.

**Binary instrumentation based CFI:** The original CFI proposal [2] and its variants resorted to the binary based approaches. CCFIR [3] statically rewrote the PE binary file embedded with a *springboard* section, and enforced a 3-ID CFI. binCFI [4] generated the CFG for an ELF executable even without the debug and relocation information. Lock-Down [43] adopted dynamic binary translation technique to enable fine-grained binary based CFI with shadow stack enforcement. RCAP Stack [5] was another binary-based approach that systematically resolved the challenges of shadow stack, like automatically analyzing all possible non-standard returns. Opaque CFI [6] combined the fine-grained code-randomization and coarse-grained CFI checking, and made it a bound checking problem. TypeArmor [7] implemented the binary-level use-def and liveness analysis to significantly reduce the number of possible targets for indirect call sites.

While compiler- or binary-based approaches are effective in enforcing CFI, they generally have two issues. First, they usually have to make a tradeoff between efficiency and precision as instrumenting and checking every indirect branch would incur significant runtime overhead. Second, there may be compatibility issues for the protected software. While binary-based approaches have better compatibility compared to compiler-based approaches by requiring no compilation, binary rewriting still needs to change the code layout and signatures, which may be incompatible with some existing security mechanisms (e.g., Window 7 system library protection, remote attestation, etc.), and usually have difficulties in dealing with shared libraries.

## 8.2 Hardware-assisted CFI Enforcement

Hardware-assisted approaches address the compatibility issue by using transparent runtime monitoring. There are generally two directions: designing new hardware features and reusing existing ones.

**Designing new hardware:** SCRAP [14] defined a set of formal grammars to represent the code reuse attacks, changed the microarchitecture of the superscalar processors, and implemented the detection logic at the commit stage of the pipeline for the signature-based protection. ControlFreak [15] pre-computed the signature for each basic block using its instructions as well as its possible successors, then designed a hardware watchdog to calculate the runtime signature and detect violation. Similarly, approaches like BR [13] and HCFI [16] also proposed new hardwares for both forward and backward edges checking, but they still required rewriting the binaries.

**Reusing existing hardware:** Vasudevan et al. [44] and Yuan et al. [45] were first to present case studies of using PMU to detect code reuse attacks on AMD and Intel processors accordingly. CFIMon [17] was a first effort to leverage BTS for transparent CFI enforcement. kBouncer [18] and ROPecker [19] both leveraged LBR to record the runtime flow, and used heuristic approaches to detect control flow violation. Similar with kBouncer, PathArmor [20] triggered CFI checking in some specific sensitive points, and traced the runtime data using LBR. Furthermore, it proposed a context-sensitive verification, and the binaries were instrumented to enforce the context sensitive CFI invariants on the monitored paths. FlowGuard continues this line of research and is the first to make a novel reuse of IPT designated for offline analysis to runtime CFI checking, which embraces precision, efficiency and transparency.

## 8.3 Retrofitted Hardware Features

Aside from using hardware for CFI, there are also a large body of research in the architecture community leveraging commodity hardware features for software security and reliability, including applying performance counters to improve malware detection [46], using control speculation for information flow tracking [47, 48], adopting hardware transactional memory for virtual machine introspection [49], as well as reusing cache allocation technology for side channel protection [50]. FlowGuard adds to the literature by showing that IPT designed for debugging and profiling can also be retrofitted for control flow protection.

# 9. CONCLUSION

This paper described FlowGuard, an efficient and transparent approach that effectively enforces CFI by a novel reuse of Intel Processor Trace to collect and check runtime control flow. FlowGuard addresses the challenges such as slow decoding and incomplete control traces by constructing an IPT-compatible control flow graph such that IPT traces can be directly searched over the CFG. FlowGuard embraces efficiency and precision through separating fast and slow path checking. Evaluation confirmed the security and efficiency of FlowGuard. And hardware suggestions are proposed by experience for a better CFI enabler.

## Acknowledgment

We thank the anonymous reviewers for their constructive comments. This work is supported in part by National Key Research and Development Program of China (No. 2016YFB1000104), China National Natural Science Foundation (No. 61572314, 61525204), a research grant from Huawei Technologies, Inc., National Top-notch Youth Talents Program of China, Zhangjiang Hi-Tech program (No. 201501-YP-B108-012), a grant from Microsoft, and Singapore NRF (CREATE E2S2).

# 10. REFERENCES

[1] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security*, 2012.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[3] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *S&P*, 2013.

[4] M. Zhang and R. Sekar, "Control flow integrity for cots binaries.," in *USENIX Security*, 2013.

[5] R. Qiao, M. Zhang, and R. Sekar, "A principled approach for rop defense," in *ACSAC*, 2015.

[6] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity.," in *NDSS*, 2015.

[7] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *S&P*, 2016.

[8] B. Niu and G. Tan, "Modular control-flow integrity," in *PLDI*, 2014.

[9] 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 *USENIX Security*, 2014.

[10] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: Cryptographically enforced control flow integrity," in *CCS*, 2015.

[11] B. Niu and G. Tan, "Per-input control-flow integrity," in *CCS*, 2015.

[12] W. Arthur, B. Mehne, R. Das, and T. Austin, "Getting in control of your control flow with control-data isolation," in *CGO*, 2015.

[13] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev, "Branch regulation: Low-overhead protection from code reuse attacks," in *ISCA*, 2012.

[14] M. Kayaalp, T. Schmitt, J. Nomani, D. Ponomarev, and N. Abu-Ghazaleh, "Scrap: Architecture for signature-based protection from code reuse attacks," in *HPCA*, 2013.

[15] S. Arnautov and C. Fetzer, "Controlfreak: Signature chaining to counter control flow attacks," in *RDS*, 2015.

[16] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "Hcfi: Hardware-enforced control-flow integrity," in *CODASPY*, 2016.

[17] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *DSN*, 2012.

[18] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent rop exploit mitigation using indirect branch tracing," in *USENIX Security*, 2013.

[19] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG, *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[20] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical context-sensitive cfi," in *CCS*, 2015.

[21] P. Yuan, Q. Zeng, and X. Ding, "Hardware-assisted fine-grained code-reuse attack detection," in *RAID*, 2015.

[22] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *EuroS&P*, 2016.

[23] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *VEE*, 2014.

[24] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *S&P*, 2014.

[25] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security*, 2014.

[26] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *CCS*, 2015.

[27] "American fuzzy lop." http://lcamtuf.coredump.cx/afl/.

[28] "Fuzz testing." https://en.wikipedia.org/wiki/Fuzz_testing.

[29] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *CCS*, 2010.

[30] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *S&P*, 2015.

[31] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *USENIX Security*, 2015.

[32] "Dyninst api." http://www.dyninst.org/dyninst.

[33] "Preeny project." https://github.com/zardus/preeny.

[34] "Intel pt decoder library." https://github.com/01org/processor-trace.

[35] N. Carlini and D. Wagner, "Rop is still dangerous: Breaking modern defenses," in *USENIX Security*, 2014.

[36] E. Bosman and H. Bos, "Framing signals: A return to portable shellcode," in *S&P*, 2014.

[37] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis, "Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard," in *USENIX Security*, 2014.

[38] W. Arthur, S. Madeka, R. Das, and T. Austin, "Locking down insecure indirection with hardware-based control-data isolation," in *MICRO*, 2015.

[39] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *S&P*, 2013.

[40] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "Sok: Automated software diversity," in *S&P*, 2014.

[41] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *S&P*, 2012.

[42] J. Criswell, N. Dautenhahn, and V. Adve, "Kcofi: Complete control-flow integrity for commodity operating system kernels," in *S&P*, 2014.

[43] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *DIMVA*, 2015.

[44] A. Vasudevan, N. Qu, and A. Perrig, "Xtrec: Secure real-time execution trace recording on commodity platforms," in *HICSS*, 2011.

[45] L. Yuan, W. Xing, H. Chen, and B. Zang, "Security breaches as pmu deviation: detecting and identifying security attacks using performance counters," in *APSys*, 2011.

[46] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *ISCA*, 2013.

[47] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From speculation to security: Practical and efficient information flow tracking using speculative hardware," in *ISCA*, 2008.

[48] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew, "Control flow obfuscation with information flow tracking," in *MICRO*, 2009.

[49] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *HPCA*, 2014.

[50] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.