CFInsight: A Comprehensive Metric for CFI Policies

Tommaso Frassetto, Patrick Jauernig, David Koisser, and Ahmad-Reza Sadeghi
Technical University of Darmstadt
tommaso.frassetto@trust.tu-darmstadt.de, patrick.jauernig@trust.tu-darmstadt.de,
david.koisser@trust.tu-darmstadt.de, ahmad.sadeghi@trust.tu-darmstadt.de

***Abstract*—Software vulnerabilities are one of the major threats to computer security and have caused substantial damage over the past decades. Consequently, numerous techniques have been proposed to mitigate the risk of exploitation of vulnerable programs. One of the most relevant defense mechanisms is Control-Flow Integrity (CFI): multiple variants have been introduced and extensively discussed in academia as well as deployed in the industry. However, it is hard to compare the security guarantees of these implementations as existing metrics (such as AIR) do not consider the different usefulness to the attacker of different basic blocks, which are the fundamental components that constitute the code of any application.**

This paper introduces BlockInsulation and CFGInsulation, novel metrics designed to overcome this limitation by modeling the usefulness of basic blocks for an attacker trying to traverse the program's control-flow graph. Moreover, we propose a new CFI policy generator, named NumCFI, which is orthogonal to existing policy generators and prevents the attacker from taking shortcuts from vulnerable code to a system call instruction. We evaluate NumCFI, as well as a number of other CFI policy generators, using BlockInsulation, CFGInsulation, and existing metrics. Lastly, we describe L+TCFI, our implementation that combines NumCFI and an existing label-based policy, with a performance overhead of just 1.27%.

I. Introduction

Since their invention, computer systems have become responsible for increasingly complex tasks. As a result, computer programs have become increasingly complex as well. Due to this complexity and the presence of legacy code, most modern software projects are plagued by several security vulnerabilities. A number of approaches have been suggested to find these security vulnerabilities, including software testing, fuzzing, and formal methods. However, proving that software is free from vulnerabilities is only feasible for small programs. Thus, researchers have proposed a number of strategies that aim to mitigate vulnerabilities in running programs.

These run-time mitigations are usually based on one of two principles: either preventing the adversary from learning some information that is necessary to perform an attack, or inserting additional checks in the program to make an attack impossible. An example of the former, which is currently deployed in most operating systems, is Address Space Layout Randomization (ASLR) [37], which randomizes the memory layout of a program and hides it from the adversary. An example of the latter, also widely adopted in the software industry, is Control-Flow Integrity (CFI) [2], which ensures that a part of a program can only transfer control to a different part if this transfer was intended by the programmer.

This paper is focused on CFI, whose main idea is to add checks to all *indirect control flow transfer instructions*, i.e., machine instructions that transfer control to a dynamically-computed address. CFI allows each of these instructions to only transfer control to a subset of targets, according to a *control-flow graph* (CFG). Since its introduction by Abadi et al. in 2005 [2], CFI has been the focus of a large corpus of research works. There are many variants with different granularity and based on either hardware or software. Given the number of different approaches, it is important to be able to compare them in terms of both performance overhead and effectiveness against memory-corruption attacks. While there is a widely accepted metric to compare their performance overhead, i.e., the run time overhead of the execution of a standard benchmark, there is no single metric that is widely recognized by the community to compare the security protection provided by different approaches. The best-known metric is Average Indirect-target Reduction (AIR) [55], which is defined as the average reduction of allowed targets across every indirect control flow transfer instruction (the higher the better). However, AIR is not a good metric to compare different policies [51], as most CFI papers that rely on AIR report similar values greater than 99% [6]. Yet, even implementations with very high AIR are still vulnerable [13], [21]: in other words, missing even less than 1 percentage point in AIR is enough to perform attacks. Hence, AIR is not a good instrument to distinguish between and compare CFI approaches. After AIR, a number of other metrics, including AIA [18], QuantitativeSecurity [6], and CTR [34], have been proposed, as we discuss in Section IX. However, they all share a major shortcoming: they do not consider the usefulness of different basic blocks to construct an attack, instead only considering their quantity. Hence, there is a need for a new approach that leverages not only local information regarding single basic blocks, but also their position and connectivity in the full CFG. In this paper, we introduce CFInsight, a new CFI evaluation methodology and framework that achieves that.

CFInsight. A very important building block for a run-time attack is the possibility to invoke a system call with controlled parameters. This is useful, e.g., to start a new

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.23165
www.ndss-symposium.org

malicious process or to change the memory protection settings. Performing a system call is also the only way to exfiltrate files or further compromise the machine, and is used in real-world exploits [10]. Hence, our CFI evaluation framework CFInsight is based on the assumption that the attacker found a vulnerable basic block and wants to perform a controlled system call. Usually, many blocks containing system call instructions exist in the program: some are unreachable from the vulnerable block, while others can be reached using a number of different paths through the CFG. We construct a novel metric, CFGInsulation, which considers the number and length of these paths from a basic block to a system call, and quantifies how easy it is for an attacker to build an exploit. We model a number of CFI policy generators: a theoretical perfect one, generators based on matching function types or number of arguments, and a generator that allows transfers to any valid function. We show that we can apply CFInsight to the generated policies and compute their CFGInsulation to compare them, showing how CFGInsulation allows to distinguish between policies with very similar AIR.

NumCFI. Moreover, we leverage the data generated by CFInsight to define and evaluate a new CFI policy generator, dubbed NumCFI. NumCFI assigns each basic block a *tag*, which is the length of the shortest path from the block to a system call instruction, and it enforces the property that a block with tag t can only call blocks with tag $\geq t-1$. In other words, any attack that starts in a basic block and requires a system call needs to go through as many basic blocks as the shortest legal path from the starting node to a system call instruction; the attacker cannot "take shortcuts," but has to go through the specified number of basic blocks instead. We show that NumCFI has a comparable or better CFGInsulation than a type-based policy generator, and that combining them leads to significant improvements over either one. We demonstrate that this combination is practical with a prototype implementation, which we call L+TCFI, and show that it has a very low run-time overhead (1.27% on benchmarks of the SPEC CPU2017 suite).

Contributions. In this paper we make the following contributions:

- We describe, design, and implement a novel CFI evaluation framework, CFInsight, based on measuring expressive properties of the CFGs of real programs, instead of simply counting reachable basic blocks. We plan to open source CFInsight so it can be useful to the community.
- We apply CFInsight to better compare the relative security characteristics of multiple state-of-the-art CFI policy generators, using our new CFI metric, CFGInsulation. We compare CFGInsulation with four existing CFI metrics.
- We leverage the knowledge generated by CFInsight to define a new CFI policy generator, NumCFI, and show that it significantly improves the security guarantees of other widely used CFI policy generators.
- We design a generic CFI implementation, L+TCFI, which can be used to enforce a combination of NumCFI with a classic label-based CFI, and we show that it has a very low run-time overhead (1.27% on benchmarks of the SPEC CPU2017 suite).

The rest of the paper is organized as follows: Section II introduces a number of topics that are required to understand the rest of the paper; Section III describes our approach and our metrics; Section IV describes our analyzer, which computes these metrics; Section V applies the analyzer to a number of existing CFI policy generators and discusses the resulting metrics; Section VI describes NumCFI; Section VII adds NumCFI to our analysis; Section VIII describes and evaluates our CFI implementation L+TCFI; Section IX discusses related works and Section X concludes the paper.

II. Background

This section introduces control-flow graphs, run-time attacks and control-flow integrity.

A. Control-Flow Graphs

A *control-flow graph* (CFG) is a directed graph representing the control flow of a program. It consists of nodes, which represent the basic blocks in the program, and edges representing legal transitions from one basic block to another. A basic block is a contiguous sequence of instructions that does not have any internal branch: branch instructions can only be the last instruction of a basic block, and instructions targeted by a branch can only be the first instruction of a basic block. CFGs (and the basic blocks they contain) are a popular abstraction used to analyze computer programs. However, generating CFGs is not trivial. They can be generated either statically or dynamically. Static generation leverages compiler passes (or an equivalent for binaries) to decide based on the observed instruction whether a new basic block is formed or if there is a transition from one basic block to another. These transitions are caused by branches. Determining all the possible destinations of a branch is hard in practice, as a common construct used in programs are indirect jumps. Indirect jumps get their target from a register, hence, this target cannot be resolved statically in the general case, but only approximated. While modern techniques like symbolic execution can help to solve this problem, the generated CFG is still an approximation in practice. In contrast, dynamic approaches monitor the behavior of the program at run time. Hardware features like Intel PT or debugging functionality allow to extract the actual targets of indirect jumps. Nonetheless, this approach also cannot fully solve the problem, as dynamic approaches can only monitor the control flow for taken branches. Since the information observed depends on the program's input, a large set of inputs might be needed to generate a close-to-perfect CFG, which can be achieved through the use of automated testing (fuzzing) or a test suite.

B. Run-time Attacks

Run-time attacks have been a persistent threat for modern computing platforms for more than three decades. These attacks exploit vulnerabilities in software to achieve arbitrary code execution. Memory corruption attacks have a long-standing history. The very first attacks exploited buffer overflows in memory to inject new code into the data section and execute it later, which effectively added a new node to the CFG. However, these attacks were still primitive, and easy to mitigate. By introducing a write-xor-execute (W$\oplus$X) policy, attackers could no longer inject executable data, stopping

code injection attacks altogether. This mitigation has been deployed broadly, and is most prominently known as Data Execution Prevention (DEP). Although this mitigation raised the bar, attackers found new strategies to bypass these defenses using more sophisticated attacks that do not add nodes to the CFG, but add new paths between existing nodes, hence called code-reuse attacks. Code-reuse attacks can be categorized into full-function reuse attacks (e.g., return-to-libc [49]) and return-oriented programming (ROP) [45], [9]. ROP uses small sequences of instructions to form gadgets, which can be used as building blocks to mount a more complex attack or achieve Turing-complete computation. While simple defenses like Address-Space Layout Randomization (ASLR) were deployed in real systems, ROP remains challenging to prevent, especially since code-reuse attacks can be combined with information leakage (e.g., the JIT-ROP attack [48]). These enhanced attacks spawned advanced defenses both in hardware and software. Prominent examples of defenses are Control-Flow Integrity (CFI) [2], [12], [19], [3], Code-Pointer Integrity (CPI) [28], or sophisticated randomization techniques [11], [48]. Some defenses are already deployed in products, e.g., Microsoft's Control-Flow Guard (CFGuard), Clang's CFI [30] which is used in Google Chrome, Intel's Control-flow Enforcement Technology (CET) [25] and ARM's Pointer Authentication (PAC) [41]. Due to the progressive adoption of some of these defenses, a more advanced type of attack has been introduced in the academic world. In a Data-oriented Programming (DOP) attack [24], [26], non-control data is manipulated to reuse valid paths under CFI to achieve Turing-complete computation. While schemes like Data-Flow Integrity [8], [50] solve this theoretically, they come at a significant performance and hardware overhead. As a result, solving this problem remains challenging in practice.

C. Control-Flow Integrity

Control-Flow Integrity relies on the fact that most functions in a program only call a very limited subset of the other functions. Given a CFG of a program, a CFI implementation instruments the code such that only these transfers are allowed, and any attempt to deviate from the CFG is detected. Only function calls that compute their target at run time, i.e., *indirect* function calls, are potentially vulnerable and need to be instrumented; direct function calls have a hard-coded target that cannot be changed at run time. A *context-insensitive* CFI policy specifies, for every indirect function call site, which other functions can legitimately be called from that site. A *context-sensitive* CFI policy considers not only the identity of the call site and the callee, but also other criteria, like the value of a variable or the top of the call stack, in order to decide whether an indirect call is legal. Moreover, the call to a function (forward edge) is not the only one that needs to be protected, the return (backward edge) needs it too [7], e.g., in the form of a shadow stack [5], a data structure keeping secure copies of return addresses.

Deploying CFI poses a number of challenges. One such challenge is *overapproximation* of the allowed control flow transfers, which is mostly introduced in the name of performance. A precise run-time instrumentation needs to check if a specific target is allowed for the specific caller, which can be relatively slow. Thus, most *CFI policy generators* introduce overapproximations in order to streamline the checks and make

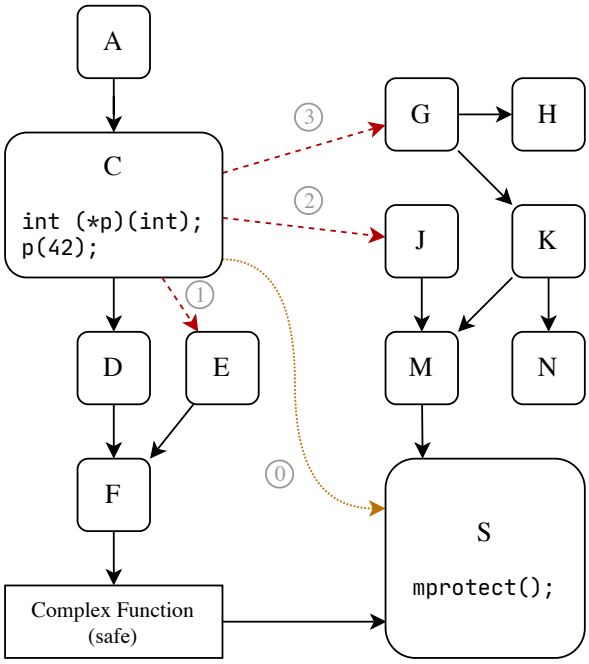

Fig. 1. A CFG for a simple program with a corruptible pointer.

them faster. A common method to simplify CFI checks is to assign a single numeric *label* to every indirect caller and callee and check that the label of the caller matches that of the callee. This effectively splits the nodes into *equivalence classes*, one for each label, and allows the run-time check to be a simple (and fast) integer comparison. As an example, a CFI implementation can label functions according to the return type and type of the parameters [38]. However, it introduces overapproximation because unrelated nodes need to have the same label in order for the scheme to work, i.e., it is not possible to distinguish between targets in the same equivalence class. This overapproximation has been shown to be sufficient to attack protected programs [13], [17]. In practice, most of the deployed CFI implementations use either type-based policies [30] or simple heuristics, like checking whether the callee address is the beginning of a function [32].

A recent trend is to add hardware support for CFI [25], [29], improving performance and providing better integrity protection for the CFI mechanism itself.

III. CFInsight

We begin our description of CFInsight by looking at the sample CFG in Figure 1. In this program, basic block C contains a function pointer `p` which can be corrupted by the adversary. The adversary can leverage this vulnerability to launch a code-reuse attack, and wants to reach block S, which contains an invocation to the system call `mprotect`. If the adversary can reach this block and control the parameters to the system call, it is trivial to disable memory protection and then perform a classic code injection attack. If the program is not protected by CFI, the adversary can simply redirect the control flow to block S using edge ⓪ and then perform the rest of the attack. However, if the program is protected by CFI, only CFG edges allowed by the *CFI policy* can be followed;

as a result, the attacker is limited to these allowed edges. In the case of a perfect CFI policy (which only allows black solid edges from Figure 1), the only path to S goes through a complex function, which we assume to be implemented with secure programming techniques: as a result, it is likely that the attacker can only invoke the system call using safe parameters and, hence, cannot launch the attack.

Real-world CFI implementations, however, are not perfect, as we mention in Section II-C, and often use *overapproximated policies*, i.e., they allow edges that should be forbidden. The impact of this overapproximation on the security of the program depends on which illegal edge is incorrectly allowed. As an example, if edge ① is allowed, the adversary gains no advantage, since the only path to S still goes through the safe function. If one of the edges ② and ③ is included, the attacker can instead jump to J or G and follow more nodes until the control flow reaches S.

From the perspective of existing metrics, like AIR [55], AIA [18], or CTR [34], a policy that includes edge ① is equivalent to one that includes edge ②, or one that includes ③, as they allow the same number of edges starting from C. However, they are not equivalent in practice. If only ① is allowed, the attacker has no advantage over a perfect CFI, since there is no additional path to S. If only ② is allowed, the attacker has a substantial advantage: the attacker can jump to J and then follow the flow to M and S. Lastly, if only ③ is allowed, the attacker still has an advantage, but smaller than the previous case: The attacker must jump to G and try to follow the chain all the way to S. In order for this to be successful, the adversary needs to ensure that the desired branch to K is taken in G, instead of the branch to H (similarly in K, with the branch to M). Which branch is taken depends on a condition, which could be out of the attacker's control.

The purpose of CFInsight is to compare CFI policies considering their graph structure and connectivity. We focus on context-insensitive CFI policies, since most CFI policies deployed in practice fall in this category [2], [30], [25]; however, our approach can also be applied to context-sensitive CFI policies, as we discuss in Section IV-D. For each indirect function call, we measure the quantity and length of possible paths that lead to a system call instruction.

In the following, we first describe our threat model, then we explain how our metric is defined and how we compute it.

A. Threat Model and Assumptions

With CFInsight we aim to model how most run-time attacks start in the real world. Thus, we make the following assumptions about the victim program and the capabilities of the adversary:

- **A0** The adversary wants to attack a vulnerable program. More concretely, the goal of the adversary is to invoke a system call with controlled parameters, e.g., to start a new malicious process or to change the memory protection settings. Performing a system call is the only way to exfiltrate files or further compromise the machine, and is used in real-world exploits [10].
- **A1** The adversary has access to a vulnerability in the program that allows arbitrary read operations to readable memory and arbitrary write operations to writable memory.
- **A2** The adversary can leverage the arbitrary write primitive to corrupt the memory such that an indirect call will be redirected to an unintended target. As an example, this can be done in the presence of a buffer overflow vulnerability. The adversary can corrupt pointers and hijack the control flow multiple times. If a CFI policy is in place, all of the hijacked calls need to comply with the CFI policy.
- **A3** We assume W⊕X (see Section II-B) to be in place and working, i.e., the adversary cannot overwrite the application code or inject new code.
- **A4** We assume that a shadow stack implementation [5], or equivalent, is deployed on the victim, hence, the attacker cannot target the function returns. Protecting function returns is a very different problem than protecting function calls, and this paper is focused on the latter.
- **A5** We assume the adversary to be able to bypass any randomization-based defense in use, e.g., ASLR; thus, we do not consider them in our model.
- **A6** In principle, our approach can be applied to any operating system. However, a number of low-level details differ between them. Hence, we focus on Linux, in line with related work [6], [7], [16], [17].
- **A7** We expect the victim program to be built using the current best practices for Linux software, e.g., full RELRO [47], which makes the Procedure Linkage Table (PLT) read-only. Thus, the attacker cannot overwrite PLT entries.

B. Our Observations: Single-Node Metric

In CFInsight, we aim to define quantifiable properties of a graph that measure how easily an attacker can build a successful attack. We begin by considering a given node in the CFG that calls a vulnerable code pointer, and a specific system call site the adversary needs to reach. To reach this goal, the adversary needs to follow a number of CFG edges, which need to be legal according to the current CFI policy. Let us consider one such path. Each basic block on this path contains machine instructions, which perform a number of operations, and ends with a (possibly conditional) branch instruction. As a result, traversing each basic block poses two challenges for the adversary. First, if the branch is conditional, the adversary needs to make sure the value of the branch condition is true if the branch is to be taken, or false otherwise. Second, the code in the basic block often writes data to memory or to a register; this might overwrite some data the adversary prepared for the attack, e.g., a parameter of the system call or the operand of a branch condition. Our first observation follows:

- **O1** The more basic blocks an attack needs to traverse, the harder the attack is.

However, there usually are multiple paths between a node and a system call site. The attacker only needs one path that supports an attack, and hence:

- **O2** The more paths are available for an attack, the higher the likelihood that at least one of them is viable for the attack.

We leverage these observations to build our metric to measure the effectiveness of CFI policies. As a first approximation, our metric is the ratio between the length of paths to any system call, and the number of these paths. Our metric is directly proportional to the length of the paths, due to

Observation O1, and inversely proportional to their number, due to Observation O2; higher values of the metric indicate that the attack is harder. However, this approximation needs to be refined to be applicable in practice. First, there are multiple paths of varying lengths starting in a given node and ending in some system call site; since it is not computationally feasible to examine all paths in a complex CFG, we consider instead the lower bound of their lengths, i.e., the shortest path from the node to any system call. Second, it is also infeasible to know the exact number of paths from a given node to a system call; a useful approximation is to consider the number of *linearly independent* paths, which can be computed efficiently[1].

The result is our metric that quantifies the difficulty of an attack starting in a basic block b and reaching any system call site. We call this metric BLOCKINSULATION(b) and we define it as:

$$\frac{\text{length of shortest path } b \to \mathit{syscall}}{\text{№ linearly independent paths } b \to \mathit{syscall}}$$

If there is no path between b and any system call site, we define BLOCKINSULATION(b) $= \infty$, since any attack is impossible in this case.

C. Whole-Program Metric

In general, considering the whole distribution of values of BLOCKINSULATION of all basic blocks gives the most complete picture. However, it can also be useful to define a single numeric metric to summarize the distribution of the BLOCKINSULATION. Simply averaging the values is impractical, since the metric we defined can assume values from ≈ 0 to ∞. We instead decide to take the median value of the distribution, which is often a finite value. If more than half of the values are ∞ and the median is infinite, we instead take the maximum finite value. A greater value of CFGINSULATION indicates a program that is harder for an attacker to exploit.

CFINSIGHT leverages this new metric to compare the security guarantees of different CFI policy generators.

IV. CFINSIGHT ANALYZER

In the previous section we introduced a metric to measure the security guarantees of a CFI policy. In Figure 2 we show the overall design of the analyzer we designed to compute this metric. Each component is described in detail below.

A. CFG Generation

In order to compute our metric for a program we need its CFG. As we explain in Section II-A, there are two main approaches to generate a CFG: either through dynamic or static analysis. Both approaches have different advantages and limitations. While we consider the problem of enhancing CFG generation techniques to be orthogonal to the scope of this paper, the quality of our analysis does depend on the quality of the CFG it uses. Hence, we leverage both approaches: we trace a number of executions of the program on a set of inputs and we also perform a static analysis of the program.

[1]The number of linearly independent paths in a graph, also known as McCabe's cyclomatic complexity [31], can be easily computed as $|E| - |N| + 2$, where $|E|$ is the number of edges and $|N|$ the number of nodes of the graph.

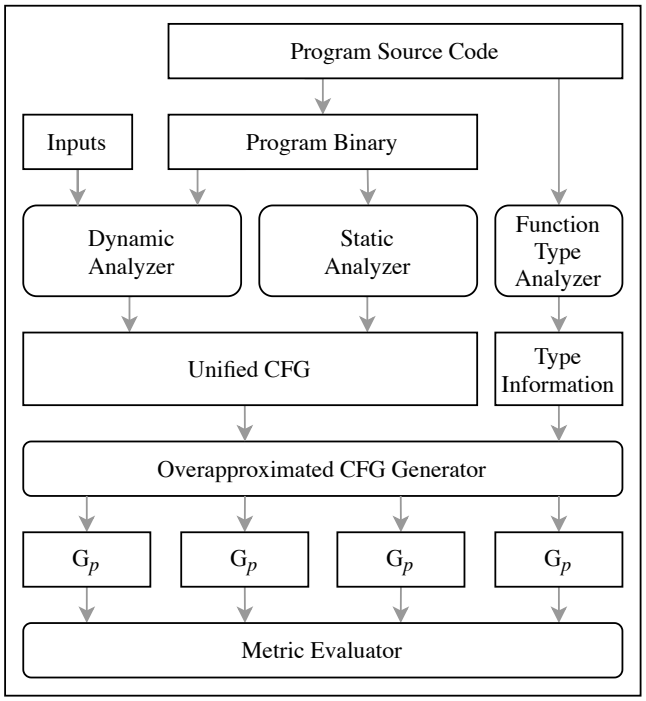

Fig. 2. Architecture of the CFINSIGHT analyzer.

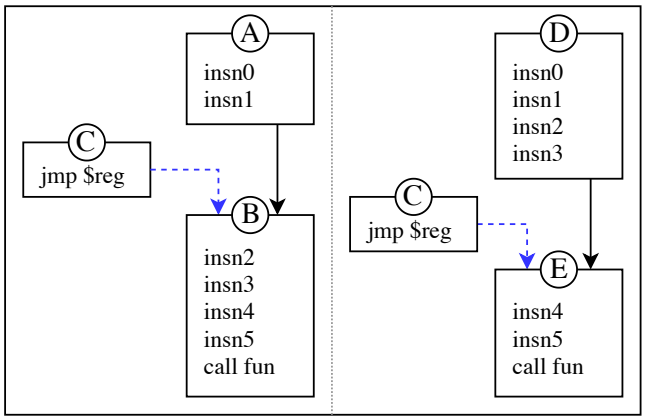

Fig. 3. Depending on the target of the jump instruction in block C (dashed blue edge), two different pairs of basic blocks are generated: A+B or D+E.

We then combine the CFGs generated by these tools into a single unified CFG for the program under analysis. Combining multiple CFGs is not trivial, since different tools can split a binary into different sets of basic blocks. As an example, Figure 3 shows two different CFGs generated for the same program. Assume that the jump instruction in block C can legally jump to either *insn2* or *insn4*, and that different tools generate CFGs which only contain one of these edges each. As a result of the different edges, in the CFG on the left the code is split resulting in blocks A and B, while in the CFG on the right the split results in blocks D and E.

While it is easy to determine whether an instruction terminates a basic block (any branch instruction does), determining where a basic block starts is more complex. By definition, a basic block is a sequence of instructions that will be always executed one after another. Since the tools often generate dif-

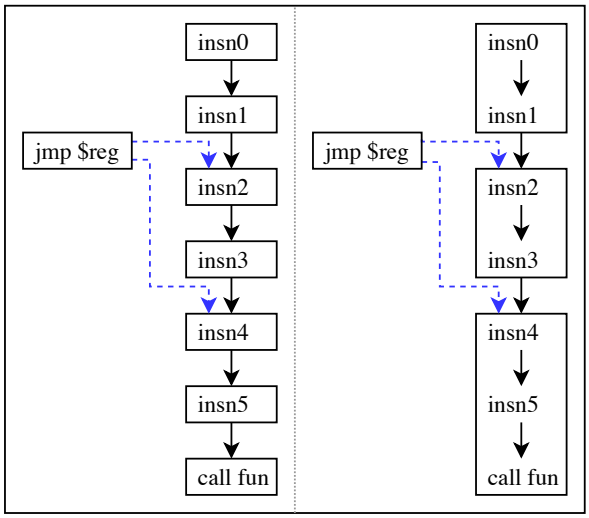

Fig. 4. Merging the graphs of Figure 3. Left: decomposition of the basic blocks in single instructions. Right: recomposed basic blocks.

ferent sets of edges, and since the way instructions are grouped into blocks depends on the known edges, different tools (and concrete executions on different inputs) often produce different basic block sets for the same instructions.

We address the problem by splitting all basic blocks into instructions, building a list of all edges on the instruction level (see left side of Figure 4). We then find the new basic block boundaries based on all known edges for the program (right side of Figure 4), thus generating a *unified CFG*.

In order to make the next steps more straightforward, we add an additional node to the CFG, called *target*, which symbolizes the attacker's goal. In our attacker model (Assumption A0), the adversary's goal is to reach a system call, so we add an edge from any block containing a system call to *target*.

B. Overapproximating the CFG

Once we obtain a unified CFG for a program, we need to model the effect of the various CFI polices on the program. As we mentioned in Section II-C, CFI policy generators introduce overapproximations in order to simplify their construction or improve their performance. For each generated CFI policy p, we model its effect on a program's CFG and generate a list of edges that are not present in the CFG but are allowed by the policy. We then add these edges to the original CFG and generate G^p, i.e., the graph of all allowed control flow transfers under policy p.

We model a number of CFI policy generators used in widely deployed mitigation schemes, as well as the lack of CFI:

SoFCFI a CFI policy generator that allows all functions to be the target of any indirect function call, similarly to what can be done with Intel CET [25];

TypeCFI a CFI policy generator that only allows indirect function calls if the type signature of the callee matches the type expected at the caller side, like RAP [38];

NumArgCFI a simplified variant of TypeCFI, which only checks that the number of arguments of the callee matches the number provided at the caller side. This policy generator is an idealized version of TypeArmor [53];

NoCFI the absence of CFI can be modeled by a policy that allows each indirect call site to call any other basic block[2] in the binary.

In order to compute SoFCFI, we only need the addresses of the functions, which we can extract from the symbols in the binary. For TypeCFI and NumArgCFI, we additionally need type information. We extract the types of functions from the debug symbols, while we leverage a custom compiler pass to extract the expected function type at the indirect call sites.

C. Computing Our Metric

Once we generate the overapproximated graph G^p for a policy p, we can use it to compute BLOCKINSULATION (see Section III-B). The first step is to compute, for every indirect call site b, the subgraph containing all paths to *target* (see Section IV-A). Since we are only interested in nodes that have a path to *target*, i.e., its *ancestors*, we focus only on them for efficiency reasons. We compute the set of ancestors by performing a depth-first search, starting in *target*, in a copy of the graph where all edges are reversed. Afterwards, for each indirect call site b, we perform a depth-first search, only considering the ancestor nodes we found before. The nodes found in this search compose the subgraph we wanted to build. After this subgraph is known, our metric can be computed in a straightforward manner.

The naive representation of the G^p graphs in memory is challenging, since naively representing some CFI policies requires a large number of edges. As an example, in NoCFI, any indirect call site can transfer control to any basic block (see Section IV-B), which produces $|I| \times |N|$ edges for a CFG with $|I|$ indirect call sites and $|N|$ basic blocks. In order to produce a more tractable representation of this graph, we introduce a synthetic node, called *any*. We then create an edge $i \rightarrow any$ for every $i \in I$, and an edge $any \rightarrow b$ for every $b \in N$. This leads to a graph with the same connectivity, with only $|I| + |N|$ edges, which is a substantially lower number. However, if not accounted for, this optimization would lead to different result for our metric. Hence, while computing our metric in the presence of synthetic nodes, we take this difference into account, in order to compute the value the metric would have[3] in the naive version of the graph.

D. Extensions and Discussion

The framework is built in a modular way and it can easily be extended. For example, an analyst can add an additional CFG generator, mark further blocks as the attacker's target, or model a new CFI policy generator. In particular, CFINSIGHT can also be extended to consider a context-sensitive CFI policy (see Section II-C). Representing a context-sensitive CFI policy

[2] In a variable-length instruction set like x86, in the absence of CFI, the attacker can also jump in between instructions; we do not consider this in our model, since the adversary does not need this possibility to very easily reach a system call (without CFI).

[3] A synthetic node with i incoming and j outgoing edges represents the fact that each of these i predecessors can reach any of the j successors. As a result, these $i + j$ edges in the graph actually represent $i \times j$ edges. We then adjust the edge count by adding $i \times j - (i + j)$ and the node count by subtracting one.

requires having multiple nodes in the CFG for the same basic block, one for each context. Our methods can then be applied to this extended CFG.

In the presence of a multi-threaded application, CFInsight considers each thread separately. As a result, it does not directly model an attack where two or more threads are exploited at the same time and collaborate to perform a system call. However, in this case, we focus on the thread that performs the system call. Its control flow needs to reach a system call site, starting from a legitimate block; as a result, our analysis still applies.

V. CFInsight: Implementation and Results

In this Section we describe our CFInsight implementation and we present its results.

A. Analyzer: Implementation Notes

We implemented our prototype of the CFInsight analyzer as a number of Python scripts, totaling approximately 4 000 lines of code.

We compile all binaries with the Clang compiler (version 11.0.1), which we extend with a custom IR pass to produce a list of expected function types at indirect call sites (see Section IV-B). For our static analysis we use the angr framework [46], which can generate the CFG of a program using static analysis and symbolic execution. For our dynamic analysis we choose CFGgrind [43], a Valgrind-based tool that dynamically records control flow transitions as they happen during program execution. In angr, we generate both a *fast* and an *emulated* CFG, while in CFGgrind we generate a separate CFG for every input file; all of them are then combined into a unified CFG, like we explain in Section IV-A. We extract the function types for TypeCFI from DWARF debug symbols, which encode the types of the functions (together with other information) in the binary itself. We decode this data in our DWARF parser, which is based on pyelftools [4]. We also retrieve the detached debug symbols for the system libraries from the Debian package manager, then we decode them with our DWARF parser.

The most processing-intensive part of the analysis pipeline is the evaluation of the metrics on an overapproximated graph. Since this evaluation is mostly independent for each basic block, we split the work between multiple threads (up to the number of CPU cores available), while the main thread is responsible for collecting the results.

B. Results

In this section, we report the results of our CFInsight framework on state-of-the-art CFI policy generators. In Section VII, we compare these metrics with our novel CFI policy generator NumCFI as well.

Experimental setup. In order to test CFInsight and to compare different CFI policy generators, we compute our metrics for a number of benchmarks. From SPEC CPU2017, the most recent version of a widely used benchmarking suite, we select all benchmarks written in C, C++, or a mix of the two, in their *speed* variant. For each benchmark, we statically generate its CFG with angr and we use CFGgrind to dynamically trace the execution; we use all input files that compose the three SPEC workloads (*test*, *train* and *refspeed*). Moreover, we prepare our own benchmark of the web server nginx [1]: as inputs we use the official nginx test suite, which is composed of 388 different configurations. We ran all tests on a machine running Debian Sid, last updated in April 2021, with a 32-core Intel Xeon Silver 4110 processor and 128 GB of RAM.

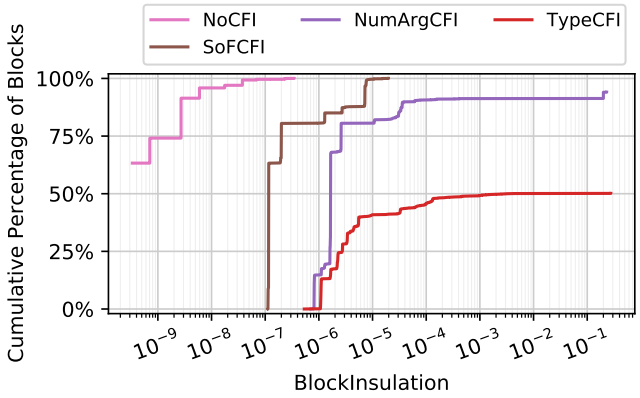

Fig. 5. CDF of the distribution of BlockInsulation of basic blocks containing indirect calls, under different CFI policies.

BlockInsulation. We evaluated BlockInsulation for every indirect call site in our benchmarks under four CFI policy generators: NoCFI, SoFCFI, NumArgCFI, and TypeCFI (in increasing order of strictness; we define them in Section IV-B). In Figure 5, we visualize the distribution of these metrics by plotting the Cumulative Distribution Function (CDF) of the BlockInsulation over all indirect call sites in all of our benchmarks[4]. Each point with coordinates $(x, y\%)$ in these curves means that $y\%$ of the blocks have a BlockInsulation $\leq x$. Since greater values of BlockInsulation indicate that attacks are harder to perform, a curve that is lower and to the right of the figure indicates a more secure CFI policy generator. As expected, the least secure policy generator is NoCFI, followed by SoFCFI, NumArgCFI, and lastly TypeCFI.

CFGInsulation. While we stress that a CDF of BlockInsulation (like Figure 5) is the most complete way to compare different policies, it is often useful to summarize the results into a simpler numeric metric, which we introduce in Section III-C. We define CFGInsulation as the median BlockInsulation value for the indirect call sites of a program (or the maximum finite value if the median is infinity). Figure 6 shows the values of this metric for all benchmarks we consider. For all benchmarks, the CFGInsulation values are in the expected order (TypeCFI, NumArgCFI, SoFCFI, NoCFI). TypeCFI improves the CFGInsulation by 3 to 7 orders of magnitude compared to NoCFI, by 1 to 5 orders of magnitude compared to SoFCFI, and up to 3 orders of magnitude compared to NumArgCFI.

[4]The graphs only show data about the main binaries. Our model also considers the dynamic libraries, but we only use the information that can be extracted from their binary and debug symbols, since compiling libraries such as libc is a very complex process.

TABLE I. COMPARISON OF CFI POLICY GENERATORS USING EXISTING METRICS AND CFGINSULATION.

CFI policy generator	mean(fAIR)*	mean(fAIA)†	sum(iCTR)†	geomean(QS)*	total CFGINSULATION*
NoCFI	0.00000%	6023518.4	324855240247	0.00000020	$3.348766 \cdot 10^{-10}$
SoFCFI	99.94011%	4504.8	406122077	0.00040833	$1.154559 \cdot 10^{-07}$
NumArgCFI	99.99284%	584.6	59811668	0.00233956	$1.644561 \cdot 10^{-06}$
TypeCFI	99.99720%	228.1	29684972	0.05289177	$3.712601 \cdot 10^{-03}$

* *Higher is better.* † *Lower is better.*

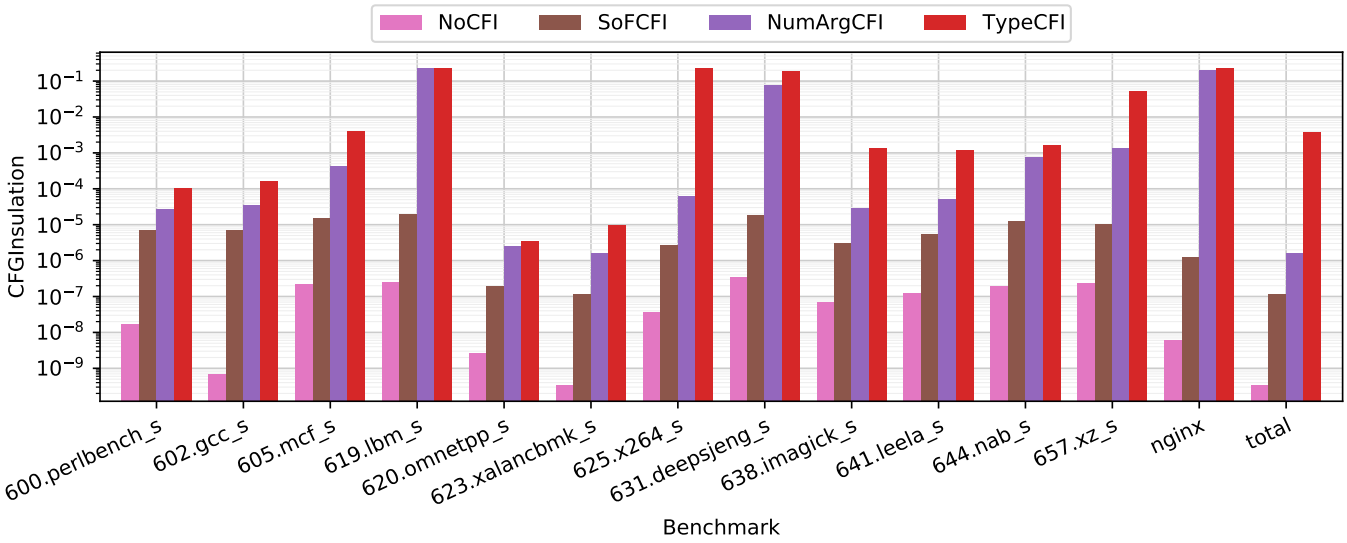

Fig. 6. CFGINSULATION for each benchmark we consider.

Other metrics. In order to validate our findings and compare the results with other existing metrics, we compute a number of CFI metrics over the same CFGs. Specifically, we compute:

- fAIR, the forward-edge variant of AIR [55]: the average reduction in the number of allowed target for every indirect function call (higher is better);
- a forward-edge variant of AIA [18] which we dub fAIA: the average number of allowed targets for every indirect function call (lower is better);
- iCTR [34]: the sum of the number of allowed targets for every indirect function call (lower is better);
- QS (QuantitativeSecurity [6]), the product of the number of equivalence classes and the inverse of the size of the largest class (higher is better).

We compute all of these metrics for each of our benchmarks. Since the metrics are defined in different ways, we use different mathematical functions to summarize them. fAIR and fAIA are defined as arithmetic means; hence, we report the arithmetic mean of the individual results from the benchmarks. iCTR is defined as a count, so we report the sum of the single results; QS is a ratio, so we report its geometric mean. These values, along with CFGINSULATION, are shown in Table I. The metrics confirm that TypeCFI offers more security than NumCFI, which is better than SoFCFI and NumArgCFI. In Section VII we extend this analysis with our novel CFI policy generators NumCFI.

VI. NUMCFI

We mentioned earlier that existing CFI metrics, like AIR, consider basic blocks with the same label equivalent to each other, leading to the division of basic blocks in equivalence classes. Our answer to this shortcoming is to propose CFINSIGHT, which analyzes a CFI-protected program in terms of how easy it is for an attacker to reach a system call instruction. The core insight is that a node that is close to a system call instruction (e.g., node J in Figure 1) is more useful to an attacker than farther nodes (e.g., node G). The same insight can be applied to produce a novel CFI policy generator as well, which led us to the definition of NumCFI.

The idea of NumCFI is to assign each basic block a *tag*, which is the number of basic blocks on the shortest path from the block to a system call instruction. As an example, Figure 7 shows the tags that NumCFI assigns to the program in Figure 1, assuming the path within Complex Function to be 10 blocks long. At run time, we enforce the property that the tag can decrease by at most 1 for every call, i.e., a block b can call a block c only if their tags t_b, t_c satisfy this property:

$$t_c \geq t_b - 1 \tag{1}$$

This prevents the attacker from "taking shortcuts" when planning an attack, i.e., if the attacker wants to hijack the control flow in a block with tag t, the attack chain needs to go through at least t blocks before it reaches a system call

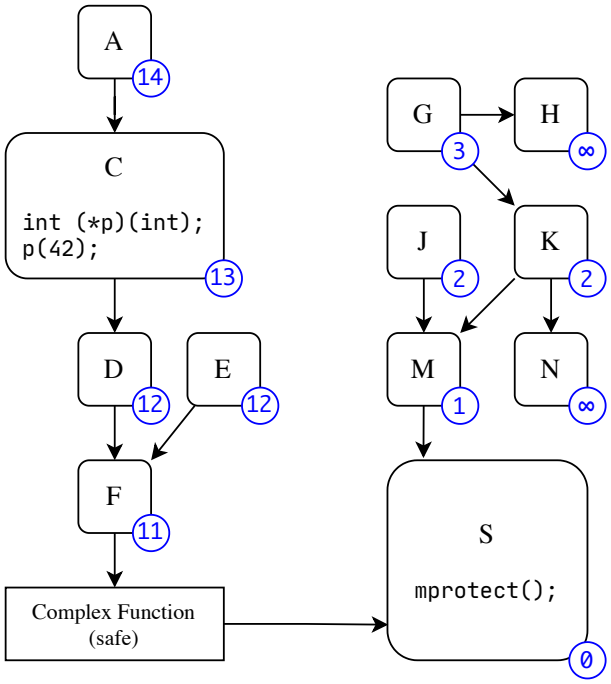

Fig. 7. The same program of Figure 1, with the NumCFI tags for each block in the blue circles. In this example we assume the path through the Complex Function to be 10 blocks long.

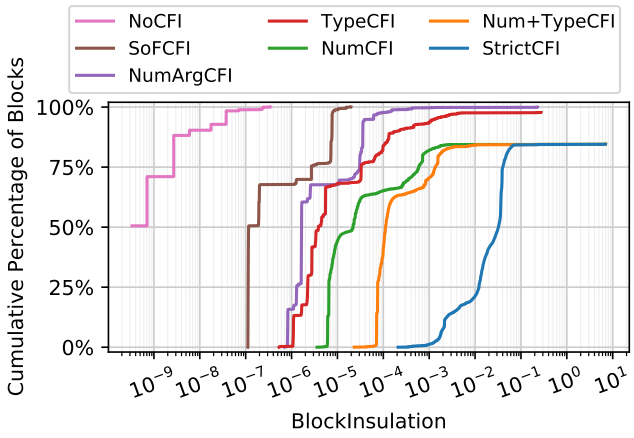

Fig. 8. CDF of the distribution of BLOCKINSULATION of basic blocks containing indirect calls, under different CFI policies.

instruction. Every basic block with no path to a system call instruction receives a special tag ∞, which only allows it to transfer control to other blocks with the tag ∞. NumCFI can also be combined with an orthogonal CFI policy generator based on labels, e.g., TypeCFI. We name this combined policy generator Num+TypeCFI.

An interesting consequence of deploying NumCFI is that it allows security analysts to focus their attention on a small number of basic blocks with a low tag value, since they are the blocks that the adversary might use to mount an attack. Blocks with higher tag values can receive less attention, since they would require long attack chains; blocks with tag ∞ can be outright ignored, since they cannot reach system call instructions at all.

Below, we discuss how NumCFI prevents the nginx attack described in [17]. We then describe how NumCFI and Num+TypeCFI compare with other policy generators in Section VII. Lastly, we write an implementation of Num+TypeCFI and we evaluate its performance overhead in Section VIII.

Case Study: Nginx. Farkhani et al. [17] construct an attack on nginx, protected by type-based CFI implementation RAP [38] The attack leverages a collision between functions with the same type. Specifically, the code of function `ngx_worker_process_exit` contains an indirect function call to a function that takes no arguments. The attack leverages this fact to hijack the control flow and call a different function, `ngx_master_process_cycle`, which also takes no arguments; from there, the control flow eventually reaches an invocation of the system call `execve`. We applied NumCFI to nginx 1.16.1 compiled for x86_64 Linux and we verified that the tags of these two basic blocks differ by more than 1, i.e., Equation (1) is not satisfied and this control flow is disallowed. Hence, NumCFI protects nginx from the attack described in [17].

A downside of all CFI implementations is that, if a valid edge is missing in its input CFG, attempting to follow that edge in the program will lead to a false-positive CFI violation. For example, the authors of RAP [38], encountered missing edges caused by incorrect type signatures in the source code of programs. The issue of missing edges also presents itself in NumCFI. The root cause of this issue is the well-known difficulty of generating a precise CFG of a program. While improving CFG generation techniques is outside of the scope of this paper, there are ways to mitigate this issue. First, one can combine multiple CFG generation approaches. In our prototype implementation of CFINSIGHT, we combine angr and CFGgrind as representatives of static and dynamic analysis tools. However, any other more advanced static analysis tool can be used as well. The dynamic analysis can be improved by increasing the number and quality of the input files. Good software engineering practices recommend the presence of a test suite which is as thorough as possible. As a result, tracing the execution of this comprehensive test suite ensures that any tested functionality is covered in the generated CFG. The coverage generated by the test suite can further be improved with other techniques such as fuzzing. Finally, if the core functionality of the program is covered by tests, any false positive that is still present is by definition only incurred in rare circumstances. These false positives can then be manually addressed just like any other rare bug.

VII. CFINSIGHT: COMPARISON OF NUMCFI

In this section, we extend the results of Section V by considering NumCFI as well. We use the same experimental setup and benchmarks we describe in Section V. Due to an imperfect CFG generation, for some indirect call sites we do not know of any legal outgoing edge: in order to ensure a fair comparison between policy generators, we omit these nodes from the following analysis.

In addition to the existing CFI policy generators (TypeCFI, NumArgCFI, SoFCFI, NoCFI), we analyze NumCFI, our new

TABLE II. COMPARISON OF NUMCFI WITH OTHER CFI POLICY GENERATORS USING EXISTING CFI METRICS AND CFGINSULATION.

CFI policy generator	mean(fAIR)*	mean(fAIA)†	sum(iCTR)†	geomean(QS)*	total CFGINSULATION*
NoCFI	0.00000% (7)	6023518.4 (7)	75153429826 (7)	0.00000020 (7)	$3.348766 \cdot 10^{-10}$ (7)
SoFCFI	99.94011% (6)	4504.8 (6)	83776616 (6)	0.00040833 (6)	$1.161153 \cdot 10^{-07}$ (6)
NumArgCFI	99.98757% (4)	1039.4 (4)	22424457 (4)	0.00197575 (4)	$1.647973 \cdot 10^{-06}$ (5)
TypeCFI	99.99498% (3)	390.6 (3)	8077681 (3)	0.04507330 (3)	$3.856191 \cdot 10^{-06}$ (4)
NumCFI	99.95866% (5)	3055.5 (5)	60548479 (5)	0.00045113 (5)	$2.184315 \cdot 10^{-05}$ (3)
Num+TypeCFI	99.99665% (2)	260.1 (2)	5615941 (2)	0.05484967 (2)	$1.109534 \cdot 10^{-04}$ (2)
StrictCFI	99.99996% (1)	2.0 (1)	10974 (1)	1.65245297 (1)	$3.225806 \cdot 10^{-02}$ (1)

* *Higher is better.* † *Lower is better.* *(The number in parentheses is the rank of each CFI policy generator according to each metric.)*

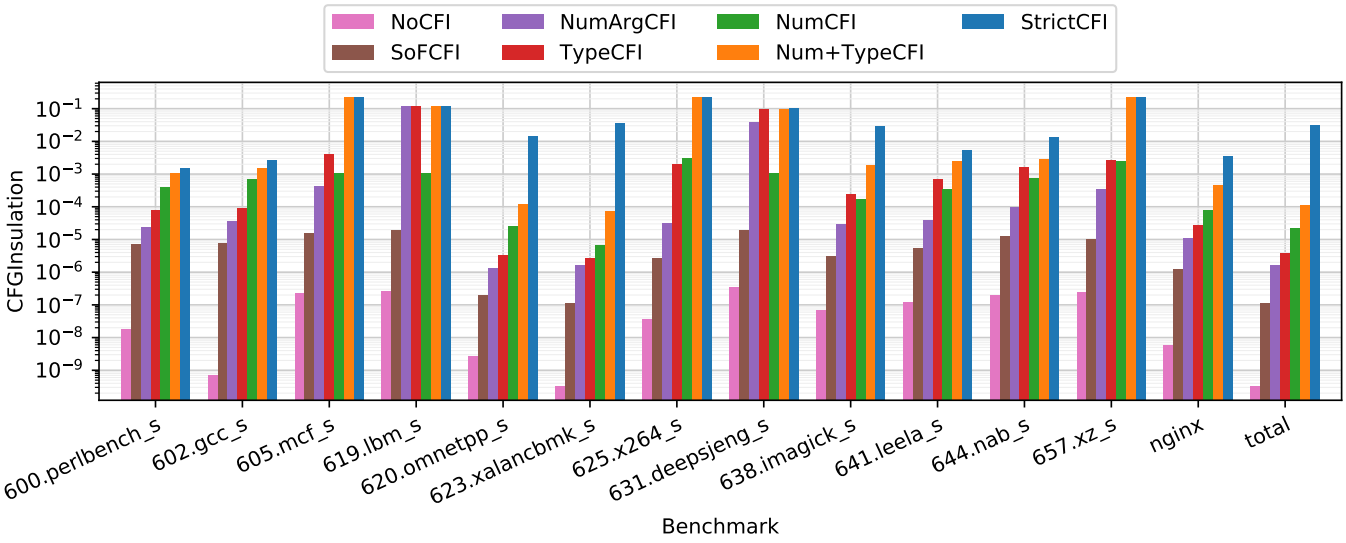

Fig. 9. CFGINSULATION for each benchmark we consider.

CFI policy generator, as well as Num+TypeCFI, which is the combination of NumCFI and TypeCFI. Finally, we define StrictCFI as a CFI policy that only allows legal edges, without any overapproximation. StrictCFI serves as an indication of the best possible context-insensitive CFI policy that can be obtained for the binaries we consider.

BLOCKINSULATION. In Figure 8, we show the cumulative distribution (CDF) of BLOCKINSULATION over the call sites. As we mentioned earlier, curves that are lower and to the right indicate more secure CFI policies. As expected, StrictCFI is the most secure policy, since it does not introduce any overapproximation. The BLOCKINSULATION of NumCFI is approximately one order of magnitude greater than TypeCFI, which is the best CFI policy generator currently deployed on a large scale. Moreover, Num+TypeCFI combines the strengths of both its components, further increasing the BLOCKINSULATION by one order of magnitude.

CFGINSULATION. As we mentioned, the most expressive way to compare two policies with the help of CFINSIGHT, is to look at a CDF (like Figure 8). However, we also define a numeric summary, CFGINSULATION. In Figure 9, we show the CFGINSULATION values for every benchmark and every CFI policy generator we consider.

The CFGINSULATION of NumCFI is approximately 5 times greater than TypeCFI considering all call sites together. Considering the benchmarks separately, in six of them NumCFI has a higher CFGINSULATION than TypeCFI, while for seven benchmarks TypeCFI has a higher CFGINSULATION. The latter seven are the benchmarks with the lowest amount of basic blocks and, hence, a distance-based CFI policy is less effective for them. For more complex applications, like the other six benchmarks, NumCFI shows a better performance than TypeCFI.

Moreover, Num+TypeCFI has a better CFGINSULATION than TypeCFI in every benchmark. Considering all call sites together, Num+TypeCFI has a CFGINSULATION which is approximately 29 times greater than TypeCFI. For eight benchmarks the improvement is at least tenfold.

Other metrics. We also compare NumCFI and Num+TypeCFI using other the existing metrics we select in Section V: fAIR [55], fAIA [18], iCTR [34] and QS [6]. We report the value of these metrics in Table II. We also report, in parentheses, the rank of every policy generator according to each metric (1 marks the best and 7 marks the worst).

We can make a number of observations from Table II. First, according to all metrics we examine (both our metrics and existing metrics), Num+TypeCFI outperforms both NumCFI and TypeCFI, always ranking second after the baseline. Second, all CFI policy generators have very high fAIR (above 99.9%), proving the point that AIR is not an effective metric to evaluate

the security guarantees of CFI policies. Third, other metrics, like fAIA, iCTR, and QS, show a more significant variation between the policy generators we evaluate. However, they have a disadvantage. They only consider the number of targets that are reachable; yet, they neglect to take into consideration the usefulness of the blocks for an attacker. BLOCKINSULATION and CFGINSULATION overcome this limitation, taking into account the usefulness of each basic block to the adversary.

VIII. L+TCFI

In the previous Section, we evaluate the security of a number of CFI policy generators, including NumCFI and Num+TypeCFI. This Section shows that the generated policies can be implemented in an efficient way, with a low run-time overhead.

To do so, we design L+TCFI, a generic CFI enforcement mechanism. In L+TCFI, every basic block b has two properties, a label l_b and a tag t_b. The label is determined by a label-based CFI policy generator like TypeCFI, while the tag is determined by a distance-based policy generator like NumCFI. The mechanism is designed to allow control flow transfers between a block b and a block c if and only if both of these conditions are met:

$$t_c \geq t_b - 1 \quad (1)$$
$$l_c = l_b \quad (2)$$

Ensuring the enforcement of these conditions requires two components: 1) a way to encode the metadata (tags and labels) in the program itself, and 2) a run-time component that decides whether indirect jumps are allowed depending on the encoded information. We discuss both of them in the following.

A. Metadata Encoding

A common strategy for CFI metadata encoding is to embed it in the executable code itself. As an example, RAP [38] inserts the metadata immediately before the beginning of every function in the program. This way, when the run-time checker needs to decide whether a jump to a pointer should be allowed, it can simply read a fixed number of bytes before the pointer and retrieve the metadata. We use this approach as well.

However, embedding metadata in the executable code must be done carefully, in order to not introduce any incompatibility or vulnerability in the application. We do this by embedding our metadata inside of *CFI marks*. Our CFI marks are interpreted by the CPU as a `nop` instruction, which do not produce any result (the name stands for "no operation"). The advantage of embedding data in `nop` instructions is that, unlike raw data, the processor can execute them without changing its state, so they can be easily inserted into the code during the build process.

On x86, `nop` instructions can have different lengths; we choose the 9-byte variant because of its convenience for our purposes. This longer variant of the `nop` instruction is achieved by encoding information on various operands (register and immediate), which are then ignored by the processor. For our purposes, we can consider the leftmost four bytes of the instruction fixed (bytes 8 to 5 in Figure 10). The following byte (4) has multiple legal values which do not influence the behavior of the processor; we choose value `0x9a`, since it is different from the value in Intel's recommended 9-byte `nop` instruction and thus very unlikely to occur in any regular code. The rightmost four bytes of the instruction (bytes 3 to 0 in Figure 10) can be set to any arbitrary value; we decide to encode the label in bytes 3 to 1, and the tag in byte 0. This layout is advantageous for the run-time checker, as Section VIII-B explains; it allows us to encode approximately 16 million labels and 256 tags, which is sufficient in our testing. We encode tag ∞ as 255, any tag $\geq$ 254 as 254, and any other tag as itself.

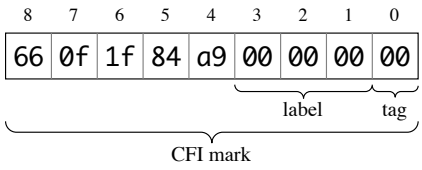

Fig. 10. A CFI mark which embeds metadata in a `nop` instruction.

We insert the CFI marks in the program by 1) instructing the compiler to create an ELF section for every function, and 2) using a custom linker script to insert these instructions between them in the final binary. We insert the marks in assembly files by rewriting them on the fly and adding the required instruction before every function definition.

B. Run-time Checker

The run-time checker has the goal of examining every indirect control flow transfer and decide whether it is allowed or not according to the metadata. We instrument every indirect control flow transfer by developing a custom pass for the Clang C/C++ compiler. Our compiler pass, which was developed for LLVM 11.0.1 and consists of approximately 60 lines of C++ code, finds all indirect function calls and instruments them to check the target address before it is used. Our proof-of-concept implementation does not embed checks in assembly code, which is only a tiny portion of the application code.

The instrumentation code checks that the target address is preceded by a valid CFI mark and that the tags and labels are correct (satisfying Equations (1) and (2) respectively). This can be done with just two comparisons: a single 64-bit equality test can check the presence of a CFI mark and that the labels match (2), while a 8-bit comparison can check whether the target label is greater than the threshold (1).

C. Security Considerations

The goal of L+TCFI is to allow an indirect call if and only if Equations (1) and (2) hold. Our run-time checker (Section VIII-B) is designed to check for CFI marks, which contain the label and tag of a function, before the indirect control flow transfer succeeds. Since we assume W$\oplus$X to be in place (Assumption A3), the adversary is unable to insert counterfeit CFI marks into the application. The adversary could, however, leverage data which accidentally matches the format of a CFI mark and is included in the code of the application. To investigate this possibility, we scanned for the

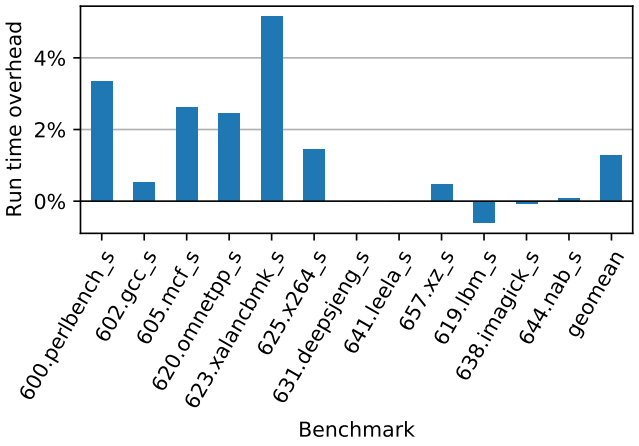

Fig. 11. Run-time overhead of SPEC CPU2017 benchmarks when using L+TCFI.

prefix of our CFI marks (the binary string `660f1f84a9`) all the baseline binaries used in our performance evaluations, as well as all binaries in the directories `/bin`, `/sbin`, and `/lib/x86_64-linux-gnu` on our test system. We did not find any match. We can then assume that accidental matches are very unlikely and, hence, the attacker cannot trick the run-time checker into calling an unintended target.

D. Handling Dynamic Libraries

Binaries are often distributed independently of the dynamic libraries they require to work. As a result, it can be impractical to apply the CFI marks to all the libraries as well as the main binary. This can be addressed by using *trampolines* that intercept indirect function calls between different libraries. Each trampoline has a CFI mark that contains the expected distance of the target function in a different library. As a result, even if the dynamic library is independently updated, the CFI marks on the trampolines remain the same and functionality is maintained.

E. Performance Evaluation

After describing L+TCFI, we analyze its performance using the run time overhead of the benchmarks we selected from SPEC CPU2017 benchmarks, as well as measuring the reduction of available throughput of an nginx instance.

Benchmarks from SPEC CPU2017. First, we analyze the performance of SPEC CPU2017 benchmarks when protected by L+TCFI compared to an unprotected baseline (Figure 11). We run every benchmark three times on the same machine mentioned in Section VII; we report the median of the three values, as recommended by SPEC. The geometric mean of the overheads is 1.27%. Only two benchmarks have an overhead higher than 3%: *600.perlbench_s* (3.33%) and *623.xalancbmk_s* (5.16%). Unsurprisingly, these benchmarks have a higher proportion of indirect function calls compared to other benchmarks.

Nginx Throughput. In addition to SPEC, we also test the effect of L+TCFI on the throughput of an nginx instance. We configure nginx to only use one worker thread, then we

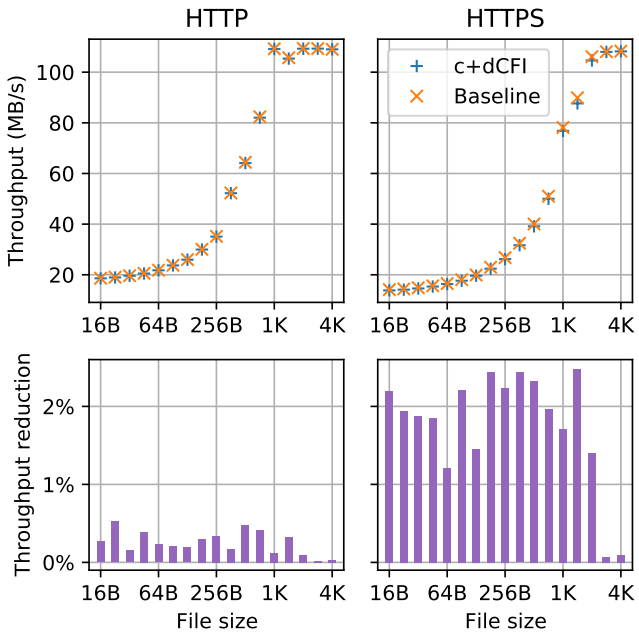

Fig. 12. Throughput and throughput reduction in nginx when using L+TCFI.

run the tool *wrk* [20] on a different machine to determine the connection throughput of a build of nginx protected by L+TCFI compared to an unprotected baseline (Figure 12). The client machine has an Intel Xeon CPU E5-2630 processor and uses its 16 threads to maintain 1024 simultaneous connections. The two machines are connected through a Gigabit Ethernet switch. In our tests we use a number of randomly-generated files having size between 16 bytes and 1 MB, and we access those files through unencrypted HTTP and HTTPS. Our tests show that any file of at least 2 KB is sufficient to saturate the Gigabit Ethernet connection while using HTTP, and any file of at least 3 KB saturates it while using HTTPS; in both cases, there is no measurable overhead above these marks. For that reason, we do not show files bigger than 4 KB in the figure. For smaller file sizes we can measure an overhead: considering only the files of size 1 KB or smaller, the geometric mean of the throughput reduction is 0.29% for HTTP and 1.99% for HTTPS.

Both our tests show performance reductions in the order of 1% to 2%, which attests that L+TCFI can be deployed in practice.

IX. Related Work

In this section we give an overview of works in the fields of CFI policies, benchmarks, as well as attacks on CFI implementations.

A. CFI Schemes

A common design aspect of CFI policies is to assign equivalence classes to every indirect caller and callee, and check that the label of the caller matches that of the callee. The first CFI policy, as proposed by Abadi et al. [2], uses CFGs generated by static analysis to derive labels for valid control-flow transfers between callers and callees, then enforces their match at run

time using inserted checks. Later, Zhang et al. [55] extend this idea to binaries using binary instrumentation. Similarly, Zhang et al. [54] propose a randomized "Springboard section" to encode a CFI policy implicitly by knowing the correct entry for the indirect jump in this springboard. This technique is also used by Tice et al. [51] in combination with a vtable protection to create a fine-grained, forward-edge CFI compiler pass for GCC and LLVM. Lockdown [39] uses dynamic binary instrumentation to inject CFI checks dynamically at run time. For backward-edge protection, Lockdown uses a shadow stack that is also guarded by dynamic checks. However, this flexibility incurs a higher performance overhead.

Another promising approach is enforcing type-based policies to restrict control-flow transfers. For example, TypeArmor [53] leverages binary-analysis techniques to infer the parameter count of a function, to restrict call targets to functions with less or equal amount of parameters than prepared by the caller. τCFI [33] extends this approach by also taking the parameter types into consideration and leverages a points-to analysis for the return instruction to protect the backward edge. MARX [36], as well as VCI [15], augment CFI mechanisms with efficient vtable protection by leveraging reconstructed class hierarchies to reduce the overapproximation of, e.g., type-based CFI policies [33]. Type-based CFI policies often imply a relatively low performance overhead, hence, the clang compiler frontend of LLVM also features a type-based CFI policy [30] that checks a variety of dynamic types. All of these label-based CFI implementations do not consider the distance of blocks to a system call, which we introduce with NumCFI, that prevents the attacker from taking shortcuts from the vulnerability to a system call.

A different line of research investigates *context-sensitive* CFI schemes, which consider some form of context to decide whether an indirect call should be allowed. PathArmor [52] compares the latest 16 taken branches against a statically generated list whenever the application calls sensitive system calls. πCFI [35] dynamically constructs a CFG at run time, in order to restrict the legal targets of return instructions, but it is similar to context-insensitive CFI for forward edges. PittyPat [14] intercepts security-sensitive system calls and validates the control flow of the program based on online points-to analysis of a subset of control-relevant data. μCFI [23] extends this analysis to include more constraint data and further refine the sets of allowed targets from any indirect call. OS-CFI [27] focuses on reducing the size of the biggest equivalence class by leveraging information about the origin of the code-pointer used by the indirect call.

B. CFI Benchmarks

In order to compare the security of CFI policies, it is crucial to quantify and compare how restrictive they are. A number of metrics have been proposed for this purpose. The best-known metric is Average Indirect-target Reduction (AIR) [55], which is defined as the average reduction of allowed targets across every CFG node (the higher the better). However, AIR is not a good metric to compare different policies [51], as most CFI papers that rely on AIR report similar values greater than 99% [6]. Other metrics also have been proposed, e.g., QuantitiveSecurity [6], which is based on the number and size of equivalence classes, AIA [18], which measures the average number of allowed indirect targets, or Calltarget Reduction (CTR) [34], which measures the absolute number of remaining call targets after applying a CFI defense. However, all of these metrics have a common pitfall: they consider every basic block equivalent to each other. The goal of an attacker in most cases is to leak the content of some memory, exfiltrate some files, or install malware on the victim machine. The first goal is trivially possible in the common CFI threat model which includes arbitrary data read capabilities. The second and the third goal require accessing system resources, for which the attacker *needs* to use system calls. While other metrics do not consider whether an edge is useful to allow the attacker to reach a system call, CFInsight is the only CFI evaluation framework that considers this goal rather than merely the count of reachable blocks.

C. Attacks on CFI

Although CFI can be a strong defense even in practical scenarios, bypasses are possible, and can also be found in practice. While Göktas et al. [21], as well as Davi et al. [13], demonstrated that coarse-grained CFI can be bypassed with new types of ROP gadgets due to the low number of labels, the first work presenting a bypass for more secure fine-grained CFI defenses was Control-Flow Bending [7]. It shows that a single `printf` call can lead to Turing-complete computation by abusing the right format specifiers, allowing the attacker to overwrite the return address of `printf`, even in the presence of a fully-precise static CFI implementation. Control Jujutsu [16], instead, exploits insecure programming patterns and the impreciseness of static analysis approaches to find legal but unintended control flows, which can be leveraged for attacks. Most of this impreciseness is caused by the central element needed for CFG creation: the complete points-to analysis for pointers, which is undecidable [42], [22], and, hence, hard to achieve in practice. Another work uses specifics of C++, namely vtable pointers, to chain virtual function calls through existing call sites, allowing the attack to be resistant against even fine-grained CFI enforcement [44]. Recently, multiple CFI bypasses switched to *data-only attacks*, in which the adversary corrupts only non-control data. These attacks can inherently bypass any kind of static CFI, as they chain only legitimate control-flow paths. While there are already solutions that automatically generate payloads [26], [40], they are still limited in target executable size due to heavy use of static analysis.

X. Conclusion

In this paper, we present CFInsight, a novel framework to evaluate the security guarantees of CFI policies. With our novel metrics BlockInsulation and CFGInsulation we measure the usefulness of any basic block to constructing a code-reuse attack targeting a system call instruction. We introduce NumCFI, a novel CFI policy generator based on the distance between each basic block and the closest system call instruction. We use CFInsight to analyze seven CFI policy generators, including NumCFI, using five different metrics, including CFGInsulation. Lastly, we describe L+TCFI, a fast implementation of NumCFI combined with a type-based policy, with a performance overhead of just 1.27% on benchmarks from the SPEC CPU2017 suite.

ACKNOWLEDGMENTS

This work was supported in part by the Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, by the European Space Operations Centre with the Networking/Partnering Initiative, by Huawei within the OpenS3 Lab, by the German Federal Ministry of Education and Research and the Hessian State Ministry for Higher Education, Research and the Arts within ATHENE, and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 952697).

REFERENCES

[1] "nginx," http://nginx.org.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "CFI: Principles, implementations, and applications," in *Proc. ACM Conference and Computer and Communications Security*, 2005.

[3] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan, "HAFIX: Hardware-Assisted Flow Integrity Extension," in *52nd Design Automation Conference*, 2015.

[4] E. Bendersky, "pyelftools," https://github.com/eliben/pyelftools, Jul 2020.

[5] N. Burow, X. Zhang, and M. Payer, "SoK: Shining light on shadow stacks," in *2019IEEE Symposium on Security and Privacy*, 2019.

[6] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Comput. Surv.*, vol. 50, no. 1, Apr. 2017. [Online]. Available: https://doi.org/10.1145/3054924

[7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium*, 2015.

[8] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[10] W. Chen, "Here's that FBI Firefox exploit for you (cve-2013-1690)," https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690, 2013.

[11] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *36th IEEE Symposium on Security and Privacy*, 2015.

[12] L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Design Automation Conference*, 2014.

[13] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium*, 2014.

[14] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, "Efficient protection of path-sensitive control security," in *26th USENIX Security Symposium*, 2017.

[15] M. Elsabagh, D. Fleck, and A. Stavrou, "Strict virtual call integrity checking for C++ binaries," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, 2017.

[16] 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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[17] R. M. Farkhani, S. Jafari, S. Arshad, W. Robertson, E. Kirda, and H. Okhravi, "On the effectiveness of type-based control flow integrity," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[18] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *2016 IEEE European Symposium on Security and Privacy*, 2016.

[19] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-grained control-flow integrity for kernel software," in *1st IEEE European Symposium on Security and Privacy*, 2016.

[20] W. Glozer, "wrk - a http benchmarking tool," https://github.com/wg/wrk, Apr 2019.

[21] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*, 2014.

[22] S. Horwitz, "Precise flow-insensitive may-alias analysis is np-hard," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 1, 1997.

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

[24] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[25] Intel Corporation, "Control-flow enforcement technology preview." https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2017.

[26] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[27] M. R. Khandaker, W. Liu, A. Naser, Z. Wang, and J. Yang, "Origin-sensitive control flow integrity," in *28th USENIX Security Symposium*, 2019.

[28] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-Pointer Integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[29] A. Limited, "Arm® a64 instruction set architecture: Future architecture technologies in the a architecture profile," https://developer.arm.com/docs/ddi0602/f/base-instructions-alphabetic-order/bti-branch-target-identification, 2020.

[30] LLVM, "Clang documentation, control-flow integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, Jul 2020.

[31] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, 1976.

[32] Microsoft, "Control flow guard for clang/llvm and rust," https://msrc-blog.microsoft.com/2020/08/17/control-flow-guard-for-clang-llvm-and-rust/, 2020.

[33] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert, "τCFI: Type-assisted control flow integrity for x86-64 binaries," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2018.

[34] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert, "Analyzing control flow integrity with LLVM-CFI," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, p. 584–597. [Online]. Available: https://doi.org/10.1145/3359789.3359806

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

[36] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, "Marx: Uncovering class hierarchies in C++ programs." in *Symposium on Network and Distributed System Security*, 2017.

[37] PaX Team, "Pax address space layout randomization (ASLR)," http://pax.grsecurity.net/docs/aslr.txt.

[38] PaX Team, "RAP: RIP ROP," 2015.

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

[40] J. Pewny, P. Koppe, and T. Holz, "Steroids for doped applications:

A compiler for automated data-oriented programming," in *2019 IEEE European Symposium on Security and Privacy*, 2019.

[41] Qualcomm Technologies Inc., "Pointer authentication on armv8.3," https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[42] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, 1994.

[43] A. Rimsa, "Cfggrind," https://github.com/rimsa/CFGgrind, Jul 2020.

[44] 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 *2015 IEEE Symposium on Security and Privacy*, 2015.

[45] H. Shacham, "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*, 2007.

[46] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[47] H. Sidhpurwala, "Hardening ELF binaries using Relocation Read-Only (RELRO)," Red Hat Blog, https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro, 2019.

[48] 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 *IEEE Symposium on Security and Privacy*, 2013.

[49] Solar Designer, "Getting around non-executable stack (and fix)," https://seclists.org/bugtraq/1997/Aug/63, 1997.

[50] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," in *2016 IEEE Symposium on Security and Privacy*, 2016.

[51] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium*, 2014.

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

[53] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, 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 *2016 IEEE Symposium on Security and Privacy*, 2016.

[54] 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 *2013 IEEE Symposium on Security and Privacy*, 2013.

[55] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium*, 2013.