

Control-Flow Integrity: Precision, Security, and Performance

Nathan Burow*, Scott Carr*, Stefan Brunthaler†,
Mathias Payer*, Joseph Nash†, Per Larsen†, Michael Franz†
*Purdue University, †SBA Research, *University of California, Irvine.

Abstract—Memory corruption errors in C/C++ programs are still the most common source of security vulnerabilities in today’s systems. Modern “control-flow hijacking” attacks exploit memory corruption vulnerabilities to divert program execution away from the intended control-flow. There is intense research interest in defenses based on Control-Flow Integrity (CFI), and this technique is now being integrated into several production compilers. However, so far no study has systematically compared the various proposed CFI mechanisms, nor is there even any protocol on how to compare such mechanisms.

We compare a range of CFI mechanisms using a unified nomenclature based on (i) a qualitative discussion of the conceptual security guarantees, (ii) a quantitative security evaluation, and (iii) an empirical evaluation of their performance on the same test environment. For each mechanism, we evaluate (i) protected types of control-flow transfers, (ii) the precision of the protection for forward and backward edges, and, for open-sourced compiler-based implementations additionally (iii) quantitatively the generated equivalence classes and target sets, and (iv) runtime performance.

I. INTRODUCTION

Low-level programming languages such as C and C++ give programmers a high degree of freedom to optimize and control how their code uses available resources. While this facilitates the construction of highly efficient programs, in practice requiring the programmer to manually manage memory and observe typing rules typically leads to security vulnerabilities. Chief among these are buffer overflows and other types of memory corruption that are routinely exploited by attackers. Despite significant research efforts in devising exploit mitigations, very few of these mitigations have entered practice [67]. The combination of three such defenses, (i) address space layout randomization (ASLR) [54], (ii) stack canaries [70], and (iii) data execution prevention (DEP) [40] guarantees protection against *code injection attacks*, but only offer probabilistic guarantees against *code-reuse attacks*. Modern attacks hijack the control-flow of the application and reuse existing code to achieve the objective of the attacker, e.g., arbitrary code execution, privilege escalation, or leaking sensitive information.

Control-flow integrity (CFI) [1] is a defense mechanism that observes that control-flow transfers differ between legitimate and illegitimate program executions, and attempts to detect these differences. This allows CFI mechanism to defend against code reuse attacks that rely on memory corruption. Conceptually, a CFI mechanism follows a two step process. An *analysis* phase constructs the Control-Flow Graph (CFG) corresponding to legitimate program execution. This CFG information is then used at runtime by an *enforcement* component that ensures that the observed branching behavior corresponds to the legitimate CFG.

During the analysis phase, which usually occurs ahead of

time, the CFG is computed by analyzing either the source code or binary of a given program. In either case, the limitations of static program analysis lead to an over-approximation of the actual control-flow edges shared among legitimate program traces. Over-approximation limits precision, and thereby also the security of the enforced CFI policy.

The enforcement phase ensures at runtime that indirect branches occur only along edges that are also present in the CFG that models legitimate program execution. Because DEP prevents injection of new code and modification of existing code, CFI can be enforced by instrumenting indirect control-flow transfers. The types of indirect branches that are subject to such validation and the number of valid targets per branch varies greatly between different CFI defenses and has a major impact on the associated security and performance characteristics. Direct branches do not need instrumentation as the target is encoded in the (immutable) code and cannot be controlled by adversaries. Call and return instructions are generally paired in the sense that a callee returns to its caller. This stateful property is not captured by any static analysis so far. Therefore, strong CFI defenses have been using a shadow stack to ensure that a callee returns to the actual caller rather than any of the potential callers. Alternatively, several CFI defenses focus on forward edges only and leave the protection of backward edges to other defenses.

CFI does not seek to prevent memory corruption, the root cause of the security defects particular to code written in low-level programming languages. While mechanisms that enforce spatial [45] and temporal [46] memory safety eliminate memory corruption (and thereby control-flow hijacking attacks), existing mechanisms are considered prohibitively expensive. In contrast, CFI defenses offer reasonably low overheads while making it substantially harder for attackers to gain arbitrary code execution in vulnerable programs. Moreover, CFI requires few changes to existing source code which allows complex software to be protected in a mostly automatic fashion.

Since Abadi et al.’s seminal paper on CFI [1] was published over a decade ago, the research community has proposed a large number of variations on the original idea. More recently, CFI implementations have been integrated into production-quality compilers, tools, and operating systems.

Current CFI mechanisms can be compared along two axes: performance and security. In the scientific literature, performance overhead is usually measured through the SPEC CPU2006 benchmarks. Unfortunately, sometimes only a subset of the benchmarks is used for evaluation. To evaluate security, many authors have used the Average Indirect target Reduction (AIR) [77] metric that counts the overall reduction of targets for any indirect control-flow transfer.

A problem of the performance measurements is that they

are all in the same range, between 0% and 20% across different benchmarks with only slight variations for the same benchmark. Since the benchmarks are evaluated on different machines with different compilers and software versions, these numbers are close to the margin of measurement error. For AIR, most mechanisms report a reduction of $> 99\%$ of targets (i.e., out of 100 possible targets, only 1 remains). AIR is not a desirable metric for two reasons. First, all CFI mechanisms report similar AIR numbers which makes AIR unfit to compare individual CFI mechanisms against each other. Second, even a large reduction of targets often leaves enough targets for an attacker to achieve the desired goals [8, 19, 26], making AIR unable to evaluate security of CFI mechanisms on an absolute scale.

We systematize the different CFI mechanisms (where “mechanism” captures both the analysis and the enforcement component of an implementation) and compare them against metrics for security and performance. By introducing metrics for these areas, our systematization allows the objective comparison of different CFI mechanisms both on an absolute level and relatively against other mechanisms. Our systematization allows potential users to assess the trade-offs of individual CFI mechanisms and choose the one that is best suited.

To evaluate the security of CFI mechanisms we follow a *comprehensive* approach, classifying them according to a *qualitative* and a *quantitative* analysis. In the qualitative security discussion we compare the strength of individual solutions on a theoretical level by evaluating the CFI policy flavour of each mechanism along several axes: (i) supported directions, (ii) precision on the forward edge, (iii) precision on the backward edge, (iv) supported control-flow transfer types according to the source programming language, and (v) reported performance. In the quantitative evaluation we evaluate each CFI mechanism and compare the produced sets for the SPEC CPU2006 benchmarks. The precision and security guarantees of a CFI mechanism depends on the *tightness* of target sets used at runtime, i.e., across all control-flow transfers, how many superfluous targets are reachable through an individual control-flow transfer. We compute these target sets for all available CFI mechanisms and compare the ranked sizes of the sets against each other. This methodology allows us to compare the actual sets used for the integrity checks of one mechanism against other mechanisms. In addition, we collect all indirect control-flow targets used for the individual SPEC CPU2006 benchmarks and use these sets as a lower bound of required targets. We use this lower bound to compute how close a mechanism is to an *ideal* CFI mechanism.

As a second metric we evaluate the performance impact of open-sourced, compiler-based CFI mechanisms. In their corresponding original publications, each mechanism was evaluated on different hardware, with different kernels, different libraries, and different systems, using either the full or a partial set of SPEC CPU2006 benchmarks. While we cannot change the difference in compilers used (e.g., different CFI mechanisms use different LLVM and GCC versions) we evaluate available CFI mechanisms on the same hardware and with the same system configuration.

Overall, we present the following contributions:

- 1) systematization of knowledge on CFI mechanisms with a focus on discussing the major different CFI

- flavors and their trade-offs,
- 2) presentation of both a qualitative and quantitative security metric and the evaluation of existing CFI mechanisms along these metrics,
- 3) a detailed performance study of existing CFI mechanisms.

II. FOUNDATIONAL CONCEPTS

This section introduces foundational concepts to evaluate different CFI mechanisms. We first define the baseline of CFI and discuss the two components of every CFI mechanism: (i) the *analysis* that defines the CFG (which inherently limits the precision that can be achieved) and (ii) the runtime instrumentation that *enforces* the generated CFG. Secondly, we classify and systematize different types of control-flow transfers and how they are used in programming languages. Thirdly, we discuss related work in static analysis.

A. Control-Flow Integrity

CFI is a policy that restricts the execution flow of a program at runtime to a predetermined CFG by validating indirect control-flow transfers. On the machine level, indirect control-flow transfers may target any executable address of mapped memory but on the source language (C or C++) the targets are greatly restricted due to the control-flow and data-flow of the program as well as the type system. Languages in the C family rely on checks inserted by the programmer to enforce memory safety. Missing checks allow an attacker to modify data outside of the context of the programming language, possibly changing either code pointers or any pointer dereferencing chain that ends in a code pointer. For example, the programming language may statically enforce strict type safety guarantees. If memory locations can be changed orthogonally to the guarantees and proofs of the source language, then these guarantees and proofs no longer hold. Without either memory safety and type safety or CFI, an attacker can use an out of bounds or temporal memory error to redirect the control-flow to an arbitrary location, resulting in a control-flow hijacking attack. CFI restricts the allowed transfers to a smaller set of target locations where the set may depend on the source location.

The security guarantees of a CFI mechanism depend on the precision of the CFG. CFG construction through source analysis inherently results in over-approximation due to imprecision and ambiguities of the static analysis. For example, assuming a type-based analysis, the same type of function pointer may be used in two different locations, but according to the data flow only a disjoint set of functions can be used at either location. These sets will end up in the same equivalence class, leading to over-approximation.

Indirect control-flow transfers allow runtime-dependent targets, e.g., the target may be stored in a register or in a memory location. This indirection allows flexibility as, e.g., the target of a function may depend on a call-back that is passed from another module. Another example of indirect control-flow transfers are return instructions that read the return address from the stack. Without such an indirection, a function would have to explicitly enumerate all possible callers and check to which location to return to based on an explicit comparison. Indirect control-flow transfers are problematic because the attacker can

potentially control the calculated address. In contrast, direct control-flow transfers target statically determined addresses. The CFG used for CFI does not consider direct control-flow transfers, e.g., conditional direct jumps, direct jumps, or direct calls.

The CFI enforcement component validates target addresses before they are used in an indirect control-flow transfer. This approach detects code pointers (including return addresses) that were modified to targets that differ from the statically determined set.

B. Classification of Control-Flow Transfers

Control-flow transfers can broadly be separated into two categories: (i) *forward* and (ii) *backward*. Forward control-flow transfers are those that move control to a new location inside a program. When a program returns control to a prior location, we call this a backward control-flow¹.

A CPU's instruction-set architecture (ISA) usually offers two forward control-flow transfer instructions: call and jump. Both of these are either direct or indirect, resulting in four different types of forward control-flow:

- *direct jump*: is a jump to a constant, statically determined target address. Most local control-flow, such as loops or if-then-else cascaded statements, use direct jumps to manage control.
- *direct call*: is a call to a constant, statically determined target address. Static function calls, for example, use direct call instructions.
- *indirect jump*: is a jump to a computed, i.e., dynamically determined target address. Examples for indirect jumps are switch-case statements using a dispatch table, as well as the threaded code interpreter dispatch optimization [5, 21, 36].
- *indirect call*: is a call to a computed, i.e., dynamically determined target address. The following three examples are relevant in practice:

Function pointers are often used to emulate object-oriented method dispatch in classical record data structures, such as C structs, or for passing callbacks to other functions.

vtable dispatch is the preferred way to implement dynamic dispatch to C++ methods. A C++ object keeps a pointer to its *vtable*, a table containing pointers to all virtual methods of its dynamic type. A method call, therefore, requires (i) dereferencing the vtable pointer, (ii) computing table index using the method offset determined by the object's static type, and (iii) an indirect call instruction to the table entry referenced in the previous step. In the presence of multiple inheritance, or multiple dispatch, dynamic dispatch is slightly more complicated.

Smalltalk-style send-method dispatch that requires a dynamic type look-up. Such a dynamic dispatch using a *send-method* requires walking the class hierarchy

(or the prototype chain in JavaScript) and selecting the first method with a matching identifier. This expensive procedure is required for all method calls and therefore impacts performance negatively.

Jump instructions feature another property and can be either conditional or unconditional. For all intents and purposes of this paper this distinction is not relevant.

Instruction-set architectures support backward control-flow transfers using just one instruction: return, or just *ret*. A return instruction is the symmetric counterpart of a call instruction, and a compiler emits function prologues and epilogues to form such pairs. A call instruction pushes the address of the immediately following instruction onto the native machine stack. A return instruction pops the address off the native machine stack and updates the CPU's instruction pointer to point to this address. Notice that a return instruction is conceptually similar to an indirect jump instruction, since the return address is unknown a priori. Furthermore, compilers are emitting call-return pairs by *convention* that hardware usually does not enforce. By way of modifying return addresses on the stack, an attacker can "return" to all addresses in a program, the foundation of return-oriented programming [10, 58, 63].

Control-flow transfers can become more complicated in the presence of more involved features. Exception handling complicates control-flows locally, i.e., within a function, for example by moving control from a try-block into a catch-block. Global exception-triggered control-flow manipulation, i.e., interprocedural control-flows, require unwinding stack frames on the current stack until a matching exception handler is found.

Further topics requiring attention from a control-flow transfer perspective are: (i) separate compilation, (ii) dynamic linking, and (iii) compiling libraries. Finally, keep in mind that, in general, not all control-flow transfers can be recovered from a binary.

Summing up, our classification scheme of control-flow transfers is as follows:

- **CF.1**: backward control-flow,
- **CF.2**: forward control-flow using direct jumps,
- **CF.3**: forward control-flow using direct calls,
- **CF.4**: forward control-flow using indirect jumps,
- **CF.5**: forward control-flow using indirect calls supporting function pointers,
- **CF.6**: forward control-flow using indirect calls supporting vtables,
- **CF.7**: forward control-flow using indirect calls supporting *send-style* method dispatch,
- **CF.8**: complex control-flow to support exception handling,
- **CF.9**: control-flow supporting language features such as dynamic linking, separate compilation, etc.

According to this classification, the C programming language uses control-flow transfers 1–5 and 9, whereas the C++ programming language allows all control-flow transfers except no. 7.

¹Note the ambiguity of a backward edge in machine code (i.e., a backwards jump to an earlier memory location) which is different from a backward control-flow transfer as used in CFI.

<code>o o;</code>		
<code>o = new ClassA();</code>	$o \rightarrow \text{ClassA}$	
<code>...</code>	<code>...</code>	$o \rightarrow \{\text{ClassA}, \text{ClassB}\}$
<code>o = new ClassB();</code>	$o \rightarrow \text{ClassB}$	
(a) Flow-sensitivity example.	(b) Flow-sensitive result.	(c) Flow-insensitive result.

<code>Obj id(Obj o) { return o; }</code>		
<code>x = new A();</code>	$x \rightarrow A$	$x \rightarrow A$
<code>y = new B();</code>	$y \rightarrow B$	$y \rightarrow B$
<code>a = id(x);</code>	$a \rightarrow A; \text{id}_1 \rightarrow A$	$a \rightarrow \text{id}; \text{id} \rightarrow A$
<code>b = id(y);</code>	$b \rightarrow B; \text{id}_2 \rightarrow B$	$b \rightarrow \text{id}; \text{id} \rightarrow \{A, B\}$
(d) Context-sensitivity example.	(e) Context-sensitive result.	(f) Context-insensitive result.

Figure 1: Effects of flow/context sensitivity on precision.

C. Prior Results in Static Analysis

Static analysis research has attracted significant interest and attention from the research community. Following our classification of control-flows in Section II-B, we are particularly interested in static analysis that identifies indirect calls/jumps targets. Researchers refer to this kind of static analysis as *points-to analysis*. The wealth of advanced information and results in points-to analysis goes well beyond the scope of this paper. We focus our attention on how points-to analysis affects CFI precision.

1) *A Theoretical Perspective*: Many compiler optimizations benefit from points-to analysis. As a result, points-to analysis must be sound at all times and therefore conservatively over-approximates results. The program analysis literature (e.g., [31, 32, 47, 65]) expresses this conservative aspect as a *may*-analysis: A specific object “may” point to any members of a computed points-to set.

For the purposes of this paper, the following orthogonal dimensions in points-to analysis affect precision:

- *flow-sensitive* vs. *flow-insensitive*: this dimension states whether an analysis considers control-flow (sensitive) or not (insensitive).
- *context-sensitive* vs. *context-insensitive*: this dimension states whether an analysis considers various forms of context (sensitive) or not (insensitive). The literature further separates the following context information sub-categories: (i) call-site sensitive: the context includes a function’s call-site (e.g., call-strings [64]), (ii) object sensitive: the context includes the specific receiver object present at a call-site [42], (iii) type sensitive: the context includes type information of functions or objects at a call-site [66].

Both dimensions, context and flow sensitivity, are orthogonal and a points-to analysis combining both yields higher precision.

a) *Flow-Sensitivity*: Figures 1a – 1c show the effect of flow sensitivity on points-to analysis. A flow-sensitive analysis

considers the state of the program per line. We see, for instance, in Figure 1b how a flow-sensitive analysis computes the proper object type per allocation site. A flow-insensitive analysis, on the other hand, computes sets that are valid for the whole program. Or, simply put, it lumps all statements of the analyzed block (intra- or interprocedural) into one set and computes a single points-to set that satisfies all of these statements. From a CFI perspective a flow-sensitive points-to analysis offers higher precision.

b) *Context-Sensitivity*: Figures 1d – 1f show the effects of context sensitivity on points-to analysis. In Figure 1d we see that the function `id` is called twice, with parameters of different dynamic types. Context-insensitive analysis (cf. Figure 1f), does not distinguish between the two different calling contexts and therefore computes an over-approximation by lumping all invocations into one points-to set (e.g., the result of calling `id` is a set with two members). A context-insensitive analysis, put differently, considers a function independent from its callers, and is therefore the forward control-flow transfer symmetric case of a backward control-flow transfers returning to many callers [47]. Context-sensitive analysis (cf. Figure 1e), on the other hand, uses additional context information to compute higher precision results. The last two lines in Figure 1e illustrate the higher precision by inferring the proper dynamic types `A` and `B`. From a control-flow integrity perspective a context-sensitive points-to analysis offers higher precision.

c) *Object-Oriented Programming Languages*: A C-like language requires call-string or type context-sensitivity to compute precise results for function pointers. Due to dynamic dispatch, however, a C++-like language should consider more context provided by object sensitivity [38, 42]. Alternatively, prior work describes several algorithms to “devirtualize” call-sites. If a static analysis identifies that for a given call-site only one receiver is possible (i.e., if the points-to set is a singleton) a compiler can sidestep expensive dynamic dispatch via the vtable and generate a direct call to the referenced method. Class-hierarchy analysis (CHA) [20] and rapid-type analysis (RTA) [4] are prominent examples that use domain-specific information about the class hierarchy to optimize virtual method calls. RTA

differs from CHA by pruning entries from the class hierarchy from objects that have not been instantiated. As a result, the RTA precision is higher than CHA precision [27]. Grove and Chambers [27] study the topic of call-graph construction and present a partial order of various approaches' precision (Figure 19, pg. 735). From a CFI perspective, higher precision in the call-graph of virtual method invocations translates to either (i) more de-virtualizations of call-sites, which replace an indirect call by a direct call, or (ii) decreasing the points-to sets, which reduce an adversary's attack surface. Note that the former, de-virtualization of a call-site also has the added benefit of removing the call-site from a points-to set and transforming an indirect control-flow transfer to a direct control-flow transfer that need not be validated by the CFI enforcement component.

2) *A Practical Perspective:* Points-to analysis over-approximation reduces precision and therefore restricts the optimization potential of programs. The reduced precision also reduces precision for CFI, opening the door for attackers. If, for instance, the over-approximated set of computed targets contains many more "reachable" targets, then an attacker can use those control-flow transfers without violating the CFI policy. Consequently, prior results from studying the precision of static points-to analysis are of key importance to understanding CFI policies' security properties.

Mock et al. have studied dynamic points-to sets and compared them to statically determined points-to sets [43]. More precisely, the study used an instrumentation framework to compute dynamic points-to sets and compared them with three flow- and context-insensitive points-to algorithms. The authors report that static analyses identified 14% of all points-to sets as singletons, whereas dynamic points-to sets were singletons in 97% of all cases. In addition, the study reports that one out of two statically computed singleton points-to sets were optimal in the sense that the dynamic points-to sets were also singletons. The authors describe some caveats and state that flow and context sensitive points-to analyses were not practical in evaluation since they did not scale to practical programs. Subsequent work has, however, established the scalability of such points-to analyses [28, 29, 30], and a similar experiment evaluating the precision of computed results is warranted.

Concerning the analysis of devirtualized method calls, prior work reports the following results. By way of manual inspection, Rountev et al. [60] report that 26% of call chains computed by RTA were actually infeasible. Lhotak and Hendren [38] studied the effect of context-sensitivity to improve precision on object-oriented programs. They find that context sensitivity has only a modest effect on call-graph precision, but also report substantial benefits of context sensitivity to resolve virtual calls. In particular, Lhotak and Hendren highlight the utility of object-sensitive analyses for this task. Tip and Palsberg [69] present advanced algorithms, XTA among others, and report that it improves precision over RTA, on average, by 88%.

3) *Backward Control Flows:* Figure 2 shows two functions, f and g , which call another function h . The return instruction in function h can, therefore, return to either function f or g , depending on which function actually called h at run-time. To select the proper caller, the compiler maintains and uses a stack of activation records, also known as stack frames. Each stack frame contains information about the CPU instruction

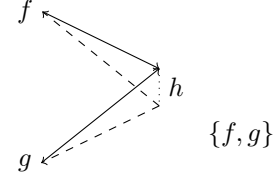


Figure 2: Backward control-flow precision.

pointer of the caller as well as bookkeeping information for local variables.

Since there is only one return instruction at the end of a function, even the most precise static analysis can only infer the set of callers for all calls. Computing this set, inevitably, leads to imprecision and all call-sites of a given function must therefore share the same label/ID such that the CFI check succeeds. Presently, the only known alternative to this loss of precision is to maintain a shadow stack and check whether the current return address equals the return address of the most recent call instruction.

4) *Classification of Static Analysis Precision:* For the intents and purposes of this paper, the following classification summarizes prior work to determine forward control-flow transfer analysis precision:

- **SAP.F.0:** No forward branch validation
- **SAP.F.1a:** ad-hoc algorithms and heuristics
- **SAP.F.1b:** context- and flow-insensitive analysis
- **SAP.F.1c:** labeling equivalence classes
- **SAP.F.2:** class-hierarchy analysis
- **SAP.F.3:** rapid-type analysis
- **SAP.F.4a:** flow-sensitive analysis
- **SAP.F.4b:** context-sensitive analysis
- **SAP.F.5:** context- and flow-sensitive analysis
- **SAP.F.6:** dynamic analysis (optimistic)

For the intents and purposes of this paper, the following classification summarizes prior work to determine backward control-flow transfer analysis precision:

- **SAP.B.0:** No backward branch validation
- **SAP.B.1:** Labeling equivalence classes
- **SAP.B.2:** Shadow stack

Note that there is well established and vast prior work in static analysis that goes well beyond the scope of this paper. The goal of our systematization is merely to summarize the most relevant aspects and use them to shed more light on the precision aspects of CFI.

D. Nomenclature and Taxonomy

Prior work on CFI usually classifies mechanisms into fine and coarse-grained. Over time, however, these terms have been used to describe different systems with varying granularity and have, therefore, become overloaded and imprecise. In addition, prior work only uses a rough separation into forward and backward control-flow transfers without considering sub types or precision.

III. SECURITY

In this section we present a security analysis of existing CFI implementations. Drawing on the foundational knowledge in Section II, we present a qualitative analysis of the theoretical security of different CFI mechanisms based on the policies that they implement. We then give a quantitative evaluation of a selection of CFI implementations. Finally, we survey previous security evaluations and known attacks against CFI.

A. Qualitative Security Guarantees

Our qualitative analysis of prior work and proposed CFI implementations relies on the classifications of the previous section (cf. Section II) to provide a higher resolution view of precision and security. Figure 3 summarizes our findings among five dimensions while Figure 4 adds a quantitative argument as well.

- supported directions of an approach (i.e., either forward, or forward and backward);
- supported control-flow transfers (cf. Section II-B);
- reported performance numbers;
- static-analysis precision of forward control-flows (cf. Section II-C4); and
- static-analysis precision of backward control-flows (cf. Section II-C4).

The shown CFI implementations are ordered chronologically by publication year, and the colors indicate whether a CFI implementation works on the binary-level (blue), relies on source-code (green), or uses other mechanisms (red), such as hardware implementations.

Our classification and categorization efforts for reported performance were hindered by methodological variances in benchmarking. Experiments were conducted on different machines, different operating systems, and also different or incomplete benchmark suites. Classifying and categorizing static analysis precision was impeded by the high level, imprecise descriptions of the implemented static analysis by various authors. Both of these impediments, naturally, are sources of imprecision in our evaluation. We hope that future work adopts our classification to further scientific progress.

Comprehensive protection through CFI requires the validation of both forward and backward branches. This requirement means that the reported performance impact for forward-only approaches (i.e., SafeDispatch, T-VIP, VTV, IFCC, vfGuard, and VTint) is restricted to partial protection. When seeking a holistic perspective, the performance impact for backward control-flows must be considered as well.

CFI mechanisms satisfying SAP.B.2, i.e., using a shadow stack to obtain high precision for backward control-flows are: original CFI [1], MoCFI [17], HAFIX [2, 18], and Lockdown [55]. PathArmor emulates a shadow stack through validating the last-branch register (LBR).

Increasing the precision of static analysis techniques that validate whether any given control-flow transfer corresponds to an edge in the CFG decreases the performance of the CFI mechanism. Most implementations choose to combine precise results of static analysis into an equivalence class. Each such equivalence class receives a unique identifier, often

referred to as a label, which the CFI enforcement component validates at runtime. By not using a shadow stack, or any other comparable high-precision backward control-flow transfer validation mechanism, even high precision forward control-flow transfer static analysis becomes imprecise due to labeling. The explanation for this loss in precision is straightforward: to validate a control-flow transfer, all callers of a function need to carry the same label. Labeling, consequently, is a substantial source of imprecision (see Section III-B for more details). The notable exception in this case is π CFI, which uses dynamic information, analogous to dynamic points-to sets [43], to build a high-resolution, precise control-flow graph. Borrowing a term from information-flow control [61], π CFI can, however, suffer from *label creep* by accumulating too many labels from the static CFG.

CFI implementations introducing imprecision via labeling are: the original CFI paper [1], control-flow locking [6], CF-restrictor [56], CCFIR [75], MCFI [49], KCoFI [14], and RockJIT [50].

According to the criteria established in analyzing points-to precision, we find that at the time of this writing, π CFI [51] offers the highest precision due to leveraging dynamic points-to information. π CFI's predecessors, RockJIT [50] and MCFI [49], already offered a high precision due to the use of context-sensitivity in the form of types. PathArmor also scores well when subject to our evaluation: high-precision in both directions, forward and backward, but hampered by limited hardware resources (LBR size). Lockdown [55] offers high precision on the backward edges but derives its equivalence classes from the number of libraries used in an application and is therefore inherently limited in the precision of the forward edges. IFCC [68] offers variable static analysis granularity. On the one hand, IFCC describes a Full mode that uses type information, similar to π CFI and its predecessors. On the other hand, IFCC mentions less precise modes, such as using a single set for all destinations, and separating by function arity. With the exception of Hypersafe [72], all other evaluated CFI implementations offer lower precision of varying degrees below SAP.F.3.

B. Quantitative Security Guarantees

Quantitatively assessing how much security a CFI mechanism provides is challenging as attacks are often program dependent and different implementations might allow different attacks to succeed. So far, the only existing quantitative measure of the security of a CFI implementation is Average Indirect Target Reduction (AIR). Unfortunately, AIR is known to be a weak proxy for security [68]. A more meaningful metric must focus on the number of targets (i.e., number of equivalence classes) available to an attacker. Furthermore, it should recognize that smaller classes are more secure, because they provide less attack surface. Thus an implementation with a small number of large equivalence classes is more vulnerable than an implementation with a large number of small equivalence classes.

One possible metric is the product of the inverse of the number of equivalence classes (EC) and the size of the largest class (LC), see Equation 1. Smaller products would indicate a more secure mechanism as the product decreases with increases

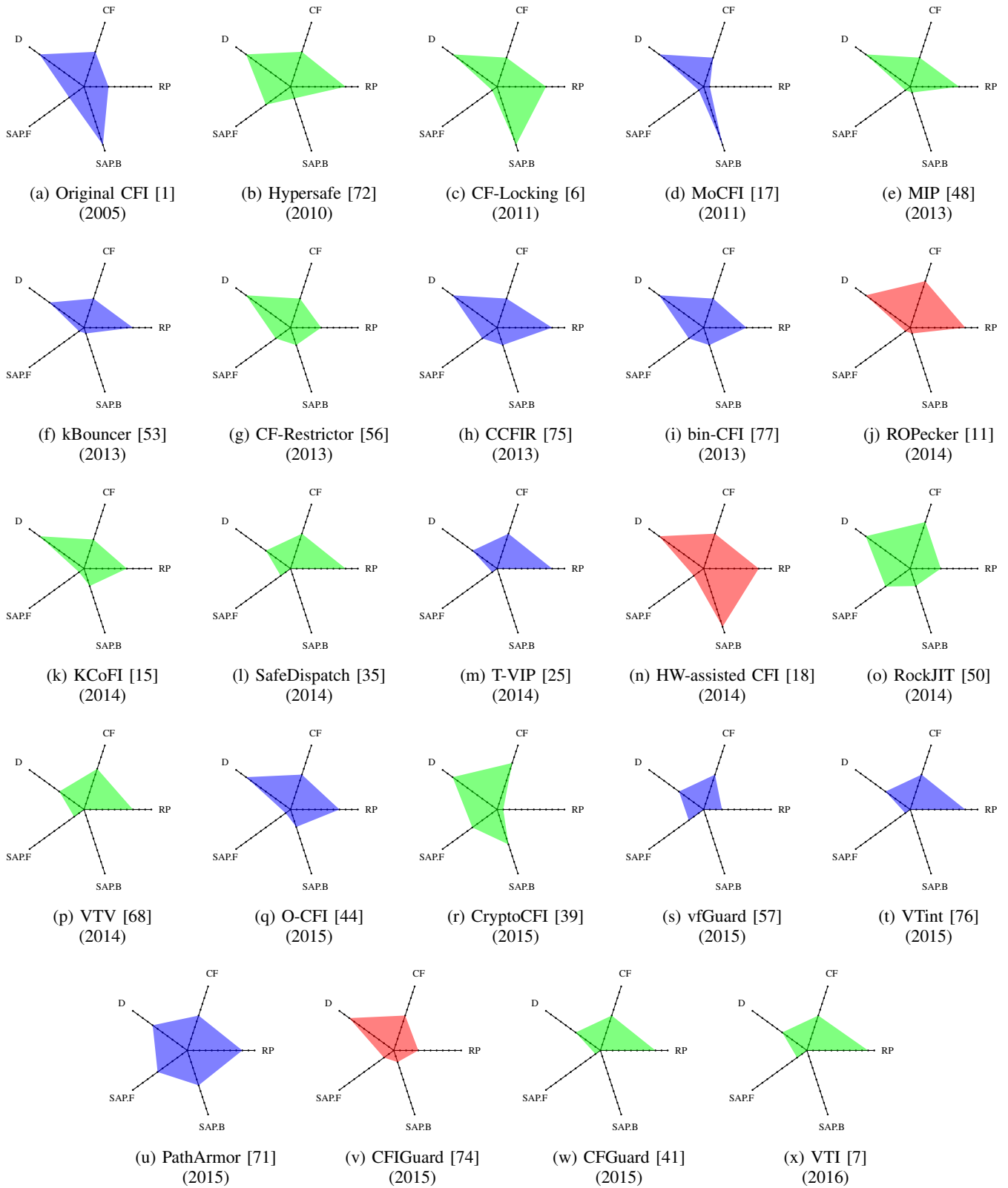


Figure 3: CFI implementation comparison: direction (D), supported control-flows (CF), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B). Color coding of CFI implementations: binary are blue, source-based are green, others red.

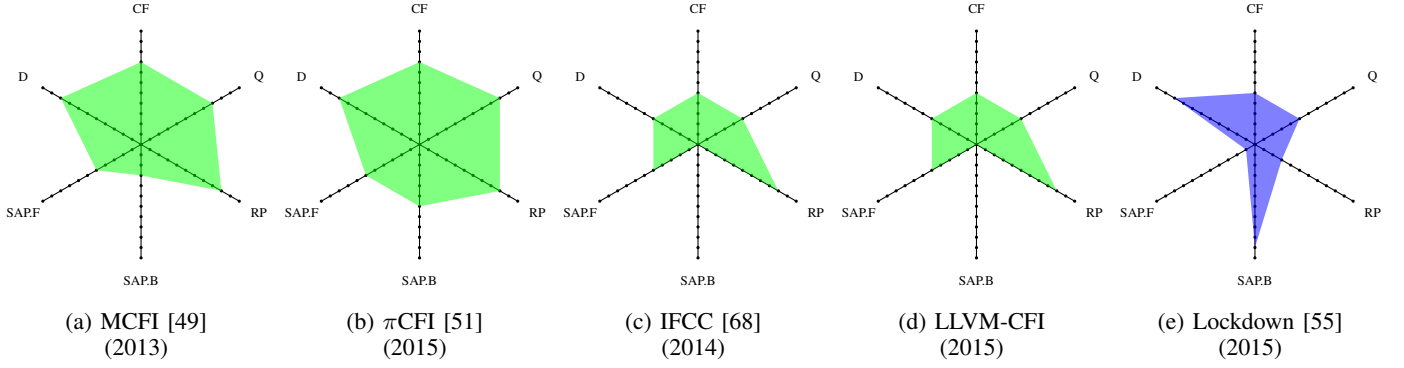


Figure 4: Quantitative comparison: direction (D), control-flows (CF), quantitative security (Q), reported performance (RP), static analysis precision: forward (SAP.F) and backward (SAP.B).

in both the number of equivalence classes and the size of the largest set. A more sophisticated version would also consider the usability of the sets, by considering whether or not they are located on an attacker accessible “hot” path, and if so how many times they are used. A large equivalence class that is pointed to by many indirect calls on the hot path poses a higher risk because it is more accessible to the attacker.

$$\frac{1}{EC} * LC = QuantitativeSecurity \quad (1)$$

This section measures the number and sizes of sets to allow a meaningful, direct comparison of the security provided by different implementations. Moreover, we report the dynamically observed number of sets and their sizes. This provides a lower bound on the precision required from the implementations’ CFG analysis. It also provides a yardstick to measure their achieved precision against.

1) Implementations: We evaluate four compiler-based, open-source CFI mechanisms IFCC, LLVM-CFI, MCFI, and π CFI. For IFCC and MCFI we also evaluated the different analysis techniques available in the implementation. In addition to the compiler based solutions, we also evaluate Lockdown, which is a binary based CFI implementation.

MCFI and π CFI already have a built-in reporting mechanism. For the other mechanisms we extend the instrumentation pass and report the number and size of the produced target sets. We then used the implementations to compile, and for π CFI run, the SPEC CPU2006 benchmarks to produce the data we report here².

IFCC³ comes with four different CFG analysis techniques: *single*, *arity*, *simplified*, and *full*. *Single* creates only one equivalence class for the entire program, resulting in the weakest possible CFI policy. *Arity* groups functions into equivalence classes based on their number of arguments. *Simplified* improves on this by recognizing three types of arguments: composite, integer, or function pointer. *Full* considers the precise return

type and types of each argument. We expect full to yield the largest number of equivalence classes with the smallest sizes, as it performs the most exact distribution of targets.

Both MCFI and π CFI rely on the same underlying static analysis. The authors claim that disabling tail calls is the single most important precision enhancement for their CFG analysis [52]. We measure the impact of this option on our metric. MCFI and π CFI are also unique in that their policy and enforcement mechanisms consider backwards edges as well as forward edges. When comparing to other implementations, we only consider forward edges. This ensures direct comparability for the number and size of sets. The results for backwards edges are presented as separate entries in the figures.

As of LLVM 3.7, the latest released version, LLVM-CFI cannot be directly compared to the other CFI implementations because its policy is strictly more limited. Instead of considering all forward, or all forward and backward edges, LLVM-CFI focuses on virtual calls and ensures that virtual, and non-virtual calls are performed on objects of the correct dynamic type. There is ongoing work in the development branch of LLVM, version 3.8, to add support for all indirect calls.⁴

Lockdown is a CFI implementation that operates on compiled binaries and supports the instrumentation of dynamically loaded code. To protect the backward edge, Lockdown enforces a shadow stack. For the forward edge, it instruments libraries at runtime, creating one equivalence class per library. Consequently, the set size numbers are of the greatest interest for Lockdown. Lockdown’s precision depends on symbol information, allowing indirect calls anywhere in a particular library if it is stripped. Therefore, we only report the set sizes for non-stripped libraries where Lockdown actually operates.

To collect the data for our lower bound, we wrote an LLVM pass. This pass instruments the program to collect and report the source line for each indirect call, the number of different targets for each indirect call, and the number of times each of those targets was used. This data is collected at runtime. Consequently, it represents only a subset of all possible indirect calls and targets. As such, we use it to present a lower bound on the number of equivalence sets (i.e. unique indirect call sites)

²We will release our instrumentation passes, the source modifications, reporting utilities, and full scripts with publication.

³Note that the IFCC patch was pulled by the authors and will be replaced by LLVM-CFI.

⁴If LLVM 3.8 is released before publication, we will update our analysis to include this feature.

and size of those sets (i.e. the number of different locations called by that site).

2) *Results*: We conducted three different quantitative evaluations in line with our proposed metric for evaluating the overall security of a CFI mechanism and our lower bound. For IFCC, LLVM-CFI, MCFI it is sufficient to compile the SPEC CPU2006 benchmarks as they do not dynamically change their equivalence classes. π CFI uses dynamic information, so we had to run the SPEC CPU2006 benchmarks. Similarly, Lockdown is a binary CFI implementation that only operates at run time. We highlight the most interesting results in Figure 4, see Table III in Section A for the full data set.

Figure 5 shows the number of equivalence classes for the four CFI implementations that we evaluated, as well as their sub-configurations. As advertised, IFCC *Single* only creates one equivalence class. We obtain, unsurprisingly, that this IFCC mode offers the least precision. The other IFCC analysis modes only had a noticeable impact for perlbench and soplex. Indeed, on the sjeng benchmark all four analysis modes produced only one equivalence class.

On forward edges, MCFI and π CFI are more precise than IFCC in all cases except for perlbench where they are equivalent. No other implementation considers backwards edges. Relative to each other, π CFI’s dynamic information decreases the number of equivalence classes available to the attacker by 21.6%. The authors of MCFI and π CFI recommend disabling tail calls to improve CFG precision. This only impacts the number of sets that they create for backwards edges, not forward edges. As such this compiler flag does not impact most CFI implementations, which rely on a shadow stack for backwards edge security.

LLVM-CFI creates a number of equivalence classes equal to the number of classes used in the C++ benchmarks. However, due to the partial support of only a subset of indirect control-flow transfer types we cannot compare LLVM-CFI with other mechanisms directly. However, we present the results in Figure 5 and Figure 6 to show the relative cost of protecting vtables in C++ to protecting all indirect call sites.

Second, we quantify the set sizes for each of the four implementations in Figure 6. We show box and whisker graphs of the set sizes for each implementation. The red line is the median set size and a smaller median set size indicates more secure mechanisms. The blue box extends from the 25th percentile to the 75th, smaller boxes indicate a tight grouping around the median. An implementation might have a low median, but large boxes indicating that there are still some large equivalence classes for an attacker to target. The top whisker extends from the top of the box for 150% of the size of the box. Data points beyond the whiskers are considered outliers and indicate large sets. This plot format allows an intuitive understanding of the security of the distribution of equivalence class sizes. Lower medians and smaller boxes are better. Any data points above the top of the whisker show very large, outlier equivalence classes that provide a large attack surface for an adversary.

Note that IFCC only creates a single equivalence class for xalancbmk and namd (except for the Full configuration on namd which is more precise). Entries with just a single equivalence class are reported as only a median. IFCC data

points allow us to rank the different analysis methods, based on the results for benchmarks where they actually impacted set size: perlbench and soplex. In increasing order of precision (least precise to most precise) they are: *single*, *arity*, *simplified*, and *full*. This does not necessarily mean that the more precise analysis methods are more secure, however. For perlbench the more precise methods have outliers at the same level as the median for the least precise (i.e., *single*) analysis. For soplex the outliers are not as bad, but the *full* outlier is the same size as the median for *arity*. While increasing the precision of the underlying CFG analysis increases the overall security, edge cases can cause the incremental gains to be much smaller than anticipated.

The MCFI forward edge data points highlight this. The MCFI median is always smaller than the IFCC median. However, for all the benchmarks where both ran, the MCFI outliers are greater than or equal to the largest IFCC set. From a quantitative perspective, we can only confirm that MCFI is at least as secure as IFCC. The effect of the outlying large sets on relative security remains an open question, though it seems likely that they provide opportunities for an attacker.

While we cannot directly compare LLVM-CFI to the other implementations, it is important to note that LLVM-CFI sets do not have extreme outliers. Additionally, Figure 6 shows that the equivalence classes that are created have a low variance, as seen by the more compact whisker plots that lack the large number of outliers present for other techniques. As such, LLVM-CFI does not suffer from the edge cases that effect more general analyzes.

Lockdown consistently has the largest set sizes, as expected because it only creates one equivalence class per library and the SPEC CPU2006 benchmarks are optimized to reduce the amount of external library calls. These sets are up to an order of magnitude larger than compiler techniques. However, Lockdown isolates faults into libraries as each library has its independent set of targets compared to a single set of targets for other binary-only approaches like CCFIR and bin-CFI.

The lower bound numbers were measured dynamically, and as such encapsulate a subset of the actual equivalence sets in the static program. Further, each such set is at most the size of the static set. Our lower bound thus provides a proxy for an ideal CFI implementation in that it is perfectly precise for each run. However, all of the IFCC variations report fewer equivalence classes than our dynamic bound. On this basis, we classify IFCC as a coarse grained CFI implementation. MCFI and π CFI find more sets, and so we consider them fine grained.

The whisker plots for our dynamic lower bound in Figure 6 show that some of the SPEC CPU2006 benchmarks inherently have outliers in their set sizes. For perlbench, gcc, gobmk, h264ref, omnetpp, and xalancbmk our dynamic lower bound and the static set sizes from the compiler based implementations all have a significant number of outliers. This provides quantitative backing to the intuition that some code is more amenable to protection by CFI. What coding styles and practices make code more or less amenable to CFI is out of scope here, but interesting future work.

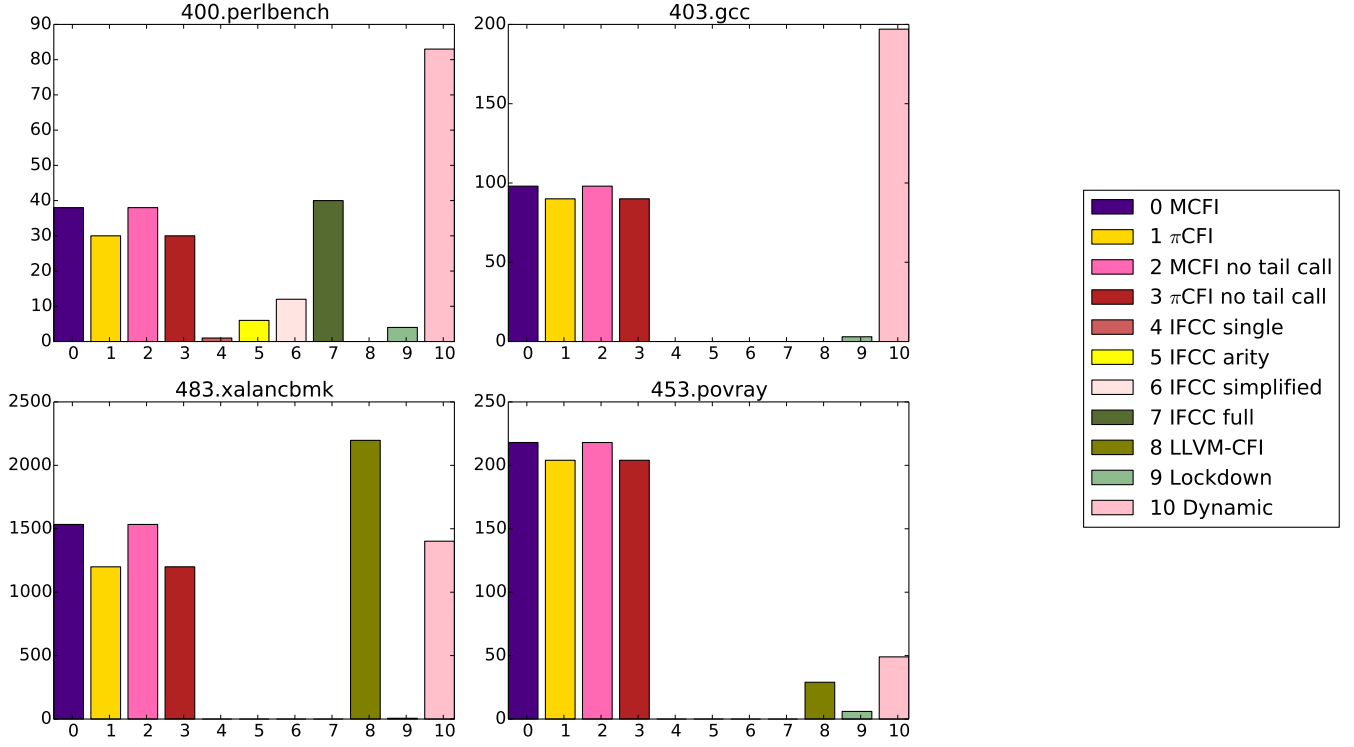


Figure 5: Total number of equivalence classes for different mechanisms when running SPEC CPU2006 (higher is better).

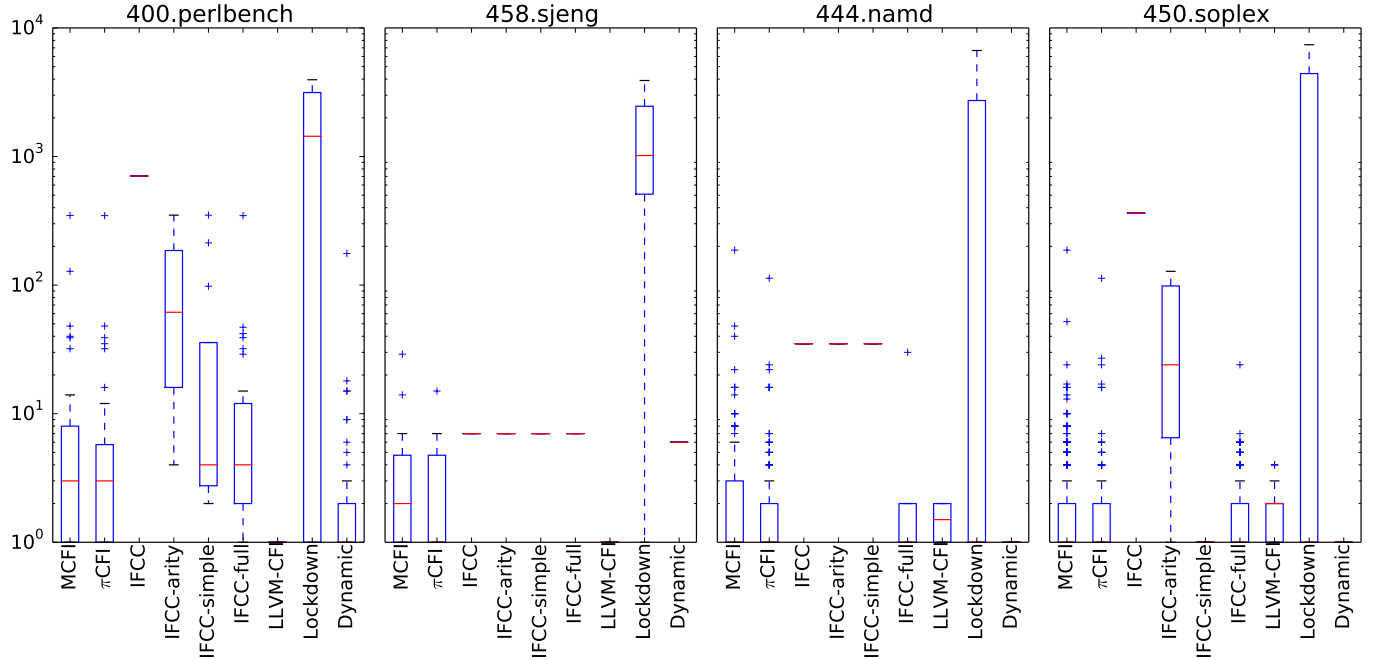


Figure 6: Whisker plot of equivalence class sizes for different mechanisms when running SPEC CPU2006. (Smaller is Better)

C. Previous Security Evaluations and Attacks

Evaluating the security of a CFI implementation is challenging because exploits are program dependent and simple metrics do not cover the security of a mechanism. The Average Indirect target Reduction (AIR) metric [77] captures the average reduction of allowed targets, following the idea that an attack is less likely if fewer targets are available. This metric was then used to measure new CFI implementations, generally reporting high numbers of more than 99%. Such high numbers give the illusion of relatively high security but, e.g., if a binary has 1.8 MB of executable code (the size of the glibc on Ubuntu 14.04), then an AIR value of 99.9% still allows 1,841 targets, likely plenty enough for an arbitrary attack. A similar alternative metric to evaluate CFI effectiveness is the gadget reduction metric [49]. Unfortunately, these simple relative metrics give, at best, an intuition for security and we argue that a more involved metric is needed.

A first set of attacks against CFI implementations targeted *coarse-grained* CFI that only had 1-3 equivalence classes [8, 19, 26]. These attacks show that equivalence classes with a large number of targets allow an attacker to execute code and system calls, especially if return instructions are allowed to return to any call site.

Counterfeit Object Oriented Programming (COOP) [62] introduced the concept that whole functions can be used as gadgets. Virtual calls in C++ are a specific type of indirect function calls that are dispatched via vtables, which are arrays of function pointers. COOP shows that an attacker can construct counterfeit objects and, by reusing existing vtables, perform arbitrary computations. This attack shows that indirect calls requiring another level-of-indirection (e.g., through a vtable) must have additional checks that consider the types at the language level for the security check as well.

Control Jujutsu [23] extends the existing attacks to so-called *fine-grained* CFI by leveraging the imprecision of points-to analysis. This work shows that common software engineering practices like modularity (e.g., supporting plugins and refactoring) force points-to analysis to merge several equivalence classes. This imprecision results in target sets that are large enough for arbitrary computation.

Control-Flow Bending [9] goes one step further and shows that attacks against perfect CFI are possible. Perfect CFI assumes that a precise CFG is available that is not achievable in practice, i.e., if any edge would be removed then the program would fail. Even in this configuration attacks are likely possible if no shadow stack is used, and sometimes possible even if a shadow stack is used.

Several attacks target data structures used by CFI mechanisms. StackDefiler [13] leverages the fact that many CFI mechanisms implement the enforcement as a compiler transformation. Due to this high-level implementation and the fact that the optimization infrastructure of the compiler is unaware of the security aspects, an optimization might choose to spill registers that hold sensitive CFI data to the stack where it can be modified by an attack. Any CFI mechanism will rely on some runtime data structures that are sometimes writeable (e.g., when MCFI loads new libraries and merges existing sets). Missing the Point [22] shows that ASLR might not be enough to hide this secret data from an adversary.

IV. PERFORMANCE

While the security properties of CFI (or the lack thereof for some mechanisms) have received most scrutiny in the academic literature, performance characteristics play a large part in determining which CFI mechanisms are likely to see adoption and which are not. Szekeres et al. [67] surveyed mitigations against memory corruption and found that mitigations with more than 10% overhead do not tend to see widespread adoption in production environments and that overheads below 5% are desired by industry practitioners.

Comparing the performance characteristics of CFI mechanisms is a non-trivial undertaking. Differences in the underlying hardware, operating system, as well as implementation and benchmarking choices prevents apples-to-apples comparison between the performance overheads reported in the literature. For this reason, we take a two-pronged approach in our performance survey: For a number of publicly available CFI mechanisms, we measure performance directly on the same hardware platform and, whenever possible, on the same operating system, and benchmark suite. Additionally, we tabulate and compare the performance results reported in the literature.

We focus on the aggregate cost of CFI enforcement. For a detailed survey of the performance cost of protecting backwards control flows from callees to callers we refer to the recent, comprehensive survey by Dang et al. [16].

A. Measured CFI Performance

a) Selection Criteria: It is infeasible to replicate the reported performance overheads for all major CFI mechanisms. Many implementations are not publicly available or require substantial modification to run on modern versions of Linux or Windows. We therefore focus on recent, publicly available, compiler-based CFI mechanisms.

Several compiler-based CFI mechanisms share a common lineage. LLVM-CFI, for instance, improves upon IFCC, π CFI improves upon MCFI, and VTI is an improved version of SafeDispatch. In those cases, we opted to measure the latest available version and rely on reported performance numbers for older versions.

b) Method: Most authors use the SPEC CPU2006 benchmarks to report the overhead of their CFI mechanism. We follow this trend in our own replication study. All benchmarks were compiled using the `-O2` optimization level. The benchmarking system was a Dell PowerEdge T620 dual processor server having 64GiB of main memory and two Intel Xeon E5-2660 CPUs running at 2.20 GHz. To reduce benchmarking noise, we ran the tests on an otherwise idle system and disabled all dynamic frequency and voltage scaling features. Whenever possible, we benchmark the implementations under 64-bit Ubuntu Linux 14.04.2 LTS. Since CFGuard is part of Microsoft Visual C++ Compiler, MSVC, we used the latest version of MSVC to compile and run SPEC CPU2006 on a pristine 64-bit Windows 10 installation. We report the geometric mean overhead averaged over three benchmark runs using the reference inputs in Table I.

Some of the CFI mechanisms we benchmark required link-time optimization, LTO, which allows the compiler to analyze and optimize across compilation units. LLVM-CFI and VTI

both require LTO, so for these mechanisms, we report overheads relative to a baseline SPEC CPU2006 run that also had LTO enabled. The increased optimization scope enabled by LTO can allow the compiler to perform additional optimizations such as de-virtualization to lower the cost of CFI enforcement. On the other hand, LTO is less practical than traditional, separate compilation, e.g., when compiling large, complex code bases. To measure the π CFI mechanism, we applied the author’s patches for 7 of the SPEC CPU2006 benchmarks to remove coding constructs that are not handled by π CFI’s control-flow graph analysis [49]. For example, the xalancbmk benchmark contains code that casts an object instance to its sibling class which can cause a CFI violation. Therefore we applied a 4-line patch before running the benchmark with LLVM-CFI or VTI enabled. VTI was run in interleaved vtable mode which provides the best performance according to its authors [7].

c) Results: Our performance experiments show that recent, compiler based CFI mechanisms have mean overheads in the low single digit range. Such low overhead is well within the threshold for adoption specified by [67] of 5%. This dispenses with the concern that CFI enforcement is too costly in practice compared to alternative mitigations including those based on randomization [37]. Indeed, mechanisms such as CFGuard, LLVM-CFI, and VTV are implemented in widely used compilers, offering some level of CFI enforcement to practitioners.

We expect CFI mechanisms that are limited to virtual method calls—VTV, VTI, LLVM-CFI—to have lower mean overheads than those that also protect indirect function calls such as IFCC. The return protection mechanism used by MCFI should introduce additional overhead, and π CFI’s runtime policy ought to result in a further marginal increase in overhead. In practice our results show that LLVM-CFI and VTI are the fastest, followed by CFGuard, π CFI, and VTV. The reported numbers for IFCC when run in *single* mode show that it achieves -0.3%, likely due to cache effects. Although our measured overheads are not directly comparable with those reported by the authors of the seminal CFI paper, we find that researchers have managed to improve the precision while lowering the cost⁵ of enforcement as the result of a decade worth of research into CFI enforcement.

The geometric mean overheads do not tell the whole story, however. It is important to look closer at the performance impact on benchmarks that execute a high number of indirect branches. Protecting the xalancbmk, omnetpp, and povray C++ benchmarks with CFI generally incurs substantial overheads. All benchmarked CFI mechanisms had above-average overheads on xalancbmk. LLVM-CFI and VTV, which take virtual call semantics into account, were particularly affected. On the other hand, xalancbmk highlights the merits of the recent virtual table interleaving mechanism of VTI which has a relatively low 3.7% overhead (vs. 1.4% reported) on this challenging benchmark.

Although povray is written in C++, it makes few virtual method calls [76]. However, it performs a large number of indirect calls. The CFI mechanisms which protect indirect calls— π CFI, and CFGuard—all incur high performance overheads

on povray. Sjeng and h264ref also include high number of indirect calls which again result in non-negligible overheads particularly when using π CFI with tail calls disabled to improve CFG precision. The hammer, namd, and bzip2 benchmarks on the other hand show very little overhead as they do not execute a high number of forward indirect branches of any kind. Therefore these benchmarks are of little value when comparing the performance of various CFI mechanisms.

Overall, our measurements generally match those reported in the literature. The authors of VTV [68] only report overheads for the three SPEC CPU2006 benchmarks that were impacted the most. Our measurements confirm the authors’ claim that the runtimes of the other C++ benchmarks are virtually unaffected. The leftmost π CFI column should be compared to the reported column for π CFI. We measured overheads higher than those reported by Niu and Tan. Both gobmk and xalancbmk show markedly higher performance overheads in our experiments; we believe this is in part explained by the fact that Niu and Tan used a newer Intel Xeon processor having an improved branch predictor [59] and higher clock speeds (3.4 vs 2.2 GHz).

We ran π CFI in both normal mode and with tail calls disabled. The geometric mean overhead increased by 1.9% with tail calls disabled. Disabling tail calls in turn increases the number of equivalence classes on each benchmark Figure 5. This is a classic example of the performance/security precision trade-off when designing CFI mechanisms. Implementers can choose the most precise policy within their performance target. CFGuard offers the most efficient protection of forward indirect branches whereas π CFI offers higher security at slightly higher cost.

B. Reported CFI Performance

The right-hand side of Table I lists reported overheads on SPEC CPU2006 for CFI mechanisms that we do not measure. IFCC is the first CFI mechanism implemented in LLVM which was later replaced by LLVM-CFI. MCFI is the precursor to π CFI. PathArmor is a recent CFI mechanism that uses dynamic binary rewriting and a hardware feature, Last Branch Record, LBR [34] register, that traces the 16 most recently executed indirect control-flow transfers. Lockdown is a pure dynamic binary translation approach to CFI that includes precise enforcement of returns using a shadow stack. CCFI is a compiler-based approach which stores a cryptographically-secure hash-based message authentication code, HMAC, next to each pointer. Checking the HMAC of a pointer before indirect branches avoids a static points-to analysis to generate a CFG. ROPecker is a CFI mechanism that uses a combination of offline analysis, traces recorded by the LBR register, and emulation in an attempt to detect ROP attacks. Finally, the bin-CFI approach uses static binary rewriting like the original CFI mechanism; bin-CFI is notable for its ability to protect stripped, position-independent ELF binaries that do not contain relocation information.

The reported overheads match our measurements: xalancbmk and povray impose the highest overheads—up to 15% for ROPecker, which otherwise exhibits low overheads, and 1.7x for CryptoCFI. The interpreter benchmark, perlbench, executes a high number of indirect branches, which leads to high overheads, particularly for Lockdown, PathArmor, and

⁵Non-CFI related hardware improvements, such as better branch prediction [59], also help to reduce performance overhead.

Table I: Measured and reported CFI performance overhead (%) on the SPEC CPU2006 benchmarks. The programming language of each benchmark is indicated in parenthesis: C(C), C++(+), Fortran(F). CF in a cell indicates we were unable to build and run the benchmark with CFI enabled. Blank cells mean that no results were reported by the original authors or that we did not attempt to run the benchmark. Cells with bold fonts indicate 10% or more overhead.

Benchmark	Measured Performance						Reported Performance									
Citations	VTV	LLVM-CFI	VTI	CFGuard	π CFI	π CFI	VTV	VTI	π CFI	IFCC	MCFI	PathArmor	Lockdown	CCFI	ROPecker	bin-CFI
Options	[68]	[12]	[7]	[41]	[51]	no tail call				[68]	[49]	[71]	[55]	[39]	[11]	[77]
		LTO	LTO				LTO			LTO						
400.perlbench(C)					8.2	5.3			5.0	1.9	5.0	15.0	150.0		5.0	12.0
401.bzip2(C)				-0.3	1.2	0.8			1.0		1.0	0.0	8.0	5.0	0.0	-9.0
403.gcc(C)					6.1	10.5			4.5		4.5	9.0	50.0		3.0	4.5
429.mcf(C)				0.5	4.0	1.8			4.0		4.0	1.0	2.0	10.0	1.0	0.0
445.gobmk(C)				-0.2	11.4	11.8			7.5		7.0	0.0	43.0	50.0	1.0	15.0
456.hmmcr(C)				0.7	0.1	-0.1			0.0		0.0	1.0	3.0	10.0	0.0	-0.5
458.sjeng(C)				3.4	8.4	11.9			5.0		5.0	0.0	80.0	40.0	0.0	-2.5
464.h264ref(C)				5.4	7.9	8.3			6.0		6.0	1.0	43.0	45.0	1.0	28.0
462.libquantum(C)					-3.0	-1.0			-0.3		0.0	3.0	5.0	10.0	0.0	-0.5
471.omnetpp(+)	5.8	-1.9	CF	3.8	6.7	18.8	8.0	1.2	5.0	-1.2	5.0				2.0	45.0
473.astar(+)	3.6	-0.3	1.6	0.1	2.0	2.9	2.4	0.1	4.0	-0.2	3.5		17.0	75.0	0.0	14.0
483.xalancbmk(+)	24.0	7.1	3.7	5.5	10.3	17.6	19.2	1.4	7.0	3.1	7.0		118.0	170.0	15.0	
410.bwaves(F)													1.0			
416.gamess(F)													11.0			
433.mile(C)				2.0	-1.7	1.4			2.0		2.0	4.0	8.0			2.5
434.zeusmp(F)													0.0			
435.gromacs(C,F)													1.0			
436.cactusADM(C,F)													0.0			
437.leslie3d(F)													1.0			
444.namd(+)	-0.1	-0.2	-0.3	0.1	-0.3	-0.5			-0.5	-0.2	-0.5		3.0			-2.0
447.dealII(+)	0.7	CF	CF	-0.1	5.3	4.4			4.5	-2.2	4.5					
450.soplex(+)	0.5	0.5	-0.6	2.3	-0.7	0.9	-0.7	-4.0	-1.7	-4.0			12.0			3.5
453.povray(+)	-0.6	1.5	2.0	10.8	11.3	17.4			10.5	0.2	10.0		90.0			37.0
454.calculix(C,F)													3.0			
459.gemsFDTD(F)													7.0			
465.tonto(F)													19.0			
470.lbm(C)				4.2	-0.2	-0.5			1.0		1.0	0.0	2.0			-2.5
482.sphinx3(C)				-0.1	0.7	2.4			1.5		1.5	3.0	8.0			0.5
Geo Mean	4.6	1.1	1.3	2.3	4.0	5.8	9.6	0.5	3.2	-0.3	2.9	3.0	20.0	45.0	2.6	8.5

bin-CFI. Lockdown and CCFI offer a higher level of security than any other CFI mechanism surveyed in this section, but also impose the highest performance impact.

Looking at CFI mechanisms that do not require recompilation—PathArmor, Lockdown, ROPecker, and bin-CFI we see that the mechanisms that only check the contents of the LBR before system calls (PathArmor and ROPecker) report lower mean overheads than approaches that comprehensively instrument indirect branches (Lockdown and bin-CFI) in existing binaries. More broadly, comparing compiler-based mechanisms with binary-level mechanisms, we see that compiler-based approaches are typically as efficient as the binary-level mechanisms that trace control flows using the LBR although compiler-based mechanisms protect do not limit protection to a short window of recently executed branches. More comprehensive binary-level mechanisms, Lockdown and bin-CFI generally have higher overheads than compiler-based equivalents. On the other hand, Lockdown shows the advantage of binary translation: almost any program can be analyzed and protected, independent from the compiler and source code. Also note that the Lockdown overhead includes a dynamic shadow stack that none of the other mechanisms have.

Although we cannot directly compare the reported overheads of bin-CFI with our measured overheads for CFGuard, the mechanisms enforce CFI policies of roughly similar precision (compare Figure 3i and Figure 3w). CFGuard, however, has a substantially lower performance overhead. This is not

surprising given that compilers operate on a high-level program representation that is more amenable to static program analysis and optimization of the CFI instrumentation. On the other hand, compiler-based CFI mechanisms are not strictly faster than binary-level mechanisms, CCFI has the highest reported overheads by far although it is implemented in the LLVM compiler.

Table II surveys CFI approaches that do not report overheads using the SPEC CPU2006 benchmarks like the majority of recent CFI mechanisms do. Some authors, use an older version of the SPEC benchmarks [1, 44] whereas others evaluate performance using, e.g., web browsers [35, 75], or web servers [55, 73]. Although it is valuable to quantify overheads of CFI enforcement on more modern and realistic programs, it remains helpful to include the overheads for SPEC CPU2006 benchmarks.

C. Discussion

As Table I shows, authors working in the area of CFI seem to agree to evaluate their mechanisms using the SPEC CPU2006 benchmarks. There is, however, less agreement on whether to include both the integer and floating point subsets. The authors of Lockdown report the most complete set of benchmark results covering both integer and floating point benchmarks and the authors of bin-CFI, π CFI, and MCFI include most of the integer benchmarks and a subset of the floating point ones. The authors of VTV and IFCC only report subsets of integer and

Table II: CFI performance overhead (%) reported from previous publications. A label of ^C indicates we computed the geometric mean overhead over the listed benchmarks, otherwise it is the published average.

	ROPGuard [24]	SafeDispatch [35]	CCFIR [75]	kBouncer [53]	OCFI [44]	CFIMon [73]	Original CFI [1]
Benchmarks	PCMark Vantage, NovaBench, 3DMark06, Peacekeeper, Sunspider, SuperPI 16M	Octane, Kraken, Sunspider, Balls, linelayout, HTML5	SPEC2kINT, SPEC2kFP, SPEC2k6INT	wmplayer, Internet Explorer, Adobe Reader	SPEC2k	httpd, Exim, Wu-ftpd, Memcached	SPEC2k
Overhead	0.48%	2%	2.08% ^C	4% ^C	4.7%	6.1%	16.0%

floating point benchmarks where their solutions introduce non-negligible overheads. Except for CFI mechanisms focused on a particular type of control flows such as virtual method calls, authors should strive to report overheads on the full suite of SPEC CPU2006 benchmarks. In case there is insufficient time to evaluate a CFI mechanism on all benchmarks, we strongly encourage authors to focus on the ones that are challenging to protect with low overheads. These include perlbench, gcc, gobmk, sjeng, omnetpp, povray, and xalancbmk. Additionally, it is desirable to supplement SPEC CPU2006 measurements with measurements for large, frequently targeted applications such as web browsers and web servers.

Although “traditional” CFI mechanisms (e.g., those that check indirect branch targets using a pre-computed CFG) can be implemented most efficiently in a compiler, this does not automatically make such solutions superior to binary-level CFI mechanisms. The advantages of the latter type of approaches include, most prominently, the ability to work directly on stripped binaries when the corresponding source is unavailable. This allows CFI enforcement to be applied independently of the code producer and therefore puts the performance/security trade off in the hands of the end-users or system administrators. Moreover, binary-level solutions naturally operate on the level of entire program modules irrespective of the source language, compiler, and compilation mode that was used to generate the code. Implementers of compiler-based CFI solutions on the other hand must spend additional effort to support separate compilation or require LTO operation which, in some instances, lowers the usability of the CFI mechanism [67].

V. CROSS-CUTTING CONCERNS

Recent trends in CFI research target improving CFI in directions beyond new analysis or enforcement algorithms. Some approaches have sought to increase CFI protection coverage to include just-in-time code and operating system kernels. Others leverage advances in hardware to improve performance or enable new enforcement strategies. We discuss these research directions in the CFI landscape which cross-cut the traditional categories of performance and security.

Hardware-supported CFI. Modern processors offer several hardware security-oriented features. Data Execution Prevention is a classical example of how a simple hardware feature can eliminate an entire class of attacks. Many processors also support AES encryption, random number generation, secure enclaves, and array bounds checking via instruction set extensions.

Researchers have explored architectural support for CFI

enforcement [2, 18] with the goal of lowering performance overheads. A particular advantage of these solutions is that backwards edges can be protected by a fully-isolated shadow stack with an average overhead of just 2% for protection of forward and backwards edges. This stands in contrast to the average overheads for software-based shadow stacks which range from 3 to 14% according to Dang et al. [16].

There have also been efforts to repurpose existing hardware mechanisms to implement CFI [11, 53, 71, 74]. Pappas et al. [53] were first to demonstrate a CFI mechanism using the 16-entry LBR branch trace facility of Intel x86 processors. The key idea in their kBouncer solution is to check the control flow path that led up to a potentially dangerous system call by inspecting the LBR; a heuristic was used to distinguish execution traces induced by ROP chains from legitimate execution traces. ROPEcker by Cheng et al. [11] subsequently extended LBR-based CFI enforcement to also emulate what code would execute past the system call. While these approaches offer negligible overheads and do not require recompilation of existing code, subsequent research showed that carefully crafted ROP attacks can bypass both of these mechanisms. Pinghai et al.’s [74] CFIGuard mechanism uses the LBR feature in conjunction with hardware performance counters to heuristically detect ROP attacks. [73] used the branch trace store, which records control flow transfers to a buffer in memory, rather than the LBR for CFI enforcement. Mashtizadeh et al. [39]’s CCFI uses the Intel AES-NI instruction set to compute cryptographically-enforced hash-based message authentication codes, HMACs, for pointers stored in attacker-observable memory. By verifying HMACs before pointers are used, CCFI prevents control-flow hijacking. Finally, Mohan et al. [44] leverage Intel’s MPX instruction set extension by re-casting the problem of CFI enforcement as a bounds checking problem over a randomized CFG.

Protecting Just-in-time Compiled Code. Like other defenses, it is important that CFI is deployed comprehensively since adversaries only have to find a single unprotected indirect branch to compromise the entire process. Some applications contain just-in-time, JIT, compilers that dynamically emit machine code for managed languages such as Java and JavaScript. Niu and Tan [50] presented RockJIT, a CFI mechanism that specifically targets the additional attack surface exposed by JIT compilers. RockJIT faces two challenges unique to dynamically generated code: (i) the code heap used by JIT compilers is usually simultaneously writable and executable to allow important optimizations such as inline caching [33] and on-stack replacement, (ii) computing the control-flow graphs for dynamic languages during execution without imposing

substantial performance overheads. RockJIT solves the first challenge by replacing the original heap with a shadow code heap which is readable and writable but not executable and by introducing a sandboxed code heap which is readable and executable, but not writable. To avoid increased memory consumption, RockJIT maps the sandboxed code heap and the shadow heap to the same physical memory pages with different permissions. RockJIT addresses the second challenge in part by modifying the JIT compiler to emit meta-data about indirect branches in the generated code and in part by enforcing a coarse-grained CFI policy on JITed code which avoids the need for static analysis. The authors argue that a less precise CFI policy for JITed code is acceptable as long as (i) the host application is protected by a more precise policy and (ii) JIT-compiled code does not allow adversaries to make system calls.

Protecting Operating System Kernels. In monolithic kernels, all kernel software is running at the same privilege levels and any memory corruption can be fatal for security. A kernel is vastly different from a user-space application as it is directly exposed to the underlying hardware and an attacker in that space has access to privileged instructions that may change interrupts, page table structures, page table permissions, or privileged data structures. KCoFI [15] introduces a first CFI policy for commodity operating systems and considers these specific problems. The CFI mechanism is fairly coarse-grained: any indirect function call may target any valid functions and returns may target any call site (instead of executable bytes).

Protecting Interpreters. Control-flow integrity for interpreters faces similar challenges as Just-in-time compilers. Interpreters are widely deployed, e.g., two major web browsers, Internet Explorer and Safari, rely on mixed-mode execution models that interpret code until it becomes “hot” enough for just-in-time compilation [3], and some Desktop software, too, is interpreted, e.g., Dropbox’ client is implemented in Python. We have already described the “worst-case” interpreters pose to CFI from a security perspective: even if the interpreter’s code is protected by CFI, its actual functionality is determined by a program in data memory. This separation has two important implications: (i) static analysis for an interpreter dispatch routine will result in an over-approximation, and (ii) ideal non-control data attack through manipulating interpreted programs that reside in writeable data memory.

Interpreters are inherently dynamic, which on the one hand means, CFI for interpreters could rely on precise dynamic points-to information, but on the other hand also indicates problems to build a complete control-flow graph for such programs (eval further complicates this). Any CFI mechanism for interpreters needs to address this challenge.

VI. CONCLUSIONS

Control-Flow Integrity substantially raises the bar against attacks that exploit memory corruption vulnerabilities to execute arbitrary code. In the decade since its inception, researchers have made major advances and explored a great number of materially different mechanisms and implementation choices. Comparing and evaluating these mechanisms is non-trivial and most authors only provide ad-hoc security and performance evaluations. A prerequisite to any systematic evaluation is a

set of well-defined metrics. In this paper, we have proposed metrics to qualitatively (based on the underlying analysis) and quantitatively (based on a practical evaluation) assess the security benefits of a representative sample of CFI mechanisms. Additionally, we have evaluated the performance trade-offs and have surveyed cross-cutting concerns and their impacts on the applicability of CFI.

Our systematization serves as an entry point and guide to the now voluminous and diverse literature on control-flow integrity. Most importantly, we capture the current state of the art in terms of precision and performance. We report large variations in the forward and backward edge precision for the evaluated mechanisms with corresponding performance overhead: higher precision results in (slightly) higher performance overhead.

We hope that our unified nomenclature will gradually displace the ill-defined qualitative distinction between “fine-grained” and “coarse-grained” labels that authors apply inconsistently across publications. Our metrics provide the necessary guidance and data to compare CFI implementations in a more nuanced way. This helps software developers and compiler writers gain appreciation for the performance/security trade-off between different CFI mechanisms. For the security community, this work provides a map of what has been done, and highlights fertile grounds for future research. Beyond metrics, our unified nomenclature allows clear distinctions of mechanisms. These metrics, if adopted, are useful to evaluate and describe future improvements to CFI.

REFERENCES

- [1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2005.
- [2] O. Arias, L. Davi, M. Hanreich, Y. Jin, P. Koeberl, D. Paul, A.-R. Sadeghi, and D. Sullivan. HAFX: Hardware-assisted flow integrity extension. In *Annual Design Automation Conference (DAC)*, 2015.
- [3] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.
- [4] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–341, oct 1996.
- [5] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, jun 1973. ISSN 00010782.
- [6] T. Bletsch, X. Jiang, and V. Freeh. Mitigating code-reuse attacks with control-flow locking. In *Annual Computer Security Applications Conference (ACSAC)*, New York, New York, USA, 2011.
- [7] D. Bounov, R. Kici, and S. Lerner. Protecting C++ dynamic dispatch through vtable interleaving. In *Symposium on Network and Distributed System Security (NDSS)*, 2016. To appear.
- [8] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.
- [9] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015.
- [10] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.
- [11] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A generic and practical approach for defending against ROP attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
- [12] P. Collingbourne. LLVM — control flow integrity, 2015. <http://clang.llvm.org/docs/ControlFlowIntegrity.html>.
- [13] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A.-R. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.

- [14] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy*, 2014.
- [15] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete control-flow integrity for commodity operating system kernels. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [16] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and stack canaries. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2015.
- [17] L. Davi, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nürnberger, and A.-R. Sadeghi. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Symposium on Network and Distributed System Security (NDSS)*, 2012.
- [18] 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 *Annual Design Automation Conference (DAC)*, 2014.
- [19] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.
- [20] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *European Conference on Object-Oriented Programming (ECOOP)*, 1995.
- [21] E. H. Debaere and J. M. van Campenhout. *Interpretation and instruction path coprocessing*. Computer systems. MIT Press, 1990. ISBN 978-0-262-04107-2.
- [22] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point: On the effectiveness of code pointer integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [23] 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.
- [24] I. Fratric. ROPGuard: Runtime prevention of return-oriented programming attacks. http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf, 2012.
- [25] R. Gawlik and T. Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.
- [26] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [27] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, nov 2001.
- [28] B. Hackett and A. Aiken. How is aliasing used in systems software? *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 69–80, 2006.
- [29] B. Hardekopf and C. Lin. The ant and the grasshopper. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation - PLDI '07*, volume 42, page 290, New York, New York, USA, 2007. ACM Press. ISBN 9781595936332. doi: 10.1145/1250734.1250767. URL <http://portal.acm.org/citation.cfm?id=1250767http://portal.acm.org/citation.cfm?doid=1250734.1250767>.
- [30] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 289–298. IEEE, apr 2011. ISBN 978-1-61284-356-8. doi: 10.1109/CGO.2011.5764696. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5764696>.
- [31] M. Hind. Pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '01*, pages 54–61, New York, New York, USA, 2001. ACM Press. ISBN 1581134134. doi: 10.1145/379605.379665. URL <http://portal.acm.org/citation.cfm?doid=379605.379665>.
- [32] M. Hind and A. Pioli. Which pointer analysis should I use? *ACM SIGSOFT Software Engineering Notes*, 25(5):113–123, sep 2000.
- [33] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1994.
- [34] Intel Inc. Intel 64 and IA-32 architectures. software developer’s manual, 2013.
- [35] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.
- [36] Kogge. An Architectural Trail to Threaded-Code Systems. *Computer*, 15(3):22–32, mar 1982. ISSN 0018-9162. doi: 10.1109/MC.1982.1653970. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1653970>.
- [37] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.
- [38] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? *Compiler Construction*, pages 47–64, 2006.
- [39] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: cryptographically enforced control flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [40] Microsoft. Data Execution Prevention (DEP). <http://support.microsoft.com/kb/875352/EN-US/>, 2006.
- [41] Microsoft. Visual studio 2015 — compiler options — enable control flow guard, 2015. <https://msdn.microsoft.com/en-us/library/dn919635.aspx>.
- [42] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. *ACM SIGSOFT Software Engineering Notes*, 27(4):1, 2002.
- [43] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets. In *ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, 2001.
- [44] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [45] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.
- [46] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. CETS: Compiler enforced temporal safety for C. In *ISMM'10*, 2010.
- [47] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-642-08474-4. doi: 10.1007/978-3-662-03811-6.
- [48] B. Niu and G. Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.
- [49] B. Niu and G. Tan. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
- [50] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.
- [51] B. Niu and G. Tan. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, 2015.
- [52] B. Niu and G. Tan. Mcfi readme. <https://github.com/mcfi/MCFI/blob/master/README.md>, 2015.
- [53] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *USENIX Security Symposium*, 2013.
- [54] PaX-Team. PaX ASLR (Address Space Layout Randomization). <http://pax.grsecurity.net/docs/aslr.txt>, 2003. URL <http://pax.grsecurity.net/docs/aslr.txt>.
- [55] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, 2015.
- [56] J. Pewny and T. Holz. Control-flow restrictor: Compiler-based CFI for iOS. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.
- [57] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [58] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions*

- [59] E. Rohou, B. N. Swamy, and A. Sez nec. Branch prediction and the performance of interpreters: Don't trust folklore. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015.
- [60] A. Rountev, S. Kagan, and M. Gibas. Evaluating the imprecision of static analysis. In *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '04*, page 14, New York, New York, USA, 2004. ACM Press. ISBN 1581139101. doi: 10.1145/996821.996829. URL <http://portal.acm.org/citation.cfm?doid=996821.996829>.
- [61] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, jan 2003. ISSN 0733-8716. doi: 10.1109/JSAC.2002.806121. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1159651>.
- [62] F. Schuster, T. Tendyck, C. Liebhchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.
- [63] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS'07*, 2007.
- [64] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis*. Prentice Hall, 1981.
- [65] Y. Smaragdakis and G. Balatsouras. Pointer Analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015. ISSN 2325-1107. doi: 10.1561/25000000014. URL <http://www.nowpublishers.com/article/Details/PGL-014>.
- [66] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well. *ACM SIGPLAN Notices*, 46(1):17, jan 2011.
- [67] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [68] 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 Symposium*, 2014.
- [69] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. *ACM SIGPLAN Notices*, 35(10):281–293, oct 2000.
- [70] A. van de Ven and I. Molnar. Exec shield. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.
- [71] V. van der Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. PathArmor: Practical ROP protection using context-sensitive CFI. In *ACM Conference on Computer and Communications Security (CCS)*, 2015.
- [72] Z. Wang and X. Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE S&P'10*, 2010.
- [73] Y. Xia, Y. Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP Conference on Dependable Systems and Networks (DSN)*, 2012.
- [74] P. Yuan, Q. Zeng, and X. Ding. Hardware-assisted fine-grained code-reuse attack detection. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.
- [75] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity & randomization for binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.
- [76] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)*, 2015.
- [77] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

This appendix presents the full quantitative security results. An abbreviated version of these was presented in Section III-B. The full results are presented here for completeness. Table III contains the number of equivalence sets for each benchmark and every CFI mechanism that we evaluated. Figure 7 contains the full set of box and whisker plots. As this data is fundamentally three dimensional, these plots are the best way to display it. As a final note, the holes in this data reflect the fact that the CFI mechanisms that we evaluated cannot run the full set of SPEC CPU2006 benchmarks. This greatly complicates the task of comparatively evaluating them, as there is only a narrow base of programs that all the CFI mechanisms run.

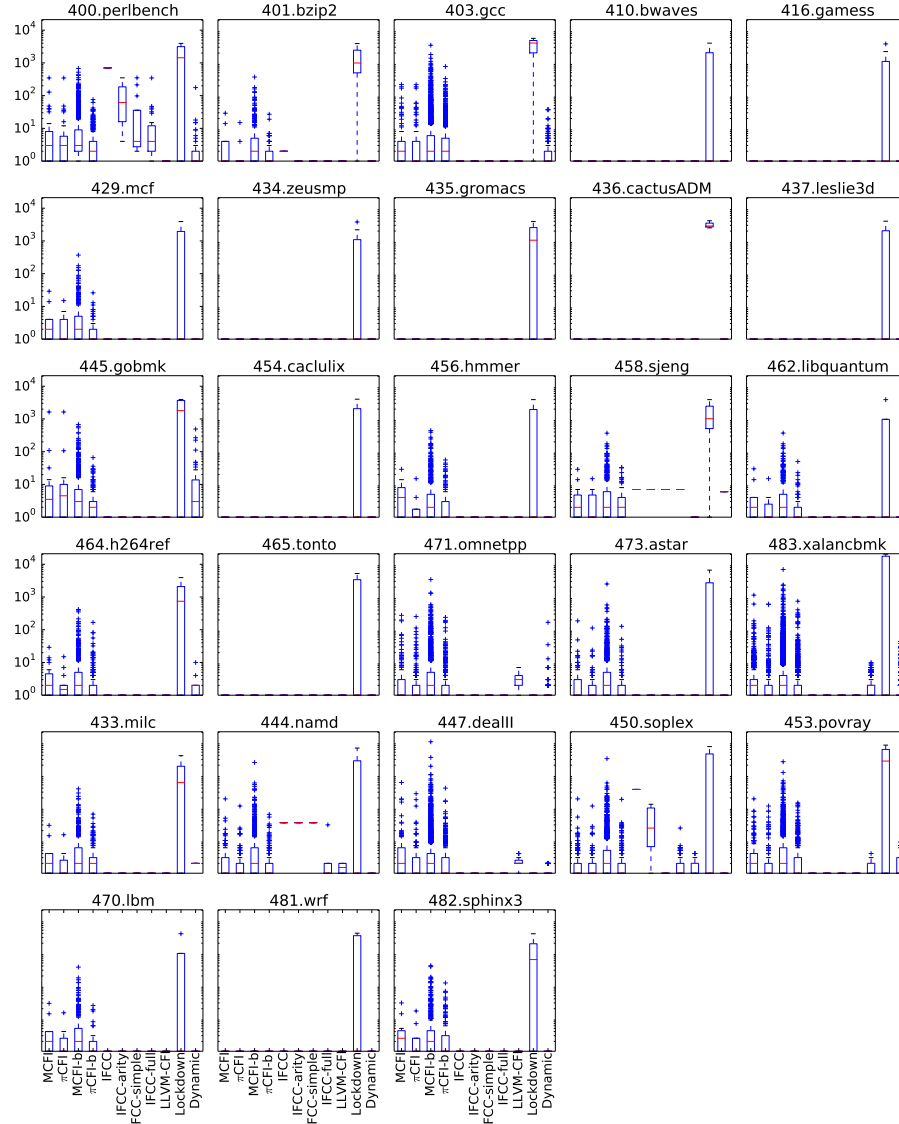


Figure 7: Whisker plot of equivalence classes size for all SPEC CPU2006 benchmarks across all implementations (smaller is better).

Benchmark	CFI Implementation												LLVM-CFI	Lockdown	Dynamic
	MCFI	π CFI back edge	MCFI edge no tail call	π CFI	MCFI	π CFI	MCFI forward edge no tail call	π CFI	IFCC						
									single	arity	simplified	full			
400.perlbench	978	310	1192	429	38	30	38	30	1	6	12	40	0	4	83
401.bzip2	484	82	489	86	14	10	14	10	1	2	2	2	0	3	12
403.gcc	2219	1260	3282	1836	98	90	98	90	0	0	0	0	0	3	197
429.mcf	475	96	475	96	12	8	12	8	0	0	0	0	0	3	0
445.gobmk	922	283	1075	230	21	17	21	17	0	0	0	0	0	4	0
456.hmmer	663	134	720	147	14	9	14	9	0	0	0	0	0	4	9
458.sjeng	540	119	557	125	13	9	13	9	1	1	1	1	0	3	1
462.libquantum	495	88	519	102	12	8	12	8	1	1	1	1	0	4	0
464.h264ref	773	285	847	327	21	15	21	15	0	0	0	0	0	4	59
471.omnetpp	1693	581	1784	624	357	321	357	321	0	0	0	0	114	0	224
473.astar	1096	226	1108	237	166	150	166	150	0	0	0	0	1	6	1
483.xalancbmk	6161	2381	7162	2869	1534	1200	1534	1200	0	0	0	0	2197	6	1402
433.milc	602	169	628	180	13	9	13	9	0	0	0	0	0	4	3
444.namd	1080	217	1087	224	166	150	166	150	1	1	1	5	4	6	12
447.dealII	2952	817	3468	896	293	258	293	258	0	0	0	0	43	0	95
450.soplex	1444	432	1569	479	321	291	321	291	1	7	0	186	41	6	157
453.povray	1748	650	1934	743	218	204	218	204	0	0	0	0	29	6	49
470.lbm	465	70	470	74	12	8	12	8	0	0	0	0	0	4	0
482.sphinx3	633	239	677	257	13	9	13	9	0	0	0	0	0	4	2

Table III: Full Quantitative Security results for Number of Equivalence Classes