Integrating Static Analyses for High-Precision Control-Flow Integrity

Florian Kasten
Fraunhofer AISEC
Garching near Munich, Germany
florian.kasten@aisec.fraunhofer.de

Philipp Zieris
Fraunhofer AISEC
Garching near Munich, Germany
philipp.zieris@aisec.fraunhofer.de

Julian Horsch
Fraunhofer AISEC
Garching near Munich, Germany
julian.horsch@aisec.fraunhofer.de

ABSTRACT

Memory corruptions are still one of the most prevalent and severe security vulnerabilities in today's programs. For this reason, several techniques for mitigating software vulnerabilities exist and are used in production systems. An important mitigation involves the prevention of invalid control flow transfers. Attackers often corrupt function pointers to subvert a forward-edge in a program's call graph. Forward-edges can be protected using Control-Flow Integrity (CFI), for which practical implementations already exist. However, current CFI implementations are often imprecise, allowing more control flow transfers than necessary. This often leaves sufficient leeway for an attacker to successfully exploit a program. This paper presents High-Precision CFI (HPCFI), a concept and implementation for precise forward-edge CFI protection of indirect calls in C and C++ programs using a combination of type analysis and static data-flow analysis for determining valid forward-edges. HPCFI is implemented as LLVM compiler passes that perform a precise type analysis and utilize the Static Value-Flow (SVF) framework to conduct a static data-flow analysis. The combination of type analysis and static data-flow analysis offers higher precision than conventional heuristic-based approaches. Our evaluation, using all compatible benchmarks from SPEC CPU 2017, demonstrates that HPCFI can be effectively applied to large projects with an average performance overhead of only 1.3%, while improving the precision of established CFI mechanisms, such as Clang CFI, by up to 99% and 40% on average.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Control-Flow Integrity, Static Data-Flow Analysis, Whole-Program Analysis, Pointer Analysis, LLVM, SVF, Multi-Layer Type Analysis

ACM Reference Format:
Florian Kasten, Philipp Zieris, and Julian Horsch. 2024. Integrating Static Analyses for High-Precision Control-Flow Integrity. In *The 27th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2024), September 30–October 02, 2024, Padua, Italy*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3678890.3678920

RAID 2024, September 30–October 02, 2024, Padua, Italy
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0959-3/24/09
https://doi.org/10.1145/3678890.3678920

1 INTRODUCTION

One of the most prevalent security vulnerabilities in today's systems are memory corruptions, which has also been recognized in a 2024 report released by the White House [38]. This issue is corroborated by the fact that 70% of all vulnerabilities assigned a CVE by Microsoft each year, as well as 70% of high-severity security bugs in the Chromium project, are related to memory safety [12, 24]. In 2023, MITRE identified out-of-bounds writes as the most dangerous software weakness [25] for the third consecutive year. Memory corruptions arise from programming errors in unsafe languages like C and C++, where data on a program's stack or heap can be overwritten by illegitimate values. This may enable attackers to manipulate the intended control flow of a program using techniques such as Return-Oriented Programming (ROP) [35], Jump-Oriented Programming (JOP) [6], and Call-Oriented Programming (COP) [10].

The intended control flow of function calls in a program can be represented by a call graph, where nodes reflect the program's functions and edges represent function calls (forward-edges) and function returns (backward-edges) [1]. While safeguarding backward-edges through the protection of stack-based return addresses is already widely employed in production programs (e.g., stack canaries [26]), the protection of forward-edges is less common and more intricate. An attacker may, for instance, subvert a forward-edge by overwriting a function pointer used in an indirect call. Control-Flow Integrity (CFI) is a security mechanism designed to ensure that a program adheres to its intended call graph. Typical software-based CFI mechanisms operate in two steps: First, an analysis determines the call graph that should be enforced; second, instructions are inserted into the program to enforce that call graph at runtime [7]. While some CFI mechanisms aim to protect both forward- and backward-edges [1, 2, 5, 29, 36], more recent approaches primarily focus on protecting the forward-edges of a call graph [13, 19, 23, 39]. These CFI mechanisms are meant to be combined with a suitable backward-edge protection such as shadow stacks, which protect against stack buffer overflows by maintaining a secure copy of the call stack's return addresses [1, 8, 14]. How the call graph is constructed determines the precision and, thus, the effectiveness of the CFI mechanism.

A simple approach to building a call graph for forward-edge CFI is to allow any function to be called from any indirect call site. For instance, this approach is implemented in Intel Control-Flow Encorcement Technology (CET), a hardware-based CFI mechanism [36]. Compared to an unprotected program, in which an attacker can manipulate a code pointer to jump to any code location, an attacker can now only jump to the beginning of functions with CET. Microsoft Control Flow Guard (Microsoft CFG) refines this approach by only allowing indirect calls to functions whose addresses have been taken [19]. Both CET and Microsoft CFG are considered *coarse-grained* CFI mechanisms and are deemed ineffective due to their imprecision. They often provide attackers with sufficient leeway to successfully exploit a program, even when restricted to jumping only to function beginnings [9, 16]. More precise and *fine-grained* CFI solutions, such as Clang CFI [39], construct a *type-based* call graph by considering function signatures at indirect call sites and allowing calls only to functions with matching signatures [39]. Similarly, TypeDive [23] utilizes a Multi-Layer Type Analysis (MLTA) to construct a call graph which additionally takes into account the type hierarchy of different compound types (e.g., structs) from which a function pointer was loaded.

However, even though type-based CFI mechanisms like Clang CFI and TypeDive already offer better precision and security guarantees than coarse-grained mechanisms, they still permit many invalid edges in the call graph. For example, in SPEC CPU 2017 [17], the `gcc` benchmark contains over 1,000 functions with the signature `(struct rtx_def*, ...)` → `struct rtx_def*`, resulting in sets of 1,000 possible targets in Clang CFI for all call sites to a function with this signature.

In this paper, we improve the precision of purely type-based CFI by combining the call graphs generated through a type-based analysis and a multi-layer type analysis with a call graph generated through a whole-program pointer analysis tailored specifically for CFI. We create a novel and highly precise call graph by intersecting these three call graphs and integrate it into High-Precision CFI (HPCFI), a forward-edge CFI mechanism implemented as an LLVM compiler pass. The compiler pass constructs the three call graphs during Link Time Optimization (LTO), where all source code modules are available. This allows the pass to construct a pointer analysis-based call graph of the entire program. The resulting call graph is then intersected with a type-based call graph similar to the one used by Clang CFI [13] and a multi-layer type-based call graph similar to the one used by TypeDive [23].

The intersection with the multi-layer type-based call graph yields an orthogonal improvement to the pointer analysis-based call graph by addressing a limitation of pointer analyses: the inability to differentiate between different fields of complex structs. Such analyses often resort to a field-insensitive analysis to ensure soundness, which results in unanalyzed pointers in these structs. Similarly, the type-based call graph enhances the precision of the final call graph in cases where the pointer analysis is imprecise and MLTA cannot be applied. The final call graph consists of a set of all possible targets for each indirect call site and always matches or surpasses the precision of each individual call graph, typically yielding greater accuracy. The combined call graph is sound if, and only if, all the individual call graphs are sound, as the intersection will never remove edges that are present in all the individual call graphs.

Based on our refined call graph, the compiler pass inserts instructions into the program to enforce the call graph at runtime. We employ an efficient checking mechanism, similar to Clang CFI [13] but necessarily adapted for our final call graph, that validates a set of targets at a call site in constant time using jumptables. Our evaluation of the LLVM-based prototype illustrates that our approach significantly reduces the average number of possible jump targets

Listing 1: Example showing the modification necessary to remove a function pointer cast from a program.

(a) Function pointer cast from `(short)`→`short` to `(int)`→`int`

```
int A(int);
short B(short);




int F(int x) {
  int(*f)(int)=x>1?A:B;
  return f(2);
}
```

(b) Clean version of the function pointer cast.

```
int A(int);
short B(short);
int B2(int arg) {
  return B(arg);
}

int F(int x) {
  int(*f)(int)=x>1?A:B2;
  return f(2);
}
```

while incurring a negligible performance overhead. To summarize, our main contributions are:

- A novel, highly precise approach for constructing a call graph at compile-time that combines type analysis, multi-layer type analysis, and a novel whole-program pointer analysis for function pointers using the SVF framework [37].
- A forward-edge CFI solution that enforces our combined call graph using runtime checks with minimal overhead.
- An LLVM-based prototype demonstrating the practicability of our approach, available as open-source.[1]
- A thorough evaluation using all compatible SPEC CPU 2017 benchmarks, indicating that our approach incurs only negligible performance overhead while substantially reducing the number of possible targets compared to previous approaches.

The rest of the paper is structured as follows. First, we cover type-based call graphs and call graphs based on multi-layer type analysis in Section 2 and Section 3, respectively. Section 4 provides details on how the pointer analysis-based call graph is built. The combination of all three call graphs into the final call graph enforced by HPCFI is presented in Section 5. Section 6 provides details on how HPCFI enforces the call graph at runtime. The implementation of HPCFI is described in Section 7, followed by an evaluation of HPCFI Section 8. Section 9 covers related work before we conclude in Section 10.

2 TYPE-BASED ANALYSIS

Type-based mechanisms rank among the most precise CFI mechanisms, with Clang CFI [13] being the most prominent example. These mechanisms restrict the set of valid call targets at each call site to functions that have their address taken and share the same signature as the call site. Such a set of functions with the same signature is referred to as an *equivalence class.* For type-based CFI mechanisms to function correctly, the dynamic type of a function at runtime must match the static type expected at the call site; otherwise, this mismatch causes a false-positive CFI violation. Consequently, function pointer casts can disrupt this CFI scheme, requiring developers to avoid incompatible function pointer casts. Notably, Clang CFI can be enabled for the Chrome browser on

[1]Source code: https://github.com/Fraunhofer-AISEC/hpcfi

Listing 2: Example code illustrating the effectiveness of MLTA.

```
1  struct A {
2    int x;
3    struct B {
4      int y;
5      void (*g)();
6    } b;
7  };
8
9  struct A a;
10
11 int main() {
12   a.b.g = funcA;
13   a.b.g(); // MLTA target: funcA
14   return 0;
15 }
```

Listing 3: Exemplary struct from the perlbench benchmark in SPEC CPU 2017 [17].

```
struct _PerlIO_funcs {
  ...
  SSize_t(*Read) (pTHX_ PerlIO *f, ...
  SSize_t(*Write) (pTHX_ PerlIO *f, ...
  ...
}
```

Linux x86-64 [11], demonstrating that adhering to such a policy is feasible even for large projects.

Furthermore, removing function pointer casts from existing codebases is a straightforward process, as demonstrated by the example code in Listing 1. In Sublisting 1a, function `B` is cast from `(short)` $\rightarrow$ `short` to `(int)` $\rightarrow$ `int` in Line 8. Consequently, a type-based CFI mechanism would not recognize function `B` as a valid target for the indirect call `f(2)` in Line 9, since the dynamic type `(int)` $\rightarrow$ `int` of `f(2)` does not match the static type `(short)` $\rightarrow$ `short` of function `B`. The removal of a function pointer cast can generally be achieved by creating a wrapper function with the correct signature and calling the original function inside the wrapper. In the given example, the function pointer cast can be eliminated by creating the wrapper function `B2` for function `B` in Sublisting 1b. Consequently, only the arguments and the return value need to be cast, rather than the entire function signature.

3 MULTI-LAYER TYPE ANALYSIS

Type-based CFI mechanisms traditionally focus solely on the type of function pointers to create equivalence classes. However, this approach overlooks the fact that function pointers are often stored within structs in C programs. To address this limitation, TypeDive [23], the first CFI mechanism to employ MLTA, leverages this language characteristic to enhance precision beyond conventional type-based approaches. MLTA narrows the potential targets for certain indirect calls by considering the type hierarchy of objects from which a function pointer is retrieved, namely, a multi-layer type hierarchy [23].

Take, for example, Listing 2: On Line 5, the function pointer `g` is a member of `struct B`, which is itself a member of `struct A`. The multi-layer type for `g` is determined by the types of all its containing layers, that is, `[struct A, struct B, void (*)()]`. MLTA identifies multi-layer types by analyzing program instructions that index a struct field of function pointer type. The potential targets of a multi-layer type are then identified by following the indexed field to the corresponding store instructions. In Listing 2, `funcA` is stored to the multi-layer type `[struct A, struct B, void (*)()]` in Line 12. If MLTA determines that finding all stores is too complex or if the stored value cannot be traced back to a target function, MLTA cannot be applied to this specific multi-layer type. For instance, should a reference to `a.b.g` be stored to other variables, MLTA would need to identify all related store instructions, or, as a conservative measure, assume that any function could be a valid target for the multi-layer type `[structA, struct B, void (*)()]`. This conservative approach is necessary to avoid missing any potential targets, which could cause false positive CFI violations during runtime. Additionally, MLTA also analyzes type casts between multi-layer types and falls back to a more permissive subtype if required. The multi-layer type for an indirect call can be determined by finding the instruction that indexes the field of the struct holding the call's function pointer. A program is then instrumented to only allow targets with the same multi-layer type as the call site [23]. In the example in Listing 2, `funcA` is the sole valid target in Line 13 for the multi-layer type `[struct A, struct B, void (*)()]`. In contrast, type-based CFI mechanisms would allow all address-taken functions with the signature `void` $\rightarrow$ `void`.

Lu et al. show that in practice each layer beyond the second provides only a marginal increase in precision [23]. The 2-layer type in the example corresponds to `[struct B, void (*)()]`. In contrast to TypeDive, our MLTA scheme uses 2-layer MLTA (refer to Section 7).

4 WHOLE-PROGRAM POINTER ANALYSIS

Pointer analysis, or points-to analysis, is a static code analysis technique used to determine which memory locations a pointer may point to. The analysis produces a points-to map that correlates pointers with the memory locations they may reference during program execution [3]. In our approach, we employ a whole-program pointer analysis that considers the entire program rather than analyzing individual source code modules separately. This allows for an interprocedural analysis, providing a more precise understanding of how functions interact with each other and resulting in a more precise analysis [22]. Given that fully precise pointer analysis is undecidable [34], the results of a pointer analysis can only yield an approximation of the true points-to map.

Depending on the goals of the analysis, the calculated results can either be an over- or underapproximation. A conservative analysis, or *may-analysis*, generates an overapproximating result that includes all possible points-to relationships, potentially introducing some false relationships. An aggressive analysis, or *must-analysis*, produces an underapproximating result that includes only guaranteed points-to relationships, potentially omitting some valid relationships. Therefore, a may-analysis captures all possible targets of

Table 1: Assessing the effectiveness and analysis times of field-sensitivity, flow-sensitivity, and context-sensitivity in SVF [37] through the analysis of average indirect call targets in SPEC CPU 2017 benchmarks [17].

SVF Field-Sens	☐	☒	☒	☒
SVF Flow-Sens	☐	☐	☒	☒
SVF Context-Sens	☐	☐	☐	☒
perlbench	119.4	119.4	119.4	119.4
	953s	1,051s	1,761s	2,695s
mcf	1.8	1.8	1.8	1.8
	1s	1s	1s	1s
x264	20.8	18.9	18.9	18.9
	33s	41s	869s	1,580s
xz	19.4	14.9	14.9	14.9
	1s	1s	17s	21s
imagick	3.8	3.8	3.8	3.8
	36s	36s	99s	151s
omnetpp	399.9	397.6	397.6	397.6
	1,403s	1,692s	3,371s	7,990s
povray	89.5	89.1	89.1	89.1
	111s	145s	207s	305s
parest	211.5	1.5	1.3	1.3
	6,008s	983s	1,131s	1,452s
xalancbmk	740.0	699.7	699.7	699.7
	2,228s	3,022s	8,725s	32,642s

an indirect call that could occur during program execution, including some that may not occur in reality, while a must-analysis only captures indirect call targets that are certain to occur [3]. For this work, we employ a may-analysis that overapproximates the possible targets at indirect calls, as underapproximating could result in false-positive CFI violations.

Pointer analyses are typically classified in terms of field-sensitivity, flow-sensitivity, and context-sensitivity. These properties influence both the precision and the complexity of the analysis. Table 1 assesses the effectiveness of field-sensitivity, flow-sensitivity, and context-sensitivity in SVF with respect to accurately resolving indirect call targets. It presents the average number of indirect call targets for the SPEC CPU 2017 benchmarks [17], as well as the SVF analysis times.

Field-Sensitivity. When analyzing pointers in aggregate types like structs, pointer analyses must decide whether to treat each field separately (field-sensitive) or equivalently (field-insensitive) [32]. Consider the example in Listing 3, which shows a struct containing function pointers, a common design pattern in large C programs. In a field-insensitive analysis, the pointer analysis does not differentiate between assignments to the `Read` and `Write` fields. Consequently, the call graph would include all `Read` functions as potential targets for the `Write` function, and vice versa. The pointer analysis used in our approach is field-sensitive. Table 1 shows that field-sensitivity can significantly improve the precision of the pointer analysis. For instance, in the case of parest, field-sensitivity reduces the average number of targets from 211.5 to 1.5. Although field-sensitivity introduces a slight analysis overhead for the SPEC

Listing 4: Example code showing flow-sensitivity.

```
1 void func(struct _PerlIO_funcs* Funcs) {
2   Func->read = read1;
3   ...
4   Funcs->read = read2;
5   Funcs->read(...); // read2, (read1)
6 }
```

CPU 2017 benchmarks, this is not true for parest. Paradoxically, in the case of parest, the analysis time without field-sensitivity is substantially higher, despite being significantly less precise. We hypothesize that the decreased analysis time in the field-sensitive analysis results from the gain of precision.

Flow-Sensitivity. Flow-sensitive pointer analyses consider the order of program statements, whereas flow-insensitive analyses do not. In the example provided in Listing 4, a flow-insensitive analysis would identify `read1` as a valid target for the indirect call in Line 5, even though `Funcs->read` is overwritten in Line 4. Although this example illustrates the potential benefits of flow-sensitivity, a flow-sensitive analysis also incurs significant compile-time overhead, especially for large programs [3]. For this reason, we employ a flow-insensitive pointer analysis in our approach. Moreover, Table 1 shows that incorporating flow-sensitivity into the SVF analysis results in only a marginal improvement in precision.

Context-Sensitivity. Context-sensitivity refers to the ability of a pointer analysis to work interprocedurally by analyzing functions separately for each calling context. A calling context refers to a list of call sites, including in particular the arguments of each function call. In contrast, context-insensitive analyses examine a function for all possible calling contexts simultaneously [3].

Listing 5 provides an example highlighting the differences between a context-sensitive and a context-insensitive analysis. A context-insensitive analysis would consider the calling context of the `id` function to be the combined context of all `id()` calls, i.e., the set {`id(funcA)`, `id(funcB)`}. Since `id` simply returns its argument, the set {`funcA`, `funcB`} represents the possible return values for the `id` function. Consequently, both `funcA` and `funcB` are determined to be potential targets at the two indirect calls in Line 7 and Line 8. In contrast, a context-sensitive analysis would analyze the `id` function for the arguments `funcA` and `funcB` separately, allowing it to infer that `funcA` is the only possible target in Line 7 and `funcB` is the only possible target in Line 8.

Context-sensitive analyses are particularly useful for dynamic pointer analyses at runtime, where the context can be part of the query to the pointer analysis. However, pointer analyses during compile-time rarely employ context-sensitivity due to limited practical benefits and poor scalability for large programs [21]. In fact, Table 1 shows that context-sensitivity did not improve the precision of our analysis in practice. Therefore, our approach uses a context-insensitive pointer analysis. Nonetheless, despite its reduced precision, the analysis must still be interprocedural in order to track value flows across function boundaries.

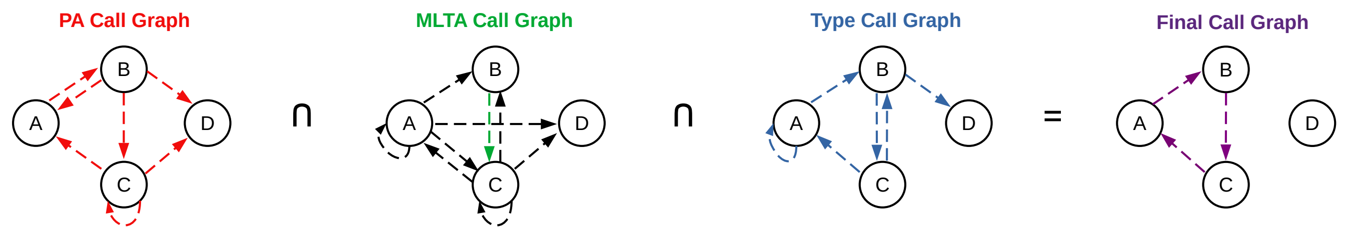


Figure 1: Combination of Pointer Analysis Call Graph, MLTA Call Graph, and Type Call Graph to the final call graph to enforce by the example of the code in Listing 6.

Listing 5: Example code showing context-sensitivity.

```
void (*)() id(void (*f)()) {
  f(); // funcA / funcB
  return f;
}

void func() {
  id(funcA)(); // funcA, (funcB)
  id(funcB)(); // funcB, (funcA)
}
```

Listing 6: Example code showing the differences in precision between a pointer analysis, MLTA, and a type-based analysis.

```
1  typedef struct S {
2    void (*f)(void (*)());
3    void (*g)();
4    void (*h)(void (*)());
5  } S;
6
7  void A() {
8    void (*f)() = B;
9    f(); //Targets: PA={B}, MLTA=n/a, Type={A,B}
10 }
11
12 void B() {
13   S s = { C, A, D };
14   s.f(s.g); //Targets: PA={A,C,D}, MLTA={C},
15             //         Type={C,D}
16 }
17
18 void C(void (*f)()) {
19   f(); //Targets: PA={A,C,D}, MLTA=n/a, Type={A,B}
20 }
21
22 void D(void (*f)()) {
23
24 }
```

5 CALL GRAPH COMBINATION

To generate the most precise call graph possible, we utilize a combination of function signature-based type analysis (refer to Section 2), Multi-Layer Type Analysis (MLTA) (refer to Section 3), and whole-program pointer analysis (refer to Section 4). To ensure the correctness of a program enforcing the combined call graph, we guarantee that the static analyses never underapproximate the set of possible targets at a call site. Underapproximations would result in enforcing an incomplete call graph, potentially leading to erroneous CFI violations where a program is aborted even though no actual CFI violation occurred. Conversely, overapproximating a set of possible targets would result in enforcing an imprecise call graph, giving an attacker more leeway than necessary.

The example in Listing 6 demonstrates three indirect calls and their corresponding targets as determined by the static analyses. For illustrative purposes, a field-insensitive pointer analysis is used. Each analysis builds its own call graph, as depicted in Figure 1. In the call graphs, circles represent functions, and edges represent possible indirect function calls identified by the analysis of the same color. In the MLTA call graph, black edges depict indirect function calls where MLTA could not be applied, for example, when a function pointer is not loaded from a struct. If MLTA cannot be applied at an indirect call, all functions are valid targets of this call. Each individual call graph is an overapproximation of the runtime call graph, resulting in more edges than necessary. The final call graph is the intersection of the three separate call graphs created in the three analyses, containing only the edges present in the pointer analysis-based call graph, the MLTA-based call graph, and the type-based call graph.

An essential part of constructing the combined call graph is the whole-program pointer analysis. Although such an analysis could theoretically construct the most precise call graph possible, in practice, this is not feasible for large programs due to high runtime complexity [3]. Consequently, a trade-off between performance and precision is necessary. For instance, even though we employ a field-sensitive pointer analysis, its result may not be field-sensitive in practice. In C programs, object fields can reference each other in a way that resolving their points-to values introduces infinite constraints. To handle this, a common approach is to make the entire object field-insensitive, ensuring termination and soundness of the analysis [37]. However, this solution comes at the cost of precision. In the example provided in Listing 6, the field-insensitivity of struct `S` leads to mix-ups between its two fields `f` and `g`. Consequently, the pointer analysis overapproximates the set of possible targets (`{A, C, D}`) in Line 14 and Line 19.

Since using structs to store function pointers is a common idiom in C programs, we employ MLTA to improve the precision of resolving indirect calls with function pointers loaded from structs. In the example provided in Listing 6, MLTA is able to determine that function C is the only possible target in Line 14. This is because C is the only function assigned to the field s.f. However, MLTA cannot be applied to the other two indirect calls in the example since the corresponding function pointers are not loaded from a struct.

To enhance precision in cases where MLTA cannot be applied, a standard type-based analysis is employed. Although this approach may result in underapproximations when dealing with function pointer casts, removing function pointer casts from existing code is a straightforward process that we require prior to the analysis (refer to Section 2). The type-based analysis improves the precision of the final call graph for the example code in Listing 6 by determining that function B is not a valid target for the indirect call in Line 19. This is because the function signature at the call site does not match the signature of function B.

The final call graph is the intersection of the pointer analysis-based call graph, the MLTA-based call graph, and the type-based call graph. Since each call graph determined by the different static analyses is an overapproximation of the runtime call graph, the intersection of the three call graphs will also be an overapproximation of the runtime call graph. Thus, the final call graph will never miss any actual edges but only remove superfluous ones, thereby increasing precision. In the case of the example code in Listing 6, the final call graph depicted in Figure 1 represents the most precise call graph possible, despite the individual analyses containing excessive edges. This demonstrates that the analyses complement each other effectively.

6 CALL GRAPH ENFORCEMENT

To enforce the combined call graph at runtime, we insert instructions before each indirect call during compilation, similar to Clang CFI [13]. These instructions, called CFI checks, terminate a program immediately if an indirect call that does not adhere to the combined call graph is about to be executed. Since CFI checks add runtime overhead and increase the size of the instrumented program, it is crucial to implement these checks efficiently. Section 6.1 discusses a straightforward yet inefficient method for implementing CFI checks. This method is applied at indirect call sites with only a small number of potential targets. For indirect call sites with larger target sets, we use a more sophisticated and efficient technique based on the CFI checking mechanism in Clang CFI. We describe this technique in Section 6.2. The two different types of CFI checks are explained using the example code provided in Listing 7. In the example, three indirect calls occur via the function pointers f, g, and h in Lines 12-14. A fully precise static analysis of function M in isolation would identify the potential targets {A, B} for f(), {B, C} for g(), and {D, E} for h().

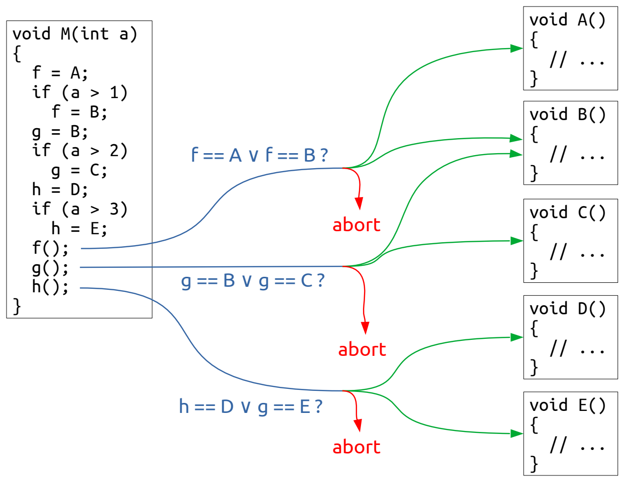
6.1 Simple Check

A straightforward method to implement CFI checks using the combined call graph from the three static analyses is to compare the function pointer of an indirect call against each authorized target before making the call. If the pointer matches one of the targets, the indirect call is allowed. Otherwise, the program is terminated with a CFI violation. Figure 2 illustrates the insertion of such CFI checks using the sample code provided in Listing 7. In this example, in order to perform the indirect call f(), it must hold that f is either equal to A or B. These two targets have been determined by the static analyses. If f does not match A or B, a CFI violation occurs, and the program is aborted.

These simple CFI checks provide optimal precision: every target in the combined call graph is permitted, while all other targets are not. However, this type of CFI check may impose significant performance overhead. At each indirect call site, up to n comparisons may be required at runtime for n potential targets. Consequently, we propose a more refined CFI checking mechanism suitable for indirect call sites with a large number of targets next.

Listing 7: Example function with annotations for valid indirect call targets by examplary static analysis.

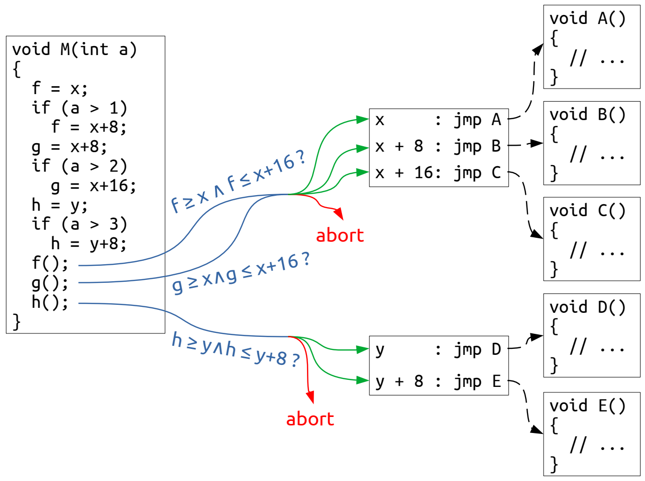
```
1  void M(int a)
2  {
3    f = A;
4    if (a > 1)
5      f = B;
6    g = B;
7    if (a > 2)
8      g = C;
9    h = D;
10   if (a > 3)
11     h = E;
12   f(); // targets: A or B
13   g(); // targets: B or C
14   h(); // targets: D or E
15 }
```



```
void M(int a)
{
  f = A;
  if (a > 1)
    f = B;
  g = B;
  if (a > 2)
    g = C;
  h = D;
  if (a > 3)
    h = E;
  f();
  g();
  h();
}
```

```
void A()
{
  // ...
}
void B()
{
  // ...
}
void C()
{
  // ...
}
void D()
{
  // ...
}
void E()
{
  // ...
}
```

Figure 2: Simple CFI checks: Each indirect call is protected by a separate CFI check.



```
void M(int a)
{
  f = x;
  if (a > 1)
    f = x+8;
  g = x+8;
  if (a > 2)
    g = x+16;
  h = y;
  if (a > 3)
    h = y+8;
  f();
  g();
  h();
}
```

```
x     : jmp A
x + 8 : jmp B
x + 16: jmp C
```

```
y     : jmp D
y + 8 : jmp E
```

```
void A()
{
  // ...
}
void B()
{
  // ...
}
void C()
{
  // ...
}
void D()
{
  // ...
}
void E()
{
  // ...
}
```

$f \geq x \wedge f \leq x+16$?
abort
$g \geq x \wedge g \leq x+16$?
$h \geq y \wedge h \leq y+8$?
abort

Figure 3: Fast CFI check: Two jumptables are created for our example. The first jumptable is used for the CFI checks protecting the indirect calls `f()` and `g()` and the second jumptable for `h()`.

6.2 Fast Check

We now present a refined CFI checking mechanism specifically designed for handling indirect call sites with a large number of potential targets. This mechanism significantly accelerates CFI checks by employing jump tables, which facilitate the execution of CFI checks for any number of targets in constant time. The concept of using jump tables is inspired by Tice et al. [39] and has been implemented in a similar form in Clang CFI [13]. Our empirical experiments indicate that simple comparison-based checks outperform jump table-based checks for indirect call sites with up to three targets. Notably, Clang CFI employs comparison-based checks only for indirect call sites with a single target.

To utilize jump tables, the CFI instrumentation requires the use of equivalence classes. At each call site, the instrumentation only permits targets from the same equivalence class, with each class corresponding to a specific jump table. An equivalence class represents a set of functions that are equivalent according to the CFI approach. In Clang CFI, equivalence classes are constructed based on functions sharing the same signatures, thus allowing a unique mapping of disjoint sets of call sites to disjoint sets of functions. Each disjoint set of functions constitutes an equivalence class. However, our threefold static analysis does not allow for such a mapping, as we identify potential targets for each indirect call site individually, rather than for a set of indirect call sites. For example, in Listing 7, function `B` is a valid target for the indirect calls `f()` and `g()`. However, function `A` is only a valid target for the indirect call `f()`, but not for `g()`. Consequently, function `B` is included in two equivalence classes: one for `f()` and another for `g()`. However, to replace function pointers with jump table entries, targets must be exclusive to a single equivalence class. Otherwise, using the same function pointer in multiple jump tables could lead to ambiguous function pointer comparisons, resulting in different program semantics.

In order to use equivalence classes with our three static analyses, we modify the combined call graph to ensure that each target belongs only to a single equivalence class. It is important to note that this modification may introduce imprecisions in enforcing the combined call graph. However, we show that these imprecisions are minimal in Section 8. Figure 3 illustrates an optimized CFI check for the example shown in Listing 7. Since the static analyses determined that function `B` is a valid target for both indirect calls `f()` and `g()`, the first jump table consists of `A`, `B`, and `C`—representing the combined targets for both call sites. Merging targets is necessary to form an equivalence class, but it also leads to less precise CFI checks. The check for `f()` now additionally allows function `C` and the check for `g()` allows function `A`. Next, all function pointers are substituted with their respective jump table entries, as depicted in the modified source code in Figure 3. For instance, `A` is replaced by `x`, and `B` is replaced by `x+8`. The use of equivalence classes enables a straightforward replacement of function pointers with jump table indices. Note that these jump table entries can be utilized in the same way as the original function pointers for both function calls and function pointer comparisons. Performing CFI checks now simply involves a range check to verify whether a function pointer falls within the designated jump table bounds. The target pointer must neither be less than the jump table's start address nor greater than the address of its last entry. Additionally, an alignment check is necessary to prevent any attempts by an attacker to hijack control by using the middle of jump table entries as jump targets. Otherwise, a CFI violation occurs and the program is aborted. It is important to note that the example was deliberately chosen such that the equivalence class `{A, B, C}` is an overapproximation of the targets from the static analyses. In Section 8, we demonstrate that the loss of precision is minimal in most cases.

7 IMPLEMENTATION

We have implemented an LLVM-based CFI mechanism called HPCFI, which enforces the combined call graph as presented in 5. Figure 4 illustrates the compilation process when using HPCFI. First, HPCFI analyzes each source file to identify indirect calls, distinguishing them from virtual calls. To reliably differentiate between these two call types, the analysis must run at compile time, prior to any optimizations. This closely resembles the virtual call identification implementation in Clang CFI [13].

HPCFI employs a Multi-Layer Type Analysis (MLTA) implementation similar to, but independent of, TypeDive [23]. Our MLTA consists of two stages: the first stage operates before any LLVM optimizations, while the second stage runs during LTO. In contrast, TypeDive runs exclusively during LTO. This is unproblematic if a target is compiled without any optimizations (the method used by the authors to evaluate TypeDive [23]), but may result in an incomplete call graph when compiled with optimizations enabled. Optimizations can obscure instructions that index struct fields, preventing MLTA from reliably identifying stores of function pointers to struct fields. To address this issue and ensure the practical usability of HPCFI in realistic compilation scenarios with enabled optimizations, we analyze function pointer stores to struct fields and call sites of such function pointers before any optimizations.

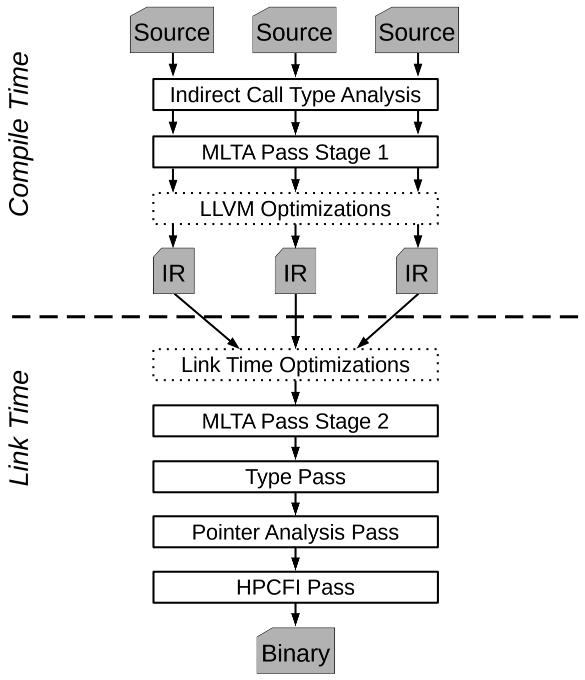


Figure 4: HPCFI Compilation Process.

Our first stage, in further contrast to TypeDive, only considers the types in the first two layers when collecting function pointers. Additional layers have been shown to only marginally improve precision [23] while significantly increasing the complexity of the analysis and the risk of omitting edges in the MLTA call graph. The results of the first stage are attached as LLVM metadata to each analyzed IR file and carried to the second stage. This metadata associates struct fields with potential function pointers and contains information about which struct fields are used as indirect call targets at specific call sites. After the compile and link time optimizations, the second MLTA stage uses the metadata from the first stage to construct the MLTA-based call graph for the whole program.

Following that, HPCFI constructs the type-based call graph. Even though the construction of the type-based call graph is very similar to Clang CFI, there are some small but crucial differences. Clang CFI is not implemented purely in the LTO phase but instead consists of an analysis pass running at compile time and an instrumentation pass running during link time. During compile time, Clang CFI inserts an `llvm.type.test` intrinsic function before each indirect call, which holds information about the expected type of the target pointer. Additionally, Clang CFI embeds metadata within functions that specify their respective signatures [13]. This is done at compile time to accommodate potential alterations to function signatures by the linker. For instance, the linker merges isomorphic structs, i.e., structurally identical structs but with different names, resulting in a change to a function's signature if such a struct is included in it. In contrast, HPCFI disables the merging of isomorphic structs, which enables HPCFI's type pass to run at link time while avoiding false-positive function signature mismatches.

Next, the pointer analysis call graph is built. The pointer analysis of HPCFI is designed to operate on the entire program and is therefore executed during LTO, where inter-modular analysis is possible. We utilize the whole-program pointer analysis from the SVF framework [37] to construct the pointer analysis-based call graph. Static Value-Flow (SVF) is a program analysis framework specifically designed for analyzing LLVM IR using interprocedural static value-flow analysis and pointer analysis [37]. In HPCFI, we employ SVF to perform a field-sensitive, context-insensitive, and flow-insensitive variant of Andersen's pointer analysis. For cases where SVF does not fully cover all aspects required for sound static analysis, such as interactions with unmodeled libraries, a fallback mechanism that allows all address-taken functions at the affected call site is necessary to ensure soundness.

Once all three call graphs have been built, they are combined into the final call graph as described in Section 5. Finally, the HPCFI pass inserts the CFI checks presented in Section 6 into the program to enforce the combined call graph.

The various compiler passes of HPCFI are implemented in C++ for LLVM 13 and can be used from within the LLVM compiler pipeline. Since HPCFI operates on LLVM IR, the compiler pass is technically independent of the source language. However, SVF currently only supports C and C++.

8 EVALUATION

In this section, we evaluate the precision, performance, and compile time of HPCFI using the SPEC CPU 2017 benchmark suite [17]. The evaluated benchmarks include all C and C++ benchmarks in SPEC CPU 2017 that contain at least one indirect call, with the exception of blender, which did not successfully compile with our version of the LLVM compiler. All benchmarks were compiled using optimization level O2. As discussed in Section 2, HPCFI's type-based call graph requires programs to be free from ambiguous function pointer casts. To ensure all benchmarks run without any false-positive CFI violations due to function type mismatches, we modified the source code of the perlbench, gcc, mcf, imagick, povray, parest, and xz benchmarks to eliminate function pointer casts. These modifications involved changing fewer than 100 lines of code. We note that the same modifications were also necessary to run the benchmarks with standard Clang CFI. Section 2 provides a detailed description of how these modifications must be performed. Moreover, due to a bug in SVF that leads to an underapproximation in the indirect call target resolution, we excluded two indirect calls in parest from the CFI instrumentation. In addition to the SPEC CPU 2017 benchmarks, we also evaluate the precision and compile time of two real-world applications: Nginx, a webserver [28], and Redis, a database [33].

The remainder of this section is organized as follows. In Section 8.1, we evaluate whether HPCFI's final call graph, as a combination of a type-based call graph, MLTA call graph, and pointer analysis call graph, can enhance precision over each individual call graph and existing CFI mechanisms. In Section 8.2, we compare the performance of HPCFI to Clang CFI. Finally, Section 8.3 examines the compilation time required by HPCFI.

Table 2: Average indirect call targets per call site of type-based CFI (similar to Clang CFI [13]), MLTA CFI (similar to TypeDive [23]), pointer analysis CFI, and HPCFI for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33].

	Program Metrics			Arithmetic and Geometric Mean of Indirect Call Targets					Improv. over Type CFI	
Benchmark	Code Size	Functions	Addr.-Taken Functions	Type CFI	MLTA ∩ Type CFI	PA CFI	**HPCFI Simple**	**HPCFI Fast**	MLTA ∩ Type CFI	**HPCFI Fast**
perlbench	1.9 MB	1,755	866	30.2 17.3	15.9 6.8	119.4 57.1	15.8 6.7	16.2 6.8	47% 61%	46% 61%
gcc	8.0 MB	8,958	3,829	75.0 16.1	71.0 10.6	105.4 14.4	45.3 6.2	45.4 6.3	5% 34%	39% 61%
mcf	16.5 KB	51	3	1.8 1.7	1.8 1.7	1.8 1.7	1.8 1.7	1.8 1.7	0% 0%	0% 0%
x264	878.8 KB	948	339	6.8 3.7	3.7 2.3	18.9 13.8	3.7 2.3	4.5 2.5	46% 38%	34% 32%
xz	93.3 KB	168	95	6.6 4.2	5.8 3.8	14.9 10.2	4.6 3.0	4.9 3.1	12% 10%	26% 26%
imagick	1.2 MB	789	60	7.5 3.5	7.5 3.5	3.8 2.5	1.6 1.2	1.6 1.2	0% 0%	79% 66%
omnetpp	1.1 MB	5,004	4,073	15.8 4.8	15.8 4.8	397.6 136.3	15.5 4.6	15.5 4.6	0% 0%	2% 4%
povray	1.0 MB	1,223	535	23.1 14.5	15.8 11.4	89.1 43.3	15.8 11.4	15.8 11.4	32% 21%	32% 21%
parest	1.7 MB	2,878	1,910	105.5 95.8	105.5 95.8	1.5 1.4	1.5 1.4	1.5 1.4	0% 0%	99% 99%
xalancbmk	2.7 MB	8,711	6,424	57.0 24.2	57.0 24.2	699.7 48.8	31.7 15.4	31.7 15.4	0% 0%	44% 36%
nginx	648 KB	3,966	2,023	16.4 10.0	14.8 8.2	189.1 111.5	11.2 6.8	11.2 6.8	10% 18%	32% 32%
redis	2.1MB	4,891	1,098	4.0 2.3	3.3 2.0	167.2 60.4	2.3 1.6	2.3 1.6	18% 13%	43% 30%
Arithm. Mean									14% 16%	40% 39%

Table 3: Comparison of average (arithmetic and geometric mean) indirect call targets of type-based CFI, MLTA CFI, and pointer analysis CFI exclusively for MLTA-applicable call sites for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33].

Benchmark	Type CFI	MLTA ∩ Type CFI	PA CFI	MLTA Appl.
perlbench	24.2 17.6	5.5 5.2	136.9 77.9	77%
gcc	63.2 7.6	53.6 2.8	55.9 2.9	41%
mcf	- -	- -	- -	0%
x264	7.7 3.9	3.5 2.1	20.7 14.5	73%
xz	7.8 6.1	6.6 5.3	13.9 9.1	63%
imagick	- -	- -	- -	0%
omnetpp	- -	- -	- -	0%
povray	41.8 35.6	23.11 19.5	121.4 98.2	39%
parest	109.0 109.0	109.0 109.0	1.5 1.4	97%
xalancbmk	76.0 76.0	76.0 76.0	2069.0 2069.0	33%
nginx	11.1 4.8	5.0 2.3	233.7 177.9	26%
redis	3.2 2.1	1.9 1.6	153.4 80.1	48%
Arithm. Mean				41%

8.1 Precision

To evaluate the precision of HPCFI, we compare the average number of allowed targets at indirect call sites for purely type-based CFI, MLTA CFI, pointer analysis CFI, and HPCFI in Section 8.1.1. The precision of our type-based CFI mechanism aligns closely with Clang CFI, with minor differences due to Clang CFI's insertion of instructions at compile time, which slightly alters input for subsequent compiler passes (refer to Section 7) [13]. Since TypeDive [23] has some limitations in combination with compiler optimizations (refer to Section 7), we use our own MLTA implementation. Nonetheless, we expect the precision of our MLTA CFI in combination with type-based CFI to be similar to TypeDive, as both function similarly. As an additional metric to the average number of indirect call targets, we evaluate the distribution of the target set sizes in Section 8.1.2.

8.1.1 Average Indirect Call Targets. Table 2 presents the arithmetic mean and geometric mean of call targets per indirect call site for type-based CFI (Type CFI), MLTA CFI combined with type-based CFI (MLTA ∩ Type CFI), pointer analysis CFI (PA CFI), and HPCFI. For a full pairwise comparison of Type CFI, MLTA CFI, and PA CFI, refer to Table 6 in the Appendix. The first column shows the code size of the unprotected binaries, which provides an indication of the number of potential targets, as any instruction in uninstrumented binaries could be a target for an indirect call. The second column displays the number of functions within the binary, reflecting the precision of Intel CET, which enforces that indirect calls must target only the start of a function [31]. Similarly, the third column shows the average number of functions from which an address was taken, which corresponds to the precision of Microsoft CFG [19]. Type-based CFI enhances the precision of coarse-grained CFI mechanisms like Intel CET [31] and Microsoft CFG [19] by taking the types of function pointers into account. The average number of targets for type-based CFI generally increases as the code size of the binary grows, suggesting that, like Intel CET and Microsoft CFG, type-based CFI becomes less effective as the code size increases.

The combination with MLTA CFI further enhances precision by considering the object type hierarchy from which function pointers are loaded. Since MLTA ∩ Type CFI uses a type-based call graph as a fallback for indirect calls where MLTA is not applicable, it is always at least as precise as type-based CFI. Table 3 shows the number of indirect calls to which MLTA can be applied in each benchmark. In certain benchmarks, only a small number of function pointers are loaded from structs, limiting the applicability of MLTA and resulting in only marginal improvements to the precision of type-based CFI. For instance, none of the 397 indirect calls in the imagick

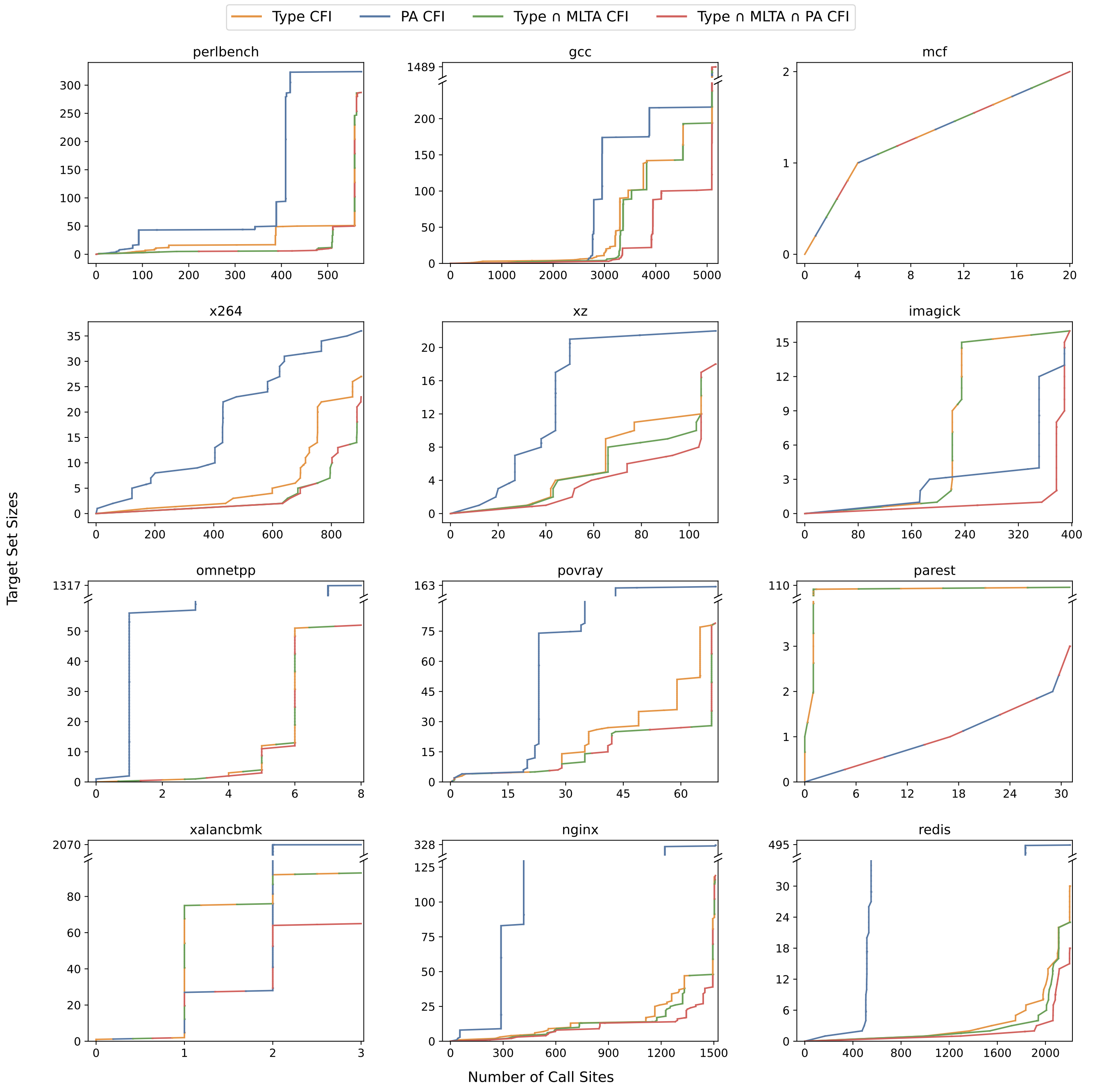


Figure 5: Cumulative target set sizes for type-based CFI (Type CFI), pointer analysis CFI (PA CFI), the combination of type-based CFI and multi-layer type CFI (Type ∩ MLTA CFI), and HPCFI for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33].

benchmark utilize MLTA. In contrast, MLTA can be applied to 73% of indirect calls in the x264 benchmark, where `struct x264_t` contains 368 function pointers. In this instance, MLTA is highly effective and reduces the average number of targets for MLTA-applicable indirect calls from 7.7 (type-based CFI) to 3.5.

The pure pointer analysis CFI mechanism, which utilizes SVF to determine targets for indirect calls, shows a precision improvement of approximately 99% for the parest benchmark and 50% for the imagick benchmark, but on average, is less precise than type-based CFI for all other benchmarks.

Even though pointer analysis CFI does not perform well on its own, the combination with type-based CFI and MLTA CFI in HPCFI significantly improves precision. For example, HPCFI improves the precision for gcc from 71.0 (MLTA ∩ Type CFI) and 105.4 (PA CFI) to 45.3, and for imagick from 7.5 (MLTA ∩ Type CFI) and 3.8 (PA CFI) to 1.6. HPCFI with fast checks is only slightly less precise compared to HPCFI with simple checks for perlbench, gcc, x264, and xz. On average, HPCFI improves the precision of type-based CFI by 40%, whereas MLTA ∩ Type CFI alone achieves an improvement of only 14%. It is worth noting that precision improvements are not always possible. For the mcf benchmark, a manual analysis revealed that type-based CFI is already 100% accurate, making further precision improvements impossible. In conclusion, HPCFI significantly improves the average indirect call targets of fine-grained type-based CFI mechanisms such as Clang CFI and MLTA CFI.

8.1.2 Target Set Size Distribution. While the average number of indirect call targets provides a good indication of the security of a CFI scheme, the distribution of target set sizes is also important. For instance, reducing the target set sizes of 10 call sites from 1,000 to 100 may not be as beneficial as reducing the target set sizes of 1,000 call sites from 10 to 1. Although both scenarios eliminate 9,000 of 10,000 possible targets, the latter results in a more significant tightening of control flow. In contrast, the former scenario still permits 100 targets per call site, which may still be vulnerable to control-flow bending attacks [9]. Figure 5 shows the cumulative target set sizes for type-based CFI (Type CFI), pointer analysis CFI (PA CFI), the combination of type-based CFI and multi-layer type CFI (Type ∩ MLTA CFI), and HPCFI for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33]. For a full pairwise comparison of Type CFI, MLTA CFI, and PA CFI, refer to Figure 7 in the Appendix. The area under a line represents the total number of allowed targets at all call sites, which corresponds to the average number of indirect call targets shown in Table 2. Generally, target sizes are not evenly distributed. Instead, many target sets are either very small (0-3 targets) or very large (up to 6,424 for xalancbmk), which is evident in Figure 5 by the convexity of the lines. The convexity is even more pronounced when analyzing the combination of pointer analysis CFI, MLTA CFI, and Type CFI, indicating that HPCFI tends to further reduce already small target set sizes rather than improving large target sets.

8.2 Performance

For a CFI mechanism to be applicable in real-world programs, it must incur minimal performance overhead. We conducted a performance evaluation of HPCFI on a 64-bit x86 AMD Ryzen 7 PRO 4750U CPU with 32GB RAM running Ubuntu 20.04. To minimize variations in performance results, we disabled frequency scaling, isolated the CPU running the benchmarks through CPU shielding, and turned off ASLR. We averaged the performance results over six runs, with a relative standard deviation of less than one percent for all benchmarks.

To evaluate the performance of HPCFI, we selected Clang CFI [13] (using the `-fsanitize=cfi-icall` flag for Clang) as the primary reference mechanism for two reasons. Firstly, Clang CFI is widely used in large real-world projects like Chrome on Linux [11] and Android. On Android, it is used in various system components and can even be enabled for the kernel [4]. Secondly, like HPCFI, Clang CFI is implemented in LLVM [13], allowing for a meaningful comparison between both mechanisms. However, it is important to note that Clang CFI already inserts instructions during compile time (refer to Section 7), which means that subsequent compiler passes will run on different input compared to HPCFI. This can result in different behavior of the subsequent compiler passes. For example, the inliner pass may choose to inline a function in HPCFI but not in Clang CFI due to variations in the number of instructions within the function. As a result, small differences in performance may arise that are not directly related to the CFI mechanisms themselves.

Table 4 shows the number of static and dynamic indirect calls for the SPEC CPU 2017 benchmarks [17]. Static indirect calls are categorized into two groups: all indirect call sites found in the program and those indirect calls with at least one possible target in the HPCFI call graph. Call sites with no valid targets are identified exclusively by the pointer analysis and may, for example, occur due to inlining. The number of dynamic indirect calls is expected to correlate with the performance overheads, which are shown in Figure 6. HPCFI-Fast demonstrates low performance overheads across all benchmarks, with an average overhead of 1.3% compared to the average overhead of 2.6% for Clang CFI. HPCFI-Simple, employing simple compare checks (refer to Section 6.1), exhibits more pronounced overheads, reaching 57.1% for the perlbench benchmark. This is mainly due to perlbench having an indirect call with a large number of potential targets on its frequently executed path. For both HPCFI-Fast and Clang CFI, the maximum overhead occurs with the mcf benchmark at 4.1% and 8.6%, respectively. Interestingly, some benchmarks show a negative performance overhead, likely resulting from measurement noise as discussed by Mytkowicz et al. [27]. Such measurement noise can arise from different UNIX environment sizes, which can affect stack alignment, among other factors. As expected, the performance overheads of Clang CFI and HPCFI-Fast are similar since they both use the same CFI checking mechanism. However, slight differences can occur for several reasons. Firstly, the compilation process slightly differs, which can result in different optimizations being performed by LLVM. Secondly, HPCFI-Fast uses compare checks for indirect calls with three or fewer targets instead of a jumptable check (refer to Section 6.2), while Clang CFI only uses a compare check when there is only a single target [13]. Additionally, HPCFI-Fast achieves higher precision than Clang CFI, potentially enabling it to replace more jumptable checks with compare checks. In summary, HPCFI-Fast exhibits negligible performance overhead.

8.3 Compile Time

Given that HPCFI requires a computationally expensive pointer analysis, it is expected to impose a significant compile time overhead. However, our results indicate that the analysis completes within an acceptable time frame, even for large programs. Table 5 presents the compile times for the SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33]. For smaller programs like mcf and xz, the overhead from the static analysis is negligible. Even for medium-sized programs such as imagick and povray, the increase in compile time remains moderate. However, very large programs like gcc, which consist of millions of lines of code and are highly complex,

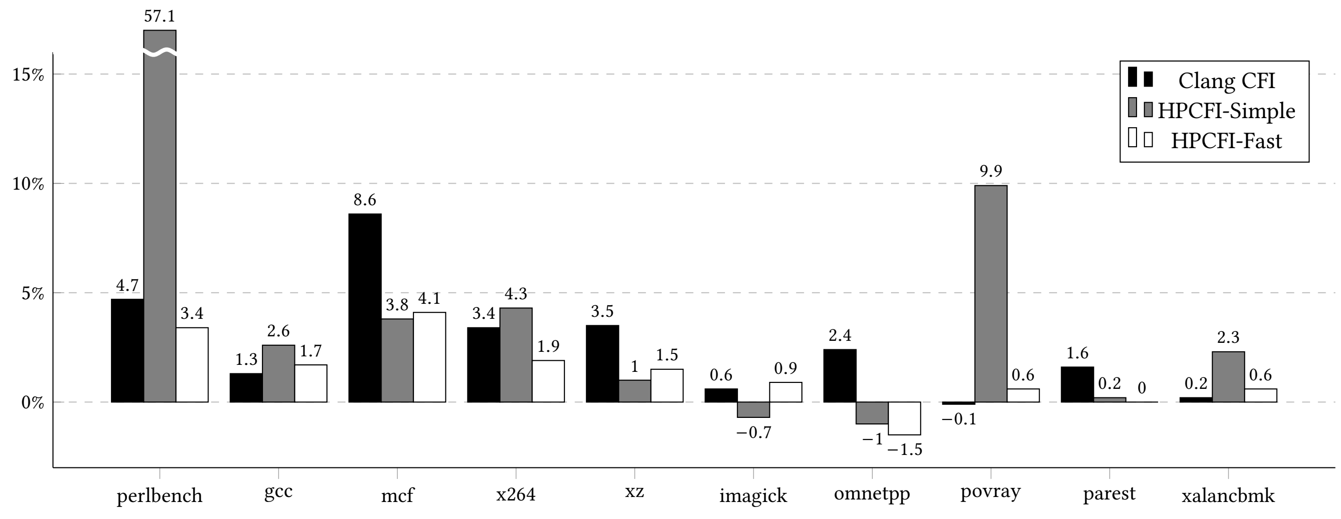


Figure 6: Performance overhead of HPCFI and Clang CFI (baseline LLVM LTO) for SPEC CPU 2017 benchmarks [17].

Table 4: Number of static and dynamic indirect calls for SPEC CPU 2017 benchmarks [17].

	#static call sites		#dynamic
Benchmark	> 0 targets	total	call sites
perlbench	572	579	13,275,847,191
gcc	5,162	5,407	1,733,358,573
mcf	20	20	20,013,516,767
x264	901	910	3,794,109,331
xz	111	184	502,633,284
imagick	397	403	37,374,914
omnetpp	8	20	109,806
povray	69	69	28,258,780,723
parest	31	86	43,972
xalancbmk	3	3	10

Table 5: Compile time overhead of HPCFI for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33].

Benchmark	KLOC	Baseline	HPCFI	Overhead
perlbench	362	1,899	2,950s	55%
gcc	1,304	4,395s	65,749s	1,396%
mcf	3	19s	20s	5%
x264	96	963s	1,004s	4%
xz	33	66s	67s	2%
imagick	259	1,353s	1,389s	3%
omnetpp	134	1,302s	2,994s	126%
povray	170	1189s	1,334s	12%
parest	427	4,798s	5,781s	20%
xalancbmk	520	5,873s	8,895s	51%
nginx	147	232s	765s	230%
redis	109	503s	3,478s	591%
Arithm. Mean				208 %

can experience a considerable compile time overhead. It is important to recognize that enabling CFI instrumentation exclusively for production builds is sufficient, allowing the static analysis overhead to be incurred only when necessary, minimizing its impact on development and testing processes.

9 RELATED WORK

Since the original work on CFI by Abadi et al. [1], there have been numerous advancements in CFI policies and designs [15, 18, 20, 23, 29, 30, 39–41]. In this section, we compare HPCFI to other recent forward-edge CFI mechanisms that leverage static analyses to achieve precise results.

A popular class of CFI mechanisms employs a simple static analysis to construct a type-based call graph and then instruments programs to enforce this call graph at runtime. This includes call graphs based on function signatures [29, 30, 39, 41] and call graphs based on the type hierarchy of structs [23], as described in Sections 2 and 3. Compared to the type-based and multi-layer type-based call graph generation, HPCFI employs a threefold generation where a type-based call graph, multi-layer type-based call graph, and pointer analysis-based call graph are computed and intersected into a single call graph. This resulting call graph benefits from the complementary nature of the individual graphs.

Another class of CFI mechanisms relies on contextual information only available at runtime. For example, this context can be the recent execution history recorded by the CPU, as implemented in PathArmor [40] and μCFI [18]. PathArmor utilizes the Last Branch Record (LBR), while μCFI uses Processor Traces (PTs). Both LBR and PT can only be accessed by the kernel. However, since transitioning into kernel space is expensive, both PathArmor and μCFI only enforce their CFI policy at select system calls, resulting in partial protection. These mechanisms check adherence to the call graph within an additional thread, where they perform a live pointer analysis to create a context-sensitive call graph. In contrast, HPCFI does not consider any runtime context when creating its call graph. Instead, the call graph is created during compile time, which helps save performance at runtime.

Another context-sensitive CFI mechanism is Origin-sensitive Control Flow Integrity (OS-CFI) [20], which takes the origin of a function pointer as context. The origin is defined by Khandaker et al. as the instruction that most recently wrote to a function pointer. OS-CFI operates by conducting a pointer analysis during compile time using SVF to identify all possible origins of indirect calls. Then, a program is instrumented at these origins by mapping the code address of the origin to a function pointer in a metadata storage. At indirect call sites, the function is used to look up the origin, which must be a valid origin for that call site as determined by the pointer analysis. Additionally, the most recent return address can be utilized to better differentiate between same origins but different execution paths. Although both HPCFI and OS-CFI use SVF for a pointer analysis during compile time, their instrumentation methods differ significantly. As a result, HPCFI demonstrates better performance, while OS-CFI can achieve higher precision. OS-CFI exhibits a performance overhead of 8.2% on the SPEC CPU 2006 benchmarks [20], while Clang CFI shows a 1% overhead on the same benchmarks [39].

Ge et al. implement a fine-grained CFI mechanism specifically for kernel software [15]. For forward-edge call graph construction, they utilize a static taint analysis to precisely identify indirect call targets. This analysis achieves high precision by operating under certain assumptions, which are typically valid in kernel code. For example, there must not exist any data pointers to a function pointer. To maintain soundness, any violations of these assumptions are reported to the user. In contrast, HPCFI does not rely on any special assumptions, which may lead to less precise call graphs in some instances.

10 CONCLUSION

In this paper, we presented HPCFI, a mechanism that combines type-based analysis, multi-layer type analysis, and pointer analysis to construct a precise call graph for CFI enforcement. We implemented a functional prototype of HPCFI using LLVM and evaluated it using the SPEC CPU 2017 benchmark suite. The evaluation demonstrated that HPCFI can be successfully applied to large programs. Our results showed that HPCFI improves the precision of established CFI mechanisms, such as like Clang CFI, by an average of 40%, while maintaining negligible performance overheads.

ACKNOWLEDGMENTS

This work was supported by research funded by the German Federal Ministry of Economic Affairs and Climate (BMWK). It was partially carried out in the RESISTANT project (contract code: 20D2116A) as part of the Aviation Research Programme (LuFo) VI-2, and partially in the PoQsiKom project under Grant No. 13I40V010A.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article 4 (11 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. 2008. Preventing Memory Error Exploits with WIT. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 263–277. https://doi.org/10.1109/SP.2008.30

[3] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph. D. Dissertation. University of Copenhagen.

[4] Android Open Source Project. 2023. *Control Flow Integrity.* https://source.android.com/devices/tech/debug/cfi

[5] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. 2011. Mitigating Code-Reuse Attacks with Control-Flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '11)*. Association for Computing Machinery, New York, NY, USA, 353–362. https://doi.org/10.1145/2076732.2076783

[6] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM symposium on information, computer and communications security.* 30–40.

[7] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)* 50, 1 (2017), 1–33.

[8] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining Light on Shadow Stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*. 985–999. https://doi.org/10.1109/SP.2019.00076

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium USENIX Security 15)*. 161–176.

[10] Nicholas Carlini and David Wagner. 2014. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*. 385–399.

[11] Chrome. [n. d.]. *Control Flow Integrity.* https://www.chromium.org/developers/testing/control-flow-integrity

[12] Chrome security team. 2020. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/

[13] Clang. [n. d.]. *Control Flow Integrity Design Documentation.* https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[14] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* 40–51.

[15] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained control-flow integrity for kernel software. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 179–194.

[16] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 575–589.

[17] John Henning. 2022. *SPEC CPU®2017 Overview.* https://www.spec.org/cpu2017/Docs/overview.html

[18] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security.* 1470–1486.

[19] John Kennedy, Kent Sharkey, David Coulter, Drew Batchelor, Nicholas Adman, Erik Swan, Mike Jacobs, and Michael Satran. 2022. *Control Flow Guard.* Microsoft. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[20] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security 19)*. 195–211.

[21] Ondřej Lhoták and Laurie Hendren. 2006. Context-sensitive points-to analysis: is it worth it?. In *International Conference on Compiler Construction.* Springer, 47–64.

[22] Donglin Liang and Mary Jean Harrold. 1999. Efficient points-to analysis for whole-program analysis. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 199–215.

[23] Kangjie Lu and Hong Hu. 2019. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1867–1881. https://doi.org/10.1145/3319535.3354244

[24] Microsoft. 2019. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/

[25] MITRE. 2023. *2022 CWE Top 25 Most Dangerous Software Weaknesses.* MITRE. https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html

[26] Ingo Molnar and Arjan van de Ven. 2004. *New Security Enhancements in Red Hat Enterprise Linux.* https://static.redhat.com/legacy/f/pdf/rhel/WHP0006US_Execshield.pdf

[27] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong! *SIGPLAN Not.* 44, 3 (3 2009), 265–276. https://doi.org/10.1145/1508284.1508275

[28] Nginx. 2024. http://nginx.org/.

[29] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. *SIGPLAN Not.* 49, 6 (5 2014), 577–587. https://doi.org/10.1145/2666356.2594295

[30] Ben Niu and Gang Tan. 2015. Per-input control-flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.*

914–926.
[31] Baiju Patel. [n. d.]. *A Technical Look at Intel's Control-flow Enforcement Technology*. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html
[32] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. 2007. Efficient Field-Sensitive Pointer Analysis of C. *ACM Trans. Program. Lang. Syst.* 30, 1 (11 2007), 4–es. https://doi.org/10.1145/1290520.1290524
[33] Redis. 2024. https://github.com/redis/redis.
[34] Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (2000), 162–186.
[35] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (3 2012), 34 pages. https://doi.org/10.1145/2133375.2133377
[36] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175
[37] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235
[38] The White House. 2024. Back to the Building Blocks: A Path Towards Secure and Measurable Software. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
[39] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 941–955. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice
[40] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 927–940.
[41] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.

A PAIRWISE PRECISION EVALUATION

The following complements the precision evaluation in Section 8 by presenting a pairwise comparison of type-based CFI, MLTA CFI, and pointer analysis CFI for the SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33]. Table 6 presents the average number of indirect call targets per call site and Figure 7 shows the cumulative target set sizes. The pure MLTA results use all address-taken functions as fallback for inapplicable call sites, resulting in a significant lack of precision for some indirect calls. However, the combination of Type CFI and MLTA CFI enhances the precision of both methods. For instance, in perlbench, the number of indirect call targets decreases from 30.2 (Type CFI) and 205.5 (MLTA CFI) to 14.9 when combined. Similarly, PA CFI substantially overapproximates the targets of some indirect calls. But the combination of PA CFI and Type CFI, for instance, improves the precision for imagick from 3.8 (PA CFI) and 7.5 (Type CFI) to 1.6. Additionally, the combination of PA CFI and MLTA CFI increases the precision for perlbench from 119.4 (PA CFI) and 205.5 (MLTA CFI) to 18.5. Notably, the combination of all three techniques yields the most precise results, demonstrating that each analysis contributes to the overall precision gains.

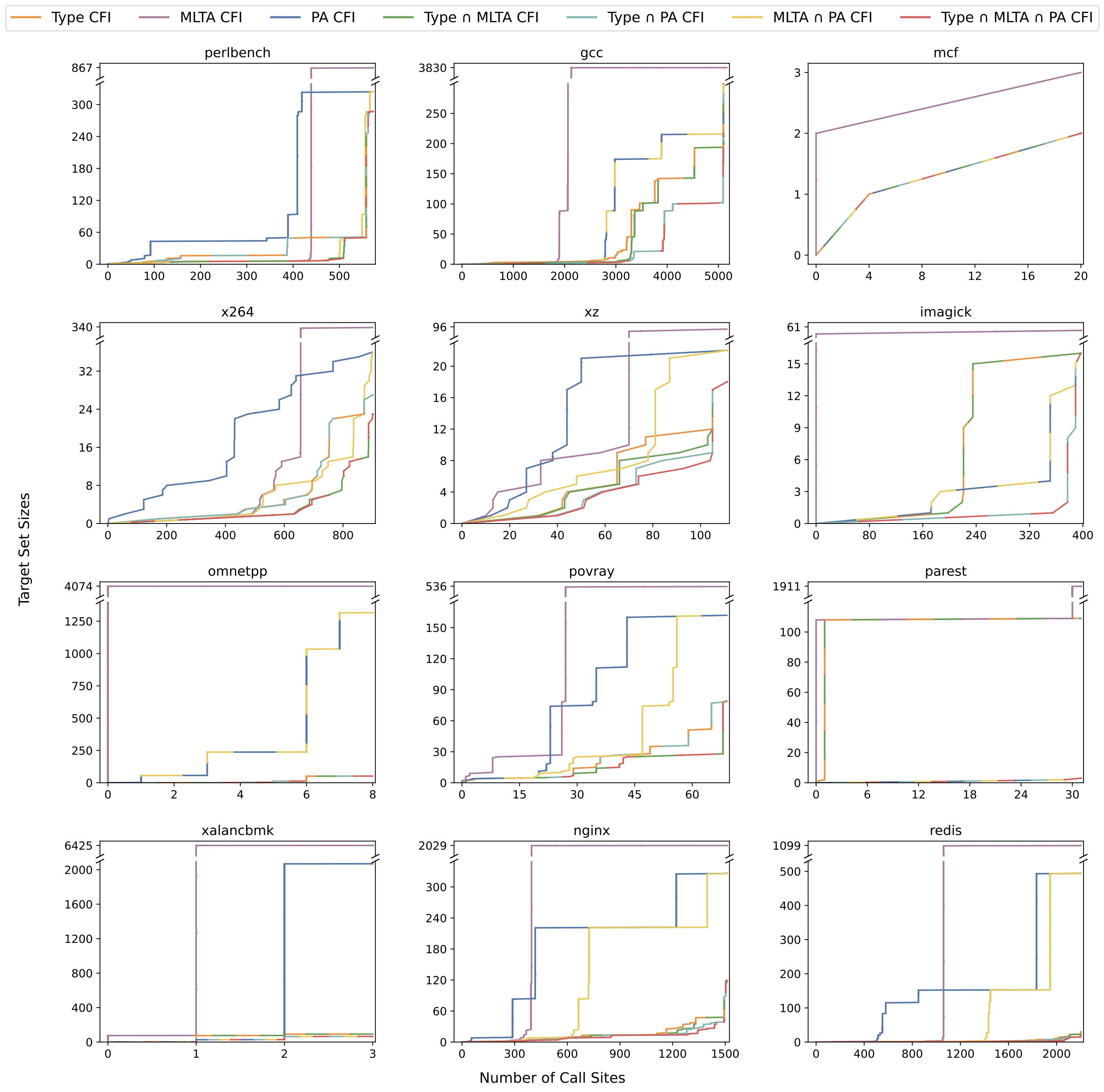


Figure 7: Cumulative target set sizes for pointer analysis CFI (PA CFI), type-based CFI (Type CFI), multi-layer type CFI (MLTA CFI), and all possible combinations for SPEC CPU 2017 benchmarks [17], Nginx [28], and Redis [33].

Table 6: Average (arithmetic and geometric) number of indirect call targets per call site of type-based CFI, MLTA CFI, pointer analysis CFI, as well as all combinations for SPEC CPU 2017 benchmarks [17].

Benchmark	Functions	Addr.-Taken Functions	Type		MLTA		PA		Type ∩ MLTA		Type ∩ PA		MLTA ∩ PA		Type ∩ MLTA ∩ PA	
perlbench	1,755	866	30.2	17.3	205.5	17.0	119.4	57.1	15.9	6.8	30.1	17.1	18.5	7.1	15.8	6.7
gcc	8,958	3,829	75.0	16.1	2,271.9	193.7	105.4	14.4	71.0	10.6	45.4	6.2	104.4	14.1	45.3	6.2
mcf	51	3	1.8	1.7	3.0	3.0	1.8	1.7	1.8	1.7	1.8	1.7	1.8	1.7	1.8	1.7
x264	948	339	6.8	3.7	94.7	8.3	18.9	13.8	3.7	2.3	6.8	3.7	6.4	3.3	3.7	2.3
xz	168	95	6.6	4.2	39.2	15.3	14.9	10.2	5.8	3.8	5.0	3.2	9.2	5.9	4.6	3.0
imagick	789	60	7.5	3.5	60.0	60.0	3.8	2.5	7.5	3.5	1.6	1.2	3.8	2.5	1.6	1.2
omnetpp	5,004	4,073	15.8	4.8	4,073.0	4,073.0	397.6	136.3	15.8	4.8	15.5	4.6	397.6	136.3	15.5	4.6
povray	1,223	535	23.1	14.5	334.7	146.3	89.1	43.3	15.8	11.4	23.1	14.5	50.7	23.0	15.8	11.4
parest	2,878	1,910	105.5	95.8	167.1	119.5	1.5	1.4	105.5	95.8	1.5	1.4	1.5	1.4	1.5	1.4
xalancbmk	8,711	6,424	57.0	24.2	4,308.0	1,463.8	699.7	48.8	57.0	24.2	31.7	15.4	31.7	15.4	31.7	15.4
nginx	3,966	2,023	16.4	10.0	1497.1	342.4	189.1	111.5	14.8	8.3	12.8	8.1	129.1	35.6	11.2	6.8
redis	4,891	1,098	4.0	2.3	570.3	47.4	167.2	60.4	3.3	2.0	2.8	1.8	94.0	9.1	2.3	1.6