


# SoK: Multi-Layer Indirect Call Analysis in the Real World

Yufei Du[1], Vasileios P. Kemerlis[2], Michalis Polychronakis[3], and Fabian Monrose[1]

[1]Georgia Institute of Technology
[2]Brown University
[3]Stony Brook University

## Abstract

Call graph analysis is foundational to a wide range of security-critical applications. A central requirement for these applications is the precise and sound identification of indirect call targets. Of late, type-based indirect call analysis (which matches address-taken functions and code pointers based on their types) has become a widely adopted solution for meeting that requirement. While scalable and nominally sound, traditional type-based analyses suffer from limited precision. In response, multi-layer type analysis was proposed as a remedy, augmenting type information with additional layers of reasoning to improve precision while retaining scalability and soundness. However, the complexity of these techniques has fueled an ongoing debate regarding both their practical precision gains and soundness guarantees in real-world settings.

In this work, we present the first systematic study of multi-layer type-based indirect call analysis, by evaluating the precision and soundness of five state-of-the-art multi-layer analysis techniques. Our study reveals a gap between the design of such techniques and their actual implementations, causing incomplete results with many indirect-call target sets missing or empty. In addition, our soundness experiments demonstrate that compiler optimizations cause every multi-layer approach to fall short of soundness. Furthermore, we conduct a case study to demonstrate that for control-flow integrity—one of the most popular downstream security applications of call graph analysis—existing multi-layer type-based techniques fall short in preventing attacks that exploit type collisions.

## 1 Introduction

Call graph analysis underpins a wide range of software security techniques, where the accuracy of the inferred graph directly shapes both effectiveness and compatibility. Directed fuzzers [5, 45], for instance, rely on call graphs to guide inputs toward target code locations. If the graph is unsound (i.e., missing valid call targets), the fuzzer may falsely conclude a path is unreachable. Likewise, if it is imprecise (i.e., including spurious targets), exploration slows down as the fuzzer wastes effort on irrelevant paths. Control-flow integrity (CFI) [14, 21, 22, 39, 48] faces similar challenges: an unsound graph can break compatibility by rejecting valid calls, while an imprecise one expands the adversary's options for code-reuse attacks. Beyond these cases, other security applications—such as debloating [1, 42], system call filtering [15, 20, 24], hardening [13], and intrusion detection [27]—also depend on precise call graphs to function reliably.

As functions can be invoked directly and indirectly, a complete call graph must include both direct and indirect call information. Extracting direct call targets is a trivial task, as the target function is hard-coded in the source code and the compiled binary. However, identifying all possible targets for indirect calls has been a challenging task because indirect calls use target addresses that are computed dynamically, at run time. Current approaches tackle this problem in various ways: using symbolic execution [4, 49], points-to analysis [3, 16], type analysis [8, 37, 48], or deep learning [12, 44, 55]. Of these, type analysis is the state-of-the-practice mainly due to its scalability [39, 48]. Because code pointers and functions both have their types defined in the source code, type analysis uses function types to filter address-taken functions for each indirect call site so that the target set of each indirect call includes only address-taken functions with matching types.

While type analysis is generally sound, it often lacks precision in real-world programs where many functions share the same type. To address this, Lu and Hu [37] introduced Multi-Layer Type Analysis (MLTA), which augments function type matching with `struct` hierarchy information when functions are assigned to pointers. By incorporating this additional layer, MLTA claims to substantially reduce false positives in identifying valid function-pointer targets within `structs`. Building on this idea, recent work [8, 30, 32, 35, 51] refined MLTA by redesigning multi-layer type matching or integrating complementary analyses to further boost precision without sacrificing soundness. Yet, despite these advances, researchers and practitioners remain divided on their practical utility.

Indeed, *ongoing debate* [36] about the precision and soundness of contemporary call graph inference techniques underscores the need for an independent evaluation. To move this debate forward, we present an in-depth study on the precision and soundness of multi-layer type-based indirect call analysis techniques. We use the single-layer type analysis approach of LLVM-CFI—the current state-of-the-practice type-based CFI scheme—as the baseline, and compare it with five multi-layer techniques: MLTA [37], DeepType [51], TFA [35], HPCFI [30], and KallGraph [32]. Specifically, we first reproduce the precision experiment of DeepType with its original dataset and configuration on the three approaches compatible with the dataset, in order to validate the reported results. Then, we conduct our independent precision and soundness experiments under our own datasets and setup, chosen to better reflect real-world conditions.

Our findings reveal a disconnect between the metrics emphasized in the aforementioned papers and their actual security impact. The original evaluations of three of the five approaches report precision *only* over the indirect calls that were successfully analyzed, while either ignoring those for which no target was found, or interpreting them as calls with no target—both of which are incorrect assumptions in practice. Our investigation shows a gap between the design of the techniques and the publicly-available implementations that causes the failed analyses. Our soundness evaluation shows that none of the multi-layer techniques could reach comparable soundness to the state-of-the-practice when analyzing programs with compiler optimizations enabled.

To explore the impact of multi-layer techniques on downstream security applications, we also conduct an offensive case study to determine if a multi-layer type-based CFI system could prevent type collision attacks [18]. The results are mixed: while some of the techniques could prevent type collisions in specific scenarios, none of the techniques could successfully prevent the attack for all the code patterns we test. We conclude our work by providing recommendations for future work in indirect call analysis and outlining promising directions for multi-layer type analyses.

Our main contributions are as follows:

1. *Reproduction of prior work*: We verify the evaluation results of MLTA [37], DeepType [51], and TFA [35] using the same dataset and setup as DeepType. Our results confirm that all approaches, when evaluated under DeepType's metric, perform as previously reported.
2. *Replication on a broader dataset*: We replicate MLTA [37], DeepType [51], TFA [35], HPCFI [30], and KallGraph [32] on a larger dataset of 81 LLVM bitcode files from 5 widely-used, open-source projects. After fixing design-implementation discrepancies that caused analysis failures, 4 of the 5 approaches outperform the baseline in precision.
3. *Soundness analysis*: We assess soundness across 11 open-source projects (13 LLVM bitcode files). Our findings reveal that while one approach is on-par with the baseline, when analyzing un-optimized LLVM bitcode files, none of the approaches could reach the same soundness as the baseline when optimizations are enabled.
4. *Offensive case study*: We explore the effectiveness of a CFI policy using call graphs from the 5 multi-layer approaches against type collision attacks. Our case study shows that for one code pattern, none of the approaches could successfully prevent type collision.

## 2 Background and Related Work

To contextualize the call graph analyses we study, Figure 1 presents an overview of a typical compilation pipeline that produces a security-enhanced binary. The process begins with the compiler translating each source file into a module in its *Intermediate Representation* (IR), where it applies standard optimization passes, like *Scalar Replacement of Aggregates* (SROA) and *Instruction Combining*. Once all source files have been lowered to IR, the linker merges them into a single module and applies link-time optimizations such as *Function Inlining* and global *Dead Code Elimination* (DCE). It is from this linked IR that call-graph analyses generate a call graph.

As shown in Figure 1, one such approach extracts the types of address-taken functions and the types of code pointers, and then includes in the call graph any address-taken function whose type matches the code pointer at an indirect call site. Security applications can then use the resulting call graph to enforce policies: for example, CFI inserts checks at indirect call sites to restrict the program's control flow. Finally, the IR is lowered to machine code for the target architecture.

All modern, high-performance toolchains include (among others) a compiler, a linker, and a machine code generator. In the case of the LLVM compiler project, the compiler is split between the `Clang` front end and the LLVM back end; the linker consists of `LLD`, while after linking, `LLD` invokes LLVM's `CodeGen` module to produce binary code.

### 2.1 Indirect Call Analysis

Unlike direct calls, whose targets are explicitly encoded in code, indirect calls rely on function pointers (or dispatch tables) and hence the respective target addresses are resolved only at runtime. Despite decades of research in program analysis, efficient and accurate solutions for identifying the precise set of indirect call targets remain an open problem.

To see why call graph construction is non-trivial, Listing 1 illustrates this difficulty with a motivating example using *type-based* CFI. In this example, the goal is to prevent each indirect call from targeting functions that would not be called under normal circumstances.

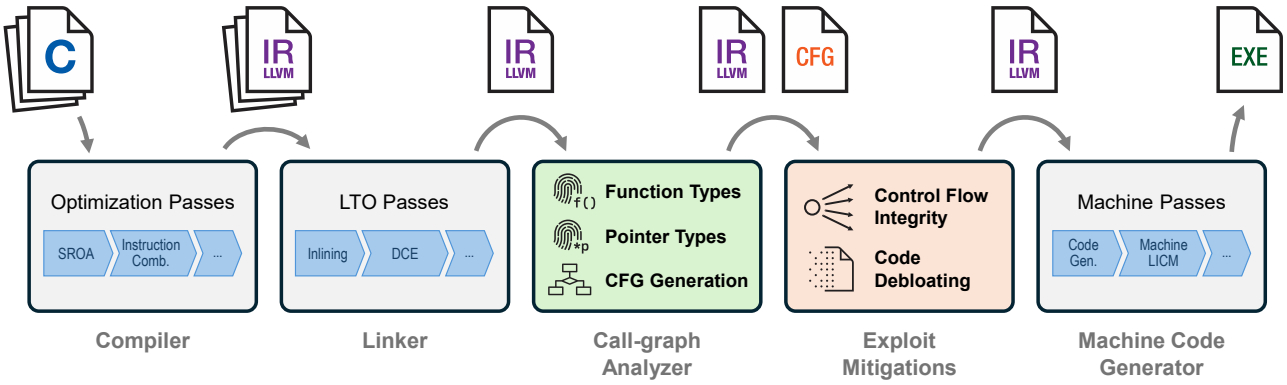



Figure 1: Placement of type-based approaches in the compilation pipeline and their use in security applications. The source code is first translated to individual modules in IR by the (LLVM) compiler front-end and optimized by the back-end. Then, the (LLVM) linker combines all modules into a single IR module and applies link-time optimizations. Call-graph analyzer uses the IR module to compute the call graph, which is used by security applications before generating the binary.

Assuming that the user's configuration always includes command line arguments (i.e., `argc` is always larger than 1), then the conditional statement at line 13 would always take the `if`-branch and not the `else`-branch. Therefore, in this configuration, the function `func_unused` should never execute. However, the current state-of-the-practice CFI defense (i.e., LLVM-CFI [48]) cannot prevent the indirect call at line 13 to target `func_unused`, because both `func_used` and `func_unused` share identical function types that match the type of the pointer `cfg_used.ptr`. On the other hand, while *points-to analysis*—which determines all possible values of a (code) pointer—could identify the single possible target (`func_used`) for the indirect call, precise points-to analysis is a known NP-Hard problem [10].

Hence, existing research in indirect call analysis focuses on balancing precision, soundness, and complexity. In practice, existing indirect call analyses follow either a *dynamic* or *static analysis* paradigm. We discuss each in turn.

```
1  struct config_used {void (*ptr) (int);};
2  struct config_unused {void (*ptr) (int);};
3
4  void func_used(int n);
5  void func_unused(int n);
6
7  int main(int argc, char *argv[]) {
8    struct config_used cfg_used =
9      { .ptr = func_used };
10   struct config_unused cfg_unused =
11     { .ptr = func_unused };
12   ...
13   if (argc > 1) {cfg_used.ptr(argc);}
14   else
15     {cfg_unused.ptr(argc);}
16   ...
17 }
```

Listing 1: Example code snippet showing the impact of indirect calls in type-based CFI.

**Inference via Dynamic Analysis** Dynamic analysis extracts control flow information by executing the program and tracing runtime information. Common dynamic analysis approaches include fuzzing [33] and concolic execution [41]. Fuzzing takes an initial input value(s) for the program and automatically executes the program with different variants of the input value(s) to explore as many code paths as possible.

Concolic execution blends dynamic execution with symbolic execution, and systematically explores all paths in a program's control flow graph. The main limitation of dynamic analysis is the tradeoff between coverage and scalability: fuzzing does not always guarantee a satisfying code coverage [33], where concolic execution still struggles with scalability: SymSan [11], one of the latest works in concolic execution, requires 17 hours to cover ~70% of `libjpeg`.

**Inference via Static Analysis** Static analysis, on the other hand, focuses on analyzing the program without executing it. Other than symbolic execution, which suffers from the same scalability issues [4] as concolic execution, static analysis approaches are more practical due to their achieved scalability and are commonly used in real-world program analysis tasks, such as control-flow integrity and debloating. Call graph analysis can be done before compilation at the source code level [23, 28], during compilation at the IR level [37, 51], or after compilation at the binary level [1, 50].

Conceptually, source code and IR analysis have similar information available (e.g., types of data structures and function boundaries); binary analysis has no access to this information but has the advantage of not requiring source code. Our study focuses on call graph analyses at the IR level as it is a popular choice by many security applications [23, 24, 45, 48].

At a high level, indirect-call static analysis approaches are primarily based on three techniques: *points-to* analysis, *type* analysis, or *machine learning-based* analysis. Points-to analysis focuses on identifying the possible values of (code) pointers, whereas type analysis filters the possible values of a (code) pointer by its type. In theory, points-to analysis should provide a precise target set of code pointers; however, as noted earlier, precise points-to analysis is an NP-Hard problem [10], forcing a tradeoff between precision and scalability [43] in real-world applications. For example, Coral [7], a state-of-the-art framework that uses a combination of pointer analysis techniques, requires over 2 hours and 60GB of RAM just to analyze a moderately-sized program (MariaDB).

Type analysis is more scalable but always over-approximates the targets of pointers. Evolved from the naïve approach where every address-taken function is considered as a possible target for every indirect call [14], state-of-the-practice approaches—like LLVM-CFI [48]—identify the type of code pointers and the type of address-taken functions and then filter the target set of each indirect call, leaving only functions with the matching type as the pointer. While this *single-layer type analysis* is sound, as Listing 1 demonstrates, it over-approximates the target set, hindering the protection provided by CFI as an attacker still has a large pool of target functions to choose from when hijacking the control flow.

To improve the precision of type matching, Lu and Hu [37] presented MLTA, which tracks the `struct` hierarchy of code pointers and address-taken function assignments. When the address of a function is assigned to a code pointer, MLTA tracks the `struct` hierarchy of the pointer; at call sites, MLTA further filters the target set to keep functions that match not only the type of the code pointer but also its `struct` hierarchy. As a simpler approach compared to points-to analysis, MLTA analyzes code pointer assignments by tracking the multi-layer types of the assigned pointer (for each address-taken function) instead of tracking the target addresses of each code pointer.

Recent work follows the same principles of MLTA but improves its design (e.g., DeepType [51]) and/or incorporates additional techniques, such as points-to analysis and *alias* analysis into multi-layer type matching (i.e., TFA [35], KELP [8], HPCFI [30], KallGraph [32]).

In recent years, researchers have also attempted to tackle indirect call analysis with machine learning-based approaches. CALLEE [55] and AttnCall [44] recover the call graph from binaries using a deep neural network, and SEA [12] uses large language models to analyze the source code with the goal of refining the call graph generated by traditional static analysis approaches. Neither of these approaches operate at the IR level, nor do they provide results that are easily interpretable.

## 2.2 Our Study

We focus on type analysis approaches because of their practicality and the significant progress they have seen in recent years. Table 1 lists the approaches we assess in this study. We use LLVM-CFI, the state-of-the-practice software-only CFI scheme, as the baseline approach because it uses single-layer type analysis for call-graph generation. Our choice to specifically evaluate MLTA, DeepType, TFA, HPCFI and KallGraph rests on several considerations.

Table 1: Approaches covered in this study and the artifact versions we use in our evaluation. For techniques, "ST" means Single-layer Type analysis; "MT" means Multi-layer Type analysis, and "PA" means Pointer Analysis.

| Approach | Technique | Version | LLVM Version |
|---|---|---|---|
| LLVM-CFI [48] | ST | LLVM 15 | 15 |
| MLTA [37] | MT | `1f2b4b7` | 15 |
| DeepType [51] | MT | `6c332d9` | 15 |
| TFA [35] | MT + PA | `1b2d45d` | 15 |
| HPCFI [30] | MT + PA | `1136477` | 13 |
| KallGraph [32] | MT + PA | `97980b7` | 14 |

First, MLTA introduced multi-layer type information and serves as the foundation for modern type analysis, while DeepType and KallGraph represent the latest advances, published at reputable venues (i.e., USENIX Security and IEEE S&P, respectively) and reporting substantial improvements over MLTA. TFA and HPCFI integrate multi-layer type analysis with data-flow analysis and are representative of approaches that combine different static analysis techniques. Second, all five approaches have publicly available artifacts, unlike KELP [8] that is unavailable. *To the best of our knowledge, this is the first independent study to compare leading type analysis approaches using a unified dataset, a widely adopted security metric, and an empirical soundness evaluation.*

# 3 Practical Multi-Layer Indirect Call Analysis

As noted earlier, we study two key aspects of indirect-call analysis: *precision* and *soundness*. Precision evaluates how effectively an approach removes false positive address-taken functions from the target set of each indirect call site. Soundness ensures that no valid targets are omitted, thereby avoiding false negatives. These two aspects play an important role on real-world security applications. An example of such a security application that relies heavily on indirect call analysis is CFI. In CFI, the precision of the constructed call graph dictates the *effectiveness* of the protection, while the soundness of the call graph is crucial for the *compatibility* of the defense.

We begin our evaluation by verifying the published *precision* results of the considered state-of-the-art multi-layer approaches using the same datasets, metrics, and experimental setup as DeepType [51]. Then, we replicate the precision evaluation using our own dataset and setup to assess whether the reported conclusions hold in real-world scenarios.

For the soundness evaluation, we collect runtime traces of indirect call invocations via fuzzing and compare them against the call graphs generated by each approach, checking for false negatives—i.e., targets observed at runtime but missing from the computed call graph.

Taken together, the setup allows us to explore the following research questions for this Systematization of Knowledge (SoK) work:

**RQ1** *To what extent can the reported precision of state-of-the-art approaches be reproduced using their original dataset and experimental setup?*

**RQ2** *To what extent do the reported precision results provide an accurate measure of the effectiveness of these approaches in security applications?*

**RQ3** *Can the precision results of state-of-the-art approaches be replicated when evaluated on alternative datasets and experimental setups?*

**RQ4** *Do state-of-the-art approaches ensure the construction of a sound call graph?*

**RQ5** *Can state-of-the-art approaches enhance the current state-of-the-practice type-based CFI policy in preventing control-flow hijacking attacks?*

In addition to studying the above research questions, we also explore new directions in type-based call graph analysis and provide recommendations for future work. While our work focuses on the precision and soundness of the multi-layer approaches, we also include a performance experiment that studies the runtime overhead on the compilation time for HPCFI in Appendix C.

**Datasets** Table 2 summarizes the datasets used in each evaluation. For the precision evaluations, we restrict the dataset to programs that all approaches successfully analyze. For the soundness evaluations, we include programs that some approaches fail to analyze in order to preserve a sufficiently large and representative dataset.

For reproducing precision results, we use the dataset available in DeepType's artifact, consisting of LLVM bitcode files for 20 C programs from open-source projects, such as `httpd` and `binutils`. We exclude one bitcode file, `vmlinux`, from our experiment because it causes TFA to crash, leaving 19 bitcode files in total. All programs in the reproduction dataset are built at optimization level `-O0`. We use an identical procedure as the one outlined in the DeepType paper [51]: we run each approach to analyze every individual LLVM bitcode file, treating each file as a separate, standalone program.

Our replication dataset consists of 81 programs across 5 C projects from the OSS-Fuzz repository [25]. On average, each project includes 119727 lines of code (LoC). Our selection of projects is based on the compatibility of the projects' build scripts with our workflow; each project needs to be: 1) compilable with LLVM-CFI; 2) compatible with compiler flags that disable opaque pointers, as this is something required by some of the considered approaches; 3) compatible with our LLVM toolchain that generates bitcode files at link time (along with the respective binaries); and 4) supported by all 5 approaches we evaluate. We exclude any program that does not contain any indirect call as they are irrelevant to our evaluation. We build the projects using the `Debug` (`-O0`) and `Release` (`-O2` or `-O3`) configurations with debug symbols enabled.

Table 2: Summary of datasets.

| Dataset | Data Source | Project Count | Bitcode Files |
|---|---|---|---|
| Precision-Reproduction | DeepType | 7 | 19 |
| Precision-Replication | OSS-Fuzz | 5 | 81 |
| Soundness | OSS-Fuzz, UNIFUZZ | 11 | 13 |

For the soundness evaluation, we use the UNIFUZZ benchmark suite [34] and three projects from the OSS-Fuzz repository: `libjpeg-turbo`, `sqlite` and `httpd`. We use 9 of the 20 programs in the UNIFUZZ benchmark suite that can be compiled using our workflow and that include indirect calls. On average, each project includes 612715 LoC; with the largest project (`binutils`) excluded, the rest of the programs contain an average of 273677 LoC. We extract the ground truth by fuzzing each project for 4 hours using the AFL++ fuzzer [2], one of the fuzzers with the highest code coverage according to Google's latest FuzzBench [38] report [26]. We collect indirect call traces via LLVM's coverage sanitizer [46], which prints the caller and callee addresses before each indirect call. Our method (i.e., collect runtime traces via fuzzing) is on a par with the methodology of previous work [12].

## 4 Multi-Layer Type Analysis Precision

For evaluating precision, we consider the following five approaches: MLTA [37], DeepType [51], TFA [35], Kall-Graph [32], and HPCFI [30]. We also include the results of LLVM-CFI [48] as the baseline. As we discussed in Section 2, our choice of prior works is based on recency, prototype availability, and dependency on a common base compiler.

### 4.1 Verification of Prior Results (RQ1-2)

First, we verify the reported precision results (of each considered approach/paper) by reproducing DeepType's precision experiment. Specifically, we use the same precision metric as DeepType—i.e., the *Average Number of Target* (ANT)

Table 3: *Reproduction (RQ1)* results using the ANT metric. Approaches with the best score are in bold. For brevity, only the average result of `binutils` (13 binaries) is listed.

| Program | MLTA | DeepType | TFA |
|---|---|---|---|
| `binutils` (avg.) | 10.98 | 2.47 | **1.97** |
| `sqlite` | 8.32 | 11.93 | **5.26** |
| `nginx` | 5.58 | 6.38 | **5.30** |
| `httpd` | 12.26 | 6.23 | **5.75** |
| `openvpn` | 1.63 | 2.35 | **2.22** |
| `proftpd` | 2.96 | 3.10 | **2.69** |
| `sshd` | 5.57 | 5.43 | **5.21** |
| **Average** | 9.43 | 3.56 | **2.74** |

metric—which is defined as:

$$ANT = \frac{num(T)}{num(IC)}$$

$num(T)$ is the total number of identified targets and $num(IC)$ is the number of indirect calls that have at least one target identified by the approach. Assuming sound results, a lower ANT value indicates higher precision, as the approach is able to filter out more functions from the target set.

For this experiment, we are only able to evaluate three approaches: MLTA, DeepType, and TFA. Because we use the dataset in DeepType's artifacts, only the LLVM bitcode files are available, and all bitcode files were compiled using LLVM v15. This means that approaches that require source code (e.g., LLVM-CFI, HPCFI) or expect a different compiler version (i.e., KallGraph) cannot be assessed.

**Experimental Results for ANT (RQ1)** Table 3 presents the results of our reproduction experiment using the metric of ANT. TFA performs best, as its results have the lowest ANT for most programs, and DeepType follows closely. Overall, all three approaches appear to achieve satisfactory precision. Even MLTA, the oldest approach of the three that yields comparatively lower results, achieves an acceptable average target set size of about nine address-taken functions per indirect call.

Our results reproduce the precision reported in the respective papers [35, 37, 51]. For DeepType, our measurements match the published results for all programs except `sqlite`. The discrepancy for `sqlite` arises because we used a version of DeepType released after the original publication. When we repeated the experiment with the version used in the DeepType paper, we obtained identical results across all programs.

For MLTA and TFA, although published results are reported using a slightly different variant of the ANT metric, and a different dataset, our findings are consistent with the ones in the respective papers. For most programs, our results also align with TFA's evaluation, which reports a 24%–59% improvement in precision over MLTA for C programs. Our MLTA measurements differ slightly from those reported in DeepType's evaluation, again because the version of MLTA we use was updated after DeepType's publication.

Table 4: Number of indirect calls with an empty target set for each program in the reproduction dataset.

| Program | MLTA | DeepType | TFA |
|---|---|---|---|
| `binutils` (avg.) | 44.8% | **14.2%** | 40.2% |
| `sqlite` | 19.1% | **4.9%** | 16.0% |
| `nginx` | 33.8% | **16.1%** | 32.9% |
| `httpd` | 29.2% | **21.5%** | 39.2% |
| `openvpn` | 49.2% | **22.2%** | 26.9% |
| `proftpd` | **24.3%** | 30.1% | 27.6% |
| `sshd` | 63.1% | **47.3%** | 50.8% |
| **Average** | 42.2% | **17.2%** | 37.7% |

> *Insight 1.* Our reproduction study (RQ1) systematically validates the reported precision outcomes of all three state-of-the-art approaches by employing DeepType's original dataset and experimental configuration. For DeepType, our results are identical to the original evaluation for all but one program. For MLTA, our results are nearly identical to the results shown in DeepType's evaluation for all programs. For TFA, while the results are not directly comparable due to different datasets, our numbers indicate similar or better improvement (24%–59%) over MLTA than the original evaluation for 14 of the 19 programs.

**Misguided Precision Metric (RQ2)** While we successfully reproduced the experiment results of the corresponding papers, we emphasize that the use of the ANT metric can lead to misguided conclusions. ANT excludes any indirect calls that have an empty target set. It is highly unlikely that an indirect call legitimately has no target in the original program. Instead, such indirect calls usually suggest that the analyzer failed to identify *any* target for the corresponding calls. In fact, we argue that *indirect calls with no identified targets are more problematic*, because there is no (known) constraint on the functions they branch to. Security applications are left with two options for these indirect calls: either assume the reported empty target sets are truly empty, which will likely result in unsoundness, or fall back to another approach, to trade precision for soundness. In the worst case, the security application would have no choice but to allow these indirect calls to target any address-taken function, to maintain soundness.

To emphasize the problem with the ANT metric, we measure the number of indirect calls with an empty target set. Table 4 shows the ratio of indirect calls with an empty target set for each program in our reproduction dataset. TFA, while achieving a better precision than the other two approaches in ANT, seems to exclude significantly more indirect calls.

Table 5: *Reproduction (RQ2)* results using the AICT metric, which does not ignore empty call target sets.

| Program | MLTA | DeepType | TFA |
|---|---|---|---|
| `binutils` (avg.) | 404.01 | **81.11** | 359.33 |
| `sqlite` | 203.01 | **57.91** | 169.19 |
| `nginx` | 243.11 | **114.80** | 236.83 |
| `httpd` | 533.70 | **213.46** | 706.83 |
| `openvpn` | 43.14 | **20.49** | 24.83 |
| `proftpd` | 120.75 | **50.38** | 136.43 |
| `sshd` | 124.58 | **79.05** | 101.26 |
| **Average** | 343.18 | **83.71** | 318.24 |

The high ratio of indirect calls with an empty target set shows that ANT cannot capture the overall picture for precision: if an approach can successfully analyze only one indirect call in a program, with good precision, it can still achieve a great ANT number by simply returning empty sets for every other indirect call even though such approach would be useless for any practical task. Moreover, our experiment reveals a gap between the design of the approaches and their implementations: even though the design of every approach claims to fall back to single-layer type analysis on failure, their implementations currently do not exhibit this behavior.

Therefore, instead of ANT, we argue that precision should be measured using the Average Indirect Call Target (AICT) metric, which is commonly used to measure the precision of control-flow integrity approaches, and—crucially— takes *all* indirect calls into consideration [31, 53, 55]. AICT is conceptually similar to ANT, but it does not exclude indirect calls. Ideally, if an approach has properly implemented a fallback behavior, any indirect call that the approach fails to analyze would include a target set filtered by single-layer type matching. While this holds in theory, in practice, because none of the approaches properly implement a fallback behavior, each indirect call with an empty target set is treated as an indirect call with every address-taken function in the program as a legal target. AICT is defined as:

$$AICT = \frac{num(T) + num(AF) \times num(IC_{empty})}{num(IC_{all})}$$

$num(T)$ is the total number of identified targets, $num(AF)$ is the total number of address-taken functions in the program, $num(IC_{empty})$ is the number of indirect calls that have no target identified by the analyzer, and $num(IC_{all})$ is the total number of indirect calls.

Table 5 reports results on the reproduction dataset using the AICT metric. In contrast to ANT, AICT presents a drastically different picture. Measured in AICT, most results are more than 10× worse than in ANT, and in some cases (e.g., certain programs in `binutils` and `httpd`) TFA degrades by over 100×. These discrepancies match the large number of indirect calls with an empty target set we see in every test program. Accounting for such unresolved calls, DeepType outperforms the rest on all programs and achieves the best overall.

> *Insight 2.* The reported precision results are not a reliable indicator of effectiveness for security applications (RQ2). Due to a gap between design and implementation, each approach fails to analyze a large number of indirect calls and reports an empty target set. When evaluated with a metric that more accurately reflects real-world usage scenarios, MLTA, DeepType and TFA exhibit degraded performance, indicating a substantial space for improvement.

## 4.2 Replication (RQ3)

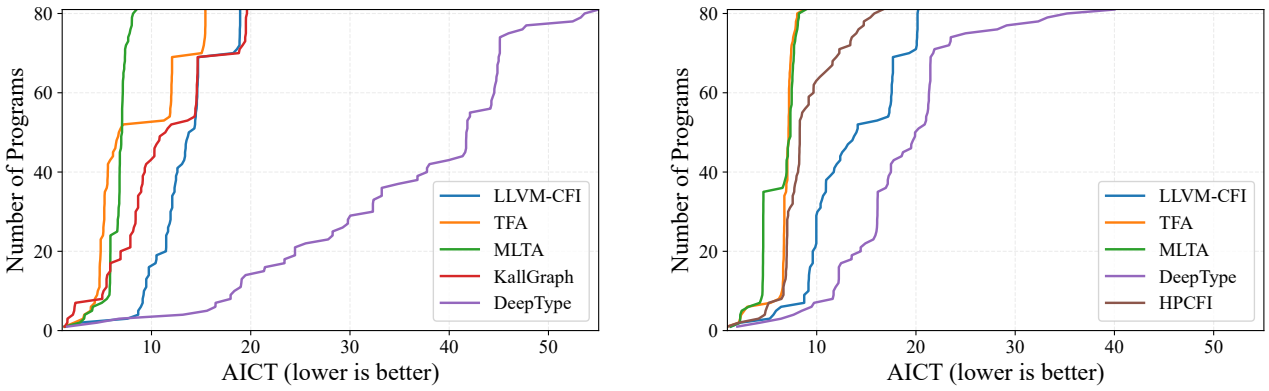
We now evaluate the precision of all five approaches using our replication dataset.

**Experimental Setup** As discussed in Section 3, we build our dataset using both `Debug` and `Release` configurations. We use two different optimization levels for the following reasons. First, the original evaluation of most approaches only focuses on one configuration, and evaluating at different optimization levels allows us to study the impact of compiler optimizations on indirect-call analysis. Second, parts of HPCFI can only be invoked at `-O1` or higher, but KallGraph crashes on most programs compiled using the `Release` configuration. Therefore, we need both the `Debug` and `Release` configurations to evaluate every approach. Because the approaches expect different versions of the LLVM compiler, we build each configuration using each compiler version.

For both configurations, we also build a version with LLVM-CFI enabled to extract the baseline call graph. As the current state-of-the-practice, LLVM-CFI uses single-layer type-based indirect-call analysis for its CFI policy. As Section 4.1 points out, the current implementations of the approaches we evaluate are missing the functionality to fall back to single-layer type analysis on failure, contrary to their design. To better simulate real-world scenarios where the approaches are implemented following their design including the fallback behavior, we use the baseline as the fallback for every approach: every time we encounter an indirect call with an empty target set, we replace the empty set with the target set of that call as computed by LLVM-CFI.

As before, precision is measured using the AICT metric.

**Results** On average, MLTA performs the best in both configurations, achieving an AICT of 6.52 in `Debug` and 6.05 in `Release`. Figure 2 presents the results of our replication experiment using Cumulative Distribution Function (CDF) plots. In both configurations, MLTA and TFA almost always outperform the other approaches. In the `Debug` configuration, KallGraph achieves a better AICT than the other approaches




(a) `Debug` (b) `Release`

Figure 2: CDF of the precision results for the replication dataset.

for a small set of programs but performs worse for larger programs. In the `Release` configuration, HPCFI achieves slightly worse AICT than MLTA and TFA for almost all programs. Finally, with a few exceptions, DeepType performs the worst in both configurations, even falling behind the baseline for the majority of the test programs.

Compared to the reported results of each approach, our numbers partially support the original findings. First, for MLTA, we measure an improvement of 49.8% (in `Debug`) and 53.7% (in `Release`) in precision compared to the baseline, but the improvements are lower than the reported 86%–98% in the original publication. Second, HPCFI outperforms the baseline LLVM-CFI by 33.8% on average, close to the original report of 40% in HPCFI's paper. On the other hand, our results are unable to support the findings in the original publications of DeepType, TFA, and KallGraph. All three approaches reported better precision than MLTA, contrary to our results. We admit that we use a newer version of MLTA that was released after the publication of all other approaches, which could lead to better results.

The replication experiment paints a drastically different picture compared to the reproduction experiment, where DeepType performs significantly better than MLTA and TFA. However, this is not unexpected because MLTA and TFA now fall back to LLVM-CFI's target set for indirect calls with empty sets, instead of the set of all address-taken functions. Since even the baseline LLVM-CFI outperforms DeepType in precision, this fallback behavior causes MLTA and TFA to achieve better precision than DeepType.

Surprisingly, other than the baseline, every approach that works on both configurations achieves a better average precision in `Release` compared to `Debug`. DeepType shows the biggest improvement of 26.5%, from an average AICT of 23 in `Debug` to an average AICT of 16.9 in `Release`. On the other hand, the baseline LLVM-CFI shows slightly worse precision in `Release` compared to `Debug`.

> *Insight 3.* Our results indicate that, with the added functionality to fall back to single-layer type analysis, the reported precision results in MLTA and HPCFI could be replicated even though our results show slightly lower improvements over the baseline (RQ3). More importantly, the drastically different results between our replication experiment and the reproduction experiment demonstrate the importance of a proper fallback method—as multi-layer approaches still face challenges identifying all indirect calls, falling back to the set of all address-taken functions is prohibitively expensive in precision. Furthermore, our results show interesting behavior at different optimization levels: all approaches, other than the baseline LLVM-CFI, achieve better precision at higher optimization levels.

## 4.3 Hybrid Approach

While our experiment results indicate that MLTA achieves the best precision among the considered approaches, there is still room for improvement as MLTA does not always generate the smallest target set for every indirect call. To further improve the precision of the call graph, we explore a hybrid scheme that uses the results generated by each approach to form a call graph with the best precision.

We follow a straightforward design for the hybrid approach: for each indirect call in the program, we query the target set generated by each approach and select the smallest non-empty target set for the call. Because the hybrid scheme uses the generated call graph of every approach, including the one of the baseline, it naturally falls back to the single-layer approach if every multi-layer approach fails. We conduct the precision evaluation again using the replication dataset with our hybrid approach included and see a noticeable improvement in both `Debug` and `Release` configurations.

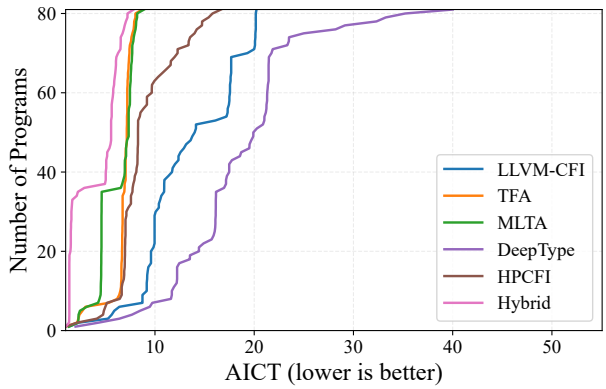



Figure 3: CDF of the precision results for the `Release` configuration with our hybrid approach.

Figure 3 shows the results for the `Release` configuration. For half of the programs, the hybrid approach achieves more than 50% better precision than the next best approach, MLTA. On average, the hybrid approach outperforms MLTA in precision by 35%.

> *Insight 4.* For precision-focused tasks, current multi-layer approaches can still be improved. While MLTA has the overall best precision, it cannot generate the most precise target set for all indirect calls inside a binary. By combining the results from different approaches, our hybrid scheme is capable of generating call graphs that are more precise than MLTA.

## 5 Revisiting Soundness (RQ4)

While precision is an important factor that shows the improvement of an approach, soundness is the minimum requirement for an approach to be worth adopting in real-world tasks. Even if an approach can eliminate all false positives for each indirect call, if its results are unsound and include false negatives, security tasks such as CFI cannot use the approach without breaking legal indirect calls in programs. Hence, we argue that soundness is also an important factor when evaluating work in indirect-call target analysis.

### 5.1 Experimental Setup

For our soundness assessment, we collect indirect call traces by fuzzing the programs in our soundness dataset, which includes 11 projects and 13 programs. We then build these programs at two optimization levels, `-O0` and `-O3`, and use each approach to analyze them. As discussed in Section 4.2, we evaluate KallGraph [32] only on un-optimized programs, because it frequently crashes when analyzing code compiled at `-O2` or `-O3`, and we evaluate HPCFI [30] only on optimized programs, because its implementation does not fully execute at `-O0`. We evaluate the other approaches (and the baseline) at both optimization levels to study the impact of compiler optimizations on soundness.

As with the replication experiment, we use the results of the baseline (LLVM-CFI) as the fallback method for the multi-layer type-based approaches in situations where the approach cannot identify any target for an indirect call. We also include the hybrid approach we discussed in Section 4.3 to explore the potential cost in soundness for maximizing precision.

We use *recall* as the metric for soundness, defined as:

$$Recall = \frac{num(TP)}{num(TP) + num(FN)}$$

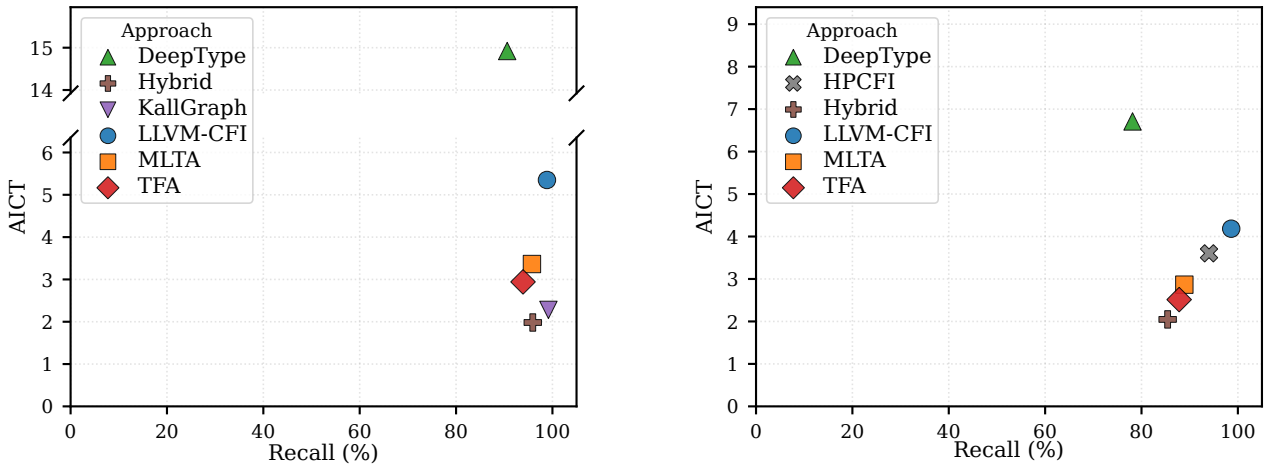
$num(TP)$ is the number of caller-callee pairs identified that are in the ground truth, and $num(FN)$ is the number of pairs in the ground truth that are missing in the analyzed result. A recall of 100% implies sound analysis, where a lower recall indicates missing call targets.

### 5.2 Results

Of the 13 programs in our soundness dataset, only five could be successfully analyzed by all approaches without crashing, hanging, or error. Figure 4 presents the average soundness (in recall) and the average precision (in AICT) across these five programs at both optimization levels. Unfortunately, none of the approaches, even including the state-of-the-practice LLVM-CFI, could guarantee soundness, as no approach reaches a recall of 100%. At `-O0`, all approaches reach a recall of 90% or higher. With the exception of DeepType, all multi-layer approaches outperform the baseline in precision. In particular, KallGraph achieves both the best precision, excluding the hybrid approach, at an average AICT of only 2.28, and the best soundness, at an average recall of 99.2%. Compared to the baseline, the average precision of KallGraph is 57.5% better while achieving the same level of soundness. However, the results at `-O3` are not as optimistic.

At `-O3`, none of the multi-layer approaches could reach the soundness of LLVM-CFI, whose average recall is 98.6%. The closest contender, HPCFI, scores a recall of 94%, but only shows a small improvement of 13.9% in precision compared to LLVM-CFI. Also, HPCFI is compatible with the least number of programs: 10 of the 13 programs can be successfully analyzed by all other approaches, and HPCFI could only succeed on 5 of them. All approaches that support both optimization levels, including MLTA, DeepType, and TFA, show degraded soundness when analyzing the same programs at `-O3` (compared to `-O0`). The average recall of all three approaches at `-O3` are below 90%: MLTA drops from 95.8% to 88.9%, TFA drops from 93.9% to 87.7%, and DeepType drops from 90.6% to 78.1%.

The hybrid approach we proposed that maximizes precision achieves an average recall of 95.9% at `-O0` and 85.4% at `-O3`.




(a) `-O0` (b) `-O3`

Figure 4: Average soundness results across the 5 programs that all approaches could successfully analyze.

At `-O0`, compared to KallGraph, the approach with the best soundness, the hybrid approach shows a better average precision of 13.1% at the cost of a worse average recall of 3.25%. At `-O3`, compared to HPCFI, the hybrid approach shows a bigger improvement in precision (43.2%) at the cost of a worse average recall (8.6%). For security applications that benefit from a precise call graph but do not require a sound call graph, such as bug reachability analysis [40], the hybrid approach could potentially be a good fit.

We discuss additional results, including the programs that HPCFI could not analyze, in Appendix A. Our findings also apply to the additional results.

> *Insight 5.* Although none of the techniques achieves full soundness across all test programs (RQ4), KallGraph attains soundness comparable to the state-of-the-practice LLVM-CFI at the `-O0` optimization level. However, under `-O3`, none of the multi-layer approaches matches LLVM-CFI's recall. For security mechanisms that depend on a sound call graph, these multi-layer techniques may therefore introduce greater compatibility issues than single-layer type analysis when compiler optimizations are enabled.

## 5.3 LLVM-CFI+Multi-Layer Type Analysis

Our results reveal that while LLVM-CFI, the baseline approach that uses single-layer type analysis, generates call graphs with better soundness than multi-layer type-based approaches, it is not without limitations, as it still cannot reach 100% recall for all programs.

On the other hand, multi-layer approaches do not inherit the same limitations, allowing them to correctly include the targets missed by LLVM-CFI. Specifically, two limitations of LLVM-CFI can be improved by multi-layer approaches.

**Issue #1: Gaps in Type Matching** LLVM-CFI extracts the type of indirect calls in the Clang front end, when source-level information is still available. Afterwards, however, LLVM-CFI waits until the linker stage to extract the type of address-taken functions and to perform type matching, at which point the source-level information no longer exists. This gap causes LLVM-CFI to miss targets in two scenarios: indirect calls targeting assembly functions and indirect calls targeting external library functions (see Appendix B).

LLVM-CFI cannot detect address-taken functions defined in assembly *even if such a function is declared with its full function type in C*. Listing 2 shows an example indirect call from `ffmpeg`, where LLVM-CFI misses a target function because it is defined in assembly. The target function is declared in C at line 1. At line 6, this function is assigned to the function pointer `fdsp->butterflies_float`. This function pointer is then used for the indirect call at line 13. During compilation, LLVM-CFI correctly identifies the type of the function pointer as `void (float*, float*, int)`. However, at the linker stage, LLVM-CFI could not identify the type of the function `ff_butterflies_float_sse`, causing the target set for the indirect call to be unsound, which could lead to compatibility issues—our indirect call traces from fuzzing show that this function has indeed been targeted. On the other hand, DeepType correctly includes this function in the target set for the indirect call because it keeps track of the multi-layer type of address-taken functions when they are assigned to a pointer, instead of relying on LLVM metadata at the linker stage.

```
void ff_butterflies_float_sse(float *av_restrict
    src0, float *av_restrict src1, int len);

av_cold void ff_float_dsp_init_x86(
    AVFloatDSPContext *fdsp)
{
  ...
  fdsp->butterflies_float    =
      ff_butterflies_float_sse;
  ...
}

static void compute_stereo(MPADecodeContext *s,
    GranuleDef *g0, GranuleDef *g1)
{
  ...
  s->fdsp->butterflies_float(...);
  ...
}
```

Listing 2: Code snippet from `ffmpeg` that causes false negatives due to Issue #1.

```
SQLITE_API const unsigned char *SQLITE_STDCALL
    sqlite3_value_text(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16le(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16be(sqlite3_value*);
```

Listing 3: Code snippet from `sqlite` that causes false negatives due to Issue #2.

**Issue #2: Strict Type Checking with No Data-flow Analysis** While the design of type-based CFI enforces each indirect call to only call functions with matching types, real-world programs may not always satisfy this requirement, as `void*` pointers are still commonly used. Without checking if a type mismatch can legitimately occur using data-flow analysis, type analysis may generate call graphs with false negatives, leading to compatibility issues for CFI.

Listing 3 shows an example in `sqlite` that includes the declaration of four address-taken functions called by the same indirect call. The first function, `sqlite3_value_text`, has a different return type (`unsigned char *`) compared to the return type of other functions (`void *`). As a result, LLVM-CFI does not consider that function to be a legal target even though our indirect call traces indicate that this function has been indirectly invoked. In contrast, all multi-layer approaches that could successfully analyze this program, including MLTA, DeepType, and TFA, identify this target function correctly, likely because they track the assignments to function pointers in addition to applying type matching.

> *Insight 6.* While our evaluation indicates that multi-layer techniques generally achieve lower soundness than LLVM-CFI, they can identify correct call targets in cases where LLVM-CFI fails. Thus, multi-layer type-based call graph analysis may help address limitations of the state-of-the-practice CFI approaches.

## 6 On The (In)-Effectiveness of Multi-layer Type-based CFI Against Type Collision Attacks (RQ5)

One major security application that relies on indirect-call analysis is CFI. Specifically, CFI is a practical defense to reduce the attack surface of C/C++ programs and has been adopted in Android and the Linux kernel [29]. Nevertheless, previous studies [9, 17] have shown that even with a perfectly precise call graph, it is possible to manually craft control-flow hijacking attacks against CFI-protected binaries; recent work by Zhang et al. [54] further extended this line of inquiry by automating gadget construction.

In what follows, we show that due to over-approximated call graphs, the current state-of-the-practice in software-based CFI remains ineffective against a form of code reuse attacks, coined type collision [18], which operates under more relaxed requirements than control-flow "bending" attacks [9, 17, 54]. A type collision attack corrupts a function pointer to target an arbitrary function that has the correct type, with the goal of executing a system call (e.g., `execve` to allow arbitrary code execution). Even armed with a more precise call graph generated by multi-layer type analysis approaches, CFI is still incapable of defending against type collision attacks.

**Model Application** For this study, we construct a model application, written in C, which mirrors the functionality of a remote shell (e.g., SSH). Specifically, it "listens" on a TCP port and accepts three types of command from a client: `authenticate`, `execute`, and `quit`. The client needs to authenticate first before sending other commands, with server-side checks implemented to disallow processing `execute` or `quit` commands without authorization. If authentication fails, the server closes the connection and re-starts.

Figure 5 shows a simplified version of the call graph of the server program, with nonessential functions omitted. Without authentication, the program can only legally execute `handle_auth` and `callback_time` from the main loop in `listen`. The *handler* (blue nodes) and *callback* functions (green nodes) share the same function type, resembling functions that share the same type in large, real-world applications. Each command from the client is stored as a `Command` object, including the two function pointers that point to the handler and callback functions.

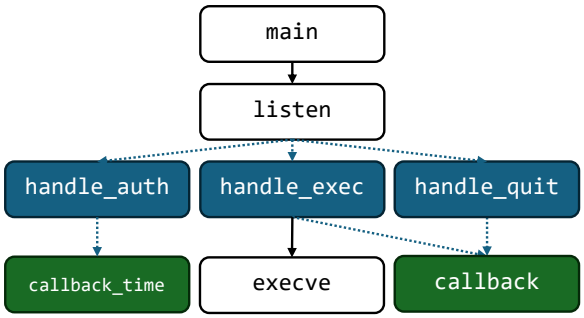



Figure 5: Simplified call graph of the model application. Solid arrows represent direct calls; dotted arrows represent indirect calls. Colored nodes correspond to address-taken functions.

```
struct Command {
    char cmd;
    const char *name;
    bool requires_auth;
    char payload[128];
#ifdef VARIANT_1 // Variant 1:
    command_fn_t ptr0; // Handler
    command_fn_t ptr1; // Callback
#elif VARIANT_2  // Variant 2:
    command_fn_t ptrs[2];
#else            // Variant 3:
    command_fn_t *ptrs;
#endif
};
```

Listing 4: Code snippet from our example server program showing the struct that stores the command.

Listing 4 shows the code for the `Command struct`. To simulate the variety of code patterns in real-world programs, and to test the capabilities of each multi-layer type analysis approach, we include three different ways to store function pointers inside the `struct` object. In Variant 1, we store each function pointer directly as a field in the `struct`. In Variant 2, the function pointers are stored inside an array (`ptrs`) in the `struct`. In Variant 3, the function pointers are stored in an array that exists in heap memory, and the pointer to the array (`ptrs`) is then stored in the `struct`.

**Experimental Setup** We assume that the attacker's goal is to execute `libc`'s `execve` function without a successful authentication. For this case study, we assume that a memory error exists within the `handle_auth` function. Therefore, the attacker aims to corrupt the function pointer that points to the callback function, used by the indirect call in `handle_auth` (`ptr1` for Variant 1 and `ptrs[1]` for Variant 2 and 3), and overwrite its value to make it point to `handle_exec`. We note that because the indirect call only targets `callback_time` in the most precise call graph, more powerful control-flow bending attacks [9, 54] are not possible because there is no legal target to "bend" the control flow to.

We run each of the five multi-layer type analysis approaches to generate the call graph of the program. We then assume that the program is protected with a CFI policy generated from the respective call graph: if the target set of the indirect call in `handle_auth` includes `handle_exec`, then the attack could succeed; otherwise, the attack would be prevented. For this case study, we run KallGraph to analyze the bitcode files compiled with both `-O0` and `-O3`, because the server program is simple enough that KallGraph can analyze bitcode at `-O3` without crashing.

**Results** Table 6 presents the results for the case study. MLTA achieves the best results overall by correctly excluding `handle_exec` in Variant 1 and 2, at both `-O0` and `-O3`. HPCFI only works at optimization levels above `-O0`, but it is the only approach that correctly excludes `handle_exec` in Variant 3. The other approaches, excluding DeepType, only correctly handle Variant 1, and even for this simplest variant, TFA is still unable to exclude the function from the target set when analyzing the `-O3` bitcode. Moreover, while the other approaches either include all address-taken functions with matching type in the target set or fall back to single-layer type analysis (which also includes all address-taken functions with matching type), KallGraph generates an unsound call graph for this indirect call when analyzing the `-O3` bitcode: its target set includes `callback` but not `callback_time`. If the binary is protected by a CFI policy using KallGraph's computed call graph, then even normal execution would lead to a CFI violation, resulting in compatibility issues.

> *Insight 7.* Even with a fine-grained CFI policy that uses multi-layer type analysis approaches to generate the respective call graph, easily constructed type collision attacks still pose a threat (RQ5).

## 7 Recommendations

**Falling Back Optimally** Our precision evaluation highlights the need for a fallback approach when the multi-layer type analysis fails. Because multi-layer approaches use a chain of type and `struct` information to match an indirect call with address-taken functions, it is expected that failures can happen, due to the complex code patterns or due to implementation bugs that are understandably challenging to identify when working at the level of compiler IR. In such cases, simply returning an empty target set for the indirect call is not enough.

For an approach to be adopted by security applications in the real world, it needs to have a coverage guarantee with well-defined exceptions (e.g., indirect calls in inline assembly code are not analyzed) and an implementation that matches the respective design.

Table 6: Type collision results. "✓" means the approach correctly excludes `handle_exec` from the target set, preventing type collision. "✗" means the approach includes `handle_exec` in the target set, enabling type collision. "⊥" means the approach generates an unsound target set, causing the program to crash under normal circumstances.

| Code Pattern | MLTA | | DeepType | | TFA | | KallGraph | | HPCFI | |
|---|---|---|---|---|---|---|---|---|---|---|
| | -O0 | -O3 | -O0 | -O3 | -O0 | -O3 | -O0 | -O3 | -O0 | -O3 |
| Variant 1 | ✓ | ✓ | ✗ | ✗ | ✓ | ✗ | ✓ | ✓ | - | ✓ |
| Variant 2 | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | - | ✗ |
| Variant 3 | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ⊥ | - | ✓ |

For approaches that are designed to fall back to a backup technique on failure, the implementation should include that backup technique.

**Embracing Compiler Optimizations** Our soundness evaluation shows that the soundness of every approach is negatively impacted by compiler optimizations. We notice that the original evaluations of the majority of approaches use only programs compiled at the `-O0` optimization level [32, 37, 51]. However, this assumption does not hold true in real-world settings, as developers routinely enable aggressive optimizations (e.g., `-O2` or `-O3`) to maximize performance. More importantly, many popular open-source programs are only thoroughly tested on advanced optimization levels. For example, GCC is only thoroughly tested when built at `-O2` [19], and PostgreSQL recommends building at `-O1` or higher optimization level [47], because `-O0` disables certain compiler warnings when using the GCC compiler. Therefore, future studies in call graph analysis should take compiler optimizations into consideration and include optimized programs in evaluation.

**Directions Beyond Racing for Precision** All approaches we study primarily focus on improving precision over single-layer type analysis. While precision improvement could benefit security applications such as CFI and debloating, there are additional aspects in call graph analysis that could benefit security applications in different ways. For example, as discussed in Section 5.3, one valid direction for future research is to address the limitations of the current state-of-the-practice LLVM-CFI that cause compatibility issues.

Also, as previous work in CFI [6] pointed out, indirect calls with a massive number of allowed targets are particularly dangerous, and even if a CFI approach could restrict most illegal indirect calls with high precision, as long as there exists one reachable indirect call with a massive number of allowed target, an attack could likely find a target to successfully launch an attack. Our study shows that the state-of-the-art approaches do not solve this challenge because of the fallback behavior: unless the multi-layer approaches could reach 100% coverage, the worst-case scenario would remain the same as the baseline single-layer type analysis.

## 8 Threats to Validity

Our evaluation required modifications to released artifacts of the studied approaches to extract the statistics and details necessary for measuring precision and soundness. Importantly, the modifications did not alter the core algorithms, and thus should not affect their results. Nevertheless, to promote transparency and reproducibility, we acknowledge this potential threat and make our modified artifacts publicly available.

Our soundness evaluation relies on a ground truth collected through executing the test programs using a fuzzer. Since fuzzing cannot guarantee exhaustive coverage of all indirect calls, the completeness of our soundness results is constrained by the code coverage achieved. Lastly, the generalizability of our results may be limited by the datasets, compiler versions, and programming languages we studied. While our experiments were designed to capture representative cases, different software ecosystems or compilers may exhibit behaviors that are not fully reflected here. Nonetheless, we believe the consistency of our findings across multiple datasets and settings increases confidence in their broader applicability.

## 9 Conclusion

We performed a quantitative and qualitative evaluation of leading compile-time multi-layer type-based indirect call analysis techniques, focusing on both precision and soundness. Our results reveal important tradeoffs. Existing implementations often omit the fallback mechanisms described in their designs, producing incomplete call graphs with unresolved indirect calls. Even when augmented with single-layer fallback analysis, multi-layer techniques remain less sound than the state-of-the-art single-layer approach, particularly on optimized binaries. Our offensive case study further shows that multi-layer type-based CFI can still permit type-collision attacks under certain code patterns.

At the same time, several approaches achieve precision gains while maintaining soundness comparable to the state of the art on unoptimized programs. We also show that multi-layer analyses can address limitations of existing single-layer techniques. Overall, we hope this study clarifies the strengths and weaknesses of multi-layer type analysis and informs future security research built on these foundations.

## Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF) through award CNS-2238467, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001123C0035, and by the Office of Naval Research (ONR) through award N00014-24-1-2054. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, DARPA, or ONR.

## References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[2] American Fuzzy Lop plus plus. AFL++. https://github.com/AFLplusplus/AFLplusplus, 2025. Accessed: 2025-08-04.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, 2017.

[6] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[7] Yuandao Cai and Charles Zhang. A Cocktail Approach to Practical Call Graph Construction. *ACM Programming Languages*, 7(OOPSLA2):1001–1033, 2023.

[8] Yuandao Cai, Yibo Jin, and Charles Zhang. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *USENIX Security Symposium (USENIX SEC)*, pages 1383–1400, 2024.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium (USENIX SEC)*, pages 161–176, 2015.

[10] Venkatesan T Chakaravarthy. New Results on the Computability and Complexity of Points–to–Analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 115–125, 2003.

[11] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 2531–2548, 2022.

[12] Baijun Cheng, Cen Zhang, Kailong Wang, Ling Shi, Yang Liu, Haoyu Wang, Yao Guo, Ding Li, and Xiangqun Chen. Semantic-Enhanced Indirect Call Analysis with Large Language Models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 430–442, 2024.

[13] Neophytos Christou, Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3913–3927, 2024.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium (USENIX SEC)*, pages 401–416, 2014.

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[16] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994.

[17] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 901–913, 2015.

[18] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the Effectiveness of Type-based Control Flow Integrity. In *Annual Computer Security Applications Conference (ACSAC)*, pages 28–39, 2018.

[19] Free Software Foundation, Inc. Installing GCC. https://gcc.gnu.org/install/index.html, 2025. Accessed: 2025-08-22.

[20] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1964–1978, 2023.

[21] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 527–546, 2023.

[22] Alexander J Gaidis, Jamie Gabbay, Joao Moreira, and Vasileios P. Kemerlis. NOPoutNG: Improving the Effectiveness of Hardware-assisted Control-flow Integrity via Dynamic Landing Pad Elision. In *Annual Computer Security Applications Conference (ACSAC) Workshops/Cyber Security Experimentation and Test Workshop (CSET)*, pages 531–545, 2025.

[23] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, 2016.

[24] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium (USENIX SEC)*, pages 1749–1766, 2020.

[25] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2022. Accessed: 2025-07-30.

[26] Google. FuzzBench: 2025-03-12 report. https://www.fuzzbench.com/reports/2025-03-12/index.html, 2025. Accessed: 2025-08-04.

[27] Rajeev Gopalakrishna, Eugene H Spafford, and Jan Vitek. Efficient Intrusion Detection using Automaton Inlining. In *IEEE Symposium on Security and Privacy (S&P)*, pages 18–31, 2005.

[28] Gerard Holzmann. Static Source Code Checking for User-defined Properties. In *Integrated Design and Process Technology (IDPT)*, 2002.

[29] Sabine Houy and Alexandre Bartel. Twenty Years Later: Evaluating the Adoption of Control Flow Integrity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(4):1–30, 2025.

[30] Florian Kasten, Philipp Zieris, and Julian Horsch. Integrating Static Analyses for High-Precision Control-Flow Integrity. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 419–434, 2024.

[31] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining Indirect Call Targets at the Binary Level. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[32] Guoren Li, Manu Sridharan, and Zhiyun Qian. Redefining Indirect Call Analysis with KallGraph. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2734–2752, 2025.

[33] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[34] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *USENIX Security Symposium (USENIX SEC)*, pages 2777–2794, 2021.

[35] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. Improving Indirect-Call Analysis in LLVM with Type and Data-Flow Co-Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 5895–5912, 2024.

[36] Kangjie Lu. A Critical Review of "Redefining Indirect Call Analysis with KallGraph". https://github.com/umnsec/mlta/blob/main/docs/review-kallgraph.md, 2025. Accessed: 2025-08-04.

[37] Kangjie Lu and Hong Hu. Where Does it Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 2019.

[38] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1393–1403, 2021.

[39] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[40] Harshvardhan Patel, Alexander Snit, and Michalis Polychronakis. Supply Chain Reaction: Enhancing the Precision of Vulnerability Triage using Code Reachability Information. In *Annual Computer Security Applications Conference (ACSAC)*, pages 92–107, 2025.

[41] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[42] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 329–339, 2018.

[43] Yangyang Shi, Liwei Chen, and Gang Shi. Techniques for Refining Indirect Call Targets in Binaries. In *International Conference on Computer and Communication System (ICCCS)*, pages 1–7, 2025.

[44] Rui Sun, Yinggang Guo, Zicheng Wang, and Qingkai Zeng. AttnCall: Refining Indirect Call Targets in Binaries with Attention. In *European Symposium on Research in Computer Security (ESORICS)*, pages 391–409, 2023.

[45] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1630–1644, 2023.

[46] The Clang Team. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html, 2025. Accessed: 2025-08-04.

[47] The PostgreSQL Global Development Group. PostgreSQL 17.6 Documentation. https://www.postgresql.org/docs/17/index.html, 2025. Accessed: 2025-08-22.

[48] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium (USENIX SEC)*, pages 941–955, 2014.

[49] Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky, and Shamil Kurmangaleev. Sydr: Cutting Edge Dynamic Symbolic Execution. In *Ivannikov ISP RAS Open Conference (ISP RAS)*, pages 46–54, 2020.

[50] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, 2020.

[51] Tianrou Xia, Hong Hu, and Dinghao Wu. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 5877–5894, 2024.

[52] Adhemerval Zanella. Building GLIBC with LLVM: The How and Why. https://www.linaro.org/blog/building-glibc-with-llvm-the-how-and-why/, 2023. Accessed: 2026-02-23.

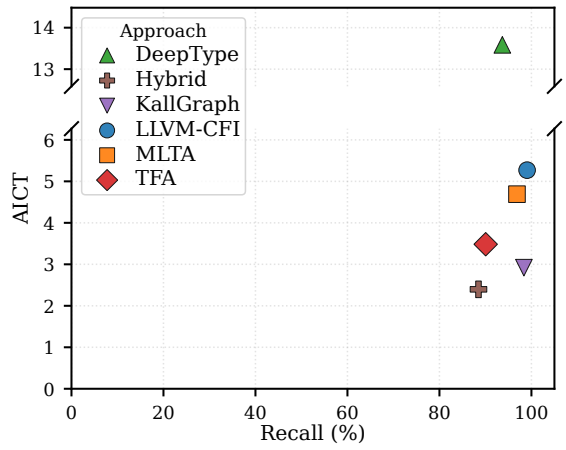
[53] Dongrui Zeng, Ben Niu, and Gang Tan. MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 810–821, 2021.

[54] Zhechang Zhang, Hengkai Ye, Song Liu, and Hong Hu. SACK: Systematic Generation of Function Substitution Attacks against Control-flow Integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2026.

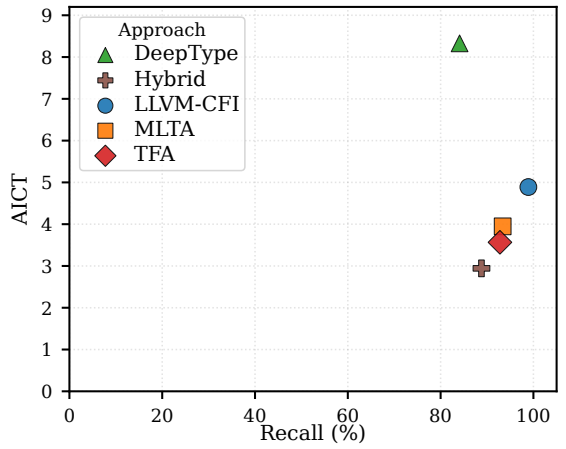
[55] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2357–2374, 2023.

## Open Science

Our artifacts, including the source code used in our evaluation and the list of programs and versions included in our datasets, is publicly available at https://github.com/yufeidu/SoK-MLTA. For repositories without an open-source license or with a non-redistributable license, we provide a list of them along with our corresponding patches.




(a) `-O0`




(b) `-O3`

Figure 6: Soundness results across the 10 programs that all approaches other than HPCFI can successfully analyze.

```
1  static struct unix_syscall {
2    const char *zName;
3    sqlite3_syscall_ptr pCurrent;
4    sqlite3_syscall_ptr pDefault;
5  } aSyscall[] = {
6    ...
7    { "unlink", (sqlite3_syscall_ptr)unlink, 0 },
8    #define osUnlink    ((int(*)(const char*))
         aSyscall[16].pCurrent)
9    ...
10 }
11
12 static int unixOpen( ... ){
13   ...
14   osUnlink(zName);
15   ...
16 }
```

Listing 5: Code snippet from `sqlite` that causes false negatives due to Issue #1.

## Ethical Considerations

From a stakeholder-focused ethics perspective, we do not anticipate any ethical concerns in this work. Our study centers on studying previously published program analysis techniques from flagship security venues and does not involve sensitive topics like vulnerability discovery on live systems or inadvertent data disclosure. In discussing our findings, we deliberately avoid criticism of the authors, instead presenting the results objectively and identifying confounding factors that limit the practical application of the studied techniques.

More broadly, our findings on the limitations of existing call graph construction approaches are intended to strengthen the robustness of tools widely used in security-critical contexts, such as integrating the approaches to improve CFI. While such improvements may ultimately aid both defenders (e.g., by building more sophisticated CFI schemes) and adversaries (e.g., by identifying type collision targets), we believe the net benefit lies in advancing program analysis.

## A Additional Soundness Evaluation Results

Figure 6 presents the average soundness and the average precision across the 10 programs that could be successfully analyzed by all approaches other than HPCFI. At `-O0`, all four approaches achieve an average recall of 90% or higher. In addition, all approaches other than DeepType outperform the baseline LLVM-CFI in average precision. KallGraph has the best average recall and the best average precision among all approaches considered. Compared to the baseline, whose average recall and precision are 99.1% and 5.27, KallGraph improves the average precision by 44.6% at a cost of 0.6% lower recall. At `-O3`, all approaches show lower recall. The average recall for MLTA, DeepType and TFA are 93.4%, 84.1% and 92.8%, respectively. In comparison, the baseline reaches an average recall of 98.9%.

Overall, our results on the larger set of 10 programs show findings consistent with those reported in Section 5.2.

## B Limitations of LLVM-CFI

LLVM-CFI cannot identify address-taken functions in pre-compiled libraries that are not processed by LLVM itself, such as `glibc`. Because pre-compiled libraries do not carry the type metadata at the linker stage, LLVM-CFI cannot determine the type of library functions—even if these are declared in included header files. While one may consider this situation to be a non-issue because it can be solved by compiling everything with LLVM, several popular libraries, such as `glibc`, are still incompatible with the LLVM toolchain [52], creating compatibility roadblocks for LLVM-CFI.

Listing 5 shows an indirect call in `sqlite` that LLVM-CFI fails to identify its correct target. The program defines an array of system calls, `aSyscall`, at line 1–10. Each entry includes a

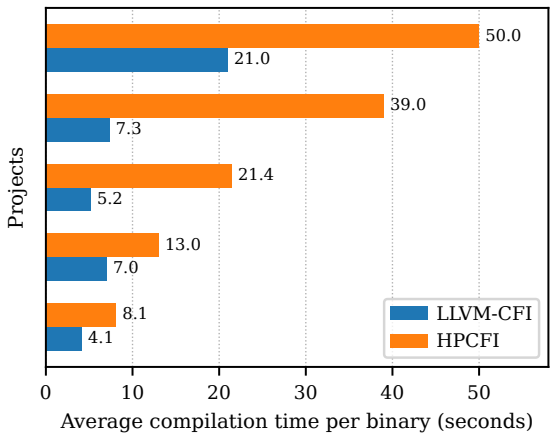



Figure 7: Compilation time of HPCFI and LLVM-CFI.

function pointer, `pCurrent`, which points to an address-taken function in the `glibc` library. The library function `unlink` is assigned to the function pointer `aSyscall[16].pCurrent` at line 7, and a macro, `osUnlink`, is defined at line 8 to dereference it. Later, at line 14, `unlink` is invoked via `osUnlink`. TFA correctly identifies the target at `-O3`; conversely, TFA misses the target function at `-O0`.

## C Performance Overhead of HPCFI

To study the performance overhead of state-of-the-art indirect call graph analysis compared to the current state-of-the-practice, we measure the compilation time of HPCFI and compare it with the compilation time of LLVM-CFI. For this experiment, we use the Precision-Replication dataset, which includes 5 projects from OSS-Fuzz [25]. We use the same version of LLVM v13 for both approaches and use the same compiler flags other than the specific flags to enable CFI. We conduct our experiment using a Lenovo Legion Slim 7 Gen 8 laptop (3.8GHz 8-core AMD Ryzen CPU, 64GB RAM) running Ubuntu 20.04 LTS. We measure the compilation time of each project in the dataset and use the metric of average compilation time per binary: if the build system of a project generates multiple binaries, we divide the total compilation time by the number of binaries.

Figure 7 presents the compilation time of each of the five projects in our dataset. On average, HPCFI takes 213% more time in compilation compared to LLVM-CFI. In almost every project, HPCFI takes more than twice as long to compile a binary. This shows that the additional analyses in HPCFI, including the pointer analysis and the multi-layer type analysis, incur significant performance overheads.