Effectiveness of Binary-Level CFI Techniques

Ruturaj K. Vaidya and Prasad A. Kulkarni

Abstract. Memory corruption is an important class of vulnerability that can be leveraged to craft control flow hijacking attacks. *Control Flow Integrity (CFI)* provides protection against such attacks. Application of type-based CFI policies requires information regarding the number and type of function arguments. Binary-level type recovery is inherently speculative, which motivates the need for an evaluation framework to assess the effectiveness of binary-level CFI techniques. In this work, we develop a novel and extensible framework to assess how the program analysis information we get from advanced binary analysis tools affects the efficacy of type-based CFI techniques. We introduce new and insightful metrics to quantitatively compare source independent CFI policies with their *ground truth* source aware counterparts. We leverage our framework to evaluate binary-level CFI policies implemented using program analysis information extracted from the IDA Pro binary analyzer and compared with the ground truth information obtained from the LLVM compiler.

1 Introduction

Software written in memory unsafe languages like `C` and `C++` is vulnerable to *Code-Reuse Attacks (CRA)* such as return-into-libc (full function reuse attack) [4], ROP (Return Oriented Programming) [18, 3] and COOP (Counterfeit Object-oriented Programming) [17, 9, 10]. *Control flow integrity (CFI)* [1] is a popular technique to prevent such control flow hijacking attacks. CFI aims to ensure that the control flow of the program stays within the legitimate targets desired by the programmer. Usually, this is achieved by computing the user intended control flow targets using a static analysis phase to insert security checks into the generated binary code. The inserted security checks monitor and enforce the control flow of the program to stay within the desired target locations at run-time.

Various CFI techniques have been proposed after the introduction of an exemplary CFI model by Abadi et al. [1]. CFI techniques could be source-code aware — implemented at source or compiler-level, or source-code independent — implemented at the binary level. Binary-level CFI techniques are necessary to secure *unprotected* and *untrusted* programs and third-party libraries that are typically shipped without their corresponding high-level source codes.

CFI techniques typically require the accurate recovery of function call-site and function signature information, including argument counts and all argument types. Lack of accurate program analysis information at the binary-level makes it extremely challenging to build a precise function call-graph for large binary software. In turn, the effectiveness of binary-level CFI techniques depend and suffer from the inaccuracies of the program information extracted by the adopted binary analysis framework. Over- or under-approximation of reachable call-targets by CFI techniques can result in false negatives (attacks go undetected) or false positives (correct control flow tagged), which can dent the usability of CFI.

Our goal is to study and quantify the correctness of binary-level CFI techniques and how they are impacted by the inaccuracies in program analysis information recovered by binary analyzers. We focus on *type-based* CFI techniques that use the number and type of arguments to match each call-site to the set of *potential* call-targets. Source-level CFI techniques have access to precise program and type information, and are therefore most likely to achieve their *design* objective. We use the output of each source-level CFI technique as the *ground truth* to assess the accuracy of the corresponding binary-level CFI technique.

In this work, we develop a novel framework, called *β-CFI*, to study and quantify the effectiveness of different binary-level CFI techniques. Our framework supports the integration of different source (compilers) based and binary-level analysis modules to gather program information required to model different CFI techniques, each at the source and binary levels. To validate our framework, we develop a source-level analysis module using the LLVM compiler [13] and a binary-level analysis module using the IDA Pro and Hex-Rays software reverse engineering (SRE) tools [8]. The analysis modules statically recover program information, including call-site and call-target argument counts, argument types, and the function return type. We also model four different CFI techniques that employ the analysis information gathered by the source/binary-level analysis modules to impose the call-target constraints.

Next, we introduce new and insightful metrics to quantitatively compare the effectiveness of CFI policies instituted at the binary level with the *ground truth* provided by their source-aware counterparts. Unlike most existing CFI metrics that only measure the *number* of call-targets reached without regards to their *correctness* compared to the ground truth set of call-targets [25, 20, 7, 2, 15, 6], our approach provides a more correct metric for evaluating the accuracy of binary-level CFI techniques.

We make the following contributions in this paper:

– We develop a modular and extensible framework[1] along with a common language to compare the accuracy and effectiveness of binary-level type-based CFI techniques.
– We develop a mechanism to model multiple different type-based CFI techniques using program information obtained from different sources.
– We develop metrics to quantitatively measure the accuracy of binary-level CFI techniques compared to *ground truth* results obtained with access to the source code.
– We employ our framework, models, metrics and mechanisms to recover program information from IDA Pro binary analyzer, and LLVM compiler, and employ that information to quantitatively assess the accuracy of four binary-level CFI techniques compared to their source-level equivalents.

[1] Our framework is available online - https://github.com/Ruturaj4/B-CFI.

2 Background and Related Work

2.1 Control Flow Integrity (CFI)

Code-Reuse Attacks (CRA) [4, 18, 3, 17, 9, 10] allow attackers to exploit spacial and temporal memory safety violations to alter the control flow of the program. CFI provides protection against such arbitrary control flow subversion. CFI techniques use static or dynamic analysis to compute the program control flow graph (CFG) and then check at run-time if the program execution follows the CFG computed in the previous analysis stage. Thus, CFI maintains program integrity by only allowing legitimate control transfers during execution.

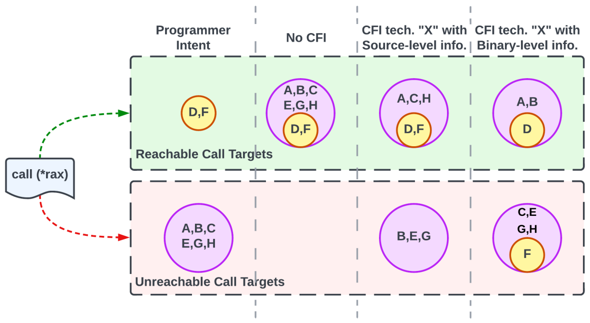

Fig. 1: High-level overview of CFI techniques

We use Figure 1 to describe, at a high-level, how CFI techniques work and also to explain our goals in this work. When coding, the developers may intend the control-flow at each *indirect call-site* to only reach a few potential function *targets* during program execution. For instance, the programmer *intent* in our example (as illustrated in Figure 1-(A)) is for the indirect call-site to only reach targets '`D`' and '`F`'. Unfortunately, this programmer intent is not explicitly encoded in the source-code, and is lost before it reaches the compiler. Without any CFI check, an attacker may be able to subvert the call-site to reach *any* reachable function target ('`A`', '`B`', '`C`', '`D`', '`E`', '`F`', '`G`', and '`H`' in our example, Figure 1-(B)).

Different CFI techniques use various *safe* approaches that constrict the set of spurious reachable targets, while ensuring that the technique does not inadvertently disallow any correct (but, unknown) programmer-intended targets. If a correct target is not in the set of reachable targets, then the CFI check may trigger a *false positive* alarm for correct program flow during execution. At the same time, if the set of reachable targets is overly broad, then the CFI technique may leave the program more vulnerable to attacks. In our example, the source-level CFI technique partitions the targets into reachable and unreachable sets, as illustrated in Figure 1-(C). For the CFI policies that employ types to determine the set of "valid" targets, these results obtained from employing a program representation with perfect type information (like the source-code) presents the best case result they can achieve.

Unfortunately, program analysis information recovered by binary-level SRE tools may be imprecise, which can cause the *same* CFI algorithm to produce different and incorrect reachable and unreachable target function sets at the binary-level for each call-site (as illustrated in Figure 1-(D)). Our goal in this work is simply to measure and study this imprecision in the output of binary-level CFI techniques as compared to their source-level counterparts.[2]

Abadi et al. introduced the idea of CFI by statically computing the CFG and restricting control flow of the program to the valid targets during run-time [1]. Since then, researchers have developed many CFI policies and algorithms that differ in their implementation, precision and cost. Several CFI approaches employ pointer analysis to construct the CFG that is needed by the algorithm [25, 24, 20, 22]. However, static points-to-analysis is imprecise, especially for program binaries [5]. Therefore, researchers have proposed CFI techniques that incorporate program invariants such as argument count and types to construct the CFG. These type of techniques are referred to as Run-time Type Checking (RTC) based CFI techniques [20, 16, 19, 21, 14, 5, 12].

In this work, we do not propose or build new CFI techniques. Instead, we develop a new framework and metrics to model and compare binary-level RTC based CFI mechanisms against a known ground-truth. We also assess the accuracy of the relevant program information recovered by state-of-the-art binary analysis tools, and their impact on the precision of binary-level CFI policies.

2.2 CFI Security Policy Comparison Metrics

Researchers have developed several mechanisms and metrics to evaluate and compare the protection provided by different CFI policies. *Average Indirect target Reduction (AIR)* [25] measures the reduction of permitted call-targets. *AIA* [7] computes the average number of call-targets per function call. Similarly, *fAIR* [20] and *fAIA* [6] are forward-edge variations of the previous metrics. The *CTR (Call-Target Reduction)* metric provides absolute values (rather than averaged results) of reachable call-targets at every indirect call-site [15]. Most of these metrics use a relative measure, such as reduction in the average number of reachable *targets* from each call-site, or reduction in the number potential gadgets, etc. to assess the accuracy and benefit of the CFI technique.

Burow et al. propose a metric called *QuantitativeSecurity* that computes the number of equivalence classes and the inverse of the size of the largest class, to quantify the security of CFI techniques [2]. Frassetto et al. develop the *BLOCK-Insulation* and *CFGInsulation* metrics to calculate the distance between a vulnerable instruction to system call at basic-block granularity [6].

[2] It is important to realize that even if the binary-level CFI technique produces a more desirable outcome (for example, by allowing all programmer-intended targets and a smaller spurious set in the reachable set), it is still considered erroneous in this work, if it does not match the output of the corresponding source-level approach, since the technique did not function as algorithmically designed (due to imprecise analysis data), and any observed "*improvement*" is merely coincidental.

None of these existing CFI metrics incorporate the notion of obtaining the actual accuracy of any CFI technique as compared to some known ground truth, and determining the false positive and false negative call-targets at each call-site. In this work, we show why such earlier CFI metrics are ill-suited for comparing the performance of binary-level CFI policies. We introduce new metrics that can quantitatively compare the accurate call-targets in each equivalence class identified by binary-level policy with that of call-targets recuperated in the corresponding equivalence class using a source aware ground truth policy.

2.3 CFI Frameworks

It is difficult to compare and assess the performance of different CFI policies as they use different settings, including compilers, operating systems and machines. Therefore, researchers have built detailed frameworks, mechanisms, and metrics to compare and assess CFI techniques uniformly.

Farkhani et al. develop a framework to analyze the ability of RTC CFI mechanisms, and compare them with a points-to analysis based CFI mechanism [5]. Li et al. introduce CScan — a framework to compute actual feasible targets using run-time checks and CBench — an extensive set of vulnerable programs to assess the effectiveness of CFI techniques [11]. *ConFIRM* [23] analytically compares various CFI policies in terms of compatibility issues in contrast to focusing on performance or security.

Our framework to evaluate binary-level CFI policies is inspired by a compiler-level CFI policy comparison framework, called LLVM-CFI [15]. This framework provides a LLVM-Clang based unified framework for statically modelling and systematically assessing various CFI techniques. LLVM-CFI leverages a link time optimization (*LTO*) pass in the LLVM compiler to impose constraints on invariants collected during compilation to implement CFI policies. The CFI policies in our current work also adhere to much of the formalization described by this earlier work. However, our goals, implementation machinery and metrics used differ considerably from LLVM-CFI.

The CFI policies in LLVM-CFI are modelled based on their idealized representation, which means that they do not consider the effect of loss in high-level information whilst modelling source insentient CFI techniques. In other words, the binary-level CFI policies in LLVM-CFI are established on the premise that the analysis primitives are all recovered correctly at the binary level. Instead, our goal in this work, which is to compare the precision of binary-level CFI policies, requires us to gather the necessary program information from both binary-level and source-level analysis tools.

None of these earlier CFI policy comparison frameworks and metrics attempt to study and assess how the loss of program information at the binary-level affects the efficacy of different binary-level CFI policies compared to some ground truth, which we do in this work. Furthermore, we also develop a new set of metrics that can more accurately determine the accuracy of binary-level CFI policies compared to a ground truth, which was not attempted by earlier CFI policy comparison frameworks.

3 Implementation

In this section we describe the design and implementation of the CFI policy comparison framework and the CFI models that we build and use for this work.

3.1 Design Overview

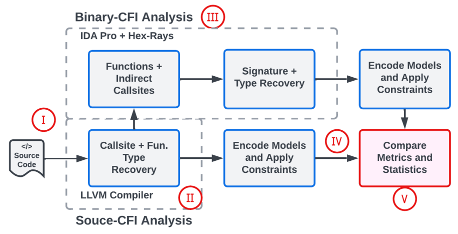
Binary-CFI Analysis
III
IDA Pro + Hex-Rays
Functions + Indirect Callsites
Signature + Type Recovery
Encode Models and Apply Constraints
I
</> Source Code
Callsite + Fun. Type Recovery
Encode Models and Apply Constraints
IV
Compare Metrics and Statistics
LLVM Compiler
II
V
Souce-CFI Analysis

Fig. 2: Block Diagram of β-CFI

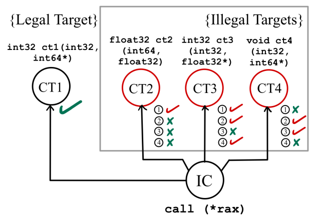

Fig. 3: Indirect call-site targeting functions in binary hardened with four different policies — ① TypeArmor, ② IFCC, ③ MCFI and ④ τCFI

In this paper, we introduce *β-CFI* — a binary level CFI comparison framework. To assess and analyze the precision of type-based binary-level CFI techniques, we design and construct an evaluator framework that is capable of comparing the results achieved by different CFI techniques at both the source-sentient and insentient levels. Figure 2 shows the high-level block diagram of our evaluation technique. The technique can be broadly classified into two stages. Firstly, relevant program analysis information is collected from both source-level (LLVM) and binary-level (IDA Pro) means and secondly, this information is fed into the CFI models, and the results are computed, compared and analyzed.

In further detail, our technique performs the following steps: (I) First, the source code to be analyzed is compiled (using the LLVM compiler, in this work). (II) During compilation, we collect various program analysis information, including function argument counts, and argument and function types at each call-site and at every call-target using a dynamically loadable *LLVM LTO* (Link Time Optimization) pass that we built for this work. This pass makes separate compilation of source files possible providing flexibility. These source-level analysis statistics are used to drive an idealized representation (or ground truth) of our type-based CFI policies. (III) The output binary is then employed for β-CFI statistics collection. In this work we leverage IDA Pro [8] — a popular reverse engineering framework to statically analyze the binaries and recuperate static analysis information, including indirect calls and program functions accompanied by their type signatures i.e. function return type, function argument counts and their types at each call-target and call-site. We also leverage Hex-Rays decompiler to refine the type information generated by IDA Pro. The advanced type inference in the decompiler assists us to model robust run-time type checking (RTC) policies at the binary level.

We invoke our LLVM LTO pass after full link time optimizations to ensure the accurate source to binary function matching. We do not consider any unmatched functions if they aren't identified correctly by IDA tool. Although we employ LLVM and IDA Pro for this work, our framework is modular and evaluators can use any other source- and binary-level static analysis tools to extract function and call-site related program analysis information.

Ⓘⓥ After the recovery of these analysis primitives at both the source level and binary-level, type-based policy constraints are applied corresponding to each deployed CFI policy. At this stage, evaluators can select and encode any CFI policy of their choice by setting various type-based constraints. Thus, this extensible and convenient framework will enable analysts to implement and verify new type-based CFI policies at the binary level without doing repetitive compilation and analysis. To validate our framework, we implement and deploy four type-based policies (explained in details in 3.2) for evaluation. Ⓥ Finally, the output of the CFI models using source-level and binary-level program information is compared and the final results are displayed to the evaluator.

3.2 Type-Based CFI Policies

In this section we describe the four type-based CFI policies we model by applying different type-based constraints. Some of these were also used and compared in the LLVM-CFI work [15]. Figure 3 displays an indirect call-site targeting four different functions in a binary hardened by modeling four different type-based CFI policies. The function shown on the far-left (`CT1`) is the only legal call-target intended to be called from indirect call (IC) instruction `call (*rax)`. Besides, three other functions (`CT2-CT4`) are illegal call-targets and should ideally be unreachable during correct program execution. We assume that the attacker controls the value of register `rax`.

We now discuss constraints and type collisions imposed by the four CFI policies we employ. However, our technique is adaptable and evaluators can introduce and model other policies with various levels of type-based precision.

① **TypeArmor** [21] was originally implemented at the binary level by using coarse-grained type invariants. The policy considers the number of arguments without explicit types. At each call-site the call is allowed only if the number of arguments at the call-target are equal or less than that at the call-site (maximum up to six). Additionally, void and non-void functions are differentiated i.e. call-sites which expect a return value must only target functions with non-void return type. Note that such assumptions can not be made on the contrary, i.e. if a call-site doesn't expect a return value, then it can call void as well as non-void functions. This relaxed policy is practical at the binary level, as it is often difficult to infer whether the function is going to return a value or not. Thus, at the example call-site in Figure 3, the TypeArmor CFI policy allows the `call (*rax)` instruction to reach `CT1`, `CT2` and `CT3` functions, which includes two illegal targets.

② **IFCC** [20] is implemented similar to the encoding explained in [15]. IFCC takes into account the argument and parameter counts, along with their basic types to match call-sites to call-targets. However, base pointers types are not considered, i.e. `void*` and `int*` are considered equivalent. Therefore, functions `CT3` and `CT4` in Figure 3 are allowed (in addition to `CT1`). Return type is not taken into consideration. Note that the types are not over-approximated i.e. they are not considered as upper bound, but are matched according to the exact type.

③ **MCFI** [16] is a CFI policy that is stricter than IFCC in terms of how pointer types are recuperated. Pointer types such as `void*` and `int*` are considered distinct. Similar to IFCC, the number of parameters and their types are matched with call-site argument count and types. However, stricter types are taken into consideration. Thus, as seen in Figure 3 only one target i.e. `CT4` is reachable with the stricter MCFI policy (in addition to `CT1`). Function return types are not considered, similar to IFCC.

④ **τCFI** [14] considers argument and parameter types along with their counts. The types are contemplated based on the size of the registers {`0,8,16,32,64`} prepared during the indirect call. According to x86-64 calling convention (System V ABI) the first 6 arguments are passed through registers during a function call. τCFI policy allows the call if 1) the number of arguments prepared at the call-site are more or equal to the number of parameters consumed at the call-target, 2) the return type recuperated at the call-site and the call-target is non-void and its size at the call-site is larger than that of the call-target return type; else, if return type recuperated at the call-site is void and then it can also call non-void functions, 3) the size of the argument types at call-site are greater than or equal to their matching arguments at call-targets. Thus, in our example displayed in Figure 3, `CT3` is the only illegal target that is allowed to be reached from the indirect call-site.

4 Evaluation

4.1 Benchmarks

We evaluate our framework using *sixteen* `C` and `C++` benchmarks from the SPEC 2006 [3] integer and floating point suite. We leave out the remaining benchmarks either because we didn't find any indirect call-sites in the optimized benchmark version (mcf, libquantum and lbm) or when the benchmarks use `Fortran` code.

Additionally, we include five popular and large real world applications for this study. Specifically, we performed our evaluation with (a) *Nginx* (v1.22.1 `C`), an open-source web server software [4], (b) *Node JS* [5] (v10.24.0 `C/C++`), an open-source, cross-platform JavaScript run-time environment, (c) *Apache Traffic*

[3] https://www.spec.org/cpu2006/
[4] https://nginx.org/en/download.html
[5] https://nodejs.org/en/download/current

server [6] (v6.2.3 `C/C++`), an open-source forward and reverse proxy web server, (d) *postgresql* [7] (v12.0 `C`), an open-source relational database management framework, and the (e) Tor Browser [8] (v0.4.8.0-alpha-dev `C`), an open-source web browser focused on privacy and security. We obtained the most primary application binary from these benchmarks for our analysis.

Our benchmarks along with the total number of indirect call-sites and call-targets in each program are listed in Table 1. All the SPECint and SPECfloat benchmarks are presented together in their respective groups in this table (and in all later results).

4.2 Experimental Configuration

We design two benchmark configurations for this study.

I. **Ideal or Baseline Scenario:** For our first configuration, we keep the debugging symbols and compile the binary with optimizations (‘`-O3`’). We refer to this configuration as the *baseline*. This *baseline* configuration can be considered as an idealized representation at the binary level where some source semantics in the form of debug symbols are available to guide the binary analysis frameworks.

II. **Practical Scenario.** For our second configuration, we strip the debugging symbols using ‘`strip --strip-debug`’. This is a practical scenario for most COTS (Commercial off-the-shelf) binaries and presents a more challenging case for the binary analysis algorithms. All benchmarks are still optimized by ‘`-O3`’.

All experiments are performed on Fedora 34 operating system with x86-64 Intel Xeon processor. The LLVM/Clang version used is (v.12.0.0) to compile binaries and get the ground truth program information, and 64-bit version of *IDA Pro* (v7.5.2) is used to conduct binary analysis and extract the program information used by the binary-level CFI models.

Table 1: Inverse of Benchmark Properties

Benchmark	SPECint	SPECfp	nginx	postgresql	trafficserver	tor	node
call-targets	15594	2341	1237	11089	6886	5761	133496
call-sites	20304	1179	448	9367	8311	273	8239

4.3 Evaluation Metric

To compare and evaluate the *precision* of binary-level CFI policies with their source aware counterparts in terms of the *correct* reachable call-targets at each

[6] https://archive.apache.org/dist/trafficserver/
[7] https://www.postgresql.org/download/
[8] https://www.torproject.org/download/

call-site, we introduce new metrics that calculate not only the number of targets reached (fewer the better), but also employ the known ground-truth targets information to check if there are any false positives or false negatives generated by the CFI policy under evaluation. Such a detailed evaluation of CFI policies is crucial, as mere call-target reduction results, as measured by most earlier CFI metrics, can not characterize the number of:

- true positives – illegal (unreachable) targets that are correctly marked by the CFI policy under evaluation,
- false positives – legal (reachable) targets in the ground truth, but are marked as illegal by the CFI policy under evaluation,
- true negatives – legal targets in the ground truth that are correctly marked by the CFI policy under evaluation, and
- false negatives – targets illegal in the ground truth that are incorrectly marked as legal by the CFI policy.

Thus, it is very important to know the exact targets reached, i.e. we not only need to check how many functions are reached using *Binary-CFI*, but also how many of these functions match the functions detected using our ground truth. We introduce new metrics named $RelativeCTR$ ($RelativeCTR_T$ and $RelativeCTR_F$) to check whether the actual targets reached when Binary-CFI policies are applied are in fact equivalent to the actual targets reached when Source-level CFI policies are applied. $RelativeCTR_T$ (higher the better) represents the number of call-targets that are accurately reached at a particular call-site using binary-CFI policy, compared to source-level CFI policy, and $RelativeCTR_F$ (lower the better) presents the call-targets that are incorrectly reached at a particular call-site using binary-level policy, compared to its source aware CFI policy counterpart.

Suppose that P is a program with total indirect call-sites IC and total reachable call-targets CT. Let IC_i be an indirect call-site in program P with number of reachable call-targets CT_i after applying the CFI constraints for source aware policy P_c and $CT_i^{'}$ be number of reachable call-targets after applying source independent policy $P_c^{'}$ at the same call-site. Then, $RelativeCTR_T$ and $RelativeCTR_F$ are defined as follows.

Definition 1. *$RelativeCTR_T$ is the ratio of the intersection of targets in Source-CFI (CT_i) and in Binary-CFI ($CT_i^{'}$) to the total number of actual targets in Source-CFI (CT_i) at an indirect call-site ICi.*

$$RelativeCTR_T \quad (R_T) = \sum_{i=1}^{n} (CT_i \cap CT_i^{'})/CT_i$$

Definition 2. *$RelativeCTR_F$ is the ratio of the total number of call-targets in ($CT_i^{'}$) reachable with Binary-CFI but not reachable with Source-CFI (CT_i) to the total number of targets in Binary-CFI ($CT_i^{'}$) at an indirect call-site ICi.*

$$RelativeCTR_F \quad (R_F) = \sum_{i=1}^{n} (CT_i^{'} \setminus CT_i)/CT_i^{'}$$

We illustrate our new metrics using the hypothetical example from Figure 1. This program has eight different functions 'A', 'B', 'C', 'D', 'E', 'F', 'G', and 'H'. For some indirect call-site IC_1 in the program, the set of reachable targets as identified by the source-level CFI policy (our ground truth) are 'A', 'C', 'D', 'F' and 'H' (CT_1, CT_3, CT_4, CT_6 and CT_8). However, the binary-level CFI policy under evaluation determines the reachable set of targets from the same call-site to be 'A', 'B' and 'D' ($CT_1^{'}$, $CT_2^{'}$ and $CT_4^{'}$). Thus, with reference to the ground truth, 'B' is an unintended target, and 'C', 'F' and 'H' are correct targets that are missed. Therefore, the $RelativeCTR_T$ for this call-site is 2/5, which indicates the correctly detected, or true negative targets (and, correspondingly, also the false positive targets). $RelativeCTR_F$ is 1/3, which indicates the incorrectly detected or the false negative call-targets. Thus, a high $RelativeCTR_T$ indicates a high true negative (and low false positive) rate for the CFI technique, i.e., a low likelihood of throwing a fault when there is none. A high $RelativeCTR_F$ indicates a more relaxed CFI policy and a higher likelihood for the CFI technique to allow unsupported control flow paths that can lead to attacks.

In addition, since we target the same weakness in all previous CFI metrics, we use only one, the popular CTR metric [15], as representative of the category of metrics that only use the measure of reduction in the number of call-targets from each call-site to rate different CFI policies. The CTR metric depicts the absolute values of the number of call-targets accessible from a call-site after hardening with a particular CFI policy. The CTR metric is defined as follows.

$$CTR = \sum_{i=1}^{n} ct_i$$

Where ct_i is number of call-targets reachable from an indirect call-site ic_i. A lower value of CTR implies a better CFI policy, as it ostensibly reduces the number of *extraneous* targets allowed from a call-site. In this paper, we highlight some important shortcomings of the CTR (and similar) metrics for our work. Specifically, such metrics do not fairly and accurately assess the precision of CFI policies compared to some known *ground truth*.

4.4 CFI Policy Comparison

We present and discuss our results in this section. We use our framework and models to collect the $RelativeCTR$ and CTR numbers for all our benchmark programs. We use *Dwarf* symbols at every call-site to match the call-sites detected during the source-level LLVM pass with the call-sites in the binary executable. We leverage the *llvm-symbolizer* tool to match *Dwarf* symbols with the address of the respective call-site in the binary. Note that the binary address to *Dwarf* mapping is one-to-many and thus we consider all the call-sites that appear in the binary for each source-level call-site.

We leverage our new $RelativeCTR$ metrics to show correctly and incorrectly reachable call-targets at each call-site. Table 2 shows the $RelativeCTR$ metrics with "Mean" values for our benchmarks in both our binary configurations **I** and

Table 2: Mean *RelativeCTR* comparison results of our 4 CFI policies (*TypeArmor*, *IFCC*, *MCFI* and τ*cfi*)

	TypeArmor				IFCC				MCFI				τCFI			
Benchmark	R_T (I)	R_F (I)	R_T (II)	R_F (II)	R_T (I)	R_F (I)	R_T (II)	R_F (II)	R_T (I)	R_F (I)	R_T (II)	R_F (II)	R_T (I)	R_F (I)	R_T (II)	R_F (II)
SPECint	0.93	0.24	0.92	0.25	0.26	0.45	0.22	0.48	0.14	0.65	0.13	0.66	0.74	0.27	0.75	0.27
SPECfp	0.91	0.13	0.87	0.28	0.49	0.44	0.29	0.53	0.40	0.51	0.19	0.67	0.89	0.16	0.71	0.28
nginx	0.92	0.03	0.91	0.19	0.68	0.30	0.35	0.37	0.47	0.43	0.24	0.72	0.89	0.03	0.67	0.12
postgresql	0.80	0.02	0.75	0.12	0.45	0.53	0.25	0.52	0.28	0.66	0.23	0.76	0.74	0.11	0.42	0.32
trafficserver	0.93	0.22	0.93	0.22	0.31	0.39	0.29	0.40	0.10	0.51	0.11	0.52	0.48	0.22	0.48	0.22
tor	0.96	0.12	0.64	0.29	0.70	0.26	0.18	0.51	0.49	0.32	0.14	0.76	0.75	0.10	0.31	0.22
node	0.99	0.05	0.95	0.24	0.74	0.17	0.31	0.38	0.64	0.22	0.28	0.51	0.92	0.16	0.69	0.38

Table 3: Mean *CTR* comparison results of our 4 CFI policies (*TypeArmor*, *IFCC*, *MCFI* and τ*cfi*)

	TypeArmor			IFCC			MCFI			τCFI		
Benchmark	Source	Bin-I	Bin-II	Source	Bin-I	Bin-II	Source	Bin-I	Bin-II	Source	Bin-I	Bin-II
SPECint	3327.33	3966.70	3967.50	1608.22	606.83	591.66	1296.44	356.30	370.67	2105.19	2092.90	2088.97
SPECfp	308.05	307.51	333.59	101.71	78.05	52.72	86.55	48.11	28.00	167.72	155.00	120.70
nginx	506.06	487.85	570.47	277.97	201.50	153.71	130.45	69.33	142.92	366.34	357.17	366.55
postgresql	6637.11	5337.80	5515.54	1825.17	1233.95	1009.84	997.45	720.76	932.53	2415.92	2372.24	1585.11
trafficserver	3049.97	3882.86	3905.23	1866.46	809.56	754.68	1699.28	229.41	250.19	1622.12	990.37	991.23
tor	2896.82	3123.42	2578.18	923.81	730.51	365.42	470.58	385.08	330.14	1610.27	1231.44	786.90
node	70251.10	73847.82	89280.00	25418.30	23934.50	10240.00	17394.30	16165.50	8000.88	37759.30	37679.10	40666.90

II. The results in Table 2 allow us to make some important observations that would be missed by earlier CFI comparison metrics that use a reduction in the number of reachable targets from each call-site as the only measure to evaluate the effectiveness of CFI techniques [25, 20, 7, 2, 15, 6].

Table 3 presents the results using the CTR metric for 4 CFI policies - ① TypeArmor, ② IFCC, ③ MCFI and ④ τCFI, and for our benchmark set when using the analysis information from LLVM (*Source-CFI*), and our two binary configurations, *Binary-CFI* (**I**) and *Binary-CFI* (**II**), respectively. The CTR metrics in Table 3 present the absolute values of reachable targets.

We employ the *RelativeCTR* and *CTR* results, presented in Tables 2 and 3, respectively to make several observations. Of the four CFI policies modelled in this work, TypeArmor is the most *permissive*, since it only considers argument counts and discards argument type information. By contrast, MCFI is most *strict* as it considers both basic types and mature pointer types. Accordingly, we can see higher CTR numbers across the board for TypeArmor and relatively lower CTR numbers for MCFI, which confirms this property about the CFI policies.

For this work though, it is more pertinent to compare the binary-level CFI CTR numbers with the corresponding Source-CFI numbers to assess the accuracy of CFI methods at the binary-level (with Source-CFI acting as ground truth for each policy). When using the CTR metric, the difference between the binary and source-level numbers indicates the potential error in the binary-level CFI models. We find that, **the binary-level CTR numbers differ significantly from the source-level CTR metrics for all our CFI models. Besides, this difference is greater for the more restrictive CFI policies.** Thus, CFI policies, such as MCFI and IFCC, that rely on more precise program data type information appear to be more erroneous as compared to the simpler CFI models, like TypeArmor and τCFI. This is an intuitive result as it indicates that errors in correctly reconstructing the type information at the binary level negatively impacts the algorithms employing such data during their computations.

While this observation derived with the CTR metric appears to be correct, a deeper analysis reveals critical issues and misleading outcomes. For instance, the results in Table 3 also show that the Binary-CFI CTR ratios are often tighter (which is better, according to the CTR metric) than the Source-CFI numbers. We find that in 3 of the 7 benchmark categories with TypeArmor, and in all of the 7 benchmark categories with IFCC, MCFI, and τCFI, the number of *mean* reachable targets from each call-site is smaller with Binary-CFI (**I**) compared with the Source-CFI numbers. This result with the CTR metric is confusing since it suggests that the binary-level techniques achieve better effectiveness with fewer extraneous call-targets compared to the source-level techniques. Likewise, in many cases, especially for the stricter CFI policies, we can observe that the CTR numbers are tighter with the *stripped* benchmarks in the Binary-CFI (**II**) configuration, compared with the Binary-CFI (**I**) configuration, which is again a confusing and likely misleading outcome.

Results with our new $RelativeCTR$ metric in Table 2 can help resolve this confusion caused when looking solely at the CTR numbers in Tables 3. Thus, we find that the tighter CTR numbers with the binary-level CFI models are not a result of only eliminating the extraneous or false negative call-target edges for each call-site. Rather, the lack of precise program analysis information at the binary-level causes the CFI models to produce significant numbers of *false positive* (indicated by $RelativeCTR_T$) and *false negative* (indicated by $RelativeCTR_F$) edges. Thus, we conclude that **all CFI models for the binary configurations display high error rates that is not captured by the existing metrics used to measure the performance of CFI policies, like CTR.**

We also observe that the *Mean* $RelativeCTR_T$ values are significantly lower for all benchmark categories with the stricter MCFI and IFCC CFI policies compared to TypeArmor and τCFI. Likewise, the *Mean* $RelativeCTR_F$ values are much higher for MCFI and IFCC compared to TypeArmor and τCFI. While this is not a particularly surprising result in hindsight, the extent of the observed error is quite staggering. Thus, we find that the *mean* number of *correct* or *true negative* ($RelativeCTR_T$) edges recovered by the MCFI policy even in the Binary config. **I** (with debug symbols available) drops to as low as 0.10 and 0.14 for the `trafficserver` and `SPECint` benchmark categories, respectively, and with less than 50% of the *true negative* edges recovered for all but one benchmark suite. Likewise, the number of *incorrect* or *false negative* ($RelativeCTR_F$) edges recovered is as high as 0.66 and and 0.65 with the MCFI policy in the Binary config. **I** for benchmark suites `postgresql` and `SPECint`, respectively. It is also interesting to note that binary-level SRE tools struggle to recover precise program analysis information even for binaries with debug symbol information available, resulting in poor performance by CFI models employing such information. This level of imprecision by binary-level CFI techniques is not something that has been observed or reported by earlier works that used simple metrics like the CTR. Thus, we conclude that, **binary-level CFI models, like MCFI and IFCC, that rely on more precise program analysis information are significantly more erroneous, compared to the simpler CFI models, like TypeArmor and τCFI.**

From Figure 2 we can also observe that in almost every case, the $RelativeCTR_T$ values are lower, while the $RelativeCTR_F$ values are higher for benchmarks that have been stripped of debug symbols (binary config. **II**) compared to programs with debug information intact (config. **I**). Thus, it is clear that the greater imprecision in static analysis information that is recovered by SRE tools for stripped binaries results in degrading the performance for security and optimization algorithms that rely on such data. While this is also an expected and intuitive result, there has never previously been an attempt or a mechanism to observe, measure, and report the amount of error in CFI policies. If anything, it is interesting to note that the magnitude of error displayed by binary-level CFI policies in config. **II** programs, with symbols stripped, is not very large in several cases, compared to the inherent error already present in CFI models in config. **I**. We even find that in a few cases, like the *mean* $RelativeCTR_T$ for `trafficserver` with the MCFI policy, and the *mean* $RelativeCTR_F$ for `postgresql` with the IFCC policy, stripped benchmarks produce marginally better performance compared with unstripped benchmarks. Overall, we can conclude that **binary-level CFI policies produce significantly more erroneous results for benchmarks that are *stripped* of debugging symbols, compared to binaries that retain their debug symbols information.**

4.5 On The Accuracy of Program Analysis Information

Our results demonstrate that all the binary-level CFI policies modelled in this work show high levels of inaccuracy. This inaccuracy may be manifested by the CFI policies allowing incorrect control flow transfers while tagging correct control flow transfers as erroneous at run-time. The limitations in binary-level CFI models are caused by the imprecision in the extracted program analysis information from binaries by the SRE tools. Therefore, we further investigated the causes of inaccuracies of the relevant program analysis information collected by advanced SRE tools (IDA Pro, in this case). We present some interesting observations from this analysis in this section.

We observe that state-of-the-art SRE tools can accurately detect *the number of* call-site (89% in **I** and 88% in **II**) and function argument counts (95% in **I** and 85% in **II**) in most cases. Interesting is the observation that the lack of symbol information (in **II**) does not significantly affect the accuracy of argument count detection. This high accuracy is reflected in the relatively high $RelativeCTR_T$ and low $RelativeCTR_F$ numbers for most benchmark suites in Table 2.

We discover that the accuracy of preliminary type detection at call-sites and functions is 62% and 89% respectively in setting **I**. But the accuracy decreases significantly (44% and 45% respectively) in **II**. Likewise, the detection accuracy of base pointer types is around 35% and 84% at call-site and call-targets respectively in setting **I**, but the decreases to about 9% and 5% in setting **II**. With some manual analysis with the `Nginx` benchmark, we found that the mischaracterization of the `struct*` type as `int64` by the binary analysis tool is one important reason for the high error rate. The poor preliminary and pointer type detection by the SRE tools, especially with config. **II**, likely results in the high error rates witnessed in the MCFI and IFCC CFI policies at the binary level.

5 Future Work and Conclusion

Our goal in this work was to explore and quantify the precision of binary-level CFI techniques, and study how that precision is impacted by the inaccuracies in the program analysis information recovered by modern SRE tools. We developed a comprehensive infrastructure, a thorough mechanism, and new metrics to achieve this goal. Our modular framework can model and evaluate different binary-level type-based CFI policies by comparing their outcomes with their source-based counterparts. We demonstrated our framework and reported results for four binary-level CFI policies. The results with our novel mechanism and metrics highlight the unresolved challenges for modern SRE tools in correctly extracting the relevant program information, and their potentially staggering impact on the precision of binary-level CFI techniques that use such data.

There are several avenues for future work. We only study type-based CFI policies in this work. In the future we will augment our current target set analysis by using advanced type propagation and pointer analysis to extend this work to other CFI mechanisms. Likewise, the *false positive* and *false negative* numbers for the evaluated binary-level CFI policies in this work report the *incorrect* call-target edges according to the CFI algorithm. In the future we will develop experiments and metrics to understand how these false edges actually cause a legal program execution to fail or increase program vulnerability at run-time.

References

1. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security. CCS '05, Association for Computing Machinery, New York, NY, USA (2005)
2. Burow, N., Carr, S.A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., Payer, M.: Control-flow integrity: Precision, security, and performance. ACM Comput. Surv. **50**(1) (apr 2017)
3. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.R., Shacham, H., Winandy, M.: Return-oriented programming without returns. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. p. 559–572. CCS '10, Association for Computing Machinery, New York, NY, USA (2010)
4. Designer, S.: Getting around non-executable stack (and fix). "http://ouah.bsdjeunz.org/solarretlibc.html" (1997)
5. Farkhani, R.M., Jafari, S., Arshad, S., Robertson, W., Kirda, E., Okhravi, H.: On the effectiveness of type-based control flow integrity. In: Proceedings of the 34th Annual Computer Security Applications Conference. p. 28–39. ACSAC '18, Association for Computing Machinery, New York, NY, USA (2018)

6. Frassetto, T., Jauernig, P., Koisser, D., Sadeghi, A.R.: Cfinsight: A comprehensive metric for cfi policies. In: 29th Annual Network and Distributed System Security Symposium. NDSS (2022)
7. Ge, X., Talele, N., Payer, M., Jaeger, T.: Fine-grained control-flow integrity for kernel software. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 179–194 (2016)
8. hexrays: https://hex-rays.com/ida-pro/. In: Interactive Disassembler (IDA) (2022)
9. Lan, B., Li, Y., Sun, H., Su, C., Liu, Y., Zeng, Q.: Loop-oriented programming: A new code reuse attack to bypass modern defenses. In: 2015 IEEE Trustcom/BigDataSE/ISPA. vol. 1, pp. 190–197 (2015)
10. Lettner, J., Kollenda, B., Homescu, A., Larsen, P., Schuster, F., Davi, L., Sadeghi, A.R., Holz, T., Franz, M.: Subversive-C: Abusing and protecting dynamic message dispatch. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16). pp. 209–221. USENIX Association, Denver, CO (Jun 2016)
11. Li, Y., Wang, M., Zhang, C., Chen, X., Yang, S., Liu, Y.: Finding cracks in shields: On the security of control flow integrity mechanisms. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. CCS '20, Association for Computing Machinery, New York, NY, USA (2020)
12. LLVM: https://clang.llvm.org/docs/controlflowintegrity.html. In: Clang (2022)
13. LLVM: https://llvm.org. In: The LLVM Compiler Infrastructure (2023)
14. Muntean, P., Fischer, M., Tan, G., Lin, Z., Grossklags, J., Eckert, C.: τcfi: Type-assisted control flow integrity for x86-64 binaries. In: Research in Attacks, Intrusions, and Defenses. pp. 423–444. Springer International Publishing, Cham (2018)
15. Muntean, P., Neumayer, M., Lin, Z., Tan, G., Grossklags, J., Eckert, C.: Analyzing control flow integrity with llvm-cfi. In: Proceedings of the 35th Annual Computer Security Applications Conference. p. 584–597. ACSAC '19, Association for Computing Machinery, New York, NY, USA (2019)
16. Niu, B., Tan, G.: Modular control-flow integrity. PLDI '14, Association for Computing Machinery, New York, NY, USA (2014)
17. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In: 2015 IEEE Symposium on Security and Privacy. pp. 745–762 (2015)
18. Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security. p. 552–561. CCS '07 (2007)
19. Team, P.: Rap: Rip rop. In: Hackers 2 Hackers Conference (H2HC) (2015)
20. Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, U., Lozano, L., Pike, G.: Enforcing forward-edge control-flow integrity in gcc & llvm. In: Proceedings of the 23rd USENIX Conference on Security Symposium. p. 941–955. SEC'14, USENIX Association, USA (2014)
21. van der Veen, V., Göktas, E., Contag, M., Pawoloski, A., Chen, X., Rawat, S., Bos, H., Holz, T., Athanasopoulos, E., Giuffrida, C.: A tough call: Mitigating advanced code-reuse attacks at the binary level. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 934–953 (2016)
22. Wang, M., Yin, H., Bhaskar, A.V., Su, P., Feng, D.: Binary code continent: Finer-grained control flow integrity for stripped binaries. In: Proceedings of the 31st Annual Computer Security Applications Conference. p. 331–340. ACSAC '15, Association for Computing Machinery, New York, NY, USA (2015)

23. Xu, X., Ghaffarinia, M., Wang, W., Hamlen, K.W., Lin, Z.: Confirm: Evaluating compatibility and relevance of control-flow integrity protections for modern software. In: Proceedings of the 28th USENIX Conference on Security Symposium. p. 1805–1821. SEC'19, USENIX Association, USA (2019)
24. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy. pp. 559–573 (2013)
25. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Proceedings of the 22nd USENIX Conference on Security. SEC'13, USENIX Association, USA (2013)