TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation

Anonymous Author(s)

ABSTRACT

Maliciously-overwritten function pointers in C programs often lead to arbitrary code execution. In principle, forward CFI schemes mitigate this problem by restricting indirect function calls to valid call targets only. However, existing forward CFI schemes either depend on specific hardware capabilities, or are too permissive (weakening security guarantees) or too strict (breaking compatibility).

We present TyPro, a Clang-based forward CFI scheme based on type propagation. TyPro uses static analysis to follow function pointer types through C programs, and can determine the possible target functions for indirect calls at compile time with high precision. TyPro does not underestimate possible targets and does not break real-world programs, including those relying on dynamically-loaded code. TyPro has no runtime overhead on average and does not depend on architecture or special hardware features.

1 INTRODUCTION

C programs are ubiquitous, and so are memory-corruption vulnerabilities in them. Code-reuse attacks exploit these vulnerabilities by overwriting return addresses or function pointers with references to malicious code, ultimately gaining arbitrary code execution. Control Flow Integrity (CFI) aims to mitigate code-reuse attacks by enforcing that return and call targets are valid [1]. CFI protects two types of control transfers: (i) the return address of any function on the stack (*backward CFI*) and (ii) the targets of indirect calls (*forward CFI*). Given that backward CFI has reached production grade and gained hardware support [38, 43], and seeing that a plethora of works protects indirect calls in C++ [2, 4, 6, 13, 21, 31], in this paper, we focus on forward CFI in C programs.

In the C language, an indirect call invokes a *function pointer*. Forward CFI schemes check this pointer before the call, ensuring it points to a "valid" target. To this end, industry-grade and widely deployed forward CFI schemes (Microsoft's Control Flow Guard [28], or Intel CET's indirect branch tracking [38]) merely test if *any* valid function is called. This crude over-approximation enables attackers to call functions that are not reachable from the given call site. Therefore, more precise forward CFI schemes compute a tailored *target set* for each indirect call, containing all function pointers that are allowed for this call. On the one hand, the target set must be large enough to allow every intended function pointer. Otherwise, the protected program may crash. On the other hand, the target sets must be minimal, as every unnecessary target represents a gadget that attackers might use in code-reuse attacks. The most promising example of a more precise scheme is Clang CFI [47]. It checks the C type of the called function and compares it to the expected call type. Such compiler-integrated analyses allow for easy integration in off-the-shelf software—significantly easing wide CFI deployment.

Unfortunately, as we will show in Section 6.1, Clang CFI and its strict type checks regularly miss valid call targets. Ultimately, this impreciseness may lead to unforeseen program crashes during runtime, which *cannot* be detected beforehand. That is, whether or not a CFI-protected program crashes is only known (at some point) during runtime. Due to these deficiencies, even hardened OSes cannot deploy Clang CFI to all applications. To quote the HardenedBSD maintainers: "*We may need to disable cfi-icall [Clang's forward CFI for C-style function pointers] for more applications, and we'll need to rely on our user base to identify edge cases.*" [17]

Clearly, this is an unsatisfying state. As a non-solution, one could revert to more permissive CFI schemes. For example, IFCC [49] just compares the *number* of arguments, but not their *types*. But this policy allows many unnecessary targets, bloating the surface of code-reuse gadgets (cf. Section 6.2). Seeing that such weaker CFI designs unnecessarily undermine security, we seek to understand the root causes of Clang CFI's failures. We identify three conceptual reasons why Clang CFI is incompatible with popular software projects such as lighttpd, nginx, or redis. First, Clang CFI lacks type propagation. Therefore, it does not allow casts to/from undefined types (`void *`) that programmers often use to build inheritance-like constructs. Second, Clang CFI does not support variadic functions. Third, Clang CFI does not support dynamic linking.

As a workaround, developers could try to rewrite programs that cause CFI incompatibilities. However, this is a non-trivial task, requires significant code revisions, and comes at the price of losing flexibility. For example, plugin interfaces (e.g., in nginx and lighttpd) heavily rely on variadic functions or generic APIs that operate on `void *` pointers. Furthermore, the lack of support for dynamic linking impedes CFI adoption and cannot be "fixed" by refactoring.

Alternatively, and the idea of this work, we can design a CFI system for Clang that supports the above compatibility features. However, the C standard foresees concepts that impose challenges, such as (i) function pointer casts, (ii) function pointers that are part of compound data types such as `struct` and `union`, or (iii) function pointers that are propagated through other indirect calls. Supporting these features is vital not to break programs. Moreover, any function pointer analysis must be context-sensitive to refine the set of valid call targets. That is, not only do we have to match function types. We *also* have to verify that a given pointer can ever be used as an indirect call target in a benign execution path. Finally, the target sets may *change* when new program parts are loaded during runtime. However, existing CFI schemes often assume a static setting and cannot support shared libraries.

To tackle some of these challenges, existing forward CFI schemes (i) use dynamic or runtime analysis [10, 19, 24], (ii) require kernel-level modifications [10, 15, 19, 51] or orthogonal defenses such as shadow stacks to be in place [23, 24], or (iii) rely on architecture-specific features to recognize valid call targets in real-time [10, 15, 19, 23, 24, 27, 33, 51]. Therefore, these solutions sacrifice generality and are not agnostic to the underlying OS and hardware. Thus, we still lack a generic and software-only forward CFI solution that is neither too permissive nor restrictive.

In this paper, we propose TyPro, a drop-in replacement for Clang's forward CFI scheme. TyPro uses static analysis to propagate function types (1) to gain a compatible CFI scheme for Clang, and (2) to tackle the open challenges in existing forward CFI systems. To this end, we extract types and casts from a program's Abstract Syntax Tree (AST), and follow how they propagate to other functions (i.e., contexts). We derive rules from the C standard that capture all permitted type propagations. We then leverage a solver that uses the type information and propagation rules to extract accurate *target sets*, i.e., functions that are valid for a given call target. We enforce these target sets by rewriting indirect function calls with switch/case constructs that can no longer be abused for function pointers other than those in the target set.

We developed TyPro as an LLVM-based open-source prototype, available at https://github.com/typro-type-propagation/TyPro-CFI. TyPro is a software-only solution, fully compatible with dynamic linking and loading of shared libraries. Our systematic conformity to the C standard pays off, especially for large real-world programs: Our protection does not break legacy code and computes target sets more precisely than industry standards (CFGuard, CET) or IFCC. Furthermore, TyPro is efficient and does not cause measurable runtime performance overhead in protected applications.

2 OVERVIEW

2.1 Attacker Model

TyPro aims to defend against function pointer corruptions in C programs, where an attacker wants to divert control flow to execute arbitrary code. We consider attackers that know the program's memory layout and can read from and write to all memory locations within the boundaries of page permissions. We assume W⊕X, i.e., that no pages exist that are both writable *and* executable. TyPro protects forward edges (function pointers), so we assume that return addresses are covered by any other orthogonal scheme [5]. Furthermore, while we consider dynamically-*loaded* code, we exclude dynamically-*generated* code such as just-in-time compilation, and suggest additional protections [36, 44, 54] if necessary.

2.2 Challenges

To build a secure forward CFI, one has to solve the challenge of finding a precise set of allowed target functions for each indirect call in C. Not every function that is ever referenced by a function pointer—we refer to those as *address-taken functions*—is a valid call target for any indirect call site. There are two validity conditions to be checked, whereas existing CFI schemes only consider the first. (1) The type of the address-taken function "matches" the type of the function pointer used in the indirect call. For example, an indirect call that passes just a single argument is clearly incompatible with functions that expect multiple arguments. Types do not necessarily have to be identical but should be "compatible". (2) There is a program execution in which the function's pointer will be passed to the respective indirect call site. That is, assume the function signatures of two functions `A` and `B` are identical, but `B`'s pointer is kept local in a function unrelated to the call. Then, `B` is never a valid call target.

Related software-only CFI schemes only perform function type checking and ignore the function's context. For example, Clang CFI [47] and MCFI [35] perform a rigid function type checking.

```
1  typedef void (*fptr_long)(long);
2  typedef void (*fptr_int)(int);
3  typedef void (*fptr_ptr)(fptr_long);
4  void f1(long a) {}
5  void f2(long a) {}
6  void f3(long a) {}
7
8  void scene1_a() {
9      fptr_int f = (fptr_int) &f1;
10     scene1_b(f);
11 }
12 void scene1_b(fptr_int f) { f(0); }        // call1
13
14 struct S { fptr_long one; fptr_int two; };
15 void scene2_a() {
16     struct S s = { &f2, 0 };
17     scene2_b(&s);
18 }
19 void scene2_b(struct S *s) { s->one(0); } // call2
20
21 fptr_long callback;
22 void set_callback(fptr_long f) { callback = f; }
23 void scene3_a() {
24     fptr_ptr some_cb_target = &set_callback;
25     some_cb_target(&f3);                    // call3
26 }
27 void scene3_b() { callback(0); }           // call4
```

Figure 1: Code example showing different ways to transfer function pointers.

While this strict type checking is intuitive and straightforward, it is too restrictive and regularly corrupts programs. Indeed, called functions may have different types than the function pointers. In the code example shown in Figure 1, we see a trivial example where this happens in lines 8-12: The indirect call in line 12 targets the function `f1` (expecting an argument of type `long`), which has a different signature than the function pointer in argument `f` (expecting an argument of type `int`). Consequently, a strict type check will not allow this function call and will mistakenly terminate the program.

As we will also experimentally show, this strict function type checking is impractical for many programs. On the other extreme, we may thus consider forward CFI schemes with less restrictive type checks. For instance, one could simply count the number of arguments instead of validating their type, like IFCC [49]. However, such generous "type matching" allows significantly more valid call targets than required for correctness, increasing the attack surface. Regarding our code example, for the indirect call in line 12, IFCC considers not only `f1` as a target, but also any other function with only one argument present in the codebase. However, in principle, the indirect call in `scene1_b`, with callee of type `fptr_int` can only target `f1`. Such over-permissive CFI systems unnecessarily bloat the attack surface for code-reuse attacks.

2.3 Methodology at a Glance

We aim for a sweet spot between the "too permissive" and "too restrictive" forward CFI schemes. In particular, our goal is to correctly track function types even in (typical) situations, such as the following three: (i) *A function pointer is cast to another type.* For example, regarding Figure 1, the invocation of `f` (line 12) requires such tracking, as discussed before. (ii) *A function type is hidden in compound data structures such as struct, union, pointer, or array.* In Figure 1, lines 14–19 provide such an example, where the function

pointer `f2` is part of struct `S` that is passed as pointer to the caller function `scene2_b`. And (iii), *a function pointer is propagated through other indirect calls*, such as the example in lines 21–27, where the call target for *call4* depends on the arguments and target of the indirect *call3*. None of the current strict CFI schemes correctly cover all these three situations, which causes them to regularly fail many real-world programs (as demonstrated in Section 6.1).

We propose a static analysis method that propagates types to tackle the challenge of collecting restricted yet correct sets of call targets. TyPro operates on source code in the form of an abstract syntax tree (AST). We track types per function, including casts between them, and we track which types are exchanged between functions, even in nested data types. If there is a propagation path from a function's type to an indirect call's type, we know this function is a valid target.

Our type propagation is roughly split into three phases. First, we extract "initial" type information from the AST. Consider the indirect call in lines 8–12 in Section 2.2. Here, our analysis collects `f1`'s and `fptr_int` type declarations, a cast from `f1`'s type `fptr_long` to `fptr_int`, and also that `fptr_int` is used as `scene1_b` argument type. We store this information as *facts*—logical formulas that are *assumed* to be true before the analysis starts. Facts represent one of the Horn clause types [18]. This encoding has been employed by a plethora of analyses for various properties [3, 14] as it allows for leveraging advanced solvers [9, 22] to compute the actual result.

Second, we use the facts to find a set of valid targets for every indirect call in the code at link-time. To this end, we first specify a set of *rules*—another Horn clause type. They are logical implications that describe how to *derive* new information (rule's body) starting from the initial facts or the information derived being true by the previous rules' applications (rule's head). In particular, our rules describe how to compute all possible type propagations permitted by the C standard. They thus form the basis for computing the final set of allowed functions (i.e., the *target set*) for each indirect call.

Third, we supply the facts and the rules to a solver that derives the target sets while applying the rules mentioned above. This process continues to the point when rule applications do not derive any new target for any indirect call. In our example, the solver concludes that `f1`'s type is the only type that propagates to *call1*, and consequently, `f1` is the only valid function in *call1*'s target set.

Finally, after the type propagation analysis stage, when generating the program binary, we *enforce* the resulting target sets. We assign an ID to each address-taken function and replace the function's address with that ID. We transform every indirect call into a switch over different direct calls that only target functions in the valid target set. To support dynamic linking and loading, a *runtime library* updates the target sets whenever new modules appear at runtime via just-in-time compilation.

2.4 Type Propagation vs. Data Flow

In contrast to the existing techniques based on data flow [10, 19, 23, 24, 51, 53], our approach does not compute the type evolution during program execution. Data flow tracking is precise but an expensive computation, as it requires several non-trivial components (e.g., control flow). Instead, our type propagation performs lightweight processing of the information extracted only from the points in the program's source code where types are created, manipulated (e.g., through casts), and used. Although employing data flow-based approaches may help to obtain even more precise approximations while computing indirect call targets, our experimental evaluation (presented in Section 6) demonstrates the effectiveness of our system in reducing the target sets for indirect calls. In other words, we do not see any drastic effects from the approximations we use, while the absence of flow information helps to make the compilation time tractable, as discussed in Section 7.

TypeContextPair : $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$	(available types and their contexts)
TypeCall : $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$	(types of indirect call parameters)
PointsTo : $\mathbb{N} \times \mathbb{N}$	(pointer to pointee mapping)
StructMember : $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$	(struct fields with memory offset)
UnionMember : $\mathbb{N} \times \mathbb{N} \times \mathbb{S}$	(union fields with field type)
Cast : $\mathbb{N} \times \mathbb{N}$	(type propagations)
ICall : $\mathbb{N} \times \mathbb{S} \times \mathbb{N}$	(indirect calls for callee type)
FunctionPointer : $\mathbb{N} \times \mathbb{S} \times \mathbb{N} \times \mathbb{B}$	(address-taken functions)
TargetSet : $\mathbb{S} \times \mathbb{S}$	(possible indirect call targets)

Figure 2: Predicate signatures for the facts used by analysis.

3 TARGET SET COMPUTATION

Our analysis technique starts by extracting *facts* about the program's code. Then we use the facts in the program-independent *rules* computing the type propagation. We use the sample code shown in Figure 1 to highlight the propagation rules necessary for our analysis's correctness. As the result of our analysis, we obtain the possible target functions set for every indirect call.

3.1 Analysis Input Generation

Our pipeline starts after Clang's parser produces an AST representation of the C program, which intuitively constitutes a collection of expressions [48]. From this tree, we extract only the information relevant to types, their propagation and usage. We then create the corresponding Horn clause facts, as introduced in Section 2.3. Our facts use several boolean relations, as shown in Figure 2. Each relation has a corresponding predicate signature which determines its domain. For instance, predicate signature $\mathbb{N} \times \mathbb{S} \times \mathbb{S}$ for relation TypeContextPair expresses that facts in this relation take three arguments, the first one is a natural number $\mathbb{N}$ (a unique identifier), the second and the third are strings $\mathbb{S}$ (in this case: a type name and a function's name). Facts are specific to each program. We collect the facts for each compilation unit (source file) and merge on linking. Overall, we split facts into four groups depending on the information they contain: initial types and context, type constructs, functions & calls, and casts & transfers:

Initial Types and Context Collection. Before we can follow a type's propagation, we first need to know all types in the program's code. In addition, we require a *context* annotation for each type. The context is a function or a global variable where C expressions occur. Using pairs of types and context instead of pure types improves the precision of the analysis—type propagations can be kept local to a group of functions. With contexts considered, type propagations from unrelated parts of the source code will not influence the computation. We leverage the AST to extract unique pairs of C types and their context to get the required information. In particular,

- For every expression e with type t in function f, we collect the pairs (t, f). For example: (`fptr_int`, `scene1_a`).
- For every function declaration f with return or parameter types $t_0 ... t_m$, we collect the pairs $(t_i, f), i \in \{0, \ldots, m\}$. For example: (`fptr_int`, `scene1_b`).
- For every global variable g of type t, we collect the pair (t, g). For example: (`fptr_long`, `callback`). If g is initialized, we also collect the pairs (t', g) for every expression e with type t' inside the initialization code.

We assign each unique pair an identifier $n \in \mathbb{N}$. This allows us to refer to each pair by number, simplifying the rules, saving storage, and improving computation time as outlined in Section 5. To this end, we define a function $N : \mathbb{S} \times \mathbb{S} \to \mathbb{N}$ that returns the identifier n for each collected type-context pair. For convenience, we define its second version $N : \mathbb{S} \to \mathbb{N}$ that returns the identifier n for each AST expression, based on its C type and the corresponding context object. We store each type/context pair (t, c) together with their corresponding identifier $n = N(t, c)$ as fact TypeContextPair(n, t, c).

In our example (Figure 1), line 9 corresponds to several AST expressions encoded as TypeContextPair(0, `fptr_long`, `scene1_a`) as we use function pointer `&f1` of type `fptr_long` in the context of function `scene1_a`, and TypeContextPair(1, `fptr_int`, `scene1_a`) as we cast `&f1` to type `fptr_int`, see Figure 7 in Appendix A. Similarly, line 19 produces the TypeContextPair(8, `struct S *`, `scene2_b`) fact, as we declare parameter of type `struct S *` for function `scene2_b`. Recall that also global variables can be used as a context. For example, we define a global variable `callback` in line 21, which adds the TypeContextPair(17, `fptr_long`, `callback`) fact.

Type Constructs. To correctly track type propagation in complex data structures, we must collect the construction information of all derived types in a program. In particular, for every fact TypeContextPair(n, t, c), we do the following depending on t:

- If t is a pointer or array type ($t := t'*$ or $t := t'[\ldots]$), we compute the identifier $n' = N(t', c)$, add another fact TypeContextPair(n', t', c), and also store the pointer or array type structure as PointsTo(n, n') fact.
- If t is a structure type ($t := \texttt{struct}\{t_0, ..., t_m\}$), we compute the identifiers $n_i = T(t_i, c)$ with $i \in \{0, \ldots, m\}$ where each type t_i corresponds to a struct field. Furthermore, we add fact TypeContextPair(n_i, t_i, c) for each field, and record the type structure as StructMember(n, n_i, x_i) facts, where x_i is the byte offset of field i in the struct's layout.
- If t is a union type ($t := \texttt{union}\{t_0, ..., t_m\}$), similar to the structure type, we add facts TypeContextPair(n_i, t_i, c) for each member, but record the type structure as UnionMember(n, n_i, t_i) facts.

Figure 8 in Appendix A visualizes the facts that we derive for our example code, they represent a type's construction in a tree-like form. The type constructs analysis starts with fact TypeContextPair(8, `struct S *`, `scene2_b`)—as obtained during the initial types and context collection—that implies that `struct S*` is a pointer type. Hence, we introduce another fact TypeContextPair(9, `struct S`, `scene2_b`) and store the points-to information as fact PointsTo(8, 9). Consequently, we check the new fact TypeContextPair(9, `struct S`, `scene2_b`) and as it is a structure type fact (as shown on the line 14) we also add facts TypeContextPair(10, `fptr_long`, `scene2_b`) and TypeContextPair(11, `fptr_int`, `scene2_b`), plus the facts capturing type structure: StructMember(9, 10, 0) and StructMember(9, 11, 8).

Functions & Calls. To determine the target sets for indirect calls, we need to identify all function pointer types that propagate to each indirect call. To this end, we need to know all functions taken as function pointers (address-taken functions) and all types of the functions that get indirectly called—the start and destination of our propagation path, respectively. To this end, we collect fact TypeContextPair(n, t, c) plus the additional information from expression e if one of the cases applies:

- **Address-taken functions**: If e represents the address of a function f outside of a direct call, we store the function with its type, its number of arguments m, and whether it accepts a variable number of arguments *vararg* in FunctionPointer$(n, f, m, vararg)$ fact. In our example, we produce facts FunctionPointer(0, `f1`, 1, false) for `&f1` in line 9, and FunctionPointer(13, `set_callback`, 1, false) for the AST expression "`&set_callback`" in line 24.
- **Indirect calls**: For indirect calls, i.e., $e := e'(a_1, ..., a_m)$, we record the callee's expression identifier, a reference to the call expression, and the number of arguments in fact ICall$(N(e'), e, m)$. In our example, line 12 produces fact ICall(10, *call2*, 1), and line 25 creates fact ICall(14, *call3*, 1).

Casts & Transfers. After having collected types, start and destination of type propagation paths, we need to know the actual propagation steps: We look for the specific AST expressions that transfer the type or context. This information populates Cast facts. In other words, we do not differentiate between type casts or context transfers in our analysis because it operates on type/context pairs. In particular, for an AST expression e, for which we also collect the fact TypeContextPair(n, t, c), we check for these cases:

- **Type casts**: For a cast $e := (t)\ e'$ and sub-expression e' with TypeContextPair(n', t', c), we produce a fact representing a cast from t' to t: Cast(n', n). This kind of fact covers all casts that C supports, including implicit casts and qualifier casts. In our example, we have a cast in line 9, casting an expression of type `fptr_long` to `fptr_int` in the context of function `scene1_a`. Using the corresponding type/context identifiers (obtained during the initial type and context collection) that the analysis stores in the TypeContextPair facts, we record this as Cast(0, 1), see Figure 7.
- **Global variable uses**: If expression $e := g$ accesses a global variable g, having a corresponding TypeContextPair(n', t, g) fact, we record this as an implicit cast: Cast(n', n). Intuitively, this cast allows us to derive the propagation of type t from the global's context to the context of expression e. If this access could be a write, we need to account not only for context transfer from the global context but to the global context itself. In other words, type t should become accessible at the global context. We capture this bidirectional context transfer by adding fact Cast(n, n'). Note that the type of the global variable t equals the type of the expression accessing it, only the context changes.
  In our example, there is a global variable `callback`, written in function `set_callback` and accessed in function `scene3_b`, see Figure 9 in Appendix A. For the write access, we record facts Cast(17, 18) and Cast(18, 17). For the read access, we record Cast(17, 19).
- **Direct calls**: Similar to global variables, direct calls also manipulate the context. In a call $e := f'(a_1, ..., a_m)$ to a function with

declaration $f'(p_1, ..., p_m)$, we produce for every argument a_i a separate cast fact: Cast $(N(a_i), N(p_i, f'))$. For the return value of type rt, we also produce a cast fact: Cast$(N(rt, f'), n)$. In our example, `scene1_b` is called in line 10, with an argument represented as TypeContextPair(1, `fptr_int`, `scene1_a`). We capture this by recording Cast$(1, 2)$, see Figure 7. Similarly, for the call from `scene2_a` to `scene2_b` in line 17, we record Cast$(7, 8)$.

- **Indirect calls**: For indirect calls $e := e'(a_1, ..., a_m)$, the type of e' is a function pointer $t_{e'} = rt(*)(p_1, \ldots, p_m)$ with parameter types p_i and return type rt. We want to store the casts as we do for a direct call. To this end, we use a new relation TypeCall, which uses indexing similar to TypeContextPair, but records a reference to indirect call e (also discussed in *Functions and Calls*) instead of a context. We add the facts TypeCall(n_i, p_i, e) and TypeCall(n_{rt}, rt, e), also we define $n_i = N(p_i, e)$ and $n_{rt} = N(rt, e)$ for $i \in \{1, \ldots, m\}$ accordingly. With these new facts, we can then add the casts similar to a direct call: Cast $(N(a_i), n_i)$ and Cast(n_{rt}, n).
  Note that a TypeCall fact does not contain the context information in which the type of the argument (in the indirect call) is defined. Ultimately, this approximation allows our analysis to extract all possible contexts for each of the types used in an indirect call, which is crucial for the correctness of the target set computation that we discuss in Section 3.2. In our example, we have indirect call *call3* in line 25. In addition to indirect call fact ICall discussed earlier, in this case, we produce a cast fact for its first argument Cast$(15, 16)$ and a TypeCall(16, `fptr_long`, *call3*) fact (see Figure 9, Appendix A). In Section 3.2 we show how these facts are used to relate this indirect call parameter type to the type of its argument FunctionPointer(15, `f3`, 1, false).

3.2 Type Analysis

Having obtained *facts* from the AST, we now describe how we use these facts for type propagation reasoning. Our type analysis is based on a collection of *rules*. We define these generic rules as constrained Horn clauses [18] *once*, i.e., they are program-independent. The rules thus specify the logic behind our analysis type reasoning. They follow type propagations across functions, through nested compound types and indirect calls. Finally, they specify how to obtain from these type propagations a set of possible targets for each indirect call as a result of TargetSet relation computation: For every function f, which is a valid target for indirect call c, the rules show how to derive TargetSet(c, f).

Figure 3 depicts the rules we use to compute TargetSet relation containing the final result, i.e., targets for each indirect call. We obtain this result when the analysis cannot extend TargetSet relation via rule application; in other words, no more information can enter the relation. Note that all variables used in the rules have the types corresponding to the predicate signature presented in Figure 2. Moreover, these variables are universally quantified—each rule can be used multiple times for each variable assignment that makes it applicable. Ultimately, TargetSet contains a mapping from indirect calls to their possible targets.

The first group of rules, which consists of the rules (T), (P), (S), and (U), allows our analysis to extend the Cast relation. Computation of this relation allows our analysis to account for transitivity of multiple propagations (rule (T)), pointer aliasing (rule (P)), and structures & unions (rules (S) and (U)). Intuitively, the Cast relation captures all possible propagation paths of the types and contexts.

The second group of rules—(F1) for fixed and (F2) for variable number of arguments—use the Cast relation to compute our final result, the TargetSet relation. In particular, we access the computed propagation paths between function pointers and indirect calls.

However, with the first two groups of rules, we can compute only a partial result for the TargetSet relation. The last group of rules (IC1, IC2) uses this partial result to enrich TargetSet, accounting for context casts from the arguments of an indirect call to its actual target, as it happens in *call3*.

In the rest of this subsection, we detail the reasoning behind the rules in each group.

Type/Context Manipulations. Rule (T) computes the relation Cast, e.g., type and context propagation paths over multiple casts. In our example code, this happens in lines 8-12 (see Figure 7 in Appendix A) for which we already extracted the facts Cast$(0, 1)$ and Cast$(1, 2)$. Using these facts, rule (T) derives the fact Cast$(0, 2)$ (grey arrow in the figure), revealing the relation between *call1* and `f1`. Later, we will use this new fact to derive one of the target set results, namely TargetSet(*call1*, `f1`).

Rule (P) handles pointer casts and pointer aliasing. Whenever a pointer is cast to another pointer type, the two pointers *alias*, they point to the same value in memory. For us, this implicates that something can be referenced by two potentially distinct pointee types, so an implicit type or context propagation can happen. This propagation is possible in both directions, depending on which pointer is accessed afterward. In our example, this happens in lines 14-19; see Figure 8 in Appendix A. From this code, we extracted two PointsTo facts, one for the type `struct S *` in `scene2_a`, and one in `scene2_b`, plus a fact for context propagation Cast$(7, 8)$ between these two. Then we use rule (P) to derive Cast$(3, 9)$ and Cast$(9, 3)$.

Rules (S) and (U) handle struct and union types. When one struct is transferred into another, this implicitly transfers all its fields. By referencing the fields by their memory offset, we map fields to each other and collect newly-introduced Cast facts. We handle unions similarly. However, values from a union can only be read as the same type as they were written. We use the field type instead of the field byte offset for unions. In our example, this happens in lines 14-19; see Figure 8, Appendix A. Having derived Cast$(3, 9)$ with the rule (P) before, now we derive Cast$(4, 10)$ (and Cast$(5, 11)$) using rule (S). These new casts connect the function pointer "`&f2`" from line 16 with *call2*, as the result of computation of Cast relation.

Target Set Computation. Using the possible type propagation paths obtained from the Cast relation, the rules (F1) and (F2) compute the actual target set TargetSet. If a function pointer propagates to the callee argument of an indirect call, rules (F1) and (F2) add this function as a possible target. In addition, these rules check that the number of parameters is valid. Rule (F1) handles functions with a fixed number of parameters, which must match the number of arguments in the indirect call. Instead, while otherwise similar, rule (F2) handles functions with a variable number of parameters.

In our example, using the previously established facts and parts of derived Cast relation computation result, rule (F1) derives target sets for indirect calls `call1`, `call2`, and `call3`: using Cast$(0, 2)$ – TargetSet(*call1*, `f1`), using Cast$(6, 10)$ – TargetSet(*call2*, `f2`), and using Cast$(13, 14)$ – TargetSet(*call3*, `set_callback`).

$$\text{Cast}(t_1, t_2) \land \text{Cast}(t_2, t_3) \implies \text{Cast}(t_1, t_3) \quad \text{(T)}$$

$$\text{Cast}(t_1, t_2) \land \text{PointsTo}(t_1, t_1') \land \text{PointsTo}(t_2, t_2') \implies \text{Cast}(t_1', t_2') \land \text{Cast}(t_2', t_1') \quad \text{(P)}$$

$$\text{Cast}(t_1, t_2) \land \text{StructMember}(t_1, t_1', x) \land \text{StructMember}(t_2, t_2', x) \implies \text{Cast}(t_1', t_2') \quad \text{(S)}$$

$$\text{Cast}(t_1, t_2) \land \text{UnionMember}(t_1, t_1', x) \land \text{UnionMember}(t_2, t_2', x) \implies \text{Cast}(t_1', t_2') \quad \text{(U)}$$

$$\text{ICall}(t_1, call, args_{call}) \land \text{FunctionPointer}(t_2, f, args_f, \text{false}) \land args_{call} = args_f \land (t_2 = t_1 \lor \text{Cast}(t_2, t_1)) \implies \text{TargetSet}(call, f) \quad \text{(F1)}$$

fixed args; param count matches; Indirect *call* to a function of type t_1; Address-taken function f of type t_2; $t_2 = t_1$ or t_2 propagates to t_1

$$\text{ICall}(t_1, call, args_{call}) \land \text{FunctionPointer}(t_2, f, args_f, \text{true}) \land args_{call} \geq args_f \land (t_2 = t_1 \lor \text{Cast}(t_2, t_1)) \implies \text{TargetSet}(call, f) \quad \text{(F2)}$$

f is vararg; enough params for req. args

$$\text{TargetSet}(call, f) \land \text{TypeCall}(t_2, type, call) \land \text{TypeContextPair}(t_3, type, f) \land \text{Cast}(t_1, t_2) \implies \text{Cast}(t_1, t_3) \quad \text{(IC1)}$$

$$\text{TargetSet}(call, f) \land \text{TypeCall}(t_2, type, call) \land \text{TypeContextPair}(t_1, type, f) \land \text{Cast}(t_2, t_3) \implies \text{Cast}(t_1, t_3) \quad \text{(IC2)}$$

Indirect *call* targets f; *type* declared in context of *call* or f; Any t_1 (t_3) propagating to (from) t_2

Figure 3: Rules for computing the final result with all possible function types for each call.

$$\text{ICall}(t_1, call, args_{call}) \land \text{FunctionPointer}(t_2, f, args_f, \_) \land args_{call} = args_f \land t_2 = t_1 \implies \text{TargetSet}(call, f) \quad \text{(ClangCFI)}$$

$$\text{ICall}(\_, call, args_{call}) \land \text{FunctionPointer}(\_, f, args_f, \_) \land args_{call} = args_f \implies \text{TargetSet}(call, f) \quad \text{(IFCC)}$$

Figure 4: Rules demonstrating the core of Clang CFI and IFCC computation of the final function types for each call.

Indirect Call Context Transfer. Function calls propagate types by changing their contexts: the argument types propagate from caller to callee context, and the return type propagates from callee to caller context. Section 3.1 shows how to collect these propagations as Cast facts for direct calls, but for indirect calls, we only collected Cast facts from argument types to a TypeCall($n, t_a, call$) fact. Now that we have partial possible targets in TargetSet from the previous rules' application, we complete type propagation computation for indirect calls: We derive a type context propagation for every argument type from caller context to every possible target function's context and vice versa for return types.

Rules (IC1) and (IC2) calculate these context transfers from indirect call arguments to indirect call target parameters. This computation is based on a partial TargetSet result, and assumes that an indirect call actually calls all functions in its target set, which might be over-approximating. Rule (IC1) computes context transfers for indirect call arguments, while (IC2) does the same for return types. These rules derive each Cast as if an indirect call would be a direct call to each function from its target set. When generating facts for an indirect call e, we added Cast($N(t, c), N(t, e)$) for each argument, where $N(t, e)$ is also used in a TypeCall fact. From this fact, rule (IC1) derives Cast $(N(t, c), N(t, f))$ for any function f the indirect call c can target, which is exactly what we would get for a direct call to f. Rule (IC2) does the same for return types.

These rules are necessary to handle higher-order functions (indirect calls with function pointer arguments), like `call4` in line 27, see Figure 9 in Appendix A. As previously discussed, facts and derived Cast information allow us to obtain TargetSet(*call3*, `set_callback`) with rule (F1), yet we cannot derive a target set for *call4* yet. Still, we establish the fact Cast(15, 16) about the call argument of *call3*. Now applying (IC1) with these two facts, we get Cast(15, 18), connecting the call argument `f3` with the actual target `set_callback`. From Cast(15, 18), we use the other rules again (namely rule (T) for Cast and rule (F1)) to derive Cast(15, 17), Cast(15, 19), and a target set for `call4`: TargetSet(*call4*, `f3`). The final shape of the TargetSet relation is as follows: *call1* has `f1`, *call2* has `f2`, *call4* has `f3`, and *call3* has `set_callback` as their targets, respectively.

Multi-module Support. So far, we have focused on generating facts for and applying rules to a single module. To combine several modules when linking, we alpha-rename type/context pair identifiers in TypeContextPair and propagate these changes across all the other facts describing the module. Renaming preserves the identifier uniqueness required for the correct rules evaluation.

Dynamic Linking. We generate a *module summary* to support dynamic linking and loading of code. It contains all facts from a program that might influence other modules. These facts are loaded with the code at runtime, combined, and the target sets are re-computed. Appendix B describes this process in more detail.

Supporting Other Forward CFI Approaches and Updates. The rule-based target set computation approach that we use for TyPro, is also general enough to support the encoding of other CFI schemes, facilitating their development and comparison. Moreover, the expressiveness of rule-based CFI encoding allowed us, for instance, to capture the core of the target set computation performed by Clang CFI [47] and IFCC [49] with only one rule for each of the approaches. In Figure 4, the rule (ClangCFI) compactly encodes that the scheme allows only the targets with the exact same types

and number of arguments, while the rule (IFCC) relaxes the requirement of type matching. Obviously, for the example in Figure 1, the rule (ClangCFI) obtains empty results for all the calls, which is too restrictive and breaks the legitimate code. Rule (IFCC) is too permissive, resolving every call to all the functions as they all have only one argument. Section 6 details how the limitations of these approaches affect the security and stability of the programs.

Furthermore, TyPro's rule system supports updates, either as new rules or refinements, *without* changing its inner workings.

4 CALL TARGET ENFORCEMENT

After having computed the target sets of all indirect calls, we enforce the CFI policy at link time within a binary: indirect calls must transfer control only to the reduced set of targets.

To enforce that only functions from the computed targets in TargetSet can be called, we replace function pointers with function identifiers in the whole program. A function identifier is a unique number for each address-taken function. It has the same bit size as a pointer so that it can replace the function pointer in memory. This way, we must only alter its initialization and usage. We replace each use of an address-taken function that is not a direct call with this identifier. Next, we replace all indirect calls with a `switch-case` structure over the function identifiers. For each possible function in the target set, we generate a `case` matching the function ID and a direct call to the respective function in the body. In the default case, i.e., when the ID does not match any allowed function ID, we terminate the program to stop a detected attack.

To stay correct and precise, we must only allow functions that are actually valid call targets according to the C specification. First, the number of arguments must match, which is enforced by the argument number checks in rule (F1)—rule (F2) similarly handles functions with a variable number of arguments. Second, it must be possible to cast all arguments to the required type in the target function; otherwise, the call would be undefined behavior. Typecasts are usually possible if both types are an integer, pointer, or float, or have the same bit width. Last, if the call's return value is not discarded, it must be possible to cast the function's return value to the return value of the call expression. We check all these conditions before generating cases and confirm their validity experimentally in Section 6.1. Still, functions with incompatible types might occur due to overapproximation in our analysis.

Generating assembly code from `switch-case` statements is left to LLVM, which lowers them into different assembly constructs, from simple comparisons over binary trees to jumptables. Also, LLVM can run additional optimizations over the new statements, that might not be possible before our changes, e.g., direct calls can be inlined if the callee is only short or rarely used.

To improve performance, TyPro tries to assign ascending identifiers to functions that occur together in the same target set. Ascending numbers lead to dense identifier sets, which can be converted to efficient jumptables, facilitating the subsequent LLVM optimization passes. Furthermore, TyPro reserves the lower numbers (up to 3) for program-specific non-function constants (like `SIG_IGN` in libc).

For dynamic modules, as detailed in Appendix B.3, a linked library generates function IDs and new switches at runtime, based on the updated analysis results.

5 IMPLEMENTATION

We build the TyPro prototype as an extension of the Clang/LLVM 10 compiler toolchain. It targets 64-bit x86, ARM, and MIPS, generating binaries without additional dependencies on hardware features or operating system. TyPro can produce fully protected binaries with a protected musl libc [34], or protected binaries linking against an unprotected GNU libc (see Appendix C for details).

In our prototype, we instrument the code generation of Clang to collect facts (as discussed in Section 3.1 and Appendix B.1) along with the compilation of the program. The collected facts are stored together with the original LLVM IR code in an object file.

After applying several optimization steps outlined in Appendix D, we extract the facts from all IR files seen in our modified version of LLVM's linker `lld`. Finally, we encode the optimized set of facts and the rules (from Section 3.2 and Appendix B.2) in datalog for the Soufflé Logic Solver [22, 42] which we leverage to compute the TargetSet relation containing the targets for each indirect call.

We implemented a runtime library that enforces dynamically-loaded targets as a stand-alone C++ library without dependencies on LLVM or other non-standard libraries. It loads the serialized facts from multiple modules, runs the target set computation, and generates switches with a built-in just-in-time compiler. We perform all operations lazily, i.e., only after the first indirect cross-module call, preventing unnecessary computation. The library is 1.3 MB large and can either be shared or included in a protected musl libc.

6 EVALUATION

We evaluate TyPro using three criteria. In Section 6.1, we measure *correctness*. Programs must not break, i.e., target sets must always include (at least) the correct targets. In Section 6.2, we evaluate *security*. Security is largely determined by the extent to which the attack surface is reduced, i.e., how many overall targets remain in the target set. Finally, in Section 6.3, we measure TyPro's *performance* in terms of runtime slowdown and size overhead.

Evaluation datasets. We evaluate each criterion on two datasets. First, we use the well-known SPEC CPU 2006 benchmark suite [45]. In particular, we consider all nine SPEC programs written in pure C and use indirect calls, namely, *bzip2*, *gcc*, *gobmk*, *h264ref*, *hmmer*, *milc*, *perlbench*, *sjeng*, and *sphinx3*. These SPEC programs are also used in most related work [10, 15, 16, 19, 23–25, 27, 35, 37, 49, 51, 56], which allows for an easy comparison with TyPro.

Our second dataset consists of 7 larger real-world programs, commonly used in related work [15, 16, 19, 23–25, 27, 51]. It includes the web servers *Apache*, *lighttpd* and *nginx*, the FTP servers *pureftpd* and *vsftpd*, a cache server *memcached*, and the database *redis*. These programs use libraries like *libpcre*, *zlib*, *libevent*, *lua* and more.

Compilation setup. We compile all programs and libraries with TyPro enabled, using `-O3` and link-time optimization. We preferred static linking, so, when compared, no disadvantage for related work without dynamic linking support (like Clang CFI) was introduced.

For comparison, we compile all programs using the same optimization settings with an unmodified Clang 10 compiler, default libraries, and Clang CFI [47] in normal (`-fsanitize=cfi-icall`) and generalized mode (`-fsanitize-cfi-icall-generalize-pointers`). We also compared TyPro's policy to IFCC [49] and CFGuard [28].

6.1 Correctness

A CFI scheme must not break existing code to foster adoption in practice. While manual code changes could resolve incompatibilities, this is not feasible in many cases.

SPEC. We run the SPEC CPU benchmarks with all input sets and verify the output. TyPro handles all nine programs without any failure, as shown in Table 1. In contrast, Clang CFI causes crashes in *gcc* and *hmmer* computing too narrow target sets. These two benchmarks failed even in Clang CFI's generalized mode with relaxed type checking. These failures were observed only post-mortem as crashes, and it might be hardly feasible to alter the source of such big projects as *gcc* to respect the typing judgments. From the related work [25], we also know that MCFI [35] crashes on *perlbench* and *gcc*, while IFCC [49] passes all benches. The lack of dynamic linking does not impact related work here, as SPEC does not use it.

Real-world. TyPro successfully compiled all seven programs without introducing errors, as shown in Table 2 (i.e., no false negatives). Also, IFCC [49] passed all tests but with much larger target sets, which we address in Section 6.2. But Clang CFI failed on four programs (*lighttpd*, *nginx*, *pureftpd*, and *redis*). Its generalized mode resolved the errors in *nginx*, but not the remaining three. We checked all introduced problems by hand—dynamic linking caused none, and all were due to functions that do not precisely match call types. Clearly, Clang CFI is too restrictive; and any extension to support *vararg* would break Clang CFI's equivalence class model.

Unit Tests. To further test TyPro, we created a set of over 220 handcrafted unit tests, checking the correctness of TyPro's support for different aspects of the C language, triggering corner cases, and checking correct interaction with libc. With these tests and the various applications, we are confident that TyPro can handle any standard-conformant C program (within limits from Section 7).

6.2 Security

As detailed in Section 4, TyPro takes all indirect calls in a C program and converts them to a set of well-typed direct calls. Therefore, no indirect call instructions remain in the compiled program, making arbitrary jumps impossible. However, attackers with memory corruption capabilities can tamper with the function identifiers stored in memory. Even in this case, they cannot invoke arbitrary code—only the execution of a minimal and limited set of functions is possible. Furthermore, the generated code is inherently safe against concurrent modifications and potential time-of-check/time-of-use (TOCTOU) vulnerabilities for two reasons: First, during computation, no intermediate values are spilled onto the stack. The branching happens in registers that are inaccessible to attackers. Second, even if manipulation would be possible, there is no arbitrary call that an attacker could try to reach, only direct calls for each function in the target set. The only indirect jumps in the program come from the compiler itself, as introduced for jumptables. However, the compiler properly bound-checks these jumps, making them irrelevant for control flow security. Thus, our switch-based target enforcement indeed limits the surface for code-reuse attacks.

The same arguments hold for the target checks between dynamic modules. Both statically-compiled and JIT-compiled switch statements are safe on their own. They are connected by a pointer in read-only memory, which attackers cannot tamper with. Only when new switches are built during load time the pointer is temporarily made writeable. However, dynamic modules are often loaded at startup before any input is processed to minimize the risk further.

Finally, we aim to understand the security benefits of finding minimal call target sets. Smaller target sets leave the attacker with fewer choices and fewer gadgets. As suggested in related literature [7, 25], we report absolute numbers of possible targets for indirect calls. We rely on CSCAN [25]'s metric of computing the average number of targets per indirect call, considering only indirect calls that are reached during the execution of the program. Because CSCAN's approach is incompatible with our function identifier and direct call approach, and because CSCAN showed problems with different compiler optimizations, we implemented an equivalent computation for this metric, collecting target sets at compile-time and indirect calls at runtime. We evaluate TyPro and compare it to Clang CFI [47] in both normal and generalized mode, IFCC [49] and CFGuard's [28] policy, all with the same compiler version and optimization settings. We do not compare to Intel CET's indirect branch tracking [38], whose precision is likely worse than CFGuard. Because Clang CFI and IFCC do not support musl libc, we link all programs against an unprotected GNU libc for this comparison, in line with related work. Results with musl libc are similar.

SPEC. Table 1 reports the average number of targets for SPEC programs. Clang CFI builds the smallest target sets but breaks programs. In contrast, TyPro has slightly more targets (0.6% on geometric mean) but keeps all programs intact without modifications. Moreover, TyPro builds smaller target sets than Clang CFI's generalized mode, which still breaks programs at a worse precision. IFCC and CFGuard do not break anything but build larger target sets than TyPro, on average 2.5× / 4.8× the size. We also compare our results to MCFI [35], analyzed by CSCAN [25]. TyPro has a slightly better precision than MCFI, and its policy works even on unmodified programs. Note that CSCAN used a different compiler version and likely different, unknown optimization settings for their results. Furthermore, it is unclear if it uses patches to SPEC. For transparency, we also show CSCAN's results for Clang CFI, which vary from us by 1.8%. If linked with musl libc, the programs use 1–15 additional indirect calls from musl, with 2–17 average targets. On average, over all programs, calls inside musl have 7 targets.

Real-world. Table 2 reports the average number of targets on several real-world programs frequently used in related literature. Again, Clang CFI builds the smallest but often incorrect target sets. Clang's generalized mode builds 91% larger but still too narrow target sets. TyPro has more allowed targets on these larger programs than Clang (+102%, +6% more than generalized mode), but the target sets do not cause crashes. Compared to other correct solutions like IFCC or CFGuard, the number of possible targets has greatly reduced—TyPro has less than one-quarter of IFCC's targets and 4.8% of CFGuard's targets while still keeping programs intact.

Having said this, TyPro's target set computation is still an approximation working exclusively on the extracted type information. The algorithm can overapproximate the possible targets to prefer correctness and speed over precision. In specific cases, sophisticated attacks on really large programs like Control Jujutsu [12] or Control-Flow Bending [7] might still be possible, even in the presence of a perfect CFI policy. But from the experiments, we conclude that TyPro hits a sweet spot between correctness and security.

Table 1: Average number of call targets per indirect call on SPEC. TyPro compared to Clang CFI, IFCC, CFGuard and MCFI. ✗: benchmark fails (CFI too restrictive). ○: best security per benchmark. ●: best security among compatibility-preserving schemes.

	Clang CFI [47]	Clang CFI (generalized)	TyPro	IFCC [49]	CFGuard [28]	Clang CFI (data from [25])	MCFI [35] (data from [25])
400.perlbench	17.71 ○	51.99	22.32 ●	180.79	821.00	22.03	23.27 ✗
401.bzip2	1.00 ○	1.00 ○	1.00 ●	1.00 ●	2.00	1.00 ○	1.00 ○
403.gcc	9.19 ✗	34.44 ✗	24.98 ●	365.12	1192.00	8.91 ✗	32.63 ✗
433.milc	2.00 ○	2.00 ○	2.00 ●	2.00 ●	2.00 ●	2.00 ○	2.00 ○
445.gobmk	631.50	631.50	631.50 ●	749.12	1786.00	600.84 ○	605.51
456.hmmer	9.00 ✗	18.00 ✗	2.78 ●	19.00	19.00	10.00	10.00
458.sjeng	7.00 ○	7.00 ○	7.00 ●	7.00 ●	7.00 ●	7.00 ○	7.00 ○
464.h264ref	2.24	2.34	2.24 ●	10.95	42.00	2.06 ○	2.06 ○
482.sphinx3	5.00 ○	5.00 ○	5.00 ●	5.00 ●	5.00 ●	5.00 ○	5.00 ○
avg. %	(base)	+41.7%	+0.6%	+157.5%	+379.8%	+1.8%	+18.4%

Table 2: Average number of call targets per indirect call on various real-world programs.

	Clang CFI	Clang CFI (gen.)	TyPro	IFCC	CFGuard
httpd	14.69 ○	41.78	36.19 ●	462.59	2267.00
lighttpd	5.99 ✗	10.93 ✗	11.26 ●	50.32	257.00
memcached	1.99 ○	2.36	2.01 ●	14.88	85.00
nginx	16.53 ✗	56.08 ○	102.28 ●	240.72	758.00
pureftpd	1.00 ✗	1.00 ✗	1.00 ●	3.00	15.00
redis	10.03 ✗	44.19 ✗	48.06 ●	247.50	1136.00
vsftpd	3.33 ○	3.33 ○	3.33 ●	6.00	35.00
avg. %	(base)	+90.8%	+102.3%	+772.3%	+4102.6%

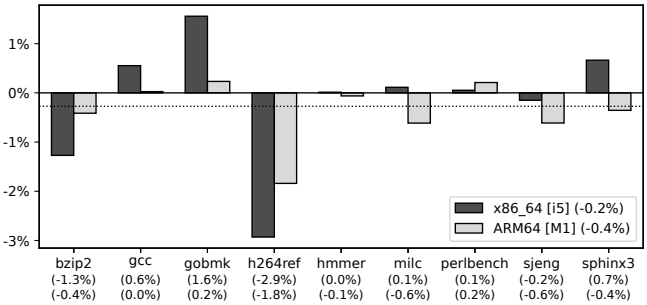

Figure 5: Runtime overhead of TyPro on SPEC 2006 benchmarks. Average overhead is −0.3%.

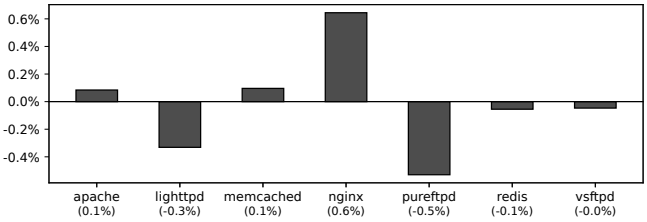

Figure 6: Runtime overhead on real-world applications.

6.3 Performance

To facilitate wide deployment, we now demonstrate that TyPro does not impose significant overheads on protected programs.

Performance Overhead. We used the SPEC CPU 2006 benchmark and our real-world servers to evaluate the performance impact. We measured the server's performance with ab [46], memaslap [57], redis-benchmark [40] and ftpbench [41]. Experiments ran on an Intel Core i5-4690, 32 GB RAM, and Debian 10. For ARM, we used an Apple M1, 16 GB Ram, and asahi Linux with kernel 5.17. We used the "performance" CPU governor, disabled CPU boost, and applied cpuset to minimize the impact of environment and operating system on the measurements. We repeated experiments at least 10×. The standard deviation on SPEC was at most 0.76%, and 0.2% on average.

We present our findings in Figure 5 and Figure 6. Protected programs get between 1.6% slower and 2.9% faster. The mean overhead is −0.3% on SPEC, i.e., programs get slightly faster, and zero on real-world applications. TyPro has a higher overhead in programs with large target sets (like *gobmk* with >600 valid targets, or nginx) and shows negative overhead in programs with small target sets (like *bzip* and *h264ref*). For many programs, the measured changes are within the standard deviation, and no real overhead is measurable.

Space Overhead. When comparing program size, we observed that the compiled binaries sometimes get larger. The generated switches and direct calls need more instructions than a single indirect call. Most programs are a few kilobytes larger after protection. In the worst case, gobmk gets 1.4 MB larger (40%). Figure 11 in Appendix E shows the additional binary size of all tested programs. On average, programs get 9.1% larger. We expect that this space demand does not prevent a CFI scheme from broader adoption, except if resources are scarce, such as for embedded systems.

6.4 Dynamic Loading

Dynamic loading support is an important feature for any CFI system such as TyPro. All SPEC programs can dynamically link against a protected standard library (musl libc). At runtime, computation and just-in-time compilation take at most 0.57s (perlbench), with 0.25s on average. Most SPEC benchmarks can avoid runtime computation completely because they do not exchange functions with libc.

redis nicely demonstrates TyPro's dynamic loading capabilities: it has a `MODULE LOAD` command that can load arbitrary shared objects at runtime. We tested this command on a protected redis instance with various protected modules and verified that they load and are usable. Even though redis had one of the largest fact sets in our tests, recomputing the target sets after a module load was a matter of seconds. Given that target sets can be cached and the JIT compiler's runtime is negligible, we consider this support and its performance practical.

7 LIMITATIONS & DISCUSSION

We now discuss the limitations of the current TyPro prototype.

First, the target set computation slows down compilation. Our non-parallelized current prototype finishes the analyses in a few seconds for small programs, in a few minutes for medium programs, and less than an hour for exceptionally complex programs like nginx. While we believe that this build time is not necessarily a deal-breaker in times when software is built by CI/CD servers, our prototype can still be improved for speed. Likewise, dynamic loading and re-computing target sets at runtime can take up to a few seconds. Assuming that the dynamically-loaded modules do not change frequently, TyPro could cache the computed target sets to speed up this process massively.

Second, like other CFI systems, our approach has limited compatibility with unprotected libraries. As soon as function pointers are exchanged with other libraries, these libraries have to be also protected. Usually, one could recompile these libraries with protection, but there might be cases where recompilation is impossible. In fact, TyPro already has semi-automated support for function pointers exchanged with unprotected libraries. But developers would have to mark functions imported from unprotected modules so that the compiler can instrument calls accordingly.

Third, TyPro does not yet support inline assembly or C++, which is sometimes combined with C code. We believe relevant use of inline assembly is rare, apart from standard libraries. Most inline assembly occurrences do not need analysis, e.g., the assembly in musl libc. While some parts of C++ are already supported (like lambda functions), it would be necessary to introduce a notion of "inheritance" and matching rules into our analysis. Next, C++ brings more types of indirect calls, e.g., virtual dispatch, which would also need additional rules.

8 RELATED WORK

TyPro positions itself in a wide range of existing CFI systems. Our evaluation has demonstrated that TyPro stands out as the most precise software-only forward CFI system that retains compatibility even with large programs. TyPro is fully open source and can protect any C software that LLVM can compile without requiring code modifications or special hardware support. Furthermore, we believe the proposed function type propagation is novel at the conceptual level. In the following, we will survey related work and briefly mention how TyPro differs from these proposals.

Software-Only Forward CFI. So far, Clang is the only compiler that has a strong forward CFI solution built-in. Clang CFI [47] checks a target function's type at indirect calls, preventing type-mismatched functions from being called. MCFI [35] is a similar protection with focus on dynamic loading support. πCFI [37] extends MCFI with runtime information; functions are not allowed to be targeted before they are referenced once at runtime. IFCC [49] checks not function types but argument numbers, providing compatibility with legacy programs at the price of much larger target sets. We have demonstrated that these schemes are either too restrictive and break programs or are too permissive. Finally, Microsoft Control Flow Guard [28] is an even weaker protection: an indirect call can target the start of any function.

Hardware-Assisted CFI. To accelerate CFI, researchers recently proposed hardware-assisted CFI schemes. In particular, Intel PT [20] was used in PT-CFI [16], PITTYPAT [10], CFIMON [53], μCFI [19], PathArmor [51] and GRIFFIN [15]. Furthermore, researchers proposed TSX-based CFI [33] and CFI-LB [23] based on Intel's transactional memory extensions, while OS-CFI [24] combines Intel MPX and TSX. CCFI [27] uses AES-NI instructions to perform cryptographic CFI checks. In contrast to TyPro, these systems require special hardware and potentially changes to the OS kernel, hindering the deployment of protected applications in many settings.

Binary-Only CFI. The protection of pre-compiled executables resolves the dependency from the source code. In particular, Opaque CFI [32], CCFIR [55], binCFI [56], Lockdown [39] and CFIMon [53] enforce a CFI scheme even if the source code is not available. In contrast to source-based schemes, these CFI solutions lack precise typing and flow information. They rely on approximative reconstructions, making them less precise than source-based approaches.

CFI for JITed Code. There are special-purpose CFI schemes targeting dynamically generated code. RockJIT [36], JITScope [54] and DCG [44] protect the execution of JIT-compiled code against control flow attacks. These schemes nicely extend TyPro because our current prototype only covers C code at compile time.

Backwards CFI. There is a whole range of schemes for *backward* CFI, which are out of scope for TyPro. For example, all major compilers inject stack canaries [11] during compilation. Shadow stacks provide an even stronger protection [8, 52], but their overhead hinders wide deployment. This situation might change soon: As part of CET [38, 43], Intel extends their processors with hardware support for shadow stacks, which reduces their performance penalty.

Type Analysis. Recently, Multi-Layer Type Analysis [26] (MLTA) has been proposed—a type analysis system designed to improve the precision of a simple base analysis (like the one used in Clang CFI). In contrast to TyPro, MLTA relies on a base analysis and can only reduce the computed target sets of this base analysis. Fixing errors coming from the inaccurate base analysis, our main contribution, is out of scope for MLTA. Older work [29] uses an inexpensive and imprecise analysis to compute a call graph for code browsing tools. Other recent tools [30, 50] use local type information and casts near memory allocations to determine types of heap objects, countering memory reuse and information leakage vulnerabilities.

9 CONCLUSION

We presented TyPro, a forward CFI scheme for C programs. TyPro protects indirect calls in legacy, real-world programs without requiring manual effort. Even multi-module programs, dynamically loaded at runtime, can be protected. TyPro's type-based approach has a precision comparable to state-of-the-art solutions used in production-grade compilers but does not underapproximate indirect call targets, leaving all protected programs intact. TyPro successfully targets the sweet spot between security and compatibility. On average, TyPro does not impose any performance overhead and only moderate binary size increase. Consequently, TyPro enables CFI deployment for legacy and modern real-world applications.

AVAILABILITY

https://github.com/typro-type-propagation/TyPro-CFI

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Markus Bauer and Christian Rossow. 2021. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking, In 6th IEEE European Symposium on Security and Privacy. *6th IEEE European Symposium on Security and Privacy*. https://publications.cispa.saarland/3364/

[3] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*. Springer, 24–51.

[4] Dimitar Bounov, Rami Gökhan Kici, and Sorin Lerner. 2016. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *NDSS*. The Internet Society.

[5] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1, Article 16 (apr 2017), 33 pages. https://doi.org/10.1145/3054924

[6] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *NDSS*.

[7] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 161–176. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[8] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (Singapore, Republic of Singapore) *(ASIA CCS '15)*. ACM, 555–566. https://doi.org/10.1145/2714576.2714635

[9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] Ren Ding, Chenxiong Qian, Chengyu Song, Bill Harris, Taesoo Kim, and Wenke Lee. 2017. Efficient Protection of Path-Sensitive Control Security. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 131–148. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding

[11] Hiroaki Etoh and Kunikazu Yoda. 2000. Protecting from stack smashing attacks. (01 2000).

[12] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 901–913. https://doi.org/10.1145/2810103.2813646

[13] Xiaokang Fan, Yulei Sui, Xiangke Liao, and Jingling Xue. 2017. Boosting the Precision of Virtual Call Integrity Protection with Partial Pointer Analysis for C++. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3092703.3092729

[14] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1075–1092. https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya

[15] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 585–598. https://doi.org/10.1145/3037697.3037716

[16] Yufei Gu, Qingchuan Zhao, Yinqian Zhang, and Zhiqiang Lin. 2017. PT-CFI: Transparent Backward-Edge Control Flow Violation Detection Using Intel Processor Trace. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy* (Scottsdale, Arizona, USA) *(CODASPY '17)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/3029806.3029830

[17] HardenedBSD. 2022. HardenedBSD - Introducing CFI. https://hardenedbsd.org/article/shawn-webb/2017-03-02/introducing-cfi

[18] Alfred Horn. 1951. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic* 16, 1 (1951), 14–21.

[19] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1470–1486. https://doi.org/10.1145/3243734.3243797

[20] Intel. 2021. Intel® Architecture Instruction Set Extensions and Future Features. https://www.intel.com/content/www/us/en/develop/download/intel-architecture-instruction-set-extensions-programming-reference.html

[21] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. 2014. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*. The Internet Society.

[22] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[23] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-Site Sensitive Control Flow Integrity. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 95–110. https://doi.org/10.1109/EuroSP.2019.00017

[24] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive Control Flow Integrity. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 195–211. https://www.usenix.org/conference/usenixsecurity19/presentation/khandaker

[25] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM, 1821–1835. https://doi.org/10.1145/3372297.3417867

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

[27] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 941–951. https://doi.org/10.1145/2810103.2813676

[28] Microsoft. 2022. Control Flow Guard - Win32 apps. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[29] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. 2004. Precise Call Graphs for C Programs with Function Pointers. *Automated Software Engineering* 11 (2004), 7–26.

[30] A. Milburn, E. van der Kouwe, and C. Giuffrida. 2022. Mitigating Information Leakage Vulnerabilities with Type-based Data Isolation. In *2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 259–275. https://doi.org/10.1109/SP46214.2022.00016

[31] M. R. Miller and K. D. Johnson. 2012. Using virtual table protections to prevent the exploitation of object corruption vulnerabilities. Patent number US 2012/0144480 A1.

[32] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *NDSS*.

[33] Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. 2016. Taming Transactions: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory. In *Symposium on Research in Attacks, Intrusion, and Defenses (RAID)* (Paris) *(RAID 16)*. Springer.

[34] musl authors. 2022. musl libc. https://musl.libc.org/

[35] Ben Niu and Gang Tan. 2014. Modular Control-Flow Integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 577–587. https://doi.org/10.1145/2594291.2594295

[36] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1317–1328. https://doi.org/10.1145/2660267.2660281

[37] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 914–926. https://doi.org/10.1145/2810103.2813644

[38] Baiju V Patel. 2020. A Technical Look at Intel's Control-flow Enforcement Technology. https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html

[39] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148* (Milan, Italy) *(DIMVA 2015)*. Springer-Verlag, Berlin, Heidelberg, 144–164. https://doi.org/10.1007/978-3-319-20550-2_8

[40] Redis Authors. 2022. Redis benchmark. https://redis.io/docs/reference/optimization/benchmarks/

[41] Giampaolo Rodola. 2016. pyftpdlib/ftpbench. https://github.com/giampaolo/pyftpdlib/blob/master/scripts/ftpbench

[42] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[43] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy* (Phoenix, AZ, USA) *(HASP '19)*. Association for Computing Machinery, New York, NY, USA, Article 8, 11 pages. https://doi.org/10.1145/3337167.3337175

[44] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *NDSS*.

[45] Standard Performance Evaluation Corporation. 2020. SPEC CPU® 2006. https://www.spec.org/cpu2006/

[46] The apache software foundation. 2022. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html

[47] The Clang Team. 2021. Control Flow Integrity - Clang 13 documentation. https://clang.llvm.org/docs/ControlFlowIntegrity.html

[48] The Clang Team. 2022. Introduction to the Clang AST - Clang 13 documentation. https://clang.llvm.org/docs/IntroductionToTheClangAST.html

[49] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 941–955.

[50] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 17–27. https://doi.org/10.1145/3274694.3274705

[51] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 927–940. https://doi.org/10.1145/2810103.2813673

[52] Vendicator. 2000. Stack Shield: A "stack smashing" technique protection tool for Linux. http://www.angelfire.com/sk/stackshield/

[53] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. 2012. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. https://doi.org/10.1109/DSN.2012.6263958

[54] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. 2015. JITScope: Protecting web users from control-flow hijacking attacks. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. 567–575. https://doi.org/10.1109/INFOCOM.2015.7218424

[55] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*. 559–573. https://doi.org/10.1109/SP.2013.44

[56] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 337–352. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[57] Mingqiang Zhuang and Brian Aker. 2022. memaslap - Load testing and benchmarking a server. http://docs.libmemcached.org/bin/memaslap.html

A COLLECTED FACTS IN EXAMPLE

Figure 7, Figure 8, and Figure 9 show the collected and derived facts for the example source code in Figure 1.

B DYNAMIC MODULES

In contrast to many prior CFI schemes [19, 23, 24, 33, 47], TyPro can handle code loaded at runtime: dynamically-linked libraries or runtime loading of shared libraries. To this end, we require that the loaded modules are also protected by TyPro. Furthermore, a generic runtime library must be present. It combines type information from

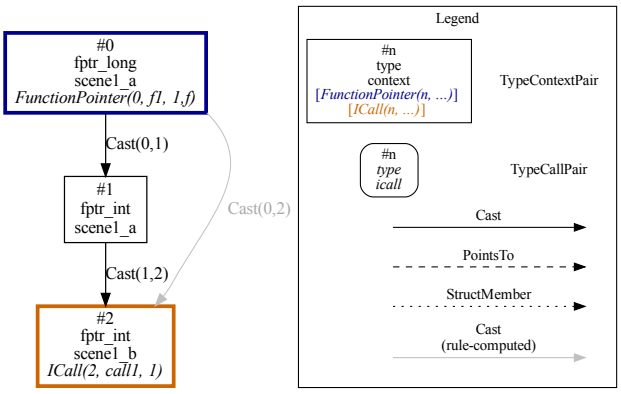

Figure 7: Graphical representation of the collected and derived facts for "scene1_a" and "scene1_b".

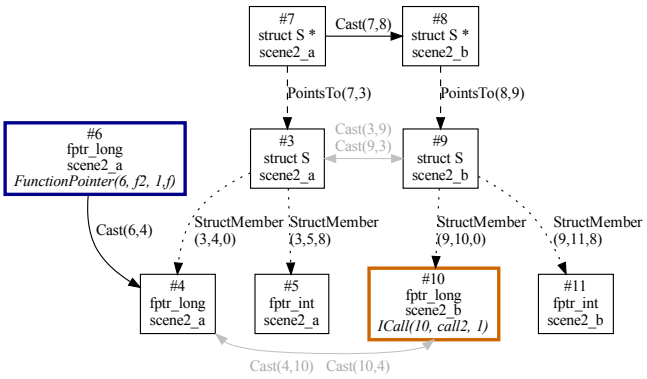

Figure 8: Graphical representation of the collected and derived facts for "scene2_a" and "scene2_b".

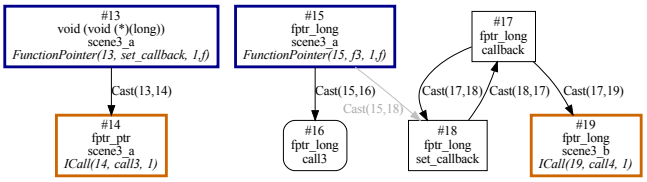

Figure 9: Graphical representation of the collected and derived facts for "scene3_a" and "scene3_b".

different modules at runtime, computing new target sets for indirect calls and updating the necessary checks. Thus, we have to export type information with every program and shared object. Therefore, we extend the target set analysis in Section 3 by a *module summary* which contains only the type/context propagation information that can influence the computation of other modules.

To compute a module summary, we add to our analysis the new relations and rules presented in Figure 10. The final result is established after computing the External relation, which intuitively contains all types that can be propagated to or from outside of the module. This relation is the index set of type/context and type/call pairs that must be exported in the module summary, if another

$$\begin{array}{ll} \text{ExternalSymbol} : \mathbb{S} & \text{(exported definitions)} \\ \text{InterfaceType} : \mathbb{S} \times \mathbb{N} & \text{(types per declaration)} \\ \text{External} : \mathbb{N} & \text{(pairs in module summary)} \end{array}$$

$$\text{ExternalSymbol}(f) \wedge \text{InterfaceType}(f, t_1) \implies \text{External}(t_1) \quad \text{(ES)}$$

$$\text{External}(t_1) \wedge \text{PointsTo}(t_1, t_2) \implies \text{External}(t_2) \quad \text{(E1)}$$

$$\text{External}(t_1) \wedge \text{StructMember}(t_1, t_2, \_) \implies \text{External}(t_2) \quad \text{(E2)}$$

$$\text{External}(t_1) \wedge \text{UnionMember}(t_1, t_2, \_) \implies \text{External}(t_2) \quad \text{(E3)}$$

$$\text{External}(t_1) \wedge \text{Cast}(t_1, t_2) \wedge \text{ICall}(t_2, \_, \_) \implies \text{External}(t_2) \quad \text{(EC)}$$

$$\text{External}(t_1) \wedge \text{ICall}(t_1, call, \_) \wedge \text{TypeCall}(t_2, \_, call) \implies \text{External}(t_2) \quad \text{(ECA)}$$

$$\text{FunctionPointer}(t_1, \_, \_, \_) \wedge \text{Cast}(t_1, t_2) \wedge \text{External}(t_2) \implies \text{External}(t_1) \quad \text{(EFP)}$$

$$\text{FunctionPointer}(t_1, f, \_, \_) \wedge \text{External}(t_1) \wedge \text{InterfaceType}(f, t_2) \implies \text{External}(t_2) \quad \text{(EFI)}$$

Figure 10: Additional predicate definitions for dynamic module support, and additional rules for module summaries.

module could contain the same pairs. For example, the summary includes the parameters of an exported function that generate the same type/context pair in every module it is imported to. The summary is the subset of all relations except TargetSet, containing *only* facts that refer to indices in the External relation.

B.1 Additional Input Generation

To determine which type/context pairs enter the module summary, we need additional facts collected from the source code, filling the relations ExternalSymbol and InterfaceType. They describe the C interface that a module exposes to other modules, containing similar information like header files in C. We iterate over all declared symbols of a module, including both imported and exported symbols, and record their interface types. If the symbol is a global g of type t, we record a fact $\text{InterfaceType}(g, N(t, g))$. If the symbol is a function f with signature $rt\ f(a_1, \ldots, a_m)$, we record facts $\text{InterfaceType}(f, N(a_i, f))$ and $\text{InterfaceType}(f, N(rt, f))$. If the symbol is visible after linking, we add $\text{ExternalSymbol}(f)$ or $\text{ExternalSymbol}(g)$.

B.2 Additional Type Analysis

Using additional input facts described in Appendix B.1, we compute the set of pair indices that must be visible to other modules. This information is expressed with External relation. It is computed using the rules shown in Figure 10 including the rule for External symbols (ES), a group of rules for type visibility ((E1), (E2), and (E3)), a group of rules for indirect calls' treatment (they are (EC) and (ECA)), and, finally, a group of rules for function pointers that should be referenced in the summary (namely, (EFP) and (EFI)). In the following, we explain these rules in more detail.

B.2.1 External Initialization. The interface of imported or exported symbols must be external because other modules can have the same symbols and, therefore, facts about the same pairs in their relations. We capture this property of the imported and exported symbols with the help of (ES) rule.

B.2.2 Type Visibility. When a type is visible externally, its structure is also visible. If an external type is a pointer type, rule (E1) marks its pointee type as external. For struct and union types, rules (E2) and (E3) respectively mark their fields as external.

B.2.3 Indirect Calls. If there is a type transfer between an external entry and an indirect call, this indirect call could receive function pointers from other modules. Rule (EC) marks this call as external, so we include its type/context pair in the module summary. In addition, rule (ECA) marks the corresponding TypeCall pair as external, including them in the module summary.

B.2.4 Function Pointers. Facts referencing a function pointer are considered external if there is a type transfer to any external fact (as captured by rule (EFP)) because the function pointer could be transferred to another module at runtime. The pointed function could become accessible at runtime, even if its symbol is not exported; therefore, we have to include its interface in the module summary: Rule (EFI) marks all arguments' types and the return value types as external.

After running the computation, External references the type/-context pairs necessary for runtime TargetSet computations. For the module summary, we filter the entire fact set to contain only those pairs. Exporting only filtered facts greatly reduces the file size of the summary and the runtime of target set computations during dynamic loading.

B.3 Dynamic Call Target Enforcement

Our approach and its enforcer also support dynamic linking, which requires additional processing. Using the information obtained by the analysis described in Appendix B.1, we can see at link time which target sets might need expansion later. If an indirect call has an associated index in External, there might be valid targets from other modules at runtime. When building the switch for such a call at link time, we do not add an error handler to its default case but add a new direct call, which we call the *trampoline*. While the trampoline target defaults to the error handler, the runtime library can overwrite it if necessary. If the target sets are amended at runtime, the trampoline target will point to a new switch, handling the additional targets.

At runtime, the third component of TyPro, the runtime library, updates the target sets of all external indirect calls if necessary. It loads the summaries of all modules (provided by the analysis discussed in Appendix B.1), combines them, and re-runs the computation from Section 3.2. If new targets appear in the target set of an indirect call, the runtime library just-in-time compiles a new switch-case statement with all new targets and modifies the trampoline to delegate to this new switch. If a module now calls a function with an identifier assigned by another module, the newly generated switch will dispatch the desired function, without allowing attackers to execute the arbitrary functions.

C C STANDARD LIBRARIES

As mentioned previously, our approach requires all dynamically linked libraries to go through the same processing as the program using it, i.e., the analyses discussed in Section 3 and Appendix B. However, many related works [19, 23, 24, 33, 47, 49] exclude the C standard library, for the following reasons: The standard GNU libc is not compilable by the Clang compiler, contains plenty of

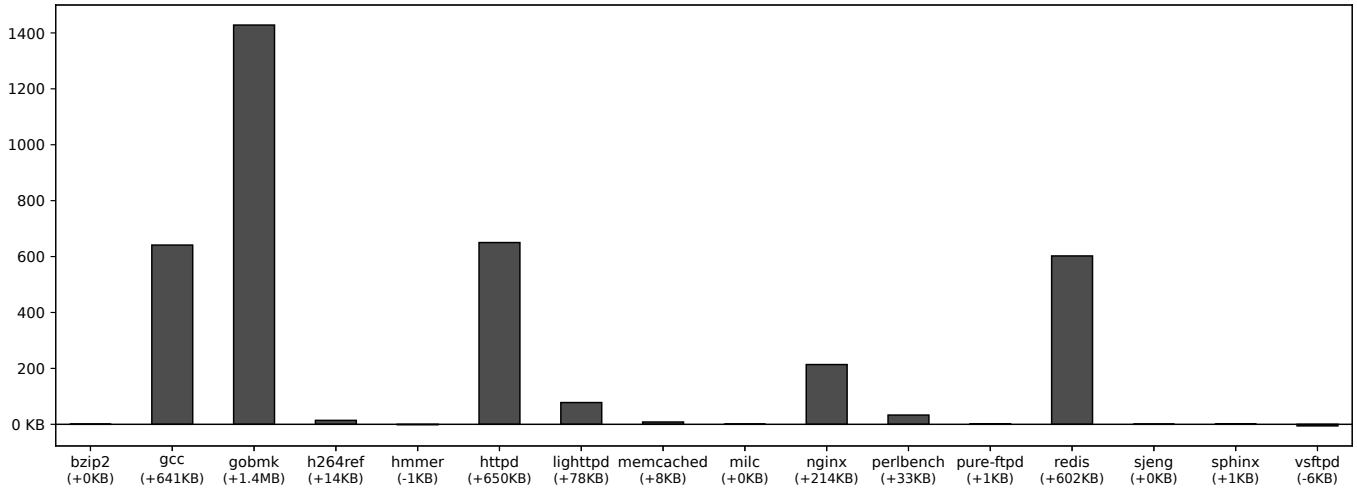

Figure 11: Additional size of SPEC and other example programs (in KB).

(typeless) inline assembly, and communicates with the Linux kernel over a syscall interface that cannot be altered.

Instead of excluding the standard library, TyPro uses and protects musl libc [34], which is compatible with Clang. Only when functions are sent to the kernel, TyPro resolves the identifiers back to function pointers before transferring them. A protected musl libc can be statically linked or used as a regular shared library.

Alternatively, programs relying on the GNU standard library can optionally link against an unprotected libc. In this case, TyPro resolves identifiers back to function pointers before transferring them to the libc, avoiding compatibility issues. To resolve a function identifier, a switch-case construct is emitted (similar to the one described in Section 4), returning actual function addresses instead of direct calls. With this method, TyPro-protected programs can use the system's unmodified standard library without breaking compatibility. This method could also be used to link with other *unprotected* libraries, assuming it is known *a priori* which library will be unprotected.

D OPTIMIZATIONS

After the facts' extraction, we perform some minor but crucial optimizations by omitting unnecessary facts for the final target computations. These optimizations do not change the result of the computation but are essential for reasonable performance. First, we only collect a TypeContextPair fact if its identifier is used in at least one another relation. If we omit a type, we also do not generate additional facts relevant to this type definition, e.g., PointsTo or StructMember facts. TypeContextPair facts unused in other relations cannot be used in any rule and are therefore irrelevant for target set computation. Second, we omit primitive C types, which are smaller (in bits) than a pointer. It is impossible to convert a function pointer to or from these types, nor can they participate in pointer aliasing or other rule-covered C structs; they are therefore irrelevant for target set computation. The most prominent example is the type `void`, which will never appear in our fact set, in contrast to `void*`, which has the same size as a function pointer and will appear in facts. Third, we collapse chains of direct casts, e.g., the expression "`(fptr_int) ((void*) &f1)`" will be seen as one cast from `fptr_long` to `fptr_int`.

Running the computation on fact sets of larger programs is very time-consuming, in particular when the facts of all input files are merged. Therefore, we use an optimization based on *equivalences* to reduce the input size for the datalog solver drastically. If two TypeContextPair facts are *equivalent*, these type/context pairs can be merged without changing the result of the target set computation, making the input fact set smaller and the computation faster. In our implementation, two simple patterns indicate equivalence:

- If we extracted facts $\mathsf{Cast}(n_1, n_2)$ and $\mathsf{Cast}(n_2, n_1)$, then type/-context pairs n_1 and n_2 can be merged.
- If we extracted $\mathsf{Cast}(n_1, n_2)$ facts and both type/context pairs n_1 and n_2 are pointer type (i.e., we extracted also $\mathsf{PointsTo}(n_1, n_1')$ and $\mathsf{PointsTo}(n_2, n_2')$ facts), then type/context pairs n_1' and n_2' can be merged.

Merging propagates along with the structure of its types: if two facts of pointer type merge, their referenced type's facts also merge. And if two facts of struct or union type merge, their field facts also merge. We found this optimization to improve the computation runtime considerably while not changing the result of the target set computation.

E SIZE OVERHEAD

Figure 11 shows the additional size of different programs after protection with TyPro for all SPEC and real-world example programs.