CFG Construction Soundness in Control-Flow Integrity

Gang Tan
The Pennsylvania State University
University Park, PA
gtan@cse.psu.edu

Trent Jaeger
The Pennsylvania State University
University Park, PA
tjaeger@cse.psu.edu

ABSTRACT
Control-Flow Integrity (CFI) is an intensively studied technique for hardening software security. It enforces a Control-Flow Graph (CFG) by inlining runtime checks into target programs. Many methods have been proposed to construct the enforced CFG, with different degrees of precision and sets of assumptions. However, past CFI work has not made attempt at justifying their CFG construction soundness using formal semantics and proofs. In this paper, we formalize the CFG construction in two major CFI systems, identify their assumptions, and prove their soundness; the soundness proof shows that their computed sets of targets for indirect calls are safe over-approximations.

KEYWORDS
Control-flow integrity; control-flow graphs; type systems; type soundness

1 INTRODUCTION
Control-Flow Integrity (CFI), proposed by Abadi et al. [1], is a specialized inlined reference monitor that enforces a pre-determined Control-Flow Graph (CFG) on a low-level program. The CFG is enforced through a combination of static and dynamic checks. Direct branches are checked statically since their targets are statically known. For indirect branches, dynamic checks are inlined to ensure that the runtime control flow follows the CFG. Indirect branches include indirect calls (i.e., function calls via register or memory operands), indirect jumps (i.e., jumps via register or memory operands), and return instructions.

CFI is important for software security since it mitigates control-flow hijacking attacks. Many software attacks hijack the control flow of programs, including stack-smashing attacks, return-to-libc attacks [8], and more recent code-use attacks such as ROP (Return-Oriented Programming [12]) attacks. By enforcing that a program’s control flow has to follow a CFG, many control-flow hijacking attacks can be effectively thwarted.

The security policy of CFI is the enforced CFG, which makes CFG construction a key problem in CFI. A program can have multiple CFGs, since a CFG is a static over-approximation of a program’s intra-procedural and inter-procedural control flow.\(^1\) There are two main considerations when constructing a program’s CFG: security and soundness. For security, a finer-grained CFG is desired in the sense that it has fewer control-flow edges; its enforcement in CFI gives attackers less room to manipulate the control flow. For instance, a CFG that allows an indirect call to target only a subset of functions is better for security than a CFG that allows the indirect call to target all functions. For soundness, a CFG needs to have all necessary control-flow edges so that its enforcement will not break the program’s legal execution.

Many methods (e.g., [5, 9–13, 15–18]) have been proposed for CFG construction in CFI, through binary-level analysis or through the help of a compiler; detailed discussion will be in the related-work section. CFG construction via the help of a compiler constructs finer-grained CFGs since it propagates extra information at the intermediate-representation level in the compiler to binary code and uses that information to compute a more precise set of targets for indirect branches.

It is relatively easy to argue the soundness of coarse-grained CFG construction. For example, certain binary-level systems allow an indirect call to target all functions [17, 18] and this is obviously sound. However, soundness is a serious concern for fine-grained CFG construction. Resolving targets of indirect branches in fine-grained CFG construction requires a special form of points-to analysis: for a code pointer used in an indirect branch, it determines what code addresses the code pointer can point to. For scalability, previous fine-grained CFI systems have proposed specialized, efficient CFG-construction methods, under some assumptions about the code. For instance, MCFL [10] uses a type-based approach and relies on the assumption that no indirect call via a function pointer invokes a function with a type signature different than the signature of the function pointer. While authors of these systems informally argued their CFG-construction soundness, they all lack formal soundness proofs.

The goal of this paper is to formalize the CFG construction and the underlying assumptions so that formal soundness can be stated and proved, for two main fine-grained CFI systems [5, 10]. The first system [10] relies on types, while the second [5] uses dataflow analysis (taint tracking) for CFG construction.

For formalization, we introduce a C-like imperative language and its formal semantics. It turns out that the formal semantics has to distinguish between memory-safe and memory-unsafe executions since CFG construction is sound only with respect to memory-safe executions. On top of the imperative language, we formalize the CFG construction and assumptions of the two CFI systems through type systems. The type systems, however, do not guarantee

\(^1\)In the CFI literature, a CFG specifies both intra-procedural and inter-procedural control flows; the program-analysis literature sometimes uses the term call graphs for inter-procedural control flows. We follow the CFI literature in this paper.
complete type safety, but only partial type safety. As we will see, the first one guarantees the safety of types that involve function-pointer types and the second one guarantees only the separation between function-pointer values and non-function-pointer values. For each type system, we will formalize partial type safety, which is sufficient to show the soundness of the corresponding CFG construction (in particular, an indirect call always stays in the constructed CFG).

The rest of the paper is organized as follows. We start with a discussion of related work. We then present a C-like imperative language in Sec. 3 and discuss its syntax and small-step operational semantics. Based on the language, Sec. 4 and Sec. 5 formalize CFG construction, the assumptions, and the soundness of a type-based approach and a taint-based approach, respectively. In Sec. 6, we discuss how violations of the assumptions in the two CFI systems can be resolved. We conclude in Sec. 7.

2 RELATED WORK

CFG construction has always been a key problem in CFI research. However, none of the existing CFI work provides a rigorous soundness proof for their CFG construction; the most some of the CFI work has done is to provide an informal argument for soundness together with some validation via testing. CFI construction roughly can be divided into two categories: the binary-analysis approach and the compiler-based approach. The binary-analysis approach analyzes binary code directly for CFG construction [9, 15, 17, 18]. Since binary code lacks structured information, the CFG determined by binary analysis is coarse grained. Among binary-analysis work so far, TypeArmor [15] provides the best precision; it performs liveness analysis to determine the arity (number of arguments) in an indirect call and uses arity matching to narrow down the set of indirect-branch targets. Another approach for CFG construction is to modify a compiler to propagate information such as types from source to binary code and uses the extra information for binary-level fine-grained CFG construction [5, 10, 11, 13, 16]. For instance, Forward-Edge CFI [13] relies on arity matching and MCFI [10] uses type signatures for pairing indirect calls and functions. This paper provides rigorous foundation for CFG construction in two recent fine-grained CFI systems.

CFG construction in CFI is a special kind of points-to analysis focusing on code pointers. General points-to analysis has been an active research area and many methods have been proposed in the past. Some algorithms consider the presence of C’s function pointers [4] and some consider how to resolve virtual method calls in object-oriented languages [2, 6, 14]. However, the soundness arguments in most points-to analysis papers are informal. Conway et al. [3] describe a formalization of pointer analysis. Similar to our work, their proof of soundness assumes memory-safe executions. One difference is that their formalization is based on abstract interpretation, while ours is based on type systems and uses standard progress and preservation proofs. Furthermore, their focus is on formalizing general points-to analysis such as Steensgaard’s algorithm and as a result their formalization does not contain functions nor function pointers; in contrast, dealing with functions and function pointers is central to any CFG construction in CFI. Finally, it is worth pointing out that formalizing the CFG soundness in CFI is perhaps more critical than formalizing general pointer-analysis soundness since CFI is a security mechanism.

3 THE LANGUAGE

We next introduce the formal syntax and operational semantics of an imperative language. The language serves as the basis for our formalization of CFG construction in two fine-grained CFI systems in Sec. 4 and Sec. 5.

The language includes the core features of C-like programs that are relevant to CFG construction for indirect branches after the programs have been compiled to an intermediate representation (such as the LLVM IR). It models pointers, direct and indirect function calls, stack and heap allocation, and includes typical types such as pointer and struct types. Since it models an intermediate representation, data stored on the stack and passed during function calls/returns have been flattened to values of atomic types (i.e., integer and pointer types) and code operations are expressed in terms of values of atomic types.

Another important point worth clarification is that the language and our formal CFG-construction soundness proofs in later sections focus on indirect calls. The CFI literature distinguishes forward edges by indirect calls/ jumps from backward edges by return instructions in CFGs. CFG construction in CFI is largely about forward-edge computation for the reason that backward edges can be computed once forward edges have been computed and also backward edges can be enforced via other mechanisms such as shadow stacks. Furthermore, the language we will introduce ignores indirect jumps since it models an intermediate language; an indirect call at this level is compiled to an indirect call in binary code, or an indirect jump in the case of a tail call.2

We next introduce some notation. We use $d$ for a sequence of $ds$; we use $e$ for the empty sequence and “$d_0; \ldots; d_i$” for a sequence whose head is $d_0$ and tail is $d_i$. We use $A \rightarrow B$ for a partial function from domain $A$ to $B$.

3.1 Syntax

Fig. 1 and Fig. 2 introduce the syntax of types and programs for the language. In types, we distinguish atomic types ($t$) from regular types ($r$). Values of an atomic type are of some unit size; without loss of generality, the unit size is assumed to be one. Values of a regular type may be of size greater than one and, when stored in memory, they are flattened to a sequence of values of atomic types.

In atomic types, we distinguish between data-pointer types and function-pointer types since, as we will see, the distinction is required by formalization of the assumptions used in fine-grained CFG construction. Syntax $r^+$ is for data-pointer values that point to data of type $r$, while “(t1 $\rightarrow$ t2) fptr” is for function-pointer values that point to functions that take $t1$ values and return $t2$ values. In C, “(t1 $\rightarrow$ t2) fptr” is written as “(t2)(t1)”. Without loss of generality, functions are assumed to take only one parameter.

Regular types include atomic types and struct types in the syntax of {i0 : t1, ..., in : tn}, which carries a list of pairs of ids and atomic types. We introduce a flatten (−) function. It maps a regular

---

1. In binary code, indirect jumps can also be used to implement jump-table based intra-procedural control flows: their targets, however, can be easily computed by the compiler when it generates jump tables.
The language omits intra-procedural control-variable, a dereference of an l-value, or a struct

```
s = (id1:int, id2:(int->int) fptr)* malloc(2);
```

```c
int y;

int bar ((id1:int, id2:(int->int) fptr)* s) {
  pointer in a struct.
}
```

expressions, or a type cast.

```c
ret x
```

Clearly, CFG construction should not capture control-flow edges

```c
that stores a function pointer and a subsequent memory update
```

via the "int*" pointer can change the function pointer to arbitrary

```c
values; then a use of the function pointer in an indirect call may
```

change the control flow in arbitrary ways. We call such executions

```c
memory-unsafe executions.
```

safety and temporal memory safety. In the operational semantics

```c
we will present, pointers are augmented with bounds information
```

that stores a function pointer and a subsequent memory update

```c
via the "int*" pointer can change the function pointer to arbitrary
```

values; then a use of the function pointer in an indirect call may

```c
change the control flow in arbitrary ways. We call such executions
```

memory-unsafe executions.

Clearly, CFG construction should not capture control-flow edges

```c
that occur only in memory-unsafe executions of the program. Therefore,
```

the soundness of CFG construction should be stated as follows:

```c
a memory-safe execution of the program always follows the constructed CFG. A memory-safe execution rules out the possibility
```

of overwriting memory via an out-of-bounds pointer. We stress that

```c
the restriction over memory-safe executions means that the constructed CFG does not contain control-flow edges that occur
```

only in memory-unsafe executions—such edges are not intended by

```c
programmers. After a CFG is constructed, a CFI system that enforces the CFG on the program execution does not necessarily need
```

to enforce memory safety; if an input triggers a memory error in

```c
the program and induces an edge outside the CFG, the CFI system
```

would raise an alarm.

Given the above discussion, it is necessary for the operational semantics to distinguish memory-unsafe executions from memory-

```c
safe ones. Memory safety can be decomposed into spatial memory
```

safety and temporal memory safety. In the operational semantics

```c
we will present, pointers are augmented with bounds information
```

so that spatial safety violations lead to an error state "merr". For simplicity, our language disallows memory deallocation so that its

```c
operational semantics can ignore temporal memory safety. Our
```

language could be augmented with memory deallocation and tem-

```c
poral safety could also be modeled in the operational semantics by
```

adding more metadata into the runtime state (e.g., following the SoftBound-CETS approach [7]).

Machine configurations. The operational semantics evaluates one machine configuration to the next machine configuration. Fig. 3

```c
presents the components in machine configurations.
```

A value can be an integer w, a pointer l(b, d), or the address of a function in the form of &f. In pointer l(b, d), l is a memory address

```c
and [b, d) is the address range of the buffer that l is supposed to be within. We define inbound(l, b, d) to hold when b \leq l \land l < d.
```

Because of pointer arithmetic, l could be an address outside the range of [b, d).
errors may occur; all the rules that result in memory errors are put

later). Also, we use a dedicated function name

does not appear in the program.

function environment \( \mathcal{F} \), which maps from function names to their
definitions and is initialized as follows:

\textbf{Definition 3.1 (Initial configuration).} For a program \( P \) of the form
\( f_d_1; \ldots; f_d_n : t_1 x_1; \ldots; t_j x_j; s_1; \ldots; s_k \), the initial configuration is
\( (m_0, \delta_0, \Lambda_0, s_1; \ldots; s_k) \) with respect to \( \mathcal{F} \), where \( \mathcal{F} \) maps function
names to their definitions according to \( f_d_1; \ldots; f_d_n \), and

\[ m_0 = \{ l_1 \mapsto \text{default}(t_1), \ldots, l_j \mapsto \text{default}(t_j) \}, \]
\[ \delta_0 = \{ x_1 \mapsto l_1, \ldots, x_j \mapsto l_j \}, \]
\[ \Lambda = \{ t_1 \mapsto t_1, \ldots, t_j \mapsto t_j \}. \]

The initial configuration initializes variables according to their
types in the following way:

\textbf{Definition 3.2 (Default values).}

\[
\text{default}(\text{int}) = 0
\]
\[
\text{default}(\tau\ast) = 0_{(0,0)}
\]
\[
\text{default}(t_1 \mapsto t_2) \text{ fptr} = \&\text{dummyFun}
\]

Note that \( 0_{(0,0)} \) is a pointer with the empty range; so the dereference
of it always leads to a memory error (the rule will be shown
later). Also, we use a dedicated function name \&dummyFun; its
described is used to initialize function-pointer variables; it is assumed
that \&dummyFun does not appear in the program.

\textbf{Evaluation rules.} Fig. 4 to Fig. 7 present the evaluation rules. At
a high level, we have rules for evaluating l-values, expressions, and
machine configurations. In all evaluation rules, we assume \( \sigma = (m, \delta, \Lambda) \) to simplify notation. During evaluation, memory
events may occur; all the rules that result in memory errors are put
into Fig. 7. For instance, a memory access via an out-of-bounds
pointer leads to memory errors.

Fig. 4 presents the rules for l-value evaluation in the form
\( \sigma, lv \mid \text{val} v : t \), meaning that \( lv \) evaluates to a value \( v \),
which stands for a memory location that holds \( t \) values. Evaluation
of variable \( x \) as an l-value returns a pointer value according to
the address of \( x \) in \( \delta \). Evaluation of \( \ast lv \) requires a memory read through
a pointer value and therefore requires the pointer be in bound.
Evaluation of \( \ast lv \mapsto \text{id}_i \) needs to consider the offset of \( \text{id}_i \) in
the flattened type list of a struct type.

Fig. 5 presents the rules for expression evaluation. Judgment
\( \sigma, e \mid \text{val} v : t \) means that \( e \) evaluates to value \( v \) of type \( t \). The rules
are straightforward and we only comment that it assumes integer
and pointer values can be freely converted back and forth using
to_int(\text{--}) and to_ptr(\text{--}) functions. When an integer is converted to
a pointer, the resulting pointer carries the empty range so that any
memory access via the pointer results in a memory error. When a
As an example, suppose variable numbers of arguments and structurally equivalent struct types. We ignore di
t3 non-function-pointer types, on the other hand, is not critical. For function of the same type. This allows CFG construction to use a 
that, MCFI maintains the 
direct call via a function pointer invokes a function with a type 
is 
extend国内外, " is allowed to invoke any function whose type 
Figure 6: Machine configuration evaluation.

4 CFG CONSTRUCTION FROM TYPE SIGNATURES

Modular CFI (MCFI [10]) adopts a type-based approach for CFG construction: an indirect call through a function pointer that is of type "(t1 → t2) fp" is allowed to invoke any function whose type is t1 → t2.3

The soundness of the type-based approach relies on that no indirect call via a function pointer invokes a function with a type signature different than the signature of the function pointer. For that, MCFI maintains the integrity of function-pointer types: a value of a function-pointer type must always contain the address of a function of the same type. This allows CFG construction to use a function pointer’s static type to compute what subset of functions an indirect call using that function pointer can target. Integrity of non-function pointers types, on the other hand, is not critical. For example, a value of type int* may well contain an integer and that would not affect the soundness of the type-based approach.

For maintaining partial type safety on function-pointer types, one assumption proposed by MCFI's authors for the input program was

A1: No type cast should involve function-pointer types.

That is, a value that is of a type that has function pointers inside cannot be cast to have a different type.

We argue there is another implicit assumption in MCFI:

A2: No pointer arithmetic or memory reads/writes through a function pointer are allowed.

If pointer arithmetic were allowed on a function pointer, the function pointer after pointer arithmetic would not be consistent with its type. A memory write through a function pointer would overwrite the code of the underlying function and invalidate any static guarantee on the code, including types. In theory, memory reads through function pointers would not invalidate the type-based approach; however, there are few reasons for reading code as data in legitimate programs and they are ruled out as well.

4.1 Formalizing MCFI’s Assumptions

To formalize assumption A1, we define the set of types that contain function pointers using the predicate has_fp"; it returns true if and only if t or τ contains a function-pointer type directly or indirectly. We abbreviate such types as has-fp types.
where

Figure 7: Memory-unsafe executions. Assume $\sigma = (m, \delta, \Lambda)$.

Definition 4.1 (has-fptr types).

\[
\begin{align*}
\text{has-fptr}(t) &= \begin{cases} 
\text{false} & \text{if } t = \text{int} \\
\text{true} & \text{if } t = (t_1 \rightarrow t_2) \text{ fptr} \\
\text{has-fptr}(r) & \text{if } t = \tau* \\
\text{has-fptr}(t_1) \land \ldots \land \text{has-fptr}(t_n) & \text{if } \tau = \{id_1 : t_1, \ldots, id_n : t_n\}
\end{cases}
\end{align*}
\]

We next present a type system that formalizes assumptions A1 and A2. The type system uses the following typing judgments, where $\Gamma$ is a type environment, mapping from variables to their types.

\[
\begin{align*}
\Gamma \vdash \text{lv} : t & \quad \text{L-value $\text{lv}$ holds $t$ values under $\Gamma$.} \\
\Gamma \vdash e : t & \quad \text{Expression $e$ has type $t$ under $\Gamma$.} \\
\Gamma \vdash s \text{ ok} & \quad \text{Statement $s$ is well typed under typing context $\Gamma$.} \\
\Gamma \vdash f \text{ ok} & \quad \text{Function declaration $f$ is well typed.} \\
\Gamma \vdash P \text{ ok} & \quad \text{Program $P$ is well typed.}
\end{align*}
\]

Figure 8 presents the fairly standard rules in the type system, except for a few cases. First, to formalize assumption A1, the rule for a type cast "($t$)" uses predicate allowed_cast($t'$, $t$), where $t'$ is the type of $e$. It is defined as follows:

\[
\text{allowed_cast}(t', t) = \neg \text{has-fptr}(t') \land \neg \text{has-fptr}(t)
\]

It makes sure that a type cast is allowed only if the original and the result types do not contain function-pointer types. Note that the restriction does not disallow other kinds of type casts, such
as from int to \( \tau^* \) or from \( \tau^* \) to \( \tau'^* \) when \( \tau \) and \( \tau' \) do not contain function-pointer types.

Second, rules for pointer arithmetic (\( e1 + e2 \)) and memory operations (via \( lv \)) take data-pointer types (\( \tau^* \)) and disallow function-pointer types; this is how assumption A2 is formalized. An expression of a function-pointer type can be used only in an indirect call or in assignments.

### 4.2 Soundness of MCFI’s CFG Construction

As explained before, soundness of type-based CFG construction relies on partial type safety for has-fptr types. That is, if the type system statically determines that an expression is of a type that contains a function-pointer type, then at runtime it must evaluate to a value that has the type. The type system, on the other hand, makes no guarantee for types that do not contain function-pointer types. Therefore, if an expression’s static type is \( t \), there is no guarantee that the runtime value that the expression evaluates to actually has type \( t \).

The soundness proof needs to capture the aforementioned invariant formally. For that, we introduce a predicate \( value_inv(\Lambda, \nu, t) \), which models the invariant that holds on a value \( \nu \), a type \( t \) (which is the type of an expression that evaluates to \( \nu \)), and a memory type \( \Lambda \); recall \( \Lambda \) is a map from locations to their declared types during allocation.

**Definition 4.2 (Value invariants for the type-based method)**

\[
value_inv(\Lambda, \nu, t) \quad \text{holds when one of the following cases holds:} \\
\begin{align*}
(1) & \quad \text{if } \nu = w, \text{ then } t = \text{int}, \text{ or } t = \tau^* \text{ for some } \tau \text{ and } \neg \text{has_fptr}(\tau); \\
(2) & \quad \text{if } \nu = l_{b,d}, \text{ and has_fptr}(t), \text{ then } t = \tau^* \text{ for some } \tau \text{ and } \\
& \quad \Lambda(l_{b,d}) = \text{extend}(\text{flatten}(\tau), d - b). \\
(3) & \quad \text{if } \nu = l_{b,d} \text{ and } \neg \text{has_fptr}(t), \text{ then for all } l' \in [b, d), \text{ we have } \\
& \quad \neg \text{has_fptr}(\Lambda(l')). \\
(4) & \quad \text{if } \nu = \&f, \text{ then } \mathcal{F}(f) = t_2 f (t_1 x_1) \ldots \text{ and } t = (t_1 \rightarrow t_2) \text{ fptr;} \\
(5) & \quad \text{if } \nu = \&\text{dummyFun}, \text{ then } t = (t_1 \rightarrow t_2) \text{ fptr for some } t_1 \text{ and } t_2.
\end{align*}
\]

We next explain \( value_inv(\Lambda, \nu, t) \) by cases.

- **Case 1:** When \( \nu \) is an integer \( w \), the static type can either be int, or some \( \tau^* \) because of type casts. For instance, \( "(\text{int}+4)" \) is of static type int*, but the runtime value is an integer. Note the invariant implicitly says that an integer can never have a function-pointer type.
- **Case 2:** The two cases for \( l_{b,d} \) capture the invariant maintained on data-pointer values. When \( \text{has_fptr}(t) \), then \( t = \tau^* \) for some \( \tau \) and the types of the locations of the underlying buffer must be consistent with the flattened type list of \( \tau \) in the memory type \( \Lambda \). Recall that extend(flatten(\( \tau \)), \( d - b \)) returns a type list of width \( d - b \) by repeating the pattern in flatten(\( \tau \)). We use the notation \( \Lambda(l_{b,d}) \) for the type list in \( \Lambda \) in the range of \( [b, d) \). This case enforces partial type safety on has-fptr types.

On the other hand, when \( \neg \text{has_fptr}(t) \), the locations of the underlying buffer may not be consistent with \( \tau \) because of type casts. For example, suppose \( l_{b,d} \) points to a buffer whose locations have type int in \( \Lambda \); initially \( l_{b,d} \) has type int*, but is then cast to have type int**; then the new type is inconsistent with the types in \( \Lambda \). In this case, we require that

\[
\begin{array}{c|c|c|c}
\text{wf}_{\text{state}}(\sigma) & \Gamma(\sigma) \vdash s \ \text{ok} & \text{wf}_{\text{conf}}(\sigma, c) \\
\text{wf}_{\text{conf}}(\sigma, e) & & \text{wf}_{\text{conf}}(\sigma, s; c) \\
\sigma = (m, \delta_0, \delta, \Lambda) & \text{wf}_{\text{state}}(\sigma) & \\
\Gamma(\sigma) \vdash e : t & \sigma_1 = (m, \delta, \Lambda) & \\
\Gamma(\sigma_1) \vdash_{\text{eval}} lv : t & \text{wf}_{\text{conf}}(\sigma_1, c) & \\
\text{wf}_{\text{conf}}(\sigma, lv = \text{ret } e; c) & & \\
\end{array}
\]

**Figure 9: Well-formed configurations.**

The types of the locations in \( \Lambda \) do not contain has-fptr types. This implies a level of separation between data pointers and function pointers: a data pointer that is of a non-has-fptr type cannot point to a buffer whose locations are of has-fptr types. If this were violated, it would be possible to modify function pointers in the buffer via the data pointer, destroying partial type safety on has-fptr types.

Note the two cases for pointer values allow out-of-bounds pointers: \( l_{b,d} \) can be of type \( \tau^* \) even if \( l \) is out of bound (but a memory error would result if such pointers were dereferenced). As a special case, \( l_{b,d} \) is of any type \( \tau^* \) when \( [b, d) \) is the empty range.

- **Case 3:** Finally, when \( \nu \) is the address of a function, the type must be exactly the same as the function’s declared type.

We say a state \( (m, \delta, \Lambda) \) is well formed if (1) the value stored in a memory location is consistent with the corresponding type in \( \Lambda \) according to value.inv(\( \neg\neg\neg\neg \neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg \)), and (2) the domains of \( m \) and \( \Lambda \) are the same and are supersets of the addresses in \( \delta \).

**Definition 4.3 (Well-formed states).** \( \text{wf}_{\text{state}}(m, \delta, \Lambda) \) holds if

\[
\begin{align*}
(1) & \quad \forall l \in \text{dom}(\Lambda), \text{value.inv}(\Lambda, m(l), \Lambda(l)); \\
(2) & \quad \text{dom}(m) = \text{dom}(\Lambda) \supseteq \text{rng}(\delta), \text{ where } \text{rng}(\delta) \text{ is the union of the ranges of stack frames in } \delta. \text{ The range of a single stack frame } \delta \text{ is defined to be the set } \{ \delta(x) | x \in \text{dom}(\delta)\}.
\end{align*}
\]

To show the standard progress and preservation for partial type safety, we first introduce well-formed configurations, defined in Fig. 9. With all the definitions, we can state the standard progress and preservation theorems.

**Theorem 4.4 (Preservation).** If \( (\sigma, c) \rightarrow (\sigma', c') \), and \( \text{wf}_{\text{conf}}(\sigma, c) \), then \( \text{wf}_{\text{conf}}(\sigma', c') \).

**Theorem 4.5 (Progress).** If \( \text{wf}_{\text{conf}}(\sigma, c) \), then either exist \( \sigma' \) and \( c' \) so that \( (\sigma, c) \rightarrow (\sigma', c') \), or \( (\sigma, c) \rightarrow \text{merr}, \text{ or } c = e \) (the final state).

Preservation is proved by case analysis over \( (\sigma, c) \rightarrow (\sigma', c') \) and progress is proved by case analysis over \( \text{wf}_{\text{conf}}(\sigma, c) \). In the appendix, we present the major lemmas required for proving progress and preservation. Note that our type progress and preservation are with respect to the definition of well-formed states, which requires only \( \text{value.inv}(\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg\neg) \) for partial type safety.

**Corollary 4.6 (Soundness of type-based CFG construction).** When a well-typed program (i.e., \( \vdash P \text{ ok} \)) executes an indirect call
We note this assumption still allows a data pointer to a struct that was discussed in the previous section: assumption A1 of the taint tracking process is performed on individual functions, so for a function, say foo, which is a function pointer that may point to foo; when it is of type "(t1 → t2) fptr", it is a function pointer that cannot point to foo. Adding taint tags to types is appropriate since the taint-based method by Ge et al. [5] is flow insensitive. For a flow-sensitive taint tracking algorithm, types can be still be used to track taints after programs have been compiled to the static-single-assignment form.

The next definition formalizes the set of types that disallow direct data pointers to function pointers. From this point on, we will use t for only those atomic types for which no_dp_to_fp(t) holds, similarly for regular types (r).

Definition 5.1 (Types with no direct data pointers to function pointers).

\[
\text{no}_\text{dp}_\text{to}_\text{fp}(t) = \begin{cases} 
\text{true, if } t = \text{int} \\
\tau 
eq ((t_1 \rightarrow t_2) fptr) \land \text{no}_\text{dp}_\text{to}_\text{fp}(\tau) \\
\text{false, otherwise} 
\end{cases}
\]

\[
\text{no}_\text{dp}_\text{to}_\text{fp}(\tau) = \begin{cases} 
\text{true, if } \tau = t \\
\text{false, otherwise} 
\end{cases}
\]

Fig. 10 presents judgment "allowed_cast(t1, t2) \sim \Psi", which tells what kinds of type casts are allowed. In the judgment, \( \Psi \) is a set of constraints on taint tags that should be satisfied. The first three rules are essentially the same as the restriction on type casts in the previous section: type casts are allowed when no function-pointer types are involved. The last rule further allows a function-pointer type to be cast to another different function-pointer type, as long as their taint tags are the same. Note there is no need to worry
about cases such as when \( t_1 = (t_1 \rightarrow t_2) \) since such types are invalid by assumption A2.

Furthermore, we note that there are two views on judgments that produce taint-tag constraints, including "allowed_cast\((t_1, t_2) \Rightarrow \Psi". First, when taint tags on function-pointer types are constants (T or F), constraints serve as checks that enforce certain rules. Another view is to first decorate function-pointer types with \( \tilde{t} \) and use those judgments to produce a set of constraints on those taint variables; a solution to those constraints produces checkable assignments from taint variables to constants.

Fig. 11 presents a type and constraint-generation system for modeling the taint-based method for CFG construction. Most judgments produce constraints on taint tags to model taint tracking. For instance, the rule for \( \Gamma \vdash t \vdash \Psi \) when \( e = \&f \) requires the taint tag of the resulting function pointer be T; when \( e = \&f \) and \( f \neq \text{foo} \), the taint tag should be F. The judgment \( \Gamma \vdash x : t \vdash \Psi \) models what kind of constraints should be generated when a value of \( t_1 \) flows to a value of \( t_2 \). The only substantial case is when \( t_1 \) and \( t_2 \) are function-pointer types; the generated constraint says that, when \( t_1 \) is tainted, then \( t_2 \) must be tainted, reflecting how taint is propagated through function-pointer assignments. Same as the last section, the type system allows pointer arithmetic and memory operations only on values of data-pointer types, not function-pointer types, formalizing assumption A3.

For the soundness proof, we define value_inv\((\Lambda, \nu, \tau)\) for modeling the runtime invariant of value \( \nu \) and its static type \( \tau \) under memory type \( \Lambda \) in the taint-based method. The only difference from the corresponding version for the type-based method is the case when \( \nu = \&f \); it requires the taint tag must be T when \( f \) is foo (i.e., the function being taint tracked).

Definition 5.2 (Value invariants for the taint-based method).

\( \text{value}_\text{inv}(\Lambda, \nu, \tau) \) holds when one of the following cases holds:

1. if \( \nu = \text{w} \), then \( t = \text{int} \) or \( t = \text{r} \) for some \( r \) and \( \neg \text{has_fptr}(r) \);
2. if \( \nu = l(b, d) \) and \( \text{has_fptr}(t) \), then \( t = \text{r} \) for some \( r \) and \( \Lambda[l(b, d)] = \text{extend}(\text{flatten}(r), d - b) \);
3. if \( \nu = l(b, d) \) and \( \neg \text{has_fptr}(t) \), then for all \( l' \in [b, d] \), we have \( \neg \text{has_fptr}(\Lambda[l']) \).
(4) if \( v = f_ptr \), then \( F(f) = t_2 \ f_ptr(t_1 \ x_1)[\ldots] \) and \( t = (t_1 \ f_ptr[t_2] \ f_ptr \ and \ if \ f = \text{foo}, \ then \ b = T. \)

(5) if \( v = \&\text{dummyFun} \), then \( t = (t_1 \rightarrow t_2) \ f_ptr \) for some \( t_1 \) and \( t_2 \).

With the new runtime invariant, we can similarly define well-formed states and configurations and show the progress and preservation theorems, which lead to the following soundness corollary.

Corollary 5.3 (Soundness of Taint-based CFG Construction). Let \( P \) be a program in which all taint tags are taint variables and it is well typed by \( P \sim \Psi \). Let \( \eta \) be a solution of \( \Psi \) and it assigns taint variables to constants \((T \ or \ F)\). Let \( \eta(P) \) be the program with all taint variables replaced by constants according to \( \eta \). Then when \( \eta(P) \) executes an indirect call \( "lv = \text{icall e(e1)}" \) in a memory-safe environment and \( e \)'s type is \( "(t_1 \rightarrow t_2) \ f_ptr" \) in its type environment, the indirect call cannot invoke the function foo.

Proof. For a well-typed program \( P \), it is easy to show that the initial configuration is well-formed. When it gets to a configuration \( \kappa = (\sigma, lv = \text{icall e(e1)}, c) \), we must have that \( \kappa \) is well-formed thanks to the preservation theorem. Therefore, \( e \) is of type \( "(t_1 \rightarrow t_2) \ f_ptr" \) in configuration \( \kappa \). Thanks to the progress theorem, if the next state is not more, then we must have \( (\sigma, e) \downarrow & f : (t_1 \rightarrow t_2) \ f_ptr \) for some \( f \), which in turn gives us value-inv \( (\sigma, A, & f, (t_1 \rightarrow t_2) \ f_ptr) \). By definition of value invariants, \( f \) cannot be foo.

By the corollary, only if \( e \) is of type \( "(t_1 \rightarrow t_2) \ f_ptr" \) in \( "lv = \text{icall e(e1)}" \), we need to add foo to the possible target list of the indirect call. By repeating taint tracking for all functions, the target list for the indirect call can be completely decided.

6 FIXING ASSUMPTION VIOLATIONS

Each of the previous two approaches for sound CFG construction makes a set of assumptions on input programs. Clearly, not all software satisfy those assumptions. In this section, we briefly discuss possible resolutions when those assumptions are violated.

The type-based approach assumes no type cast should involve function-pointer types (assumption A1). MCFI \[10\] includes a static checker that reports violations of this assumption to programmers. It further classifies violations into the kind that does not lead to unsound CFGs and the kind that requires code fixes for sound CFG construction. On SPEC2006 benchmarks, it was shown only a few code changes were required to fix the second kind of violations, mostly by adding function wrappers. For instance, in the gcc benchmark of SPEC2006, there is a generic key-comparison function pointer typed \( \text{int } \ 1 \text{ (unsigned long, unsigned long)} \). In two places, the function pointer is set to be the address of \text{strcmp}, whose type is \( \text{int } \ 1 \text{ (const char*, const char*)} \). Since the function pointer's type is incompatible with \text{strcmp}'s, MCFI's CFG generation does not connect the function pointer to \text{strcmp}. To fix the problem, a wrapper function was added that has the equivalent type as the type of \text{strcmp} and makes a direct call to \text{strcmp}. The key-comparison function pointer is then set to be the address of the wrapper function. MCFI does not report violations of assumption A2 (no pointer arithmetic or memory reads/writes through a function pointer), but it is expected that such violations are rare.

Assumptions A2 and A3 of the taint-based approach were found to hold largely for OS kernel software \[5\], where seven violations were found for the default FreeBSD kernel configuration, and these violations were fixed manually by changing the source code. The authors did not report violations of assumption A1 (function-pointer types may not be cast to or cast from non-function-pointer types), which would require more code changes. However, this did not impact sound CFG generation for the kernel software Ge et al. inspected \[5\]. Since assumption A1 on the type casts in the taint-based approach is less strict than the type-cast assumption in the type-based approach, it would involve smaller code changes for fixing the type-cast violations in the taint-based approach.

As future work, we plan to study sound relaxations of those assumptions so that fewer violations and fixes will be required for CFG construction. It would also be interesting to combine CFG-construction techniques, including the type-based approach and the taint-based approach, both for enhancing CFG precision and for relaxation of assumptions.

7 CONCLUSIONS

As attackers always aim to find weak points in software systems, solutions that enhance software security should be built on a rigorous foundation of formal semantics and proofs so that precise claims can be made on those solutions. We formalize the soundness of CFG construction in two major CFI systems, using a standard framework of type soundness and a weakened notion of type safety. We believe the same framework can be used to show CFG-construction soundness of other CFI systems and all future CFI systems should be treated with the same level of rigor.

8 ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments, which helped us substantially improve the paper. This research is based upon work supported by US NSF grants CNS-1624126, CNS-1408880, CCF-1624124, the Defense Advanced Research Projects Agency (DARPA) under agreement number N66001-13-2-4040, and Office of Naval Research Grant N00014-17-1-2498. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

REFERENCES


We next sketch the major lemmas and proofs for the soundness

**Proof.** and uses a similar set of lemmas, but with a different deﬁnition of valueinv(−, −, −).

The following lemma is easily proved by using the definition of valueinv(−, −, −).

**Lemma A.1.**

1. If ¬has_fptr(t) and ¬has_fptr(t'), then valueinv(Λ, v, t) is equivalent to valueinv(Λ, v, t').
2. If (b, d) is the empty range, then valueinv(Λ, b, d, ▷, ▷, ▷) holds for any r.
3. If valueinv(Λ, b, d, ▷, ▷), then valueinv(Λ, b, d, ▷, ▷, ▷) for any l'.
4. If valueinv(Λ, v, t') and allowed_cast(t', t), then valueinv(Λ, v, t).
5. If valueinv(Λ, v, t) and Λ ⊆ Λ', then valueinv(Λ', v, t).

**Lemma A.2 (Reads preserve invariants).** For Σ = (m, Λ, Λ), if valueinv(Λ, b, d, ▷, ▷, ▷) and inbound(l, b, d), then valueinv(Λ, m(l), t).

Proof. (a) Assume has_fptr(t). We can get Λ(l) = t, because valueinv(Λ, m(l), ▷, ▷, ▷). From valueinv(Λ, m(l), m(l), ▷, ▷), which gives us the goal valueinv(Λ, m(l), t).

(b) Assume ¬has_fptr(t). Then we get ¬has_fptr(m(l)), because valueinv(Λ, m(l), ▷, ▷, ▷) and inbound(l, b, d). From valueinv(Λ, m(l), ▷, ▷, ▷), and valueinv(Λ, m(l), ▷, ▷, ▷), which is equivalent to our goal valueinv(Λ, m(l), t), by part (1) of Lemma A.1.

**Lemma A.3.**

**Proof.** Let m' = m[l → v]. Since, dom(m) = dom(Λ) ⊇ rng(δ), we easily get dom(m') = dom(Λ) ⊇ rng(δ).

Next we show for all l' ∈ dom(Λ), valueinv(Λ, m(l'), ▷, ▷, ▷), when l' = l, we get the result from valueinv(Λ, m(l), ▷, ▷, ▷), this is proved by two cases.

1. Assume has_fptr(t). From valueinv(Λ, b, d, ▷, ▷, ▷), then inbound(l, b, d), and then valueinv(Λ, v, t), and valueinv(Λ, v, t).
2. Assume ¬has_fptr(t). Then we get ¬has_fptr(m(l)), because valueinv(Λ, m(l), ▷, ▷, ▷), and inbound(l, b, d). Then the assumption valueinv(Λ, v, t), is logically equivalent to goal valueinv(Λ, v, t), by part (1) of Lemma A.1.

From a state σ, we can construct a type environment Γ(σ) from the top static frame and the memory type in σ.

**Definition A.4.** Let σ = (m, Λ, Λ, Λ), Define Γ(σ) = λx. Λ(δ0(x)).

**Lemma A.5.**

1. If Γ(σ) lval lv : t, then lval lv : t, and lval lv : t', then t = t'.
2. If Γ(σ) lval lv : t, and lval lv : t, and lval lv : t, then valueinv(Λ, v, ▷, ▷).
3. If Γ(σ) lval lv : t, and valueinv(Λ, v, ▷, ▷), then either exists l so that lval lv : t or lval lv : lval lv.

The proof is by induction over the derivation of Γ(σ) lval lv : t.

The only diﬃcult cases lie in the proof of part (2). The case for lval lv : t is proved using Lemma A.2. The case for lval lv : t is proved by case analysis over whether the struct type of lv is a has_fptr type and case analysis over the shape of the value stored at location lv in memory; a further assumption needs to made to flatten(−): when ¬has_fptr(t), any type τ in ﬂatten(τ) must satisfy ¬has_fptr(t).

**Lemma A.6 (Lemma about expressions).** Let σ = (m, Λ, Λ).

1. If Γ(σ) e : t, and lval lv : t, then lval lv : t', then t = t'.
2. If Γ(σ) e : t, and lval lv : t, then valueinv(Λ, v, ▷, ▷, ▷).
3. If Γ(σ) e : t, and lval lv : t, then either exists l so that lval lv : t or lval lv.

The proof of the above lemma is by straightforward induction over the derivation of Γ(σ) e : t.

The preservation theorem is then proved by case analysis over whether the type of c is a has_fptr type, then using Lemmas A.5 and A.6 and the following lemma. The progress theorem is then proved by case analysis over valueinv(σ, c), using Lemmas A.5 and A.6.

**Lemma A.7.** If lval lv : t, and lval lv : t, and lval lv : t, then lval lv : t, valueinv(σ, c), and lval lv : t, and valueinv(σ, c), then valueinv(σ, c).