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, 9–11, 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, MCFI [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.

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/10.1145/3139337.3139339

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 $\overline{d}$ for a sequence of ds; we use ϵ for the empty sequence and "$d_0; \overline{d_1}$" for a sequence whose head is d_0 and tail is $\overline{d_1}$. We use $A \rightharpoonup B$ 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 (τ). 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 $\tau*$ is for data-pointer values that point to data of type τ, while "$(t_1 \to t_2)$ fptr" is for function-pointer values that point to functions that take t_1 values and return t_2 values. In C, "$(t_1 \to t_2)$ fptr" is written as "$(t_2)(*)(t_1)$". Without loss of generality, functions are assumed to take only one parameter.

Regular types include atomic types and struct types in the syntax of $\{\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n\}$, which carries a list of pairs of ids and atomic types. We introduce a flatten($-$) function. It maps a regular

[2] 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.

$$
\begin{array}{llll}
\textit{AtomicType} & t & := & \text{int} \mid \tau* \mid (t_1 \to t_2)\ \text{fptr} \\
\textit{Type} & \tau & := & t \mid \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\}
\end{array}
$$

Figure 1: The syntax of types.

$$
\begin{array}{llll}
\textit{Program} & P & ::= & \overline{fd}; \overline{t\ x}; \overline{s} \\
\textit{Function} & fd & ::= & t_2\ f\ (t_1\ x_1)\{\overline{t\ x};\ \overline{s};\ \text{ret}\ e\} \\
\textit{Stmt} & s & ::= & lv = e \mid lv = (\tau*)\ \text{malloc}(e) \\
 & & & \mid lv = \text{call}\ f(e) \mid lv_1 = \text{icall}\ e(e_1) \\
\textit{LVal} & lv & ::= & x \mid *lv \mid lv\text{->id} \\
\textit{Exp} & e & := & w \mid lv \mid \&lv \mid \&f \mid e_1 + e_2 \mid (t)e
\end{array}
$$

Figure 2: The syntax of an imperative language.

type τ to an ordered atomic-type list, which tells how values of type τ are stored in memory. To make our model general, we do not use a particular flatten function, but assume the following properties:

- flatten(τ) is the singleton list containing τ, when τ is an atomic type.
- When $\tau = \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\}$, flatten($\tau$) is a list of atomic types so that, for any i, field id_i's type t_i is included in the list; we define the index of t_i in the list to be offset(id_i, τ).

We further define size(τ) to be the length of flatten(τ).

The syntax for the language is presented in Fig. 2. A program is a sequence of function declarations, followed by a sequence of global variable declarations, and a sequence of global statements (these statements would occur in the main function of the corresponding C program). The body of a function declaration fd can have a sequence of local variable declarations, a sequence of statements, and a return at the end. Note that all variables and function parameters must be of atomic types since values of struct types have already been flattened in this language.

A statement can be an assignment, a heap allocation, a direct call on a function name f, or an indirect call (icall) via an expression. The language omits intra-procedural control-flow statements such as if-statements for the reason that CFG constructions in the two fine-grained CFI systems we examine are flow insensitive.

As in C, an l-value stands for a location in memory. It can be a variable, a dereference of an l-value, or a struct field. Expressions are pure. An expression can be a constant word w, an l-value, the address of an l-value, the address of a function, the addition of two expressions, or a type cast.

As an example, a simple program in this language is as follows. Note there is an indirect call in the `bar` function through a function pointer in a struct.

```
int foo (int x) {ret x+1};
int bar ({id1:int, id2:(int->int) fptr}* s) {
  int x;
  x = icall (s->id2) (s->id1);
  ret x
};
int y;
{id1:int, id2:(int->int) fptr}* s;
s = ({id1:int, id2:(int->int) fptr}*) malloc(2);
s->id1 = 10; s->id2 = &foo;
y = call bar(s);
```

3.2 Operational semantics

For formalizing CFG-construction soundness in CFI, it turns out that the operational semantics needs to distinguish between memory-safe and memory-unsafe executions. We first present the reason behind this.

The soundness of CFG construction means that the CFG constructed for a program is always respected by the program's execution at runtime. Programs in memory-unsafe languages including C and C++, however, can have memory errors such as out-of-bounds memory accesses or dangling pointer accesses. Such memory errors can cause the corruption of code pointers in memory and change the control flow of the program in arbitrary ways. For instance, an "int$*$" pointer may go out of bounds and point to a location that stores a function pointer and a subsequent memory update via the "int$*$" pointer can change the function pointer to arbitrary values; then a use of the function pointer in an indirect call may change the control flow in arbitrary ways. We call such executions memory-unsafe executions.

Clearly, CFG construction should not capture control-flow edges that occur only in memory-unsafe executions of the program. Therefore, the soundness of CFG construction should be stated as follows: *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 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 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 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-safe ones. Memory safety can be decomposed into spatial memory safety and temporal memory safety. In the operational semantics 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 operational semantics can ignore temporal memory safety. Our language could be augmented with memory deallocation and temporal 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 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 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 $b \le l \ \wedge \ l < d$. Because of pointer arithmetic, l could be an address outside the range of $[b, d)$.

$$
\begin{array}{llll}
\textit{Val} & v & ::= & w \mid l_{(b,d)} \mid \&f \\
\textit{Mem} & m & : & \textit{Addr} \rightharpoonup \textit{Val} \\
\textit{StkFrame} & \delta & : & \textit{Var} \rightharpoonup \textit{Addr} \\
\textit{MemType} & \Lambda & : & \textit{Addr} \rightharpoonup \textit{AtomicType} \\
\textit{State} & \sigma & ::= & (m, \overline{\delta}, \Lambda) \\
\textit{Cont} & c & := & \epsilon \mid s; c \mid lv = \text{ret } e; c \\
\textit{Config} & \kappa & := & (\sigma, c)
\end{array}
$$

Figure 3: Machine configurations.

A machine state σ contains a memory m, a series of stack frames $\overline{\delta}$, and a memory type Λ. A memory m is a partial map from addresses to values. A stack frame δ maps variables to their addresses in memory. There is a series of stack frames in a state, one for each function call. A memory type Λ records the declared types of locations during memory allocation. For instance, if the program declares a global variable x of type int$*$, then the global stack frame would map x to some address l and Λ would map l to type int$*$. Given a state σ, we write $\sigma.m$, $\sigma.\overline{\delta}$, and $\sigma.\Lambda$ for its memory, stack frames, and memory type, respectively.

A machine configuration is (σ, c), where σ is the current state and c the continuation for the rest of the program. A continuation c can be the empty list ϵ, which tells that the program has ended, or a statement s followed by another continuation, or a return continuation in the syntax of "$lv = \text{ret } e; c$"; the return continuation is for returning to the caller of the current running function.

During machine-configuration evaluation, we also assume a function environment $\mathcal{F}$, which maps from function names to their definitions and is initialized as follows:

Definition 3.1 (Initial configuration). For a program P of the form "$fd_1; \ldots; fd_n;\ t_1\, x_1; \ldots; t_j\, x_j;\ s_1; \ldots; s_k$", 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 $fd_1; \ldots; fd_n$, and

$$
\begin{array}{l}
m_0 = \{l_1 \mapsto \text{default}(t_1), \ldots, l_j \mapsto \text{default}(t_j)\}, \\
\delta_0 = \{x_1 \mapsto l_1, \ldots, x_j \mapsto l_j\}, \\
\Lambda = \{l_1 \mapsto t_1, \ldots, l_j \mapsto t_j\}.
\end{array}
$$

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

Definition 3.2 (Default values).

$$
\begin{array}{lcl}
\text{default(int)} & = & 0 \\
\text{default}(\tau*) & = & 0_{(0,0)} \\
\text{default}((t_1 \rightarrow t_2)\ \text{fptr}) & = & \&\text{dummyFun}
\end{array}
$$

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 address is used to initialize function-pointer variables; it is assumed that dummyFun does not appear in the program.

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 $\sigma = (m, \overline{\delta}, \Lambda)$ to simplify notation. During evaluation, memory errors 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.

$\boxed{(\sigma, lv) \Downarrow_{\text{lval}} v : t}$.

$$
\begin{array}{l}
(\sigma, x) \Downarrow_{\text{lval}} \overline{\delta}(x)_{(\overline{\delta}(x), \overline{\delta}(x)+1)} : \Lambda(\overline{\delta}(x)) \\
(\sigma, *lv) \Downarrow_{\text{lval}} m(l) : t, \\
\quad \text{if } (\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : t* \ \wedge\ \text{inbound}(l, b, d) \\
(\sigma, lv\text{->id}_i) \Downarrow_{\text{lval}} (l' + \mathit{off})_{(b'', d'')} : t_i, \\
\quad \text{if } (\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : \tau* \ \wedge\ \tau = \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\} \\
\quad \wedge\ \text{inbound}(l, b, d) \ \wedge\ \text{to\_ptr}(m(l)) = l'_{(b', d')} \\
\quad \wedge\ l' - b' = k * \text{size}(\tau) \ \wedge\ \mathit{off} = \text{offset}(\text{id}_i, \tau) \\
\quad \wedge\ [b'', d'') = [b', d') \cap [l' + \mathit{off}, l' + \mathit{off} + 1)
\end{array}
$$

where

$$
\overline{\delta}(x) = \delta_0(x), \text{if } \overline{\delta} = \delta_0; \overline{\delta_1} \ \wedge\ x \in \text{dom}(\delta_0)
$$

Figure 4: L-value evaluation. Assume $\sigma = (m, \overline{\delta}, \Lambda)$

$\boxed{(\sigma, e) \Downarrow v : t}$

$$
\begin{array}{l}
(\sigma, w) \Downarrow w : \text{int} \\
(\sigma, lv) \Downarrow m(l) : t, \text{if } (\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : t \ \wedge\ \text{inbound}(l, b, d) \\
(\sigma, \&lv) \Downarrow v : t*, \text{if } (\sigma, lv) \Downarrow_{\text{lval}} v : t \\
(\sigma, \&f) \Downarrow \&f : (t_1 \rightarrow t_2)\ \text{fptr}, \text{when } \mathcal{F}(f) = t_2\ f\ (t_1\ x_1)\{\ldots\} \\
(\sigma, e_1 + e_2) \Downarrow w_1 + w_2 : \text{int}, \\
\quad \text{if for } i = [1..2],\ (\sigma, e_i) \Downarrow v_i : \text{int} \ \wedge\ \text{to\_int}(v_i) = w_i \\
(\sigma, e_1 + e_2) \Downarrow (l + w * \text{size}(\tau))_{(b,d)} : \tau*, \\
\quad \text{if } (\sigma, e_1) \Downarrow v_1 : \tau* \ \wedge \text{to\_ptr}(v_1) = l_{(b,d)} \\
\quad \wedge\ (\sigma, e_2) \Downarrow v_2 : \text{int} \ \wedge\ \text{to\_int}(v_2) = w \\
(\sigma, (t)e) \Downarrow v : t, \text{if } (\sigma, e) \Downarrow v : t'
\end{array}
$$

where

$$
\text{to\_int}(v) = \begin{cases} w & \text{if } v = w \\ l & \text{if } v = l_{(b,d)} \\ \text{undefined} & \text{otherwise} \end{cases}
$$

and

$$
\text{to\_ptr}(v) = \begin{cases} w_{(0,0)} & \text{if } v = w \\ l_{(b,d)} & \text{if } v = l_{(b,d)} \\ \text{undefined} & \text{otherwise} \end{cases}
$$

Figure 5: Expression evaluation. Assume $\sigma = (m, \overline{\delta}, \Lambda)$.

Fig. 4 presents the rules for l-value evaluation in the form of "$(\sigma, lv) \Downarrow_{\text{lval}} v : t$", 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 $\overline{\delta}$. Evaluation of $*lv$ requires a memory read through a pointer value and therefore requires the pointer be in bound. Evaluation of lv->id$_i$ needs to consider the offset of id$_i$ in the flattened type list of a struct type.

Fig. 5 presents the rules for expression evaluation. Judgment "$(\sigma, e) \Downarrow v : t$" 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(−) and to_ptr(−) 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

$$(\sigma, c) \rightarrow (\sigma', c'), \text{if } \sigma = (m, \overline{\delta}, \Lambda)$$

and $c =$	then $(\sigma', c') =$
$lv = e; c_1$	$((m[l \mapsto v], \overline{\delta}, \Lambda), c_1),\ \text{if } (\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : t\ \wedge\ \text{inbound}(l, b, d)\ \wedge\ (\sigma, e) \Downarrow v : t'$
$lv = \text{call } f(e); c_1$	$(\text{stack\_alloc}(\sigma, x : t_1 = v; \overline{y : t = \text{default}(t)}), \overline{s_f}; lv = \text{ret } e_f; c_1),$ $\text{if } (\sigma, e) \Downarrow v : t_1'\ \wedge\ \mathcal{F}(f) = t_2\ f\ (t_1\ x)\{\overline{t\ y}; \overline{s_f};\ \text{ret } e_f\}$
$lv = \text{icall } e(e_1); c_1$	$(\sigma, lv = \text{call } f(e_1); c_1),\ \text{if } (\sigma, e) \Downarrow \&f : (t_1 \rightarrow t_2)\ \text{fptr}$
$lv = (\tau*)\ \text{malloc}(e); c_1$	$(m'[l \mapsto l_{1(l_1, l_1+w)}], \overline{\delta}, \Lambda'), c_1)$ $\text{if } (\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : \tau*\ \wedge\ \text{inbound}(l, b, d)\ \wedge\ (\sigma, e) \Downarrow v : \text{int}$ $\wedge\ \text{to\_int}(v) = w\ \wedge\ \text{heap\_alloc}(\sigma, w, \tau) = ((m', \overline{\delta}, \Lambda'), l_1)$
$lv = \text{ret } e; c_1$	$((m[l \mapsto v], \overline{\delta_1}, \Lambda), c_1)$ $\text{if } \overline{\delta} = \delta_0; \overline{\delta_1}\ \wedge\ (\sigma, e) \Downarrow v : t'\ \wedge\ ((m, \overline{\delta_1}, \Lambda), lv) \Downarrow_{\text{lval}} l_{(b,d)} : t\ \wedge\ \text{inbound}(l, b, d)$

$\text{stack\_alloc}((m, \overline{\delta}, \Lambda), x_1 : t_1 = v_1; \ldots; x_n : t_n = v_n) = (m \cup m_0, \delta_0; \overline{\delta}, \Lambda \cup \Lambda_0)$, where
$m_0 = \{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ so that $\text{dom}(m_0) \cap \text{dom}(m) = \emptyset$ and
$\delta_0 = \{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}$ and $\Lambda_0 = \{l_1 \mapsto t_1, \ldots, l_n \mapsto t_n\}$

$\text{heap\_alloc}((m, \overline{\delta}, \Lambda), w, \tau) = ((m \cup m_0, \overline{\delta}, \Lambda \cup \Lambda_0), l_1)$, where
$\text{extend}(\text{flatten}(\tau), w) = t_1; \ldots; t_w\ \wedge\ m_0 = \{l_1 \mapsto \text{default}(t_1), \ldots, l_1 + w - 1 \mapsto \text{default}(t_w)\}$
$\wedge\ \text{dom}(m_0) \cap \text{dom}(m) = \emptyset\ \wedge\ \Lambda_0 = \{l_1 \mapsto t_1, \ldots, l_1 + w - 1 \mapsto t_w\}$

Figure 6: Machine configuration evaluation.

pointer is converted to an integer, it loses the bounds information, losing its capability as a pointer.

Runtime evaluation of machine configurations is formalized as a small-step relation: $(\sigma, c) \rightarrow (\sigma', c')$. Fig. 6 presents the relatively standard rules. Heap allocation in the rule for "$lv = (\tau*)\ \text{malloc}(e)$" requires an extend$(-,-)$ operation; specifically, extend(flatten(τ), w) returns a type list of width w by repeating the pattern in flatten(τ). As an example, suppose flatten(τ) = [int, int$*$], then

$$\text{extend}(\text{flatten}(\tau), 5) = [\text{int}, \text{int}*, \text{int}, \text{int}*, \text{int}].$$

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 "$(t_1 \rightarrow t_2)$ fptr" is allowed to invoke any function whose type is $t_1 \rightarrow t_2$.[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_fptr$(-)$; it returns true if and only if t or τ contains a function-pointer type directly or indirectly. We abbreviate such types as has-fptr types.

[3] Instead of strict type equality, MCFI actually uses a notion of structural equivalence when matching function pointers and functions, for accommodating functions of variable numbers of arguments and structurally equivalent struct types. We ignore this aspect in this formalization, although accommodating it would not add too much difficulty.

$\boxed{(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}}$

$(\sigma, *lv) \Downarrow_{\text{lval}} \text{merr}$, if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$
$(\sigma, *lv) \Downarrow_{\text{lval}} \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} v : t \ \wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$
$(\sigma, lv\text{->id}) \Downarrow_{\text{lval}} \text{merr}$, if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$
$(\sigma, lv\text{->id}) \Downarrow_{\text{lval}} \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} v : \tau* \ \wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$
$(\sigma, lv\text{->id}) \Downarrow_{\text{lval}} \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} l_{(b,d)} : \tau* \ \wedge\ \tau = \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\}$
  $\wedge\ \text{inbound}(l, b, d) \ \wedge\ \text{to\_ptr}(m(l)) = l'_{(b',d)}$
  $\wedge\ (l' - b')$ is not a multiple of size(τ)

$\boxed{(\sigma, e) \Downarrow \text{merr}}$

$(\sigma, lv) \Downarrow \text{merr}$, if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$
$(\sigma, lv) \Downarrow \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} v : t \ \wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$
$(\sigma, \&lv) \Downarrow \text{merr}$, if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$
$(\sigma, e_1 + e_2) \Downarrow \text{merr}$, if $(\sigma, e_1) \Downarrow \text{merr}$ or $(\sigma, e_2) \Downarrow \text{merr}$
$(\sigma, (t)e) \Downarrow \text{merr}$, if $(\sigma, e) \Downarrow \text{merr}$

$\boxed{(\sigma, c) \longrightarrow \text{merr}}$

$(\sigma, lv = e; c) \longrightarrow \text{merr}$, if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$ or $(\sigma, e) \Downarrow \text{merr}$
$(\sigma, lv = e; c) \longrightarrow \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} v : t \ \wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$
$(\sigma, lv = \text{call } f(e); c) \longrightarrow \text{merr}$, if $(\sigma, e) \Downarrow \text{merr}$
$(\sigma, lv = \text{icall } e(e_1); c) \longrightarrow \text{merr}$, if $(\sigma, e) \Downarrow \text{merr}$
$(\sigma, lv = \text{icall } e(e_1); c) \longrightarrow \text{merr}$,
  if $(\sigma, e) \Downarrow \&\text{dummyFun} : (t_1 \longrightarrow t_2)\ \text{fptr}$
$(\sigma, lv = (\tau*)\ \text{malloc}(e); c) \longrightarrow \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} \text{merr}$ or $(\sigma, e) \Downarrow \text{merr}$
$(\sigma, lv = (\tau*)\ \text{malloc}(e); c) \longrightarrow \text{merr}$,
  if $(\sigma, lv) \Downarrow_{\text{lval}} v : t \ \wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$
$(\sigma, lv = \text{ret } e; c) \longrightarrow \text{merr}$, if $(\sigma, e) \Downarrow \text{merr}$
$(\sigma, lv = \text{ret } e; c) \longrightarrow \text{merr}$, if $\overline{\delta} = \delta_0; \overline{\delta_1} \ \wedge\ ((m, \overline{\delta_1}, \Lambda), lv) \Downarrow_{\text{lval}} \text{merr}$
$(\sigma, lv = \text{ret } e; c) \longrightarrow \text{merr}$,
  if $\overline{\delta} = \delta_0; \overline{\delta_1} \ \wedge\ ((m, \overline{\delta_1}, \Lambda), lv) \Downarrow_{\text{lval}} v : t$
  $\wedge\ \text{to\_ptr}(v) = l_{(b,d)} \ \wedge\ \neg\text{inbound}(l, b, d)$

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

Definition 4.1 (has-fptr types).

$$\text{has\_fptr}(t) = \begin{cases} \text{false} & \text{if } t = \text{int} \\ \text{true} & \text{if } t = (t_1 \longrightarrow t_2)\ \text{fptr} \\ \text{has\_fptr}(\tau) & \text{if } t = \tau* \end{cases}$$

$$\text{has\_fptr}(\tau) = \begin{cases} \text{has\_fptr}(t), & \text{if } \tau = t \\ \text{has\_fptr}(t_1) \ \vee\ \ldots\ \vee\ \text{has\_fptr}(t_n) & \\ \qquad \text{if } \tau = \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\} & \end{cases}$$

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

$\boxed{\Gamma \vdash_{\text{lval}} lv : t}$

$$\frac{}{\Gamma \vdash_{\text{lval}} x : \Gamma(x)} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t*}{\Gamma \vdash_{\text{lval}} *lv : t}$$

$$\frac{\Gamma \vdash_{\text{lval}} lv : \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\}*}{\Gamma \vdash_{\text{lval}} lv\text{->id}_i : t_i}$$

$\boxed{\Gamma \vdash e : t}$

$$\frac{}{\Gamma \vdash w : \text{int}} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t}{\Gamma \vdash lv : t} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t}{\Gamma \vdash \&lv : t*}$$

$$\frac{\mathcal{F}(f) = t_2\ f\ (t_1\ x_1)\{\ldots\}}{\Gamma \vdash \&f : (t_1 \longrightarrow t_2)\ \text{fptr}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : \tau* \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \tau*} \qquad \frac{\Gamma \vdash e : t' \quad \text{allowed\_cast}(t', t)}{\Gamma \vdash (t)e : t}$$

$\boxed{\Gamma \vdash s \text{ ok}}$

$$\frac{\Gamma \vdash_{\text{lval}} lv : t \quad \Gamma \vdash e : t}{\Gamma \vdash lv = e \text{ ok}} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : \tau* \quad \Gamma \vdash e : \text{int}}{\Gamma \vdash lv = (\tau*)\ \text{malloc}(e) \text{ ok}}$$

$$\frac{\Gamma \vdash_{\text{lval}} lv : t_2 \quad \mathcal{F}(f) = t_2\ f\ (t_1\ x_1)\{\ldots\} \quad \Gamma \vdash e : t_1}{\Gamma \vdash lv = \text{call } f(e) \text{ ok}}$$

$$\frac{\Gamma \vdash_{\text{lval}} lv : t_2 \quad \Gamma \vdash e : (t_1 \longrightarrow t_2)\ \text{fptr} \quad \Gamma \vdash e_1 : t_1}{\Gamma \vdash lv = \text{icall } e(e_1) \text{ ok}}$$

$\boxed{\vdash fd \text{ ok}}$

$$\frac{\Gamma = x : t, x_1 : t_1, \ldots, x_n : t_n \quad \forall j \in [1..m],\ \Gamma \vdash s_j \text{ ok} \quad \Gamma \vdash e : t'}{\vdash t'\ f\ (t\ x)\{t_1\ x_1; \ldots; t_n\ x_n;\ s_1; \ldots; s_m; \text{ret } e\} \text{ ok}}$$

$\boxed{\vdash P \text{ ok}}$

$$\frac{\begin{array}{c}\forall i \in [1..n],\ \vdash fd_i \text{ ok} \\ \forall j \in [1..k],\ x_1 : t_1, \ldots, x_m : t_m \vdash s_j \text{ ok}\end{array}}{\vdash (fd_1; \ldots; fd_n;\ t_1\ x_1; \ldots; t_m\ x_m;\ s_1; \ldots; s_k) \text{ ok}}$$

Figure 8: Typing rules for type-based CFG construction.

Judgment	Meaning
$\Gamma \vdash_{\text{lval}} lv : t$	L-value lv holds t values under Γ.
$\Gamma \vdash e : t$	Expression e has type t under Γ.
$\Gamma \vdash s$ ok	Statement s is well typed under typing context Γ.
$\vdash fd$ ok	Function declaration fd is well typed.
$\vdash P$ ok	Program P is well typed.

Fig. 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 "$(t)e$" uses predicate allowed_cast(t', t), where t' is the type of e. It is defined as follows:

$$\text{allowed\_cast}(t', t) = \neg\text{has\_fptr}(t') \ \wedge\ \neg\text{has\_fptr}(t)$$

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 τ and τ' 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(Λ, v, t), which models the invariant that holds on a value v, a type t (which is the type of an expression that evaluates to v), and a memory type Λ; recall Λ is a map from locations to their declared types during allocation.

Definition 4.2 (Value invariants for the type-based method). value_inv(Λ, v, t) holds when one of the following cases holds:

(1) if $v = w$, then $t =$ int, or $t = \tau*$ for some τ and ¬has_fptr(τ);
(2) if $v = l_{(b,d)}$ and has_fptr(t), then $t = \tau*$ for some τ and $\Lambda|_{[b,d)} =$ extend(flatten(τ), $d - b$).
(3) if $v = l_{(b,d)}$ and ¬has_fptr(t), then for all $l' \in [b, d)$, we have ¬has_fptr($\Lambda(l')$).
(4) if $v = \&f$, then $\mathcal{F}(f) = t_2\ f\ (t_1\ x_1)\{\ldots\}$ and $t = (t_1 \to t_2)$ fptr;
(5) if $v =$ &dummyFun, then $t = (t_1 \to t_2)$ fptr for some t_1 and t_2.

We next explain value_inv(Λ, v, t) by cases.

- When v is an integer w, the static type can either be int, or some $\tau*$ because of type casts. For instance, "(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.
- The two cases for $l_{(b,d)}$ capture the invariant maintained on data-pointer values. When has_fptr(t), then $t = \tau*$ for some τ and the types of the locations of the underlying buffer must be consistent with the flattened type list of τ in the memory type Λ. Recall that extend(flatten(τ), $d - b$) returns a type list of width $d - b$ by repeating the pattern in flatten(τ). We use the notation $\Lambda|_{[b,d)}$ for the type list in Λ in the range of $[b, d)$. This case enforces partial type safety on has-fptr types.
  On the other hand, when "¬has_fptr(t)", the locations of the underlying buffer may not be consistent with t because of type casts. For example, suppose $l_{(b,d)}$ points to a buffer whose locations have type int in Λ; 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 Λ. In this case, we require that the types of the locations in Λ 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.
- Finally, when v is the address of a function, the type must be exactly the same as the function's declared type.

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

Definition 4.3 (Well-formed states). wf_state($m, \overline{\delta}, \Lambda$) holds if
(1) $\forall l \in \text{dom}(\Lambda)$, value_inv($\Lambda, m(l), \Lambda(l)$).
(2) $\text{dom}(m) = \text{dom}(\Lambda) \supseteq \text{rng}(\overline{\delta})$, where $\text{rng}(\overline{\delta})$ is the union of the ranges of stack frames in $\overline{\delta}$. The range of a single stack frame δ is defined to be the set $\{\delta(x) | x \in \text{dom}(\delta)\}$.

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.

$\boxed{\text{wf\_conf}(\sigma, c)}$

$$\frac{\text{wf\_state}(\sigma)}{\text{wf\_conf}(\sigma, \epsilon)} \qquad \frac{\Gamma(\sigma) \vdash s \text{ ok} \quad \text{wf\_conf}(\sigma, c)}{\text{wf\_conf}(\sigma, s; c)}$$

$$\frac{\begin{array}{ll}\sigma = (m, \delta_0; \overline{\delta}, \Lambda) & \text{wf\_state}(\sigma) \\ \Gamma(\sigma) \vdash e : t & \sigma_1 = (m, \overline{\delta}, \Lambda) \\ \Gamma(\sigma_1) \vdash_{\text{lval}} lv : t & \text{wf\_conf}(\sigma_1, c)\end{array}}{\text{wf\_conf}(\sigma, lv = \text{ret } e; c)}$$

Figure 9: Well-formed configurations.

Theorem 4.4 (Preservation). *If* $(\sigma, c) \to (\sigma', c')$, *and* wf_conf($\sigma, c$), *then* wf_conf($\sigma', c'$).

Theorem 4.5 (Progress). *If* wf_conf(σ, c), *then either exist* σ' *and* c' *so that* $(\sigma, c) \to (\sigma', c')$, *or* $(\sigma, c) \to$ merr, *or* $c = \epsilon$ *(the final state).*

Preservation is proved by case analysis over $(\sigma, c) \to (\sigma', c')$ and progress is proved by case analysis over wf_conf(σ, 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 value_inv(−, −, −) for partial type safety.

Corollary 4.6 (Soundness of type-based CFG construction). *When a well-typed program (i.e.,* $\vdash P$ ok*) executes an indirect call*

"lv = icall $e(e_1)$" in a memory-safe execution, and e's type is "$(t_1 \to t_2)$ fptr" in its type environment, then the execution always invokes a function whose type is $t_1 \to t_2$.

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 $\kappa = (\sigma, lv = \text{icall } e(e_1); c)$, we must have that κ is well formed thanks to the preservation theorem. Therefore e is of type "$(t_1 \to t_2)$ fptr" in configuration κ. Thanks to the progress theorem, if the next state is not merr, then we must have $(\sigma, e) \Downarrow \&f : (t_1 \to t_2)$ fptr for some f, which in turn gives us value_inv$(\sigma.\Lambda, \&f, (t_1 \to t_2)$ fptr$)$. By definition, we get $\mathcal{F}(f) = t_2\ f\ (t_1\ x_1)\{\ldots\}$. □

5 CFG CONSTRUCTION BASED ON TAINT ANALYSIS

Ge *et al.* [5] recently proposed a CFG-construction method based on taint analysis (specifically, dataflow analysis on function pointers). They applied the method to enforce CFI on OS kernel software, including the FreeBSD kernel. The basic idea is to track how function pointers are propagated in the input program via taint analysis, as a way of determining what indirect-call sites function pointers can reach. In more detail, for a function f, the method first taints all function pointers that directly receive the address of f and then taints (recursively) any function pointer that may receive the value of a tainted function pointer. After the tainting process, if the function pointer used in an indirect call is tainted, then f is in the target set of the indirect call. This taint-tracking process is performed separately for each function in the program. After all functions have been processed, the target set of an indirect call is determined.

To simplify taint tracking without a full-blown points-to analysis, the method by Ge *et al.* makes a few assumptions about how programs use function pointers.

> A1: *Function-pointer types may not be cast to or cast from non-function-pointer types.*
> A2: *There exists no direct data pointers to function pointers.*
> A3: *No pointer arithmetic or memory reads/writes through a function pointer are allowed.*

Assumption A1 prohibits casts between function-pointer types and non-function-pointer types. It guarantees that function pointers can be stored only in variables of function-pointer types and therefore taint tracking can ignore variables of non-function-pointer types. Without the assumption, a function pointer could be stored in a variable of a type such as int and taint tracking would have to track such variables. Note that the type-cast assumption here is less strict than the type-cast assumption of the type-based method that was discussed in the previous section: assumption A1 of the taint-based method allows a function-pointer type be cast to a different function-pointer type; it maintains the separation between function pointers and data pointers, but integrity of function-pointer types is not guaranteed; in contrast, integrity of function-pointer types is maintained in the type-based method.

Assumption A2 excludes types such as "$((t_1 \to t_2)$ fptr$)*$", implying that a function pointer cannot be buried inside a data pointer. This further simplifies taint tracking as otherwise there would be a need to track the taints of function pointers stored in data pointers. We note this assumption still allows a data pointer to a struct that contains a function-pointer field. Finally, assumption A3 restricts

$$\boxed{\text{allowed\_cast}(t_1, t_2) \rightsquigarrow \Psi}$$

$$\frac{\neg\text{has\_fptr}(\tau)}{\text{allowed\_cast}(\text{int}, \tau*) \rightsquigarrow \emptyset} \qquad \frac{\neg\text{has\_fptr}(\tau)}{\text{allowed\_cast}(\tau*, \text{int}) \rightsquigarrow \emptyset}$$

$$\frac{\neg\text{has\_fptr}(\tau_1) \quad \neg\text{has\_fptr}(\tau_2)}{\text{allowed\_cast}(\tau_1*, \tau_2*) \rightsquigarrow \emptyset}$$

$$\text{allowed\_cast}((t_1 \xrightarrow{b} t_2)\ \text{fptr}, (t_1' \xrightarrow{b'} t_2')\ \text{fptr}) \rightsquigarrow \{b = b'\}$$

Figure 10: Allowed type casts.

what operations can be performed on function pointers and is the same as a previous assumption in the type-based method.

We next introduce a type and constraint-generation system that formalizes the aforementioned assumptions and the taint-tracking process. The three assumptions will be baked into the typing rules of the system; taint tracking will be formalized as a constraint-generation process of the system.

First, the function-pointer type is changed to add a boolean *taint tag*: $(t_1 \xrightarrow{b} t_2)$ fptr, where the taint tag b can be either T (true) or F (false). Since taint tracking is performed on individual functions, we fix a particular function, say foo, for which the system performs taint tracking. Intuitively, when a value is of type "$(t_1 \xrightarrow{\text{T}} t_2)$ fptr", it is a function pointer that *may* point to foo; when it is of type "$(t_1 \xrightarrow{\text{F}} t_2)$ 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 (τ).

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

$$\text{no\_dp\_to\_fp}(t) = \begin{cases} \text{true}, \text{ if } t = \text{int} \\ \tau \neq (t_1 \xrightarrow{b} t_2)\ \text{fptr} \ \wedge\ \text{no\_dp\_to\_fp}(\tau) \\ \quad \text{if } t = \tau* \\ \text{no\_dp\_to\_fp}(t_1) \ \wedge\ \text{no\_dp\_to\_fp}(t_2) \\ \quad \text{if } t = (t_1 \xrightarrow{b} t_2)\ \text{fptr} \end{cases}$$

$$\text{no\_dp\_to\_fp}(\tau) = \begin{cases} \text{no\_dp\_to\_fp}(t), \text{ if } \tau = t \\ \text{no\_dp\_to\_fp}(t_1) \ \wedge\ \ldots\ \wedge\ \text{no\_dp\_to\_fp}(t_n) \\ \quad \text{if } \tau = \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\} \end{cases}$$

Fig. 10 presents judgment "allowed_cast$(t_1, t_2) \rightsquigarrow \Psi$", which tells what kinds of type casts are allowed. In the judgment, Ψ 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 be cast to another different function-pointer type, as long as their taint tags are the same. Note there is no need to worry

$$\boxed{\Gamma \vdash_{\text{lval}} lv : t} \qquad \frac{}{\Gamma \vdash_{\text{lval}} x : \Gamma(x)} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t*}{\Gamma \vdash_{\text{lval}} *lv : t} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : \{\text{id}_1 : t_1, \ldots, \text{id}_n : t_n\}*}{\Gamma \vdash_{\text{lval}} lv\text{->id}_i : t_i}$$

$$\boxed{\Gamma \vdash e : t \rightsquigarrow \Psi} \qquad \frac{}{\Gamma \vdash w : \text{int} \rightsquigarrow \emptyset} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t}{\Gamma \vdash lv : t \rightsquigarrow \emptyset} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t}{\Gamma \vdash \&lv : t* \rightsquigarrow \emptyset} \qquad \frac{\mathcal{F}(\text{foo}) = t_2\, \text{foo}\, (t_1\, x_1)\{\ldots\}}{\Gamma \vdash \&\text{foo} : (t_1 \xrightarrow{b} t_2)\ \text{fptr} \rightsquigarrow \{b = \text{T}\}}$$

$$\frac{f \neq \text{foo} \quad \mathcal{F}(f) = t_2\, f\, (t_1\, x_1)\{\ldots\}}{\Gamma \vdash \&f : (t_1 \xrightarrow{b} t_2)\ \text{fptr} \rightsquigarrow \{b = \text{F}\}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \rightsquigarrow \Psi_1 \quad \Gamma \vdash e_2 : \text{int} \rightsquigarrow \Psi_2}{\Gamma \vdash e_1 + e_2 : \text{int} \rightsquigarrow \Psi_1 \cup \Psi_2}$$

$$\frac{\Gamma \vdash e_1 : \tau* \rightsquigarrow \Psi_1 \quad \Gamma \vdash e_2 : \text{int} \rightsquigarrow \Psi_2}{\Gamma \vdash e_1 + e_2 : \tau* \rightsquigarrow \Psi_1 \cup \Psi_2} \qquad \frac{\Gamma \vdash e : t' \rightsquigarrow \Psi_1 \quad \text{allowed\_cast}(t', t) \rightsquigarrow \Psi_2}{\Gamma \vdash (t)e : t \rightsquigarrow \Psi_1 \cup \Psi_2}$$

$$\boxed{t_1 <: t_2 \rightsquigarrow \Psi} \qquad \frac{}{\text{int} <: \text{int} \rightsquigarrow \emptyset} \qquad \frac{}{\tau* <: \tau* \rightsquigarrow \emptyset} \qquad \frac{}{(t_1 \xrightarrow{b} t_2)\ \text{fptr} <: (t_1 \xrightarrow{b'} t_2)\ \text{fptr} \rightsquigarrow \{b = \text{T} \implies b' = \text{T}\}}$$

$$\boxed{\Gamma \vdash e <: t \rightsquigarrow \Psi} \qquad \frac{\Gamma \vdash e : t' \rightsquigarrow \Psi' \quad t' <: t \rightsquigarrow \Psi}{\Gamma \vdash e <: t \rightsquigarrow \Psi' \cup \Psi}$$

$$\boxed{\Gamma \vdash s \rightsquigarrow \Psi} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : t \quad \Gamma \vdash e <: t \rightsquigarrow \Psi}{\Gamma \vdash lv = e \rightsquigarrow \Psi} \qquad \frac{\Gamma \vdash_{\text{lval}} lv : \tau* \quad \Gamma \vdash e : \text{int} \rightsquigarrow \Psi}{\Gamma \vdash lv = (\tau*)\ \text{malloc}(e) \rightsquigarrow \Psi}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\text{lval}} lv : t_2' \quad \mathcal{F}(f) = t_2\, f\, (t_1\, x_1)\{\ldots\} \\ \Gamma \vdash e <: t_1 \rightsquigarrow \Psi_1 \quad t_2 <: t_2' \rightsquigarrow \Psi_2\end{array}}{\Gamma \vdash lv = \text{call}\ f(e) \rightsquigarrow \Psi_1 \cup \Psi_2} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\text{lval}} lv : t_2' \quad \Gamma \vdash e : (t_1 \xrightarrow{b} t_2)\ \text{fptr} \rightsquigarrow \Psi_1 \\ \Gamma \vdash e_1 <: t_1 \rightsquigarrow \Psi_2 \quad t_2 <: t_2' \rightsquigarrow \Psi_3\end{array}}{\Gamma \vdash lv = \text{icall}\ e(e_1) \rightsquigarrow \Psi_1 \cup \Psi_2 \cup \Psi_3}$$

$$\boxed{\vdash fd \rightsquigarrow \Psi} \qquad \frac{\Gamma = x : t, x_1 : t_1, \ldots, x_n : t_n \quad \forall j \in [1..m],\ \Gamma \vdash s_j \rightsquigarrow \Psi_j \quad \Gamma \vdash e <: t' \rightsquigarrow \Psi'}{\vdash t'\, f\, (t\, x)\{t_1\, x_1; \ldots; t_n\, x_n;\ s_1; \ldots; s_m; \text{ret}\ e\} \rightsquigarrow \Psi_1 \cup \ldots \cup \Psi_m \cup \Psi'}$$

$$\boxed{\vdash P \rightsquigarrow \Psi} \qquad \frac{\forall i \in [1..n],\ \vdash fd_i \rightsquigarrow \Psi_i \quad \forall j \in [1..k],\ x_1 : t_1, \ldots, x_m : t_m \vdash s_j \rightsquigarrow \Psi_j'}{\vdash (fd_1; \ldots; fd_n;\ t_1\, x_1; \ldots; t_m\, x_m;\ s_1; \ldots; s_k) \rightsquigarrow \Psi_1 \cup \ldots \cup \Psi_n \cup \Psi_1' \cup \ldots \Psi_k'}$$

Figure 11: Typing and constraint-generation rules for taint-based CFG construction.

about cases such as when $t_1 = ((t_1 \xrightarrow{b} t_2)\ \text{fptr})*$ since such types are invalid by assumption A2.

Furthermore, we note that there are two views on judgments that produce taint-tag constraints, including "$\text{allowed\_cast}(t_1, t_2) \rightsquigarrow \Psi$". 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 *taint variables* 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 "$\Gamma \vdash e : t \rightsquigarrow \Psi$" when $e = \&\text{foo}$ 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 "$t_1 <: t_2 \rightsquigarrow \Psi$" 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 $\text{value\_inv}(\Lambda, v, t)$ for modeling the runtime invariant of value v and its static type v under memory type Λ in the taint-based method. The only difference from the corresponding version for the type-based method is the case when $v = \&f$; 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). $\text{value\_inv}(\Lambda, v, t)$ holds when one of the following cases holds:

(1) if $v = w$, then $t = \text{int}$, or $t = \tau*$ for some τ and $\neg\text{has\_fptr}(\tau)$;
(2) if $v = l_{(b,d)}$ and $\text{has\_fptr}(t)$, then $t = \tau*$ for some τ and $\Lambda|_{[b,d)} = \text{extend}(\text{flatten}(\tau), d - b)$.
(3) if $v = l_{(b,d)}$ and $\neg\text{has\_fptr}(t)$, then for all $l' \in [b, d)$, we have $\neg\text{has\_fptr}(\Lambda(l'))$.

(4) if $v = \&f$, then $\mathcal{F}(f) = t_2\, f\, (t_1\, x_1)\{\ldots\}$ and $t = (t_1 \xrightarrow{b} t_2)$ fptr, and if $f =$ foo, then $b =$ T.
(5) if $v =$ &dummyFun, then $t = (t_1 \to t_2)$ fptr 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 $\vdash P \rightsquigarrow \Psi$. Let η be a solution of Ψ 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 η. Then when $\eta(P)$ executes an indirect call "$lv =$ icall $e(e_1)$" in a memory-safe execution, and e's type is "$(t_1 \xrightarrow{\text{F}} t_2)$ fptr" 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 $\kappa = (\sigma, lv = \text{icall } e(e_1); c)$, we must have that κ is well formed thanks to the preservation theorem. Therefore e is of type "$(t_1 \to t_2)$ fptr" in configuration κ. Thanks to the progress theorem, if the next state is not merr, then we must have $(\sigma, e) \Downarrow \&f : (t_1 \xrightarrow{\text{F}} t_2)$ fptr for some f, which in turn gives us value_inv$(\sigma.\Lambda, \&f, (t_1 \xrightarrow{\text{F}} t_2)$ fptr). By definition of value invariants, f cannot be foo. □

By the corollary, only if e is of type "$(t_1 \xrightarrow{\text{T}} t_2)$ fptr" in "$lv =$ icall $e(e_1)$", 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 "`int (*)(unsigned long, unsigned long)`". In two places, the function pointer is set to be the address of `strcmp`, whose type is "`int (*)(const char*, const char*)`". Since the function pointer's type is incompatible with `strcmp`'s, MCFI's CFG generation does not connect the function pointer to `strcmp`. To fix the problem, a `strcmp` wrapper function was added that has the equivalent type as the type of the comparison function and makes a direct call to `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

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.
[2] David F. Bacon and Peter F. Sweeney. 1996. Fast Static Analysis of C++ Virtual Function Calls. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 324–341.
[3] Christopher L. Conway, Dennis Dams, Kedar S. Namjoshi, and Clark Barrett. 2008. Pointer Analysis, Conditional Soundness, and Proving the Absence of Errors. In *Proceedings of the 15th International Symposium on Static Analysis*. 62–77.
[4] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 242–256.
[5] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symposium on Security and Privacy (EuroS&P)*. 179–194.

[6] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call Graph Construction in Object-oriented Languages. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 108–124.
[7] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*. 31–40.
[8] Nergal. 2001. The Advanced Return-into-lib(c) Exploits: PaX Case Study. *Phrack Magazine, Volume 11, Issue 0x58, File 4 of 14* (2001).
[9] Ben Niu and Gang Tan. 2013. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *20th ACM Conference on Computer and Communications Security (CCS)*.
[10] Ben Niu and Gang Tan. 2014. Modular Control Flow Integrity. In *ACM Conference on Programming Language Design and Implementation (PLDI)*. 577–587.
[11] Jannik Pewny and Thorsten Holz. 2013. Control-Flow Restrictor: Compiler-based CFI for iOS. In *ACSAC '13: Proceedings of the 2013 Annual Computer Security Applications Conference*.
[12] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM Conference on Computer and Communications Security (CCS)*. 552–561.
[13] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *23rd Usenix Security Symposium*.
[14] Frank Tip and Jens Palsberg. 2000. Scalable Propagation-based Call Graph Construction Algorithms. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 281–293.
[15] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE Symposium on Security and Privacy (S&P)*. 934–953.
[16] Zhi Wang and Xuxian Jiang. 2010. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *IEEE Symposium on Security and Privacy (S&P)*. 380–395.
[17] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy (S&P)*. 559–573.
[18] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *22nd Usenix Security Symposium*. 337–352.

A APPENDIX: SOUNDNESS PROOF SKETCH

We next sketch the major lemmas and proofs for the soundness proof of type-based CFG construction. The proof structure for the soundness of taint-based CFG construction is largely the same and uses a similar set of lemmas, but with a different definition of value_inv$(-,-,-)$.

The following lemma is easily proved by using the definition of value_inv$(-,-,-)$.

Lemma A.1.

(1) If $\neg$has_fptr(t) *and* $\neg$has_fptr(t')*, then* value_inv(Λ, v, t) *is equivalent to* value_inv(Λ, v, t').
(2) If $[b,d)$ *is the empty range, then* value_inv$(\Lambda, l_{(b,d)}, \tau*)$ *holds for any* τ.
(3) If value_inv$(\Lambda, l_{(b,d)}, \tau*)$*, then* value_inv$(\Lambda, l'_{(b,d)}, \tau*)$ *for any* l'.
(4) If value_inv(Λ, v, t') *and* allowed_cast(t', t)*, then* value_inv(Λ, v, t).
(5) If value_inv(Λ, v, t) *and* $\Lambda \subseteq \Lambda'$*, then* value_inv$(\Lambda', v, t)$.

Lemma A.2 (Reads preserve invariants). *For* $\sigma = (m, \overline{\delta}, \Lambda)$*, if* wf_state$(\sigma)$*, and* value_inv$(\Lambda, l_{(b,d)}, t*)$*, and* inbound$(l, b, d)$*, then* value_inv$(\Lambda, m(l), t)$.

Proof. (a) Assume has_fptr(t). We can get $\Lambda(l) = t$, because value_inv$(\Lambda, l_{(b,d)}, t*)$, flatten$(t) = [t]$, and inbound(l, b, d). From wf_state(σ), we have value_inv$(\Lambda, m(l), \Lambda(l))$, which gives us the goal value_inv$(\Lambda, m(l), t)$.

(b) Assume $\neg$has_fptr(t). Then we get $\neg$has_fptr$(\Lambda(l))$, because value_inv$(\Lambda, l_{(b,d)}, t*)$ and inbound(l, b, d). From wf_state(σ), we have value_inv$(\Lambda, m(l), \Lambda(l))$, which is equivalent to our goal value_inv$(\Lambda, m(l), t)$ by part (1) of Lemma A.1. □

Lemma A.3 (Writes preserve invariants). *For* $\sigma = (m, \overline{\delta}, \Lambda)$*, if* wf_state$(\sigma)$*, and* value_inv$(\Lambda, l_{(b,d)}, t*)$*, and* inbound$(l, b, d)$*, and* value_inv$(\Lambda, v, t)$*, and* $\sigma' = (m[l \mapsto v], \overline{\delta}, \Lambda)$*, then* wf_state$(\sigma')$.

Proof. Let $m' = m[l \mapsto v]$. First, since dom(m') = dom(m) and wf_state(σ) gives us dom(m) = dom$(\Lambda) \supseteq$ rng$(\overline{\delta})$, we easily get dom(m') = dom$(\Lambda) \supseteq$ rng$(\overline{\delta})$.

Next we show for all $l' \in$ dom(Λ), value_inv$(\Lambda, m'(l'), \Lambda(l'))$. when $l' \neq l$, we get the result from wf_state(σ). When $l' = l$, we need to show value_inv$(\Lambda, v, \Lambda(l))$; this is proved by two cases.

(a) Assume has_fptr(t). From value_inv$(\Lambda, l_{(b,d)}, t*)$, flatten$(t) = [t]$, and inbound(l, b, d), we get $\Lambda(l) = t$. Then the goal is proved from assumption value_inv(Λ, v, t).

(b) Assume $\neg$has_fptr(t). Then we get $\neg$has_fptr$(\Lambda(l))$, because value_inv$(\Lambda, l_{(b,d)}, t*)$ and inbound(l, b, d). Then the assumption value_inv(Λ, v, t) is logically equivalent to goal value_inv$(\Lambda, v, \Lambda(l))$, by part (1) of Lemma A.1. □

From a state σ, we can construct a type environment $\Gamma(\sigma)$ from the top stack frame and the memory type in σ.

Definition A.4. Let $\sigma = (m, \delta_0; \overline{\delta_1}, \Lambda)$. Define $\Gamma(\sigma) = \lambda x.\, \Lambda(\delta_0(x))$.

Lemma A.5 (Lemma about l-values). *Let* $\sigma = (m, \overline{\delta}, \Lambda)$.

(1) If $\Gamma(\sigma) \vdash_{\text{lval}} lv : t$*, and* $(\sigma, lv) \Downarrow_{\text{lval}} v : t'$*, then* $t = t'$.
(2) If $\Gamma(\sigma) \vdash_{\text{lval}} lv : t$*, and* $(\sigma, lv) \Downarrow_{\text{lval}} v : t$*, and* wf_state$(\sigma)$*, then* value_inv$(\Lambda, v, t*)$.
(3) If $\Gamma(\sigma) \vdash_{\text{lval}} lv : t$*, and* wf_state$(\sigma)$*, then either exists* v *so that* $(\sigma, lv) \Downarrow_{\text{lval}} v : t$ *or* $(\sigma, lv) \Downarrow_{\text{lval}}$ merr.

The proof is by induction over the derivation of $\Gamma(\sigma) \vdash_{\text{lval}} lv : t$. The only difficult cases lie in the proof of part (2). The case for $*lv$ is proved using Lemma A.2. The case for lv->id 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 on flatten$(-)$: when $\neg$has_fptr(τ), any type t in flatten(τ) must satisfy $\neg$has_fptr(t).

Lemma A.6 (Lemma about expressions). *Let* $\sigma = (m, \overline{\delta}, \Lambda)$.

(1) If $\Gamma(\sigma) \vdash e : t$*, and* $(\sigma, e) \Downarrow v : t'$*, then* $t = t'$.
(2) If $\Gamma(\sigma) \vdash e : t$*, and* wf_state$(\sigma)$*, and* $(\sigma, e) \Downarrow v : t$*, then* value_inv$(\Lambda, v, t)$.
(3) If $\Gamma(\sigma) \vdash e : t$*, and* wf_state$(\sigma)$*, then either exists* v *so that* $(\sigma, e) \Downarrow v : t$ *or* $(\sigma, e) \Downarrow$ merr.

The proof of the above lemma is by straightforward induction over the derivation of $\Gamma(\sigma) \vdash e : t$.

The preservation theorem is then proved by case analysis over $(\sigma, c) \rightarrow (\sigma', c')$, using Lemmas A.5 and A.6 and the following lemma. The progress theorem is proved by case analysis over wf_conf(σ, c), using Lemmas A.5 and A.6.

Lemma A.7. *If* wf_conf(σ, c)*, and* $\sigma.\overline{\delta} = \sigma'.\overline{\delta}$*, and* $\sigma.\Lambda \subseteq \sigma'.\Lambda$*, and* wf_state$(\sigma')$*, then* wf_conf$(\sigma', c)$.