Static enforcement of security in runtime systems

Mathias V. Pedersen
Department of Computer Science
Aarhus University
mvp@cs.au.dk

Aslan Askarov
Department of Computer Science
Aarhus University
aslan@cs.au.dk

***Abstract*—Underneath every modern programming language is a runtime environment (RTE) that handles features such as automatic memory management and thread scheduling. In the information-flow control (IFC) literature, the RTE is often part of the trusted computing base (TCB), and there has been little focus on applying IFC to the implementation of the RTE itself. In this paper we address this problem by designing an IFC language, Zee, for implementing secure RTEs, thereby removing the RTE from the TCB. We implement Zee and design and implement secure versions of garbage collectors and thread schedulers using Zee. We also prove that a faithful calculus of Zee satisfies a strong variant of timing-sensitive noninterference.**

I. Introduction

Modern programming languages offer many abstraction mechanisms to simplify the development of programs, increase their throughput, or reduce their resource consumption at runtime. Such abstractions are often implemented by compiler writers in a specialized program called the runtime environment (RTE). A runtime environment is a program (possibly written in a different language) that is running alongside programs written by the user, and may use knowledge about the implementation details of the language to perform its tasks. Examples of features commonly associated with the RTE include thread scheduling and automatic memory management. For a user of the language, implementing such functionality is difficult as it often requires breaking abstractions enforced by the language designer, and careful reasoning must be done by the developers of the runtime environment to ensure that the guarantees offered by the language are not violated by the runtime environment. For instance, a garbage collector must determine which heap allocations are reachable by following pointers through the heap, starting from a set of *root* pointers. Root pointers typically include the local variables stored on the call stack, and thus the RTE must traverse every stack frame currently stored on the call stack, potentially breaking local state encapsulation [30]. Even worse is the situation from a security perspective, where operations carried out by an RTE might reveal confidential information about the data handled by the user-written programs through storage- or timing channels [24], [29], [32], [37].

Language-based information-flow control (IFC) is a popular approach to solve the problem of ensuring integrity and confidentiality of data [27]. This approach uses programming language techniques to analyze information flows in potentially untrusted programs before and/or during their execution, and verifies that the execution does not leak sensitive information.

In this paper we design and implement Zee: an IFC programming language for implementing secure runtime environments. Zee supports secure and type-safe programming on heterogeneous data (e.g., data at multiple security levels). We also define a faithful calculus of Zee and prove that well-typed programs satisfy timing-sensitive noninterference.

The calculus and its implementation are defined over an abstract *instantiation language*, which allows for Zee to be extended with additional features without redoing many of the formal proofs. Instead, each instantiation must only prove that a certain semantic interface is satisfied.

In summary, this paper makes the following contributions:

- It identifies access to the call stack and runtime type analysis as core features necessary for a practical language for programming runtime environments.
- It presents the design and implementation of Zee: an extensible language for implementing RTEs, and proves that the combination of Zee's type system and runtime semantics enforces timing-sensitive noninterference.
- It uses Zee's extensible semantics to design and implement a secure garbage collector, and a secure thread scheduler.

The rest of the paper is structured as follows: Section II gives an introduction to Zee through examples, and Section III formally defines the syntax and semantics of Zee. Section IV formalizes the attacker model and the security guarantees enforced by Zee's type system. Section V presents two case studies on how Zee can be used to implement well-typed (and hence secure) garbage collectors and thread schedulers. Section VI describes the implementation of Zee, and Section VII discusses related work. The Appendix contains full definitions and the accompanying technical report contains full proofs.

II. Programming in Zee

This section introduces Zee using three example programs. The first example demonstrates how Zee uses existentially quantified type variables to compute securely on values with different security levels located in the same data structure.[1] The second example demonstrates how a similar technique can be used to securely inspect the call-stack at runtime. Finally, the third example demonstrates how Zee uses implicit revocation of expired pointers to prevent reuse of invalidated memory.

[1]This example uses a specific instantiation language defined in Section V-A.

A. Computing on heterogeneous values

The first example uses a simple two-point lattice consisting of two elements $\mathbf{L}$ and H with the partial order $\mathbf{L} \sqsubseteq \mathsf{H}$, and $\mathbf{L} \sqcup \mathbf{L} = \mathbf{L}$ and $\ell_1 \sqcup \ell_2 = \mathsf{H}$ otherwise. We call $\mathbf{L}$ the "public" level and H the "secret" level. We use a level-partitioned heap [24] and so, in the two-point lattice, the heap consists of two partitions that we denote as the $\mathbf{L}$ partition and the H partition. As is standard for type systems for security, types are augmented with security levels, i.e., the type $\mathsf{int}_{\mathsf{H}}$ is the type of integers at security level H.

The type system also tracks which partitions pointers point into. The type $(\ell_1 \mapsto s)_{\ell_2}$ describes pointers that point into the ℓ_1 section, containing data of type s, and where the size of the allocation depends on information up to level ℓ_2.

Zee uses existential types [36] and runtime type analysis [12] for secure handling of heterogeneous data. Traditional existential types are written as $\exists \_ : \mathsf{type}.\ s$, and a value of type $\exists \_ : \mathsf{type}.\ s$ is a pair (τ, v) where τ is a runtime representation of a type,[2] and v is a value of type $s[\tau/\_]$. That is, v is of type s, but where the free type variable $\_$ has been replaced by τ. A value of an existential type can be introduced using the expression $\mathsf{pack}\ (s, e)\ \mathsf{as}\ \exists \_ : \mathsf{type}.\ s$, which evaluates s to τ and e to v before returning a pair (τ, v). Dually, given an expression e of type $\exists \_ : \mathsf{type}.\ s$, it can be eliminated using the command $\mathsf{let}\ (\_ : \mathsf{type}, x : s) := e\ \mathsf{in}\ c$, that brings the type variable $\_$, and the variable x into scope for the evaluation of command c. For information-flow control, the existential type is augmented with security levels, and a value (τ, v) is of type $(\exists \_ : \mathsf{type}_{\ell_1}.\ s)_{\ell_2}$ if τ depends only on information up to level ℓ_1. Introduction and elimination rules are augmented accordingly as $\mathsf{pack}\ (s, e)\ \mathsf{as}\ \exists \_ : \mathsf{type}_{\ell_1}.\ s$, and $\mathsf{let}\ (\_ : \mathsf{type}_{\ell_1}, x : s) := e\ \mathsf{in}\ c$ respectively. We call the latter command an *unpacking* command.

Figure 1 shows a Zee function compute that, given an array xs with elements of different security types, computes the sum of all the public integers in the array, revealing no information about the secret values in xs. The array xs is annotated with the type $(\mathbf{L} \mapsto (\exists \_ : \mathsf{type}_{\mathbf{L}}.\ \_)_{\mathbf{L}})_{\mathbf{L}}$, representing an array of heterogeneous data in the $\mathbf{L}$ partition of public length.

Function compute also specifies two additional labels (both of which are $\mathbf{L}$ in this example): the bottom label is a lower bound on the program counter label, which is classic in IFC literature [21]. The top label is novel: it represents an upper bound on the sensitivity of the information that can be learned by knowing the type of a local variable in the current activation frame. We defer the discussion of this label until Section III-C.

On lines 5 to 9, function compute loops over the elements of xs, and on lines 6 and 7 it extracts the witness $\_$ and the value of type $\_$. On line 8 the code performs runtime type analysis on the value of $\_$, and it matches the pattern $\mathsf{int}_{\mathbf{L}}$ (i.e., the type of public integers). In this branch on line 8 the value y is known to be of type $\mathsf{int}_{\mathbf{L}}$, and can be added to the public variable sum, without leaking sensitive information. Finally,

[2]The value τ is often called the *witness* of the type s.

```
1  compute(xs : (L? (9 :type_L . _)_L)_L) =^L_L
2    let n : int_L := length xs in
3    let i : int_L := 0 in
4    let sum : int_L := 0 in
5    while i < n do
6      let x : (9 :type_L . _)_L := *(xs + i) in
7      let (_ : type_L, y : _) := unpack x in
8      match _ with int_L ) sum := sum + y
9                  | _     ) skip;
10     i := i + 1
```

Fig. 1: Zee program demonstrating computations on heterogeneous values.

if $\_$ is of any other type, this value is omitted from the final sum.

This example demonstrates how Zee securely computes on heterogeneous values using existential types and runtime type analysis. In the next example, we extend such use of existential types to access the call stack while guaranteeing type-safety and security. We do this by treating the frame pointer as a pointer to an array of existentially quantified types, similar to the type of xs in Figure 1. For the remaining examples in the paper we elide some type annotations, but all of the missing type annotations are inferred by our prototype implementation of Zee.

B. Computing on the call-stack

Zee allows fine-grained reasoning about the call stack, which is important for operations such as stack traversal for many garbage collection algorithms, or unwinding the stack to handle exceptions. Figure 2 shows the structure of the call stack during an execution with two activation frames belonging to functions g and f respectively. Function g pushes two arguments on the stack: a value of type $(\mathsf{H} \mapsto \mathsf{int}_{\mathsf{H}})_{\mathsf{H}}$, and a value of type $\mathsf{int}_{\mathsf{H}}$, and then invokes f. Function f then establishes a new frame by pushing the value of the old frame pointer onto the stack, so that the previous frame can be restored upon returning from f. Furthermore, f modifies the frame pointer to point to its local variables, and modifies the stack pointer to point to the next free address on the stack. Finally, f allocates its local variables and proceeds with computation in the newly established activation frame.

As Zee features runtime type analysis, which is crucial for the implementation of many useful programming language features, the types themselves must be protected using security labels. To accomplish this, we introduce the notion of a frame label fr, which represents an upper bound on the sensitivity of the information that can be learned by knowing the type of a local variable in the current frame.

In Zee the expression FP returns a pointer to the beginning of the current activation frame. The type assigned to FP is a recursive type,[3] that reflects the layout of the stack (cf. Figure 2) and contains the types of the function parameters and local variables.

[3]We will explain the precise typing of the frame pointer in detail in Section III-C.

```
1      f(p : (H ?? int_H)_H, h : int_H) =^H_L
2        let a : int_L := 0 in
3        let b : (L ?? int_H)_L := null in
4        let c : int_H := h in skip
5      g() =^H_L ...; f(null, 42); ...
6
```

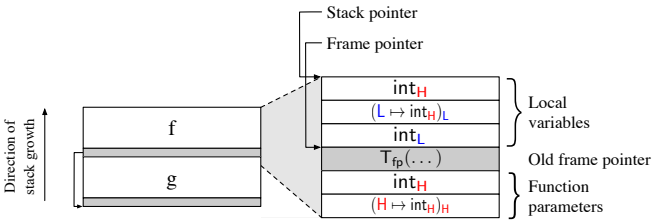


Fig. 2: Bottom: the structure of the call stack just before executing skip in f. Top: two functions g and f.

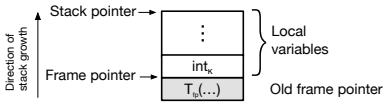
To understand the need for the frame label, consider the program in Figure 3, showing a function inspect that inspects its own frame. Unlike the compute function, the inspect function is parameterized by a secret label , which the caller provides when invoking inspect. In this example, the first element of the frame layout consists of an integer x with the security label , as declared on line 2. Using the FP construct on line 3, the type variable $_{\mathsf{args}}$ is assigned a tuple type representing the types of the arguments passed to inspect,[4] and on line 4, $_{\mathsf{locals}}$ is assigned the types of the local variables of inspect.

As no arguments has been passed to inspect, the value of $_{\mathsf{args}}$ during execution is the unit type ε. The value of $_{\mathsf{locals}}$ is $\mathsf{int}_{\ell'} \cdot \tau_{\mathsf{tail}}$, where ℓ' is the value of that has been provided when inspect was called, and τ_{tail} is the rest of the frame layout. After obtaining the type of the local variables, line 5 performs a pattern match on $_{\mathsf{locals}}$, and if the pattern $\mathsf{int}_{\ell} \cdot \_$ matches the value of $_{\mathsf{locals}}$, it follows that ℓ' is equal to ℓ, which is information classified at H. So to track the indirect flow from the label to the match command, we introduce the frame label *fr*, which assigns $_{\mathsf{locals}}$ the security level *fr* (which is equal to H as declared by the inspect function), properly tracking the dependency of in $_{\mathsf{locals}}$.

C. Fail-stop revocation of expired pointers

This section demonstrates how Zee uses a dynamic enforcement technique to prevent the reuse of pointers that point to "expired" data. Expired data include local variables of functions that have returned control to its caller, or heap data that have been reclaimed by a garbage collector. To do this we introduce a technique similar to identifier-based temporal checking [22] which we simply refer to as *versioning*. At runtime, every pointer is assigned a natural number ν called the version number. Similarly, every stack frame is assigned a version number. When a pointer to a local variable on a stack frame is created, the pointer is assigned the version number of that stack frame. When reading data pointed to by a pointer with

[4]The unroll expression is needed as Zee uses isorecursive types to represent the type isomorphism between μ : type. s and $s[\mu$: type. $s/$] (i.e., between a recursive type and its unrolling). We will omit unroll expressions throughout the paper.




```
1 inspect_h : level_H i() : =^H_L
2   let x : int  := 42 in
3   let ( _args, e) := unpack (unroll FP) in
4   let ( _locals, _) := unpack e in
5   match _locals with int_ℓ * _ )  ... //   = ℓ
6                    | _          )  skip
```

Fig. 3: Bottom: Indirect information flow from to $_{\mathsf{locals}}$ when inspect inspects its own stack frame. Top: The activation frame for function inspect.

version ν, the system checks that the stack frame, which is read from, has a version number γ such that $\gamma \leq \nu$, ensuring that this stack frame was "live"[5] when the pointer was created, and a similar check is done for writes through pointers.

Figure 4 demonstrates how an illegal flow could be constructed without versioning: on line 6, x is allocated on the stack and initialized to null.[6] Then, function f is called and writes the address of its local variable y into x through the passed pointer *px* on line 2. When f returns, the value of x is a pointer, pointing to the local variable y from the "dead" stack frame of f. Then, when g is called, the value of `secret` is stored in h, which might be located in the same offset from the base pointer as y was in function f (cf. lower part of Figure 4). So when g reads x, the value of `secret` is stored in `low`, violating the type ascribed to the variable.

Versioning prevents this leak by assigning a version number $\nu \in \mathsf{N}$ to x and the stack frame allocated on line 6. The stack frames of f and g are assigned versions $\nu + 1$ and $\nu + 2$ respectively. When g reads from x, this variable has version $\nu + 1$, while the data it points to is located on g's stack frame, which has version $\nu + 2$. This violates the version check and execution stops, successfully preventing the leak.

III. Language

This section presents a formalization of Zee. We assume a lattice $\mathcal{L}$ of security levels with a least element $\bot$ and let ℓ range over the elements of $\mathcal{L}$.

A. Syntax

Figure 5 defines the grammar for Zee. We give an informal description of each syntactic category, and delay the formal semantics until Section III-B.

1) Commands: Commands c include standard constructs for imperative languages. Non-standard commands include commands $*e := e$ and $x := *e$ that respectively write to, and read from, a location on the stack. Command at k e c raises the program counter label for command c during type checking,

[5]A live stack frame refers to a frame of a computation that is still ongoing.

[6]For now the type @ s can be read as "pointer to a value of type s", but we will define a more general version of this type in Section III.

```
1 f(px : (@(@ int_L)_L)_L) =^L_L
2   let y : int_L := 0 in *px := &y
3 g(z : (@ int_L)_L) =^L_L
4   let h : int_H := secret in
5   let low : int_L := *z in skip
6 let x : (@ int_L)_L := null in f(&x); g(x)
```

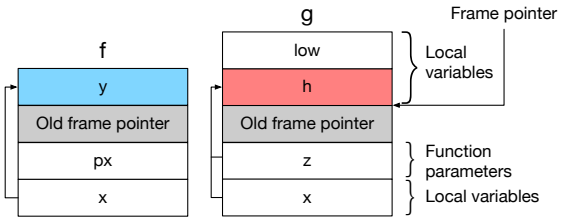


Fig. 4: Attempting to read x in function g will fail: The pointer has been implicitly revoked when f no longer was live.

and also provides lightweight[7] predictive mitigation [4], [41], [42]. The match command allows for runtime type analysis [1], [12], which is crucial for implementing many useful tasks associated with the RTE. We call the being matched on the *scrutinee*, and the patterns are defined in the syntactic category p. These include integer type patterns (int), stack pointer type patterns ((p @ p)), heap pointer type patterns (($\mapsto p$)), product type patterns ($\overline{p}$) and a "catch all" pattern that names the type value ( ). To facilitate traversing the stack at runtime, the language exposes the frame pointer, which can be obtained using the $x :=$ fp command. A function can be invoked by supplying the function name with label[8] parameters $\overline{k}$, type parameters $\overline{s}$, and expression parameters $\overline{e}$. Finally, both existentially quantified labels and types can be eliminated using the unpacking commands let ($: \mathcal{X}_k, x : s) := e$ in c for $\mathcal{X} \in \{\mathsf{type}, \mathsf{level}\}$.

2) Expressions: The meta-variable e ranges over expressions which include numbers, variables, binary operations, a special pointer value null, and pack expressions for introducing existentially quantified labels and types. We include recursive types to give the language type-safe access to the call stack, and expressions unroll e and roll e allow for simple (i.e., isorecursive) typing rules for recursive types [26]. The size of a type can be calculated using the expression sizeof s, which is also used to facilitate stack traversal. Finally, given a variable x the address of x on the stack can be obtained as $\&x$.

3) Security labels and types: The meta-variable k ranges over security labels. As labels can be existentially and universally quantified, the category includes variables . Finally, one can form the join $\sqcup$ or meet $\sqcap$ of two security labels, representing the least upper bound, or greatest lower bound of two labels respectively.

The meta-variable s ranges over security types and include base types t with a security label k, written t_k, type variables , or product types $\overline{s}$. Base types are integers, heap pointers ($k \mapsto s$) representing a pointer into the heap partition associated

[7]In particular, the predicted time for mitigation commands is given by the programmer, nor is it not automatically updated by the semantics.

[8]We distinguish between levels (i.e., elements of $\mathcal{L}$) and labels (i.e., expressions that evaluate to elements of $\mathcal{L}$)

$$
\begin{array}{rcl}
c & ::= & \mathsf{skip} \mid \mathsf{let}\ x : s := e\ \mathsf{in}\ c \mid \mathsf{if}\ e\ c\ c \\
 & \mid & \mathsf{while}\ e\ c \mid c; c \mid x := e \mid *e := e \\
 & \mid & x := *e \mid \mathsf{at}\ k\ e\ c \mid \mathsf{if}\ (k \sqsubseteq k)\ c\ c \\
 & \mid & \mathsf{match}\ \ \overline{p \Rightarrow c} \mid x := \mathsf{fp} \mid f\langle\overline{k}\rangle\langle\overline{s}\rangle(\overline{e}) \\
 & \mid & \mathsf{let}\ (\ \ : \mathsf{type}_k, x : s) := e\ \mathsf{in}\ c \\
 & \mid & \mathsf{let}\ (\ \ : \mathsf{level}_k, x : s) := e\ \mathsf{in}\ c \\
e & ::= & n \mid x \mid e \oplus e \mid \mathsf{null} \mid \mathsf{unroll}\ e \mid \mathsf{roll}\ e \\
 & \mid & \mathsf{pack}\ (s, e)\ \mathsf{as}\ \exists\ \ : \mathsf{type}_k.\ s \mid \mathsf{sizeof}\ s \\
 & \mid & \mathsf{pack}\ (k, e)\ \mathsf{as}\ \exists\ \ : \mathsf{level}_k.\ s \mid \&x \\
k & ::= & \ell \mid\ \ \mid k \sqcup k \mid k \sqcap k \\
s & ::= & t_k \mid\ \ \mid \overline{s} \\
t & ::= & \mathsf{int} \mid k \mapsto s \mid s\ @\ s \mid \exists\ \ : \mathsf{type}_k.\ s \\
 & \mid & \exists\ \ : \mathsf{level}_k.\ s \mid \mu\ \ : \mathsf{type}_k.\ s \mid \mathsf{size}[s] \\
p & ::= & \mathsf{int}\ \mid (p\ @\ p)\ \mid (\ \ \mapsto p)\ \mid \overline{p}\ \mid
\end{array}
$$

Fig. 5: The syntax of Zee.

with label k [24], stack pointers (s_1 @ s_2) representing pointers to a value of type s_2 that, on the stack, are located "above" a value of type s_1 [2], [25]. Base types also include the type of existentially quantified security types and labels, recursive types and singleton types [34] size$[s]$ describing the size of the type s. A function definition is given as $f\langle\overline{\ \ : \mathsf{level}_{k_1}}\rangle\langle\overline{\ \ : \mathsf{type}_{k_2}}\rangle(\overline{x : s}) =^{\mathit{fr}}_{\mathit{pc}} c$ that defines a function f with label parameters , type parameters , and value parameters $\overline{x}$. Label parameter $_i$ can depend on the previous label parameters $_1, \ldots, _{i-1}$, and type parameter $_i$ can depend on all label parameters and type parameters $_1, \ldots, _{i-1}$. Finally, the types $\overline{s}$ for the expression parameters can depend on all label and type parameters. Furthermore, pc is a lower bound on the side effects produced by f, and fr is an upper bound on the type of any local variable declaration. We revisit this label in Section III-C.

B. Semantics

The semantics of Zee is given by a small-step relation $\rightarrow$ on configurations C of the form $\langle c, M, P, q\rangle_\nu$. We first define each component of the configuration before describing the small-step relation.

1) Values: Figure 7 describes the syntax of values. A value v is either a number n, an address a with a *version number* $\nu \in \mathsf{N}$, a pair consisting of either a level and a value (ℓ, v), or a security type value and a value (τ, v). The meta-variable τ ranges over security type values and is either a base type value π with a security level ℓ, or a product of security type values.

Security type values also include a *nonsense* [19] type value . To motivate the need for nonsense type values, consider the program in Figure 6. When evaluating FP on line 3, the variable x with value 0 is in scope, but none of the variables e, p or y have entered the scope, and their value on the stack are "garbage" values. So $_{\mathsf{locals}}$ is a product type $\mathsf{int}_\mathsf{L} \cdot \ \cdot \ \cdot$ containing three nonsense type values as the variables e, p and y have yet to enter the scope and be assigned a value. The type variables $_{\mathsf{args}}$ and $_{\mathsf{locals}}$ live on a different stack (cf. top right of Figure 6) where extracting type-information is not possible, and therefore technically do not need a nonsense

```
1 let x : int_L := 0 in
2 let ( _args, e) :=
3   unpack FP in
4 let ( _locals, p) :=
5   unpack e in
6 let y : int_H := 0 in
7 skip
```

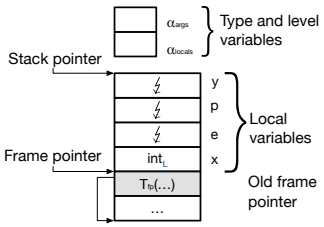


Fig. 6: Left: the value of $_{\mathsf{locals}}$ contains three nonsense type values, corresponding to the three variables e, p and y that has yet to enter the scope upon evaluating FP on line 3. Right: the stack layout when FP is evaluated.

$$
\begin{array}{rcl}
v & ::= & n \mid a_\nu \mid (\ell, v) \mid (\tau, v) \\
\tau & ::= & \pi_\ell \mid \overline{\tau} \mid \\
\pi & ::= & \mathsf{int} \mid \ell \mapsto \tau \mid \tau \mathbin{@} \tau \mid \exists \ \ : \mathsf{type}.\ s \\
 & \mid & \exists \ \ : \mathsf{level}.\ s \mid \mu \ \ : \mathsf{type}.\ s \mid \mathsf{size}[\tau]
\end{array}
$$

Fig. 7: Values in Zee.

type value, as we do not track "the type of type variables" at runtime.

The meta variable π ranges over base type values, and contain the same constructs as the base types, but where security labels not under binders $\exists$ and μ are fully evaluated, and labels on types and labels are erased (as they are only needed for type checking).

2) Exposed and private stack frames: An exposed stack frame m is a triple $(\mathsf{I}, |m|, \nu)$ where $|m| : \mathsf{I} \rightharpoonup v$ is a partial function from a nonempty interval $\mathsf{I} \subset \mathsf{N}$ to values, and a frame version $\nu \in \mathsf{N}$. We call $\min(\mathsf{I})$ the frame pointer, written $\mathsf{fp}(m)$, and $\max(\mathsf{I})+1$ the stack pointer, written $\mathsf{sp}(m)$. Intuitively, $\mathsf{fp}(m)$ is the minimum address in the stack frame, corresponding to the usual notion of a frame pointer, and similarly $\mathsf{sp}(m)$ is the address of the next available stack location. Given an exposed stack frame $m = (\mathsf{I}, |m|, \nu)$ we write $m[a \mapsto v]$ to mean $(\mathsf{I}, |m|[a \mapsto v], \nu)$, and $m(a)$ to mean $|m|(a)$. We call a list of exposed stack frames an exposed stack M.

A private stack frame p is a triple of partial functions $(p_{\mathsf{var}}, p_{\mathsf{arg}}, p_{\mathsf{local}})$ where $p_{\mathsf{var}} : \mathsf{Var} \rightharpoonup \mathcal{L} \cup \tau$ and $p_{\mathsf{arg}}, p_{\mathsf{local}} : \mathsf{Var} \rightharpoonup \tau$. Function p_{var} maps label- and type variables to levels and type values, p_{arg} maps function arguments to their security type, and p_{local} maps local variable names to their security type. The name "private" refers to the fact that this stack cannot be traversed and inspected at runtime (unlike the exposed stack). Finally, a private stack P is a list of private stack frames. We call a pair of an exposed and a private stack (M, P) a stack.

We distinguish between exposed and private stacks because Zee allows type-safe traversal of the exposed stack, but does not directly expose the private stack to programs.

As exposed stack frames include mappings from (subsets of) natural numbers, a translation from local variable names to addresses is needed. This translation is usually performed by a compiler, and many techniques exist for such translations

```
1 fib(n : int_L, r : (@ int_L)_L) =^L_L
2   if n <= 1 then *r := n
3   else let r1 : int_L := 0 in
4        let r2 : int_L := 0 in
5        fib(n-1, &r1); fib(n-2, &r2);
6        *r := r1 + r2
```

$$
\delta(x) = \begin{cases} 2 & \text{if } x = r_2 \\ 1 & \text{if } x = r_1 \\ -1 & \text{if } x = r \\ -2 & \text{if } x = n \\ \bot & \text{otherwise} \end{cases}
$$

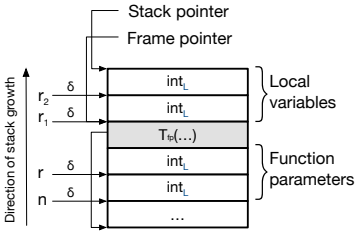


Fig. 8: Top: an implementation of a function `fib` for computing a public Fibonacci number given a public input. Bottom right: the stack frame layout for fib. Bottom left: the mapping δ between local variable names and stack frame offsets.

[3]. We abstract away the specifics of such translations by assuming a global mapping $\delta : \mathsf{Var} \to \mathsf{N}$ from variables names to offsets. Figure 8 demonstrates how a compiler might generate a mapping δ that maps variable names to activation frame offsets. At the moment we assume that the compiler does not coalesce stack locations when the lifetimes of two variables do not overlap. So given two local variables $x \neq y$ it holds that $\delta(x) \neq \delta(y)$. That is, different variables are stored at different locations on the stack. The expression $\delta(x) + \mathsf{fp}(m)$ computes the address of the local variable x in the stack frame $\mathsf{fp}(m)$.

3) Model of time: We represent time as a number $q \in \mathsf{N}$ that counts the number of operational steps in the computation. This simple model is sufficient to demonstrate that runtime environment tasks can be computed in a timing-sensitive security setting. Naturally, a realistic implementation would need to soundly relate the operational steps with the wall-clock, but that is outside of the scope of the current work.

4) Version counter: Finally, the configuration contains a version counter that keeps track of the next free version number. This is needed when constructing new stack frames, as each new frame is given a fresh version number.

5) Big-step evaluation for expressions: The big-step evaluation for expressions is defined on configurations of the form $\langle e, m, p \rangle$, where m is an exposed stack frame and p is a private stack frame. The evaluation of expressions need both the exposed, and the private stack frame, as both expression variables and type- and level variables might appear in expressions. Figure 9 shows excerpts of the big-step evaluation semantics for expressions. Rule E-NUM evaluates a literal, and E-VAR evaluates a variable by looking up its value on the stack using the global mapping δ. Finally, rule E-PACK-TY evaluates a pack expression containing a security type s and an expression e to a pair (τ, v). The remaining rules are found in the technical appendix [23].

6) Small-step relation for commands: Figure 10 shows an excerpt of the small-step relation, and the full semantics is

E-NUM
$$\langle n, m, p\rangle \Downarrow n$$

E-VAR
$$\frac{m(\delta(x) + \mathsf{fp}(m)) = v}{\langle x, m, p\rangle \Downarrow v}$$

E-PACK-TY
$$\frac{\langle s, p\rangle \Downarrow_{\mathsf{type}} \tau \qquad \langle e, m, p\rangle \Downarrow v}{\langle \mathsf{pack}\ (s, e)\ \mathsf{as}\ \exists\ \ :\mathsf{type}_k.\ s', m, p\rangle \Downarrow (\tau, v)}$$

Fig. 9: Excerpts of the big-step evaluation of expressions.

in Figure 22 in the Appendix. Rule S-ASGN evaluates an expression e and inserts the resulting value v in the memory at offset $\delta(x)$ of the current frame pointer. Rule S-FP stores the frame pointer in the variable x. In addition to the value of the frame pointer $\mathsf{fp}(m)$, the value v contains the the list of types $\mathsf{cod}(p.\mathsf{arg})$ of the arguments passed to the current function, and the list of types $\mathsf{cod}(p.\mathsf{local})$ of the local variables. Rule S-LET declares a new local variable x and (1) updates the stack location to contain the initial value of x, and (2) updates the private stack frame to contain type information about the type of x. This causes the type value of the variable x to be updated from a nonsense type value to a meaningful type value τ, which is the result of evaluating the type s. After executing c, a command $\mathsf{unscope}(x)$ removes the type information of x from the private stack p.[9] Rule S-UNPACK-TY unpacks an existential value containing a security type value and a regular value. Several maps are updated: the private stack frame is updated to contain the security type value and the type information about the newly allocated local variable. Finally, the exposed stack frame is updated to contain the regular value. Rule S-MATCH evaluates the scrutinee to a security type value and performs type analysis according to a list of patterns $\overline{p}$ using the relation τ - p containing rules such as int_ℓ - int and τ - [10] (i.e., integer patterns matches integer types, and name patterns matches any security type value). Upon finding the first (due to argmin) matching pattern the private stack frame is updated by binding relevant parts of the security type value to type variables, and execution proceeds with the command associated with the pattern.

Rule S-READ reads from a stack location a_γ. The version number γ is used to prevent attacks based on pointer reuse as was described in Section II-C. Dually, S-WRITE writes a value v to an address a, requiring the same relation between the versions of the target stack frame and the address being read from.

Rules S-AT and S-DELAY implement simple predictive mitigation of direct timing channels, i.e., channels represented directly in the control-flow of the program: S-AT reduces to the underlying command c, but ensures that the command terminates in exactly n steps, where n is the result of evaluating expression e, by delaying further commands until the command delay n has terminated.

[9]The semantics of unscope is defined in the Appendix.

[10]The complete definition of matching is found in the Appendix.

S-ASGN
$$\frac{\langle e, m, P\rangle \Downarrow v \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v]}{\langle x := e, m \cdot M, P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, P, q+1\rangle_\nu}$$

S-FP
$$\frac{v = (\mathsf{cod}(p.\mathsf{arg}), (\mathsf{cod}(p.\mathsf{local}), \mathsf{fp}(m)_\nu)) \qquad m = (\mathsf{I}, |m|, \nu) \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v]}{\langle x := \mathsf{fp}, m \cdot M, p \cdot P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, p \cdot P, q+1\rangle_\nu}$$

S-LET
$$\frac{\langle s, p\rangle \Downarrow_{\mathsf{type}} \tau \qquad \langle e, m, p\rangle \Downarrow v \qquad c' = c; \mathsf{unscope}(x) \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v] \qquad p' = p[p_{\mathsf{local}} \mapsto p_{\mathsf{local}}[x \mapsto \tau]]}{\langle \mathsf{let}\ x : s := e\ \mathsf{in}\ c, m \cdot M, p \cdot P, q\rangle_\nu \to \langle c', m' \cdot M, p' \cdot P, q+1\rangle_\nu}$$

S-UNPACK-TY
$$\frac{\langle e, m, p\rangle \Downarrow (\tau_1, v_2) \qquad \langle s, p'\rangle \Downarrow_{\mathsf{type}} \tau \qquad p' = p[\mathsf{var} \mapsto p.\mathsf{var}[\ \ \mapsto \tau_1]] \qquad p'' = p'[\mathsf{local} \mapsto p'.\mathsf{local}[x \mapsto \tau]] \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v_2]}{\langle \mathsf{let}\ (\ \ :\mathsf{type}_k, x : s) := e\ \mathsf{in}\ c, m \cdot M, p \cdot P, q\rangle_\nu \to \langle c; \mathsf{unscope}(x), m' \cdot M, p' \cdot P, q+1\rangle_\nu}$$

S-READ
$$\frac{m_i = (\mathsf{I}, |m_i|, \nu_i) \in m \cdot M \qquad a \in \mathsf{I} \qquad \nu_i \le \gamma \qquad \langle e, m, P\rangle \Downarrow a_\gamma \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto m_i(a)]}{\langle x := *e, m \cdot M, P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, P, q+1\rangle_\nu}$$

S-WRITE
$$\frac{m_i = (\mathsf{I}, |m_i|, \nu) \qquad M = M_1 \cdot m_i \cdot M_2 \qquad a \in \mathsf{I} \qquad \nu_i \le \gamma \qquad \langle e_1, M, P\rangle \Downarrow a_\gamma \qquad \langle e_2, M, P\rangle \Downarrow v \qquad m_i' = m_i[a \mapsto v]}{\langle *e_1 := e_2, M, P, q\rangle_\nu \to \langle \mathsf{stop}, M_1 \cdot m_i' \cdot M_2, P, q+1\rangle_\nu}$$

S-AT
$$\frac{\langle e, M, P\rangle \Downarrow n}{\langle \mathsf{at}\ k\ e\ c, M, P, q\rangle_\nu \to \langle c; \mathsf{delay}\ n, M, P, q+1\rangle_\nu}$$

S-DELAY
$$\frac{n \le q}{\langle \mathsf{delay}\ n, M, P, q\rangle_\nu \to \langle \mathsf{delay}\ n, M, P, n+1\rangle_\nu}$$

S-MATCH
$$\frac{\underset{i=1,\ldots,n}{\mathrm{argmin}}(\tau\ \text{-}\ p_i) = j \qquad \langle\ \ , p\rangle \Downarrow_{\mathsf{type}} \tau \qquad \mathsf{J}p_j\mathsf{K}(p, \tau) = p'}{\langle \mathsf{match}\ \ \ (p_i \Rightarrow c_i)_{i=1,\ldots,n}, M, p \cdot P, q\rangle_\nu \to \langle c_j, M, p' \cdot P, q+1\rangle_\nu}$$

Fig. 10: Semantics of Zee: commands.

C. Type system

We now describe a type system for Zee, which we will show ensures secure information-flow in Section IV. The type system integrates previous work on type safety in stack-based languages [2], [25] with dynamic security labels [44] and existential types for information-flow control [36] into a single language that is able to express complex indirect data dependencies. The typing judgment for commands has the form $\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash c$, and the typing judgment for expressions has the form $\Gamma, \Pi, \phi \vdash e : s$. We explain each component of the judgment before presenting the judgment rules.

Function $\Gamma : \mathsf{Var} \rightharpoonup s$ maps regular variable names to security types, and similarly $\Pi : \mathsf{Var} \rightharpoonup \{\mathsf{type}, \mathsf{level}\} \times k$ maps type variables to type_k and label variables to level_k. Formula ϕ is a finite conjunction of (possibly negated) flow relations such as $\ _1 \sqcup \ _2 \sqsubseteq \ _3 \wedge \ _3 \not\sqsubseteq \ _4$. These formulas are gathered during type-checking in such a way that the constructed formulae always represents flows that will be true at runtime. Adding such formulae to the typing relation improves the expressiveness of static information-flow control in the presence of dynamic security labels [16], [43], [44]. Finally, the type system tracks two labels: the program counter label pc and the frame label fr. We now explain the typing judgments involved in typing Zee programs.

1) Typing judgment for expressions: Figure 11 shows excerpts of the typing rules for expressions. Rules T-NUM and T-VAR are standard rules for literals and variables, and T-PACK-TY is the standard rule for introducing an existentially quantified type [36]. Finally, rule T-SIZEOF assigns a singleton-type $\mathsf{size}[s]$ to an expression $\mathsf{sizeof}\ s$, representing that the expression will evaluate to the size of the type s at runtime. Such expressions are crucial for secure and type-safe operations in a language with heterogeneous data like in Zee: they allow the type system to track how pointer arithmetic changes the type of the pointer. This becomes clear in rule T-BINOP, which assigns types to the result of binary expressions using the relation $s_1 \ \mathsf{J}\oplus\mathsf{K}\ s_2 \rightharpoonup s$. This states that applying operator $\oplus$ to expressions of type s_1 and s_2 results in an expression of type s. The full judgment is shown in the Appendix, and excerpts of this relation include

$$\mathsf{int}_{k_1} \ \mathsf{J}\oplus\mathsf{K}\ \mathsf{int}_{k_2} \rightharpoonup \mathsf{int}_{(k_1 \sqcup k_2)} \tag{1}$$

$$(s_1 \ @\ s \cdot s_2)_{k_1} \ \mathsf{J}+\mathsf{K}\ \mathsf{size}[s]_{k_2} \rightharpoonup (s_1 \cdot s \ @\ s_2)_{(k_1 \sqcup k_2)} \tag{2}$$

$$(s_1 \cdot s \ @\ s_2)_{k_1} \ \mathsf{J}-\mathsf{K}\ \mathsf{size}[s]_{k_2} \rightharpoonup (s_1 \ @\ s \cdot s_2)_{(k_1 \sqcup k_2)} \tag{3}$$

In words, performing a binary operation on two integers results in an integer labeled with the join of the two labels (1), adding the size of a type s to a pointer pointing to a value of type $s_1 \ @\ s \cdot s_2$ results in a pointer to a value of type $s_1 \cdot s \ @\ s_2$, and the labels are raised accordingly. Dually, one can subtract the size of a type s from a pointer of type $s_1 \cdot s \ @\ s_2$, and obtain a value of type $s_1 \ @\ s \cdot s_2$.

2) Typing judgment for commands: Figure 12 shows excerpts of the typing rules for commands. Rule T-LET states that a variable declaration $\mathsf{let}\ x : s := e\ \mathsf{in}\ c$ is well-typed if the type of e is a subtype of s. The subtype relation is standard for imperative languages for information-flow [21]. To prevent implicit flows, the program counter label should also flow to s. Finally, x is added to the typing context Γ, and the frame label fr is raised to reflect the fact that the frame layout has been influenced by the variable declaration.

Rule T-IF states that a command $\mathsf{if}\ e\ c_1\ c_2$ is well-typed when e is an expression of type $\mathsf{int}_{\mathit{pc}}$, and both branches can be shown to be well-typed. Readers familiar with IFC type systems may wonder why it is necessary to restrict the label on the type of e to be pc. This is done to facilitate predictive mitigation of direct timing channels: the pc must be explicitly

$$\text{T-NUM}\quad \frac{}{\Gamma, \Pi, \phi \vdash n : \mathsf{int}_\bot} \qquad \text{T-VAR}\quad \frac{}{\Gamma, \Pi, \phi \vdash x : \Gamma(x)}$$

$$\text{T-PACK-TY}\quad \frac{\Pi, \phi \vdash_{\mathsf{type}} s_2 \qquad \Gamma, \Pi, \phi \vdash e : s_2[s_1/x] \qquad \Pi, \phi \vdash_{\mathsf{type}} s_1 : k_1 \qquad t = \exists x : \mathsf{type}_{k_1}.\ s_2}{\Gamma, \Pi, \phi \vdash \mathsf{pack}\ (s_1, e)\ \mathsf{as}\ t : t_\bot}$$

$$\text{T-SIZEOF}\quad \frac{\Pi, \phi \vdash_{\mathsf{type}} s : k}{\Gamma, \Pi, \phi \vdash \mathsf{sizeof}\ s : \mathsf{size}[s]_k} \qquad \text{T-BINOP}\quad \frac{\Gamma, \Pi, \phi \vdash e_i : s_i \qquad s_1 \ \mathsf{J}\oplus\mathsf{K}\ s_2 \rightharpoonup s}{\Gamma, \Pi, \phi \vdash e_1 \oplus e_2 : s}$$

Fig. 11: Excerpts of the typing rules for expressions.

raised using an at command. Rule T-AT states that a command $\mathsf{at}\ k\ e\ c$ is well-typed when the label k and the computation time e only depends on information up to pc. Furthermore, the command must not lower the program counter label, and the command c must be well-typed under the raised program counter label k.

The command $x := \mathsf{fp}$ is well-typed when x is a subtype of the type $\mathsf{T}_{\mathsf{st}}(\mathit{pc}, \mathit{fr}, k)$, which abbreviates the type

$$(\mu\ \ : \mathsf{type}_k.\ (\exists\ \ : \mathsf{type}_{\mathit{fr}}.\ (\exists \gamma : \mathsf{type}_{\mathit{fr}}.\ (\ \ \cdot\ \ @\ \gamma)_{\mathit{pc}})_\bot)_\bot)_\bot$$

This type reflects the layout of the stack at runtime (cf. Figure 2). Each frame consists of some type representing the type of the arguments given to the function, followed by a pointer to the previous stack frame (which is represented as the recursive type variable), and finally the type γ representing the types of the local variables. By assigning each existentially quantified type the label fr we ensure that no type leaks information, as fr represents the upper bound of the information that can be learned from knowing the value of the types.

Rule T-MATCH states when a command $\mathsf{match}\ \ \overline{p \Rightarrow c}$ is well-typed. First, must be a type variable with a label that flows to pc, as the direct timing channels must be controlled using at commands. Judgment $\Pi \vdash p_i\ \ _k\ \Pi_i : s_i$ generates a type variable environment Π_i for type-checking the command for the i'th pattern, and the type s_i to assign in the command. Furthermore, any type and label variable bound by the new typing type variable environment Π_i is bound to the label k. Most of the rules of this judgment are of the form

$$\Pi \vdash \mathsf{int}\ \ _k\ \Pi[\ \mapsto \mathsf{level}_k] : \mathsf{int}$$

which expresses that, in environment Π, when the pattern is int and the scrutinee depends on information up to label k, the environment is updated to $\Pi[\ \mapsto \mathsf{level}_k]$ and the type of the scrutinee can be assumed to have type int in the command guarded by the pattern int . The full judgment can be found in the technical report. Finally, each command c_i is type-checked in the generated type- and variable environment.

Rule T-UNPACK-TY states when an elimination of an

T-LET

$$\frac{\Gamma, \Pi, \phi \vdash e : r \qquad \Pi, \phi \vdash_{\mathsf{type}} s : k \qquad \phi \vdash r^{\mathit{pc}} <: s \qquad \mathit{fr}' = \mathit{fr} \sqcup k \qquad \Gamma[x \mapsto s], \Pi, \phi, \mathit{pc}, \mathit{fr}' \vdash c}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{let}\ x : s := e\ \mathsf{in}\ c}$$

T-AT

$$\frac{\Pi; \phi \vdash_{\mathsf{lab}} k : \mathit{pc} \qquad \Gamma, \Pi, \phi \vdash e : \mathsf{int}_{\mathit{pc}} \qquad \phi \vdash \mathit{pc} \sqsubseteq k \qquad \Gamma, \Pi, \phi, k, \mathit{fr} \vdash c}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{at}\ k\ e\ c}$$

T-IF

$$\frac{\Gamma, \Pi, \phi \vdash e : \mathsf{int}_{\mathit{pc}} \qquad \Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash c_i \ \ i = 1, 2}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{if}\ e\ c_1\ c_2}$$

T-FP

$$\frac{\Pi; \phi \vdash_{\mathsf{lab}} \mathit{fr} : k \qquad \phi \vdash \mathsf{T}_{\mathsf{st}}(\mathit{pc}, \mathit{fr}, k)^{\mathit{pc}} <: \Gamma(x)}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash x := \mathsf{fp}}$$

T-MATCH

$$\frac{\Pi(\ \ ) = \mathsf{type}_k \qquad \phi \vdash k \sqsubseteq \mathit{pc} \qquad \Pi \vdash p_i \quad {}_k\, \Pi_i : s_i \qquad \Gamma[s_i/\ \ ], \Pi_i[s_i/\ \ ], \phi, \mathit{pc}, \mathit{fr} \vdash c_i[s_i/\ \ ]}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{match}\quad \overline{p \Rightarrow c}}$$

T-UNPACK-TY

$$\frac{\Gamma, \Pi, \phi \vdash e : (\exists\ \ : \mathsf{type}_{k_1}.\ r)_{\mathit{pc}} \qquad \phi \vdash r^{\mathit{pc}} <: s \qquad \Gamma' = \Gamma[x \mapsto s] \qquad \Pi' = \Pi[\ \ \mapsto \mathsf{type}_{k_1}] \qquad \Pi', \phi \vdash_{\mathsf{type}} r : k_2 \qquad \mathit{fr}' = \mathit{fr} \sqcup k_1 \sqcup k_2 \qquad \Gamma', \Pi', \phi, \mathit{pc}, \mathit{fr}' \vdash c}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{let}\ (\ \ : \mathsf{type}_{k_1}, x : s) := e\ \mathsf{in}\ c}$$

T-FLOWSTO

$$\frac{\Pi; \phi \vdash_{\mathsf{lab}} k_i : \mathit{pc} \qquad \Gamma, \Pi, \phi \wedge k_1 \sqsubseteq k_2, \mathit{pc}, \mathit{fr} \vdash c_1 \qquad \Gamma, \Pi, \phi \wedge k_1 \not\sqsubseteq k_2, \mathit{pc}, \mathit{fr} \vdash c_2}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \mathsf{if}\ (k_1 \sqsubseteq k_2)\ c_1\ c_2}$$

Fig. 12: Excerpts of the typing rules for commands.

existentially quantified type is well-typed. The rule follows previous work on existential types for security-typed languages [36]: the type r, in which may appear free, must be a subtype of the declared type s, and to prevent implicit flows the program counter label must also flow to the label on s. Furthermore, the frame label is raised to reflect that two new variables, each of which has a type that may depend on sensitive information, is now part of the frame layout. Finally, the command is type-checked in the updated environments with the raised frame label. Rule T-FLOWSTO branches on the runtime relation between the two labels k_1 and k_2. Each command is checked in the extended formula capturing whether $k_1 \sqsubseteq k_2$ holds at runtime.

3) Typing judgment for types and labels: The typing judgments for security types and labels are straightforward. Figure 13 shows an excerpt of the typing judgment for security types, and the judgment for labels is similar. Rule T-INT says that, if the label k depends on information up to label k' then int_k depends on information up to label k' as well. Rule T-MU states that a recursive type $(\mu\ \ : \mathsf{type}_{k_1}.\ s)_{k_2}$ depends on information up to label k if, assuming depends on information up to label k, the type s can be shown to depend on information up to k and finally, both k_1 and k_2 must also not depend on information above k.

T-MU

$$\frac{\phi \vdash k_1 \sqsubseteq k \qquad \Pi[\ \ \mapsto \mathsf{type}_k]; \phi \vdash_{\mathsf{lab}} s : k \qquad \Pi; \phi \vdash_{\mathsf{lab}} k_1 : k \qquad \Pi; \phi \vdash_{\mathsf{lab}} k_2 : k}{\Pi, \phi \vdash_{\mathsf{type}} (\mu\ \ : \mathsf{type}_{k_1}.\ s)_{k_2} : k}$$

T-INT

$$\frac{\Pi; \phi \vdash_{\mathsf{lab}} k : k'}{\Pi, \phi \vdash_{\mathsf{type}} \mathsf{int}_k : k'}$$

Fig. 13: Excerpts of the typing relation for security types.

S-LIFT

$$\frac{\langle c, M, P, q\rangle_\nu \to \langle c', m', P', q'\rangle_{\nu'}}{\langle c, M, P, h, q\rangle_\nu \to \langle c', m', P', h, q'\rangle_{\nu'}}$$

S-INST

$$\frac{\langle c, M, P, h, q\rangle_\nu \to \langle c', M', P', h', q'\rangle_{\nu'}}{\langle c, M, P, h, q\rangle_\nu \to \langle c', M', P', h', q'\rangle_{\nu'}}$$

T-INST

$$\frac{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash c}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash c}$$

Fig. 14: Extending Zee with rules for modular extensions of the reduction semantics and the typing judgment.

D. An extensible language

To allow the specification of additional operations in Zee, we include a *hole* $[\cdot]$ command:

$$c ::= \cdots \mid [\cdot]$$

We call the language without the hole construct the *base* language, and the additional commands the *instantiation* language. We let c range over commands in the instantiation language, and write $\mathcal{C}$ for the set of commands in the base language. Given an instantiation language $\mathcal{D}$ we write $\mathcal{C}[\mathcal{D}]$ for the set containing commands from both the base language and the instantiation language.[11]

We add a heap to the configuration, which can be modified by the instantiation language. A heap is a partial mapping $h : \mathsf{A} \rightharpoonup v$ from addresses to values. We write $\mathsf{dom}(h)$ for the set of addresses currently allocated in h. We add an additional rule, S-LIFT, that lifts the semantics of base commands to configurations that include a heap.

Finally, we add a rule for specifying semantics of commands in $\mathcal{C}[\mathcal{D}]$: rule S-INST delegates reduction steps to the small-step semantics of the instantiation language. The new rules are shown in Figure 14. We extend the typing judgment with an additional rule T-INST that delegates typing to the typing relation for the instantiation language using the typing judgment $\vdash$ provided by the instantiation language.

IV. SECURITY GUARANTEES

In this section we formalize the security guarantees obtained by adhering to the type system described in Section III-C. Section IV-A defines the attacker model, and Section IV-B specifies the semantic interface that each instantiation language must satisfy. Finally, Section IV-C defines *termination-insensitive timing-sensitive noninterference* (TINI) [8], [24] and shows that well-typed programs satisfy TINI.

[11] Section IV-B formally defines the notion of an instantiation language.

$$\begin{array}{rcl} \mathit{ev} & ::= & \varepsilon \mid \mathsf{asgn}(x \leftarrow v, q) \mid \mathsf{rd}(x \leftarrow v, q) \mid [\cdot] \\ & \mid & \mathsf{unp}(\ell, x : \tau \leftarrow v, q) \mid \mathsf{let}(x : \tau \leftarrow v, q) \end{array}$$

Fig. 15: Grammar for events.

$$\lfloor \varepsilon \rfloor_{\mathcal{A}} = \varepsilon \qquad \lfloor \widetilde{\mathit{ev}} \cdot t \rfloor_{\mathcal{A}} = \begin{cases} \widetilde{\mathit{ev}} \cdot \lfloor t \rfloor_{\mathcal{A}} & \widetilde{\mathit{ev}} \sqsubseteq \mathcal{A} \\ \lfloor t \rfloor_{\mathcal{A}} & \text{otherwise} \end{cases}$$

Fig. 16: $\mathcal{A}$-projected trace.

A. Attacker model

To precisely define the security condition we introduce an augmented semantics that adds observable events to the reduction rules. We associate the attacker with a fixed level $\mathcal{A} \in \mathcal{L}$, and now define what an attacker at level $\mathcal{A}$ can observe and which values $\mathcal{A}$ can distinguish.

1) Events and event semantics: The grammar of events is shown in Figure 15. We assume that only commands that modify the stack generates an event, but nothing fundamental prevents adding a more fine-grained syntax of events. Event $\mathsf{asgn}(x \leftarrow v, q)$ contains the variable x assigned to, along with the value v assigned to x, and the time q of when the assignment happened. Similarly, $\mathsf{rd}(x \leftarrow v, q)$ describes obtaining a value v from the stack by reading a pointer, and assigning the value to variable x at time q. Event $\mathsf{unp}(\ell, x : \tau \leftarrow v, q)$ describes an unpack command that declares a type (or level) variable at security level ℓ, and a variable of type τ with the initial value v at time q. Event $\mathsf{let}(x : q \leftarrow v, q)$ describes the declaration of a regular variable x of type τ with initial value v at time q.

As our events capture the time at which the events are emitted, our definition of noninterference is timing-sensitive. Finally, like commands, the language of events can be extended with *instantiation events* using a hole construct $[\cdot]$, and we write the events of the instantiation language as ev.

We denote by $\widetilde{\mathit{ev}}$ an *event tuple* of the form (ev, Γ, P) where Γ is the typing environment and P is the private stack, and we define an event semantics $\xrightarrow{\widetilde{\mathit{ev}}}$ over configurations that emits event tuples. Finally, we write $\xrightarrow{t}{}^{*}$ for the reflexive, transitive closure of the event semantics relation that concatenates all event tuples into a trace t.

2) Attacker observability: Given some level $\mathcal{A} \in \mathcal{L}$ we say that τ_ℓ is *observable* to $\mathcal{A}$ if $\ell \sqsubseteq \mathcal{A}$, and *invisible* to $\mathcal{A}$ otherwise. We lift observability to events as follows: given an event ev we write $\Gamma, P \vdash \mathit{ev} \sqsubseteq \mathcal{A}$ if $\mathcal{A}$ can observe event ev given typing environment Γ and private stack P. We write $\widetilde{\mathit{ev}} \sqsubseteq \mathcal{A}$ if $\widetilde{\mathit{ev}} = (\mathit{ev}, \Gamma, P)$ and $\Gamma, P \vdash \mathit{ev} \sqsubseteq \mathcal{A}$. Section IV-B places restrictions on the instantiation events which are necessary for the nonintereference theorem to hold.

3) Attacker equivalence: We say two values $v_i : \tau_i$ for $i = 1, 2$ are $\mathcal{A}$-equivalent given private stacks frames p_1 and p_2, written $p_1, p_2 \vdash v_1 =_{\mathcal{A}} v_2 : \tau_1 \times \tau_2$, if an attacker $\mathcal{A}$ is unable to distinguish them.

We lift $\mathcal{A}$-equivalence of values to $\mathcal{A}$-equivalence of events and write $\Gamma \mid p_1, p_2 \vdash \mathit{ev}_1 =_{\mathcal{A}} \mathit{ev}_2$ when events ev_1 and ev_2 are $\mathcal{A}$-equivalent. The typing environment Γ is needed in the judgment to associate a type with the variable x in the case of an assignment event $\mathsf{asgn}(x \leftarrow v, q)$. All judgments are spelled out in the technical appendix.

Given a trace t, Figure 16 defines the $\mathcal{A}$-*projected* trace $\lfloor t \rfloor_{\mathcal{A}}$ containing only $\mathcal{A}$-observable events. We say two traces t_1 and t_2 are $\mathcal{A}$-equivalent, written $t_1 =_{\mathcal{A}} t_2$, if $\lfloor t_1 \rfloor_{\mathcal{A}}$ and $\lfloor t_2 \rfloor_{\mathcal{A}}$ are pairwise $\mathcal{A}$-equivalent. Finally, given two pairs of exposed and private stack frames (p_i, m_i) we write $\Gamma \vdash (p_1, m_1) =_{\mathcal{A}} (p_2, m_2)$ when an attacker $\mathcal{A}$ is unable to distinguish their content.

We extend this judgment pointwise and obtain an $\mathcal{A}$-equivalence on exposed and private stacks, which we write as $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$.

B. Specification of an instantiation language

In this section we describe the specification that an instantiation language must satisfy. To define the requirements of the instantiation language, we define an augmented small-step semantics $\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash C \to C' : \Gamma', \Pi', \phi', \mathit{pc}', \mathit{fr}'$. This relation specifies that C steps to C' and updates the typing environments Γ, Π, constraints ϕ and labels pc and fr to Γ', Π', ϕ', pc' and fr' respectively. We lift this relation to the event semantics relation and write $\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash C \xrightarrow{\widetilde{\mathit{ev}}} C' : \Gamma', \Pi', \phi', \mathit{pc}', \mathit{fr}'$ when event tuple $\widetilde{\mathit{ev}}$ is emitted when evaluating $\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash C \to C' : \Gamma', \Pi', \phi', \mathit{pc}'\mathit{fr}'$. A well-formedness relation $\Gamma, \Pi, \phi \quad C$ in the technical report formalizes well-formed configurations, and given a private stack P and a constraint formula ϕ relation $P \quad \phi$ specifying that ϕ is true when evaluating all labels in ϕ in the private stack P.

Formally, an instantiation language $\mathcal{D}$ is a tuple 6-tuple $(c, \to, \vdash, \mathit{ev}, =_{\mathcal{A}}, \sqsubseteq)$ where c is a set of syntactically valid commands, relation $\to$ is a small-step relation on configurations, and $\vdash$ is a typing judgment. The set ev contains syntactically valid events, and relations $\Gamma \mid p_1, p_2 \vdash \mathit{ev}_1 =_{\mathcal{A}} \mathit{ev}_2$ and $\phi, P \vdash \mathit{ev} \sqsubseteq \mathcal{A}$ defines when two events ev_1 and ev_2 are considered equivalent by an attacker at level $\mathcal{A}$, and when an event is observable to a $\mathcal{A}$ respectively.

For the following properties, let c be a command c such that $\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash c$, and let (M, P) and h be a stack and a heap such that $\Gamma, \Pi, \phi \quad \langle c, M, h, P, q \rangle_\nu$ and $P \quad \phi$. The following three properties must then be satisfied:

Property 1 (Single-run reduction properties)**.** *If*

$$\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash \langle c, M, h, P, q \rangle_\nu \to \langle c', M', h', P', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', \mathit{pc}', \mathit{fr}'$$

it holds that $\Gamma \subseteq \Gamma'$, $\Pi \subseteq \Pi'$, $\nu \leq \nu'$, *and* $\phi' \implies \phi$. *Finally it holds that* $\Gamma', \Pi', \phi' \quad \langle c', M', h', P', q' \rangle_{\nu'}$, $P' \quad \phi'$ *and* $\Gamma', \Pi', \phi', \mathit{pc}', \mathit{fr}' \vdash c'$.

Property 1 formalizes the type-safety requirements of the instantiation language. Intuitively, the semantics of the instantiation language should preserve well-formedness of configurations (i.e., $\Gamma', \Pi', \phi' \quad \langle c', M', h', P', q' \rangle_{\nu'}$). Furthermore, the semantics should not prevent future use of variables that

are already in scope by removing them from the typing environments Γ' or Π' (i.e., $\Gamma \subseteq \Gamma'$ and $\Pi \subseteq \Pi'$). To prevent the possibility of reusing version numbers the version counter ν' should not decrease (i.e., $\nu \leq \nu'$), and finally the semantics must not weaken the constraint formula ϕ' ($\phi' \implies \phi$), but should also not strengthen the formula to the point where it is not guaranteed to hold at runtime (i.e., $P' \quad \phi'$).

Property 2 (Single-step noninterference)**.** *If* $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$ *and* $\phi \vdash \mathsf{pc} \sqsubseteq \mathcal{A}$ *and*

$$\Gamma, \Pi, \phi, \mathsf{pc}, \mathsf{fr} \vdash \langle c, M_i, h_i, P_i, q \rangle_\nu \xrightarrow{\widetilde{ev}} \langle c_i', M_i', h_i', P_i', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', \mathsf{pc}', \mathsf{fr}'$$

for $i = 1, 2$ *then* $\Gamma_1' = \Gamma_2'$, $\Pi_1' = \Pi_2'$, $\Gamma_1' \vdash (P_1', M_1') =_{\mathcal{A}} (P_2', M_2')$, $q_1' = q_2'$ *and* $\Gamma_1' \mid P_1', P_2' \vdash \widetilde{\mathsf{ev}_1} =_{\mathcal{A}} \widetilde{\mathsf{ev}_2}$.

Property 2 ensures that a command c in $\mathcal{A}$-equivalent environments results in $\mathcal{A}$-equivalent observations for a single-step.

Property 3 (Confinement)**.** *If* $\phi \vdash \mathsf{pc} \not\sqsubseteq \mathcal{A}$ *and*

$$\Gamma, \Pi, \phi, \mathsf{pc}, \mathsf{fr} \vdash \langle c, M, h, P, q \rangle_\nu \xrightarrow{\widetilde{ev}} \langle c', M', h', P', q' \rangle_{\nu'} : \Gamma', \Pi', \phi', \mathsf{pc}', \mathsf{fr}'$$

then $\Gamma \vdash (P, M) =_{\mathcal{A}} (P', M')$ *and* $\phi, P \vdash \mathsf{ev} \not\sqsubseteq \mathcal{A}$.

Property 3 ensures that the semantics does not leak sensitive information through indirect flows. That is, when the reachability of a program point depends on sensitive information (i.e., $\phi \vdash \mathsf{pc} \not\sqsubseteq \mathcal{A}$), no $\mathcal{A}$-observable event is emitted, and $\mathcal{A}$ is unable to distinguish the environments before and after the execution of c.

When properties 1, 2 and 3 are satisfied we say that the instantiation language $(c, \to, \vdash, \mathsf{ev}, =_{\mathcal{A}}, \sqsubseteq)$ is a well-formed instantiation language.

C. Security guarantees

Finally, we show that well-typed Zee programs satisfy termination-insensitive timing-sensitive noninterference. This definition permits attackers to learn information by observing the termination-behavior of the program, but it does not permit an attacker to learn information due to the timing-behavior of terminating programs. This distinction between termination and timing is unusual compared to previous literature where timing-sensitivity implies termination-sensitivity [17] but in a setting like ours, where a program can fail to terminate in many different ways, i.e., by attempting to read invalid memory or by non-exhaustive pattern matching, this definition is suitable [24].

Theorem 1 (Soundness)**.** *Let* $\mathcal{D}$ *be a well-formed instantiation language and let* $c \in \mathcal{C}[\mathcal{D}]$*, and let* Γ, Π *be typing environments. Assume* $\Gamma, \Pi \vdash c$ *and for all function definitions*

$$f\langle \overline{\quad : k_1} \rangle \langle \overline{\quad : k_2} \rangle (\overline{x : s}) =_{\mathsf{pc}}^{\mathsf{fr}} c_f$$

it holds that $\Gamma_f, \Pi_f, \top, \mathsf{pc}, \mathsf{fr} \vdash c_f$*, where*

$$\Gamma_f = \{\overline{x \mapsto s}\} \qquad \Pi_f = \{\overline{\quad \mapsto \mathsf{level}_{k_1}}\} \cup \{\overline{\quad \mapsto \mathsf{type}_{k_2}}\}.$$

If $\Gamma \vdash (P_1, M_1) =_{\mathcal{A}} (P_2, M_2)$ *and* $\langle c, M_i, P_i \rangle \xrightarrow{t_i}{}^{*}$ *for* $i = 1, 2$ *then* $t_1 =_{\mathcal{A}} t_2$.

Theorem 1 states that, if each function definition is well-typed, and the command c is well-typed, then executing c with two $\mathcal{A}$-equivalent stacks will result in $\mathcal{A}$-equivalent traces t_1 and t_2.

V. Case studies

This section presents two case studies demonstrating realistic activities of a runtime system for a modern programming language. Both case studies, and all programs presented in the paper, are executable using our prototype implementation.

The first case study is an implementation of a segregated garbage collector (GC) that splits the heap into partitions indexed by security levels from a fixed lattice [24]. The algorithm is a modified version of a mark-and-sweep collector [13], and to the best our knowledge, is the first GC algorithm that implements the abstract semantics formally proven secure in [24]. The security property obtained from the well-typedness (Theorem 1) of the GC implementation implies that the timing behavior of the garbage collector does not depend on the memory operations caused by handling sensitive information.

The second case study is an implementation of a simple cooperative thread scheduling algorithm. The security property obtained from the well-typedness of the thread scheduler implies that the scheduling of "public threads" is independent from the presence of threads spawned due to handling of sensitive information.

The programs in the case studies use a syntax more suitable for programming compared to the formal language, but it can be desugared into the core calculus presented in Section III.

A. Secure garbage collection

The job of a GC is to reclaim memory that will not be used in the future by the program. This property is in general undecidable, and GC algorithms instead only reclaim memory that is not *reachable* by the program. The GC implementation is split into two phases: The *marking* phase and the *sweeping* phase. The marking phase starts when the RTE decides that a GC is needed. Our GC implementation is a stop-the-world collector: it stops the execution of the program and marks every heap allocation that is currently reachable by the program. Guaranteeing security and type-safety for such an operation is nontrivial as data of different types, and with different security policies, must be traversed and handled differently depending on the base type of the value, and its security label.

We proceed by defining the syntax of an instantiation language MS: a language for implementing secure mark-and-sweep garbage collectors. We then define the small-step semantics and the typing relation of MS. Finally, we show that MS is a well-formed instantiation language (cf. Section IV-B).

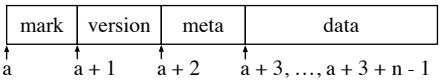
1) The instantiation language MS*:* We instantiate Zee with the instantiation language MS, whose commands are shown in Figure 17. We assume an operation on ℓ-indexed partitioning of the heap and write $\mathsf{A} \quad \ell$ for the set of addresses belonging to security level ℓ.

$$
\begin{array}{rcl}
c & ::= & \mathsf{mark}\ e \mid x := \mathsf{is\_marked}\ e \mid \mathsf{free}\ e \mid x := \mathsf{length}\ e \\
& \mid & x := \mathsf{alloc}(e, e, s, k, k) \mid x := \mathsf{start}\ k \mid x := \mathsf{next}\ e \\
& \mid & x := \mathsf{read}(e) \mid \mathsf{write}(e_1, e_2)
\end{array}
$$

Fig. 17: Instantiation language for mark-and-sweep garbage collection.

Command mark e marks a heap address, representing the information that the address is reachable in the heap. Command $x := \mathsf{is\_marked}\ e$ checks if an address, given by the evaluation of e, has previously been marked, and stores this information in variable x. Command free e reclaims the memory pointed to by e, and $x := \mathsf{alloc}(e_1, e_2, s, k_1, k_2)$ allocates e_1 number of entries, all initialized to the value e_2, in the heap partition associated with the security level k_2, and where the label k_1 denotes the sensitivity of the size of the allocation. Command $x := \mathsf{length}\ e$ stores the length of an allocation pointed to by e in variable x. The final two commands are used in the sweep phase, and implement an abstract notion of heap parsability [13]: command $x := \mathsf{start}\ k$ stores a pointer to the first allocation in the heap partition associated with the security level k in variable x, and $x := \mathsf{next}\ e$ stores the next pointer in the same heap partition as e (i.e., the allocation with the smallest address that is larger e) in variable x.

Figure 19 shows the small-step semantics for allocation in *MS*. A heap allocation of size n is structured as follows:



That is, the first address stores the *mark* of the allocation, which is used during the marking phase of garbage collection to denote that the allocation is reachable by the program. Next to the mark is the *version* entry, which ensures that it is not possible to read stale values from the heap when addresses are reused. This is similar to the technique described in Section II-C for values stored on the exposed stack. The *meta* field stores type and label information using existentially quantified labels and types: Specifically, a value of type $\exists\ : \mathsf{level}_\ell.\ \exists\ : \mathsf{type}_\ell.\ \mathsf{int}$ is stored. Here, the level ℓ is the index of the partition in which the allocation is stored on the heap. The label stores the security label on the size of the array, which is used during the sweep phase, and the type stores the type information about the type of elements in the array, which is used to traverse the heap during the marking phase, and the integer with security label stores the length of the allocation. Finally, the data of the array is stored at the end.

Rule S-ALLOC allocates such a data structure on the heap. First, an address a is found such that the range $a, \ldots, a+n+2$ is free (i.e., not part of the domain of the heap). A pointer to the first element of the array is then stored in stack variable x, and the address is given a fresh version count. Finally, the allocation structure is stored on the heap, and the version counter is incremented.

The version number in each allocation prevents leaks caused by dangling pointers and aliasing. To see an example of this, consider the program in Figure 18. On line 2 a pointer to an allocation containing public data is stored in p with a version number ν. On line 3 the memory is freed, making it possible for S-ALLOC to reuse the memory from the allocation on line 2. The memory is reused on line 5, causing p to point to secret data, even though the type of p specifies that it points to public data. However, an updated version number $\nu + 1$ is stored when the memory is being reused, so any attempt to access the secret data through p will fail the version check.

```
1 let p : (L ∄ int_L)_L := null in
2 p := alloc(10, 0, int_L, L, L);
3 free(p);
4 let q : (L ∄ int_H)_L := null in
5 q := alloc(10, h, int_H, L, L)
```

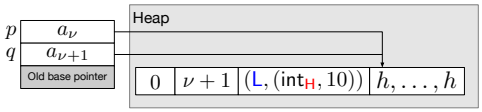


Fig. 18: Top: a program that attempts to read the secret h through the pointer p when the RTE decides to reuse the recently freed memory for the allocation on line 4. Bottom: stack- and heap layout after executing line 5 of the program.

$$
\textsc{S-Alloc}\quad
\frac{\begin{array}{c}
a = \underset{a \in \mathsf{A}\ \text{s.t.}\ a,\ldots,a+n+2 \in \mathsf{A} \restriction \ell_2}{\mathrm{argmin}}([a, \ldots, a+n+2] \cap \mathsf{dom}(h) = \mathsf{?}) \\
c = x := \mathsf{alloc}(e_1, e_2, s, k_1, k_2) \qquad \langle e_1, m, p\rangle \Downarrow n \\
P = p \cdot P' \qquad \langle e_2, m, p\rangle \Downarrow v \qquad \langle s, p\rangle \Downarrow_{\mathsf{type}} \tau \\
\langle k_i, p\rangle \Downarrow_{\mathsf{lab}} \ell_i \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto (a+3)_\nu] \\
h' = \begin{array}{l} h[a \mapsto 0, a+1 \mapsto \nu, a+2 \mapsto (\ell_1, (\tau, n))] \\ \cup \{a + i + 3 \mapsto v \mid i \in 0, \ldots, n-1\} \end{array}
\end{array}}{\langle c, m \cdot M, P, h, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, P, h', q+1\rangle_{\nu+1}}
$$

$$
\textsc{T-Alloc}\quad
\frac{\begin{array}{c}
\Gamma, \Pi, \phi \vdash e_1 : \mathsf{int}_{k_1} \qquad \Gamma, \Pi, \phi \vdash e_2 : s \\
\Pi; \phi \vdash_{\mathsf{lab}} k_1 : k_2 \qquad \Pi, \phi \vdash_{\mathsf{type}} s : k_2 \\
\Pi; \phi \vdash_{\mathsf{lab}} k_2 : \bot \qquad \phi \vdash (k_2 \mapsto s)_{k_1 \sqcup \mathit{pc}} <: \Gamma(x)
\end{array}}{\Gamma, \Pi, \phi, \mathit{pc}, \mathit{fr} \vdash x := \mathsf{alloc}(e_1, e_2, s, k_1, k_2)}
$$

Fig. 19: Static and dynamic semantics for allocation. The remaining judgments are defined in the technical report [23].

Figure 19 also defines the typing judgment: first, the expression e_1, which denotes the size of the allocation, must be of integer type with label at most k_1. This ensures that the label argument k_1 correctly captures the security of the size of the allocation at runtime. Similarly, the expression e_2, which denote the initial value of the array entries, must be typeable at type s, ensuring that the type argument s correctly captures the type information of the array at runtime. Finally, the partition label k_2 must be typeable at the label $\bot$. This guarantees that no information can be learned by knowing which partition the allocation happens in.

The events for *MS* is defined in the technical report, as well as the rest of the instantiation language. We conclude this section with the following lemma showing that *MS* satisfies

all the requirements stated in Section IV-B.

Lemma 1. *The instantiation language MS is well-formed.*

We now describe how *MS* can be used to implement a secure mark-and-sweep garbage collector. This case study is developed in a setting of the two-point lattice $\mathsf{L} \sqsubseteq \mathsf{H}$ from Section II-A.

2) Using the instantiation language: Figure 20 presents two functions representing the beginning of the marking phase. Function `mark_frames` is invoked by the `gc` function, which is invoked by the runtime.

```
1 gc ⟨pc : L, fr : L⟩() =pc,fr
2   let ( args, e) := unpack FP in
3   let ( locals, p) := unpack e in
4   let ( 2args, e2) := unpack
5     *(p - sizeof Tst(pc, fr, L)) in
6   let ( 2locals, p2) := unpack e2 in
7   mark_frames⟨pc, fr⟩ 2args, 2locals(p2); ...
```

Function `gc` starts by reading the frame pointer FP, on line 2, to obtain an existentially quantified pointer e of type $(\exists\ \text{locals} : \text{type}_{fr}.\ (\ \text{arg} \cdot \mathsf{T}_{\mathsf{st}}(pc, fr, \mathbf{L})\ @\ \text{locals})_{pc})_{\mathbf{L}}$. On line 3 e is unpacked, revealing the pointer p of type $(\ \text{arg} \cdot \mathsf{T}_{\mathsf{st}}(pc, fr, \mathbf{L})\ @\ \text{locals})_{pc}$, pointing to the beginning of `gc`'s stack frame. Lines 4 to 6 then follow the same procedure to obtain a pointer p_2 to the beginning of the previous stack frame (i.e., the function that was executing before the GC occurred). This pointer is then passed to a recursive function `mark_frames` on line 7, which traverses each stack frame. This function is shown in Figure 20. On line 3 the function checks if the pointer p is non-zero (i.e., we are not at the last stack frame). It marks allocations reachable from the stack frame starting at p using the function `mark_frame` on line 4, and computes the pointer to the previous stack frame on lines 5 to 7. On line 8 the function then invokes itself recursively to mark the previous frame starting at p_2.

Function `mark_frame` is also recursively defined, as it traverses each entry in a single stack frame. On line 3 the function performs runtime type analysis on the type locals. If the runtime representation of the type is a product type with a pointer type $(\ p \mapsto\ )$ at its head, lines 5 to 10 are executed. Line 5 performs a dynamic "flows to" check[12] to ensure that it is secure to reclaim this allocation [24]. If so, the pointer is read off the stack on line 6, and line 7 marks the pointer if it is non-null. For simplicity we elide the code that marks objects recursively reachable from this object, but the full implementation is available in our technical appendix, and the code is executable using our prototype implementation of Zee.

After having marked all allocations reachable from q on line 7, `mark_frame` calculates the offset n to the address of the next entry in the stack frame on line 9, and then invokes itself recursively on line 10 with the new address $p + n$ as its argument.

[12]The expression $_1 \equiv\ _2$ is shorthand for $_1 \sqsubseteq\ _2 \wedge\ _2 \sqsubseteq\ _1$.

```
1 mark_frames ⟨pc : L, fr : L⟩ ( args : L,  locals : L)
2  (p : ( args * Tst(pc, fr, L) @ locals)pc) =frpc
3    if p then
4      mark_frame⟨pc, fr⟩ ( args,  locals)(p);
5      let ( 2args, e) := unpack
6        *(p - sizeof Tst(pc, fr, L)) in
7      let ( 2locals, p2) := unpack e in
8      mark_frames⟨pc, fr⟩ 2args, 2locals(p2)
9    else skip

1 mark_frame⟨pc : L, fr : L⟩ ( args : L,  locals : L)
2  (p : ( args * Tst(pc, fr, L) @ locals)pc) =frpc
3    match locals with
4     ( p ?  ) * )
5      if p  pc then
6       let q := *p in
7       if q then mark(q); ... else skip
8      else skip;
9      let n := sizeof ( p ?  ) in
10     mark_frame⟨pc, fr⟩ ( args,  (p + n)
11   |  *  )
12     let n := sizeof  in
13     mark_frame⟨pc, fr⟩ ( args,  (p + n)
14   | _ ) skip
```

Fig. 20: Snippets from the GC case study: The stack frames are traversed (top) and each frame is traversed looking for pointers into the heap (bottom).

If locals is not of the form $(\ p \mapsto\ ) \cdot$, but is still a product type $\cdot$, lines 12 and 13 are executed. Line 12 computes the number of addresses that must be skipped in order to skip past the current entry in the stack frame, and line 13 then calls the function recursively with the next address $p + n$ to inspect. Finally, if local is not a product type, the frame has been completely traversed and line 14 is executed, and the function returns.

B. Secure thread scheduling

Once we have the possibility of allocating memory on the heap, we can use the same instantiation language *MS* to implement a thread scheduler. Concurrency has received a lot of attention in the literature on language-based security [15], [28], [29], [31], [32], [40], especially in the context of timing-channels. Several authors [15], [28], [32] propose special-purpose thread schedulers designed to close such timing-channels, and in this section we present an implementation of a secure cooperative thread scheduling algorithm. For the purpose of this case study, each function written by the user is assumed to have been rewritten into continuation passing style (CPS), as is standard for many compilers for functional programming languages [3], [14], and defunctionalized into a form that contains no higher-order functions, i.e., closure is an identifier followed by a heterogeneous array of local variables. Each security level from some fixed lattice is associated with a queue of closures, and a thread schedules a function f to be invoked by enqueuing its closure in the queue associated with the program counter label of f. The scheduler is a function `schedule` that receives a queue for each security level, and

a bound for how long to run sensitive computation. We have implemented a small security-typed queue datastructure in Zee that supports operations such as checking if the queue is empty, as well as queueing and dequeuing elements. We use a pseudocode-style description of the scheduling algorithm, and refer interested readers to the implementation for a precise description.

```
1  schedule(n : int_L,
2    sched_L : (L ∀ (9   : type_L .   )_L)_L,
3    sched_H : (L ∀ (9   : type_L .   )_H)_H) =^H_L
4    while nonempty(sched_L) do
5      let ( , proc) := unpack dequeue(&sched_L)
6      in run⟨L⟩ i(proc);
7      at H with bound n do
8        if nonempty(sched_H) then
9          let ( , proc) :=
10           unpack dequeue(&sched_H)
11         in run⟨H⟩ i(proc);
12       else skip
```

For simplicity, the initial program counter label is L, meaning that threads with program counter L do not need a bound on their computation time, but the scheduler can only be called when the program counter label is L. The frame label, on the other hand, is set to H allowing any sensitive information to flow to the types of the data, but any attempts to compute information based on types will be assigned the label H.

The thread scheduler executes a *public quanta* (i.e., the execution of one closure in the $\mathsf{sched}_\mathsf{L}$ queue), followed by one *secret quanta* (i.e., the execution of one closure in the $\mathsf{sched}_\mathsf{H}$). There are positives and negative things to point out about the design of this scheduler: on the positive side, the bound on secret computations n only needs to bound one quanta, and the function that calls schedule does not need to consider the number of closures in $\mathsf{sched}_\mathsf{H}$. On the negative side, to guarantee timing-sensitive noninterference, each public quanta must be followed by n steps of computation, no matter if there is any secret threads to execute or not. Furthermore, in order to run secret threads, there must also be public threads available, as the while loop terminates when the $\mathsf{sched}_\mathsf{L}$ queue is empty. An alternative strategy would be a scheduler that provides a bound on the total computation time on high threads. With this approach secret threads can run without the presence of public threads. Having developed this prototype, we leave the design of a more practical thread scheduler as future work.

VI. Implementation

We have implemented a type checker and interpreter for Zee in Haskell in about 3600 lines of code, and the case studies consists of about 500 lines each. Providing an instantiation language corresponds to an implementation of a particular type class, and the type checking and evaluation of instantiation language constructs is delegated to the relations provided by the instantiation language, similar to how the judgments in the paper are defined. The implementation can be found at: https://www.dropbox.com/s/bl2jusn8nqukqhu/zee.zip.

VII. Related work

Our work on securing runtime environments combines previous efforts of memory safety for unsafe programming languages, extensible reasoning about type systems and information-flow control. We review the relevant literature in each of these classes separately.

A. Stack typing and memory safety

The work on typed assembly languages initiated by Morrisett *et al.* [20] paved the way for type systems for low-level programming languages. As the target language was expressed in continuation-passing style, there was no need for a stack. Morrisett *et al.* [19] introduced local stack variables, but as the goal of that work is type preserving compilation it does not support reasoning about stack traversal, and so the "previous frame pointer" is not available on the stack for accessing the previous stack frames. For this reason Morrisett *et al.* do not consider runtime type analysis.

Our stack typing discipline is inspired by the bunched adjacency logic of Ahmed and Walker [2]. They use logic formulae $\mathsf{more}^{\leftarrow}$ and $\mathsf{more}^{\rightarrow}$ to describe the type of an infinite sequence of locations that increases "to the left" and "to the right" respectively, similar to our use of stack pointer types.

Our version-based enforcement mechanism is inspired by CETS's [22] identifier-based temporal checking. CETS is a program transformation that adds temporal memory safety checking capable of detecting dangling pointer dereferences and double frees errors at runtime. Our version-based enforcement mechanism could be replaced with static reasoning about regions. Regions were introduced by Tofte and Talpin [35] and later used to provide memory safety for a safe dialect of C [10]. We believe the use of regions is orthogonal to our choice of a dynamic enforcement mechanism.

B. Attacks on runtimes

The work on observational determinism by Zdancewic and Myers [40] contains a detailed collection of common scheduler-related attacks. Other parts of the RTE that has been attacked include garbage collectors [24]. Pedersen and Askarov [24] present a series of attacks on the garbage collectors of the Java virtual machine and the V8 JavaScript engine, and design a type system and a small-step semantics for a high-level language with automatic memory management for which they prove a noninterference result similar to ours. Finally, Vassena *et al.* [37] present attacks that combine concurrent execution and lazy evaluation for leaking sensitive information. They propose a new construct for Haskell called lazydup, which lazily duplicates thunks on the heap when entering secret contexts (i.e., when the program counter label is H, as they only consider a two-point lattice).

C. Securing runtimes

Vassena *et al.* [38] present a new foundation for a dynamic information flow control parallel runtime system. The goal of their work is securing the execution platform of LIO [33], a dynamic information flow control library for Haskell. Similar

to our work, Vassena *et al.* [38] consider a setting in which an attacker can obtain the current global time as a natural number counting execution steps, and (unlike our model) the current size of the heap. They design a system for hierarchically managing space and time resources with some amount of burden on the programmer: a parent thread has to manually kill their child thread to reclaim resources. Their end goal is an implementation of a modified GHC runtime system, but such a modified runtime has yet to be implemented.

The work by Sabelfeld and Sands [28] contains an interesting observation:

> *Abstractly we will take a scheduler to be a mechanism for selecting threads which itself satisfies some noninterference property, i.e., its behaviour is independent of high data.*

This is exactly the approach we have taken: a thread scheduler is a program written in our language, and Theorem 1 proves that, since this program is well-typed, its public observable behavior is independent of high data.

D. Static information flow control

There is a large body of literature focusing on static information flow control, starting with the seminal work by Denning and Denning [6] and later formulated as a type system by Volpano *et al.* [39]. Sabelfeld and Myers [27] survey the different enforcement techniques and security definitions. Zheng and Myers [44] introduce the technique of including a formulae expressing which flows are guaranteed to hold at specific program points, allowing for static reasoning about information flow policies that vary at runtime. The use of existentially quantified labels is introduced by Tse and Zdancewic [36], and we follow the same typing discipline for such values. Dependent type systems for IFC has also been explored by Lourenço and Caires [16], Zhang *et al.* [43] and Gregersen *et al.* [9].

E. Verified runtimes

There has been much work on verifying runtime system components such as garbage collectors [5], [7], [18] and thread schedulers [11] using program logics. We view our work complementary to these efforts. The constructs needed for implementing secure runtime environments, that we identify in this work, may serve as guidelines when applying enforcement techniques different from our type system. Additionally, a program logic may be used to verify the requirements of the instantiation languages used in Zee.

REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin, Dynamic Typing in a Statically-typed Language, in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1989,

[2] A. Ahmed and D. Walker, The Logical Approach to Stack Typing, in *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ACM, 2003,

[3] A. W. Appel, Compiling with continuations. Cambridge University Press, 2006.

[4] A. Askarov, D. Zhang, and A. C. Myers, Predictive Black-box Mitigation of Timing Channels, in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, ACM, 2010,

[5] L. Birkedal, N. Torp-Smith, and J. C. Reynolds, Local Reasoning About a Copying Garbage Collector, in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2004,

[6] D. E. Denning and P. J. Denning, Certification of Programs for Secure Information Flow, *Commun. ACM*, Jul. 1977.

[7] P. Gammie, A. L. Hosking, and K. Engelhardt, Relaxing Safely: Verified On-the-fly Garbage Collection for x86-TSO, in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2015,

[8] J. A. Goguen and J. Meseguer, Security Policies and Security Models, in *1982 IEEE Symposium on Security and Privacy*, IEEE, Apr. 1982.

[9] S. Gregersen, S. E. Thomsen, and A. Askarov, A Dependently Typed Library for Static Information-Flow Control in Idris, *CoRR*, 2019. arXiv: 1902.06590.

[10] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, Region-Based Memory Management in Cyclone, *ACM SIGPLAN Notices*, May 2002.

[11] Y. Guo, X. Feng, Z. Shao, and P. Shi, Modular Verification of Concurrent Thread Management, in *Programming Languages and Systems*, R. Jhala and A. Igarashi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012,

[12] R. Harper and G. Morrisett, Compiling Polymorphism Using Intensional Type Analysis, in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1995,

[13] R. Jones, A. Hosking, and E. Moss, The Garbage Collection Handbook: The Art of Automatic Memory Management, 1st. Chapman & Hall/CRC, 2011.

[14] S. L. P. Jones, Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine, *Journal of Functional Programming*, Apr. 1992.

[15] A. Karbyshev, K. Svendsen, A. Askarov, and L. Birkedal, Compositional Non-interference for Concurrent Programs via Separation and Framing, in *Principles of Security and Trust*, Cham: Springer International Publishing, 2018,

[16] L. Lourenço and L. Caires, Dependent Information Flow Types, in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 2015,

[17] H. Mantel and A. Sabelfeld, A Generic Approach to the Security of Multi-Threaded Programs, in *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, IEEE Computer Society, 2001,

[18] A. McCreight, Z. Shao, C. Lin, and L. Li, A General Framework for Certifying Garbage Collectors and Their Mutators, in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2007,

[19] G. Morrisett, K. Crary, N. Glew, and D. Walker, Stack-based Typed Assembly Language, in *Types in Compilation*, X. Leroy and A. Ohori, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998,

[20] J. G. Morrisett, D. Walker, K. Crary, and N. Glew, From System F to Typed Assembly Language, in *POPL*, 1998.

[21] A. C. Myers and A. C. Myers, JFlow: Practical Mostly-static Information Flow Control, in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1999,

[22] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, CETS: Compiler enforced temporal safety for C, in *Proceedings of the 2010 International Symposium on Memory Management*, ACM, 2010,

[23] Pedersen, Mathias V., Askarov, Aslan, "Static enforcement of secure runtime systems: technical report," Tech. Rep., 2018.

[24] M. V. Pedersen and A. Askarov, From Trash to Treasure: Timing-Sensitive Garbage Collection, in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, May 2017.

[25] F. Perry, C. Hawblitzel, and J. Chen, Simple and Flexible Stack Types, Jul. 2007,

[26] B. C. Pierce, Types and Programming Languages, 1st. The MIT Press, 2002.

[27] A. Sabelfeld and A. C. Myers, Language-based Information-flow Security, *IEEE J.Sel. A. Commun.*, Sep. 2006.

[28] A. Sabelfeld and D. Sands, Probabilistic noninterference for multi-threaded programs, in *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*, IEEE Comput. Soc.

[29] A. Sabelfeld, The Impact of Synchronisation on Secure Information Flow in Concurrent Programs, in *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2001,

[30] L. Skorstengaard, D. Devriese, and L. Birkedal, Reasoning About a Machine with Local Capabilities, in *Programming Languages and Systems*, A. Ahmed, Ed., Cham: Springer International Publishing, 2018,

[31] G. Smith and D. Volpano, Secure Information Flow in a Multi-threaded Imperative Language, in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1998,

[32] D. Stefan, A. Russo, P. Buiras, A. Levy, J. C. Mitchell, and D. Maziéres, Addressing Covert Termination and Timing Channels in Concurrent Information Flow Systems, in *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ACM, 2012,

[33] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, Flexible Dynamic Information Flow Control in Haskell, in *Proceedings of the 4th ACM Symposium on Haskell*, ACM, 2011,

[34] C. A. Stone and R. Harper, Extensional Equivalence and Singleton Types, *ACM Trans. Comput. Logic*, Oct. 2006.

[35] M. Tofte and J.-P. Talpin, Region-Based Memory Management, *Inf. Comput.*, Feb. 1997.

[36] S. Tse and S. Zdancewic, Run-time Principals in Information-flow Type Systems, *ACM Trans. Program. Lang. Syst.*, Nov. 2007.

[37] M. Vassena, J. Breitner, and A. Russo, Securing Concurrent Lazy Programs Against Information Leakage, in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, IEEE, Aug. 2017.

[38] M. Vassena, G. Soeller, P. Amidon, M. Chan, and D. Stefan, Towards parallel information flow control foundations, in *Principles of Security and Trust*, 2019.

[39] D. Volpano, C. Irvine, and G. Smith, A Sound Type System for Secure Flow Analysis, *J. Comput. Secur.*, Jan. 1996.

[40] S. Zdancewic and A. Myers, Observational determinism for concurrent program security, in *16th IEEE Computer Security Foundations Workshop, 2003. Proceedings.*, IEEE Comput. Soc.

[41] D. Zhang, A. Askarov, and A. C. Myers, Language-based Control and Mitigation of Timing Channels, in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, 2012,

$$\frac{}{\mathsf{int}_\ell \text{ - } \mathsf{int}} \qquad \frac{\tau \text{ - } p}{(\ell_1 \mapsto \tau)_{\ell_2} \text{ - } (\ _1 \mapsto p)\ _2}$$

$$\frac{\tau_i \text{ - } p_i \ \ i = 1,2}{(\tau_1 \mathbin{@} \tau_2)_\ell \text{ - } (p_1 \mathbin{@} p_2)} \qquad \frac{}{\tau \text{ - }}$$

$$\frac{|\overline{\tau}| = n \qquad |\overline{p}| = m \qquad m \le n \qquad \forall i \in \{1,\dots,m-1\} \,.\, \tau_i \text{ - } p_i \qquad \tau_{m\dots n} \text{ - } p_m}{\overline{\tau} \text{ - } \overline{p}}$$

(a) Pattern Matching relation.

$$\mathsf{Jint}\ \ \mathsf{K}(p, \mathsf{int}_\ell) = p[\ \ \mapsto \ell]$$

$$\frac{\mathsf{J}p_1\mathsf{K}(p,\tau_1) = p' \qquad \mathsf{J}p_2\mathsf{K}(p',\tau_2) = p''}{\mathsf{J}(p_1 \mathbin{@} p_2)\ \ \mathsf{K}(p,(\tau_1 \mathbin{@} \tau_2)_\ell) = P''[\ \ \mapsto \ell]}$$

$$\frac{\mathsf{J}p\mathsf{K}(p,\tau) = p' \qquad p'' = p'[\ _1 \mapsto \ell_1,\ \ _2 \mapsto \ell_2]}{\mathsf{J}(\ _1 \mapsto p)\ _2\mathsf{K}(p, \ell_1 \mapsto \tau_{\ell_2}) = p''} \qquad \mathsf{J}\ \ \mathsf{K}(p,\tau) = p[\ \ \mapsto \tau]$$

$$\frac{|\overline{\tau}| = n \qquad |\overline{p}| = m \qquad m \le n \qquad p_0 = p \qquad \forall i \in \{1,\dots,m-1\} \,.\, \mathsf{J}p_i\mathsf{K}(p_{i-1},\tau_i) = p_i \qquad \mathsf{J}p_m\mathsf{K}(p_{m-1},\tau_{m\dots n}) = p'}{\mathsf{J}\overline{p}\mathsf{K}(p,\overline{\tau}) = p'}$$

(b) Binding free variables in the pattern p by deconstructing the type value τ.

$$\Pi \vdash \mathsf{int} \qquad {}_k\ \Pi[\ \ \mapsto \mathsf{level}_k] : \mathsf{int}$$

$$\Pi \vdash \qquad {}_k\ \Pi[\ \ \mapsto \mathsf{type}_k] :$$

$$\frac{\Pi \vdash p_1 \quad {}_k\ \Pi_1 : s_1 \qquad \Pi_1 \vdash p_2 \quad {}_k\ \Pi_2 : s_2}{\Pi \vdash (p_1 \mathbin{@} p_2) \quad {}_k\ \Pi_2[\ \ \mapsto \mathsf{level}_k] : (s_1 \mathbin{@} s_2)}$$

$$\frac{\Pi \vdash p \quad {}_k\ \Pi' : s}{\Pi \vdash (\ _1 \mapsto p)\ _2 \quad {}_k\ \Pi'[\ _1 \mapsto \mathsf{level}_k,\ \ _2 \mapsto \mathsf{level}_k] : (\ _1 \mapsto s)\ _2}$$

$$\frac{\Pi_0 = \Pi \qquad \Pi_{i-1} \vdash p_i \quad {}_k\ \Pi_i : s_i \quad i = 1,\dots,n}{\Pi \vdash p_1,\dots,p_n \quad {}_k\ \Pi_n : s_1,\dots,s_n}$$

(c) Relation for closing the free type variables in the pattern p, and computing the type of the scrutinee when it matches pattern p.

[42] ——, Predictive Mitigation of Timing Channels in Interactive Systems, in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, 2011,

[43] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, A Hardware Design Language for Timing-Sensitive Information-Flow Security, in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2015,

[44] L. Zheng and A. C. Myers, Dynamic Security Labels and Static Information Flow Control, *Int. J. Inf. Secur.*, Mar. 2007.

APPENDIX

S-IF-T
$$\frac{\langle e, M, P\rangle \Downarrow n \qquad n \neq 0}{\langle \mathsf{if}\ e\ c_1\ c_2, M, P, q\rangle_\nu \to \langle c_1, M, P, q+1\rangle_\nu}$$

S-IF-F
$$\frac{\langle e, M, P\rangle \Downarrow 0}{\langle \mathsf{if}\ e\ c_1\ c_2, M, P, q\rangle_\nu \to \langle c_2, M, P, q+1\rangle_\nu}$$

S-WHILE-F
$$\frac{\langle e, M, P\rangle \Downarrow 0}{\langle \mathsf{while}\ e\ c, M, P, q\rangle_\nu \to \langle \mathsf{stop}, M, P, q+1\rangle_\nu}$$

S-WHILE-T
$$\frac{\langle e, M, P\rangle \Downarrow n \qquad n \neq 0}{\langle \mathsf{while}\ e\ c, M, P, q\rangle_\nu \to \langle c; \mathsf{while}\ e\ c, M, P, q+1\rangle_\nu}$$

S-SKIP
$$\langle \mathsf{skip}, M, P, q\rangle_\nu \to \langle \mathsf{stop}, M, P, q+1\rangle_\nu$$

S-ASGN
$$\frac{\langle e, m \cdot M, P\rangle \Downarrow v \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v]}{\langle x := e, m \cdot M, P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, P, q+1\rangle_\nu}$$

S-WRITE
$$\frac{m_i = (m_\mathsf{l}, \nu_i) \in M \qquad a \in \mathsf{l} \qquad \nu_i \le \gamma \qquad \langle e_1, M, P\rangle \Downarrow a_\gamma \qquad \langle e_2, M, P\rangle \Downarrow v}{\langle *e_1 := e_2, M, P, q\rangle_\nu \to \langle \mathsf{stop}, M[a \mapsto v], P, q+1\rangle_\nu}$$

S-READ
$$\frac{M = m \cdot M' \qquad m_i = (m_\mathsf{l}, \nu_i) \in M \qquad a \in \mathsf{l} \qquad \nu_i \le \gamma \qquad \langle e, M, P\rangle \Downarrow a_\gamma \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto M(n)]}{\langle x := *e, m \cdot M, P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M', P, q+1\rangle_\nu}$$

S-AT
$$\frac{\langle e, m, P\rangle \Downarrow n}{\langle \mathsf{at}\ k\ e\ c, m, P, q\rangle_\nu \to \langle c; \mathsf{delay}\ n, m, P, q+1\rangle_\nu}$$

S-DELAY
$$\frac{n \le q}{\langle \mathsf{delay}\ n, m, P, q\rangle_\nu \to \langle \mathsf{delay}\ n, m, P, n+1\rangle_\nu}$$

S-FP
$$\frac{v = (\mathsf{cod}(p.\mathsf{arg}), (\mathsf{cod}(p.\mathsf{local}), \mathsf{fp}(m)_\nu)) \qquad m = (\mathsf{l}, |m|, \nu) \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v]}{\langle x := \mathsf{fp}, m \cdot M, p \cdot P, q\rangle_\nu \to \langle \mathsf{stop}, m' \cdot M, p \cdot P, q+1\rangle_\nu}$$

S-LET
$$\frac{M = m \cdot M' \qquad \langle s, p\rangle \Downarrow_{\mathsf{type}} \tau \qquad \langle e, m, p\rangle \Downarrow v \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v] \qquad p' = p[p_{\mathsf{local}} \mapsto p_{\mathsf{local}}[x \mapsto \tau]]}{\langle \mathsf{let}\ x : s := e\ \mathsf{in}\ c, M, p \cdot P, q\rangle_\nu \to \langle c; \mathsf{unscope}(x), m' \cdot M', p' \cdot P, q+1\rangle_\nu}$$

S-UNSCOPE
$$\frac{p' = p[\mathsf{local} \mapsto p.\mathsf{local}[x \mapsto\ ]]}{\langle \mathsf{unscope}(x), M, p \cdot P, q\rangle_\nu \to \langle \mathsf{stop}, M, p' \cdot P, q+1\rangle_\nu}$$

S-UNPACK-LEV
$$\frac{P = p \cdot P' \qquad \langle s, p'\rangle \Downarrow_{\mathsf{type}} \tau \qquad \langle e, m, p\rangle \Downarrow (\ell_1, v_2) \qquad p' = p[p_{\mathsf{var}} \mapsto p_{\mathsf{var}}[\ \mapsto \ell_1], p_{\mathsf{local}} \mapsto p_{\mathsf{local}}[x \mapsto \tau]] \qquad M = m \cdot M' \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v_2]}{\langle \mathsf{let}\ (\ \ : \mathsf{level}_k, x : s) := e\ \mathsf{in}\ c, M, P, q\rangle_\nu \to \langle c; \mathsf{unscope}(x), m' \cdot M', p' \cdot P', q+1\rangle_\nu}$$

S-UNPACK-TY
$$\frac{P = p \cdot P' \qquad \langle s, p'\rangle \Downarrow_{\mathsf{type}} \tau \qquad \langle e, m, p\rangle \Downarrow (\tau_1, v_2) \qquad p' = p[p_{\mathsf{var}} \mapsto p_{\mathsf{var}}[\ \mapsto \tau_1], p_{\mathsf{local}} \mapsto p_{\mathsf{local}}[x \mapsto \tau]] \qquad M = m \cdot M' \qquad m' = m[\delta(x) + \mathsf{fp}(m) \mapsto v_2]}{\langle \mathsf{let}\ (\ \ : \mathsf{type}_k, x : s) := e\ \mathsf{in}\ c, M, P, q\rangle_\nu \to \langle c; \mathsf{unscope}(x), m' \cdot M', p' \cdot P', q+1\rangle_\nu}$$

S-EPILOGUE
$$\langle \mathsf{epilogue}, (\mathsf{l}_1, |m_1|, \nu_1) \cdot (\mathsf{l}_2, |m_2|, \nu_2) \cdot M, p \cdot P, q\rangle_\nu \to \langle \mathsf{stop}, (\mathsf{l}_2, |m_2|, \max(\nu_1, \nu_2) + 1) \cdot M, P, q+1\rangle_\nu$$

S-MATCH
$$\frac{\operatorname*{argmin}_{i=1,\ldots,n}(\tau\ \text{-}\ p_i) = j \qquad \langle\ \ , p\rangle \Downarrow_{\mathsf{type}} \tau \qquad \mathsf{J}p_j\mathsf{K}(p, \tau) = p'}{\langle \mathsf{match}\quad (p_i \Rightarrow c_i)_{i=1,\ldots,n}, M, p \cdot P, q\rangle_\nu \to \langle c_j, M, p' \cdot P, q+1\rangle_\nu}$$

S-CALL
$$\frac{\begin{array}{c}\mathsf{F}(f) = \langle\ _1, \ldots,\ _n\rangle\langle\ _1, \ldots,\ _m\rangle(x_1 : s'_1, \ldots, x_r : s'_r) = c \\ \langle k_i, P\rangle \Downarrow_{\mathsf{lab}} \ell_i \qquad \langle s_i, P\rangle \Downarrow_{\mathsf{type}} \tau_i \qquad \langle e_i, M, P\rangle \Downarrow v_i \\ \langle s'_i, P'\rangle \Downarrow_{\mathsf{type}} \tau'_i \qquad M = m \cdot M' \qquad m' = (\mathsf{l}', |m'|, \nu) \\ P = (p_{\mathsf{var}}, p_{\mathsf{args}}, p_{\mathsf{local}}) \cdot P' \qquad p' = (p'_{\mathsf{var}}, p'_{\mathsf{arg}}, p'_{\mathsf{local}}) \\ p'_{\mathsf{var}} = \{\ _i \mapsto \ell_i \mid i = 1, \ldots, n\} \cup \{\ _i \mapsto \tau_i \mid i = 1, \ldots, m\} \\ p'_{\mathsf{arg}} = \{x_i \mapsto \tau'_i \mid i = 1, \ldots, r\} \qquad p'_{\mathsf{local}} = \{x \mapsto \bot \mid x \in c\} \\ \mathsf{l}' = \{\delta(x_i) + \mathsf{sp}(m) \mid i = 1, \ldots, r\} \cup \{\mathsf{sp}(m)\} \cup \{\delta(z) + \mathsf{sp}(m) \mid z \in c\} \\ |m'| = \{\mathsf{sp}(m) \mapsto (\mathsf{cod}(p_{\mathsf{arg}}), (\mathsf{cod}(p_{\mathsf{local}}), \mathsf{fp}(m)_\nu))\} \cup \{\delta(x_i) + \mathsf{sp}(m) \mapsto v_i \mid i = 1, \ldots, r\}\end{array}}{\begin{array}{c}\langle f\langle k_1, \ldots, k_n\rangle\langle s_1, \ldots, s_m\rangle(e_1, \ldots, e_r), M, P, q\rangle_\nu \\ \to \langle c; \mathsf{epilogue}, m' \cdot M, p' \cdot P, q+1\rangle_{\nu+1}\end{array}}$$

S-SEQ-CONT
$$\frac{c'_1 \neq \mathsf{stop} \qquad \langle c_1, m, P, q\rangle_\nu \to \langle c'_1, m', P', q'\rangle_{\nu'}}{\langle c_1; c_2, m, P, q\rangle_\nu \to \langle c'_1; c_2, m', P', q'\rangle_{\nu'}}$$

S-SEQ-STOP
$$\frac{\langle c_1, m, P, q\rangle_\nu \to \langle \mathsf{stop}, m', P', q'\rangle_{\nu'}}{\langle c_1; c_2, m, P, q\rangle_\nu \to \langle c_2, m', P', q'\rangle_{\nu'}}$$

S-INST
$$\frac{\langle c, m, P, h, q\rangle_\nu \to \langle c', m', P', h', q'\rangle_{\nu'}}{\langle c, m, P, h, q\rangle_\nu \to \langle c', m', P', h', q'\rangle_{\nu'}}$$

Fig. 22: Small-step relation for commands.