Secure Information Flow and CPS

Steve Zdancewic Andrew C. Myers

Cornell University, Ithaca NY 14853, USA
{zdance,andru}@cs.cornell.edu

Abstract. *Security-typed languages* enforce secrecy or integrity policies by type checking. This paper investigates continuation-passing style as a means of proving that such languages enforce non-interference and as a first step towards understanding their compilation. We present a low-level, secure calculus with higher-order, imperative features. Our type system makes novel use of *ordered linear continuations*.

1 Introduction

Language based mechanisms for enforcing secrecy or integrity policies are attractive because, unlike ordinary access control, static information flow can enforce end-to-end policies. These policies require that data be protected despite being manipulated by programs with access to various covert channels. For example, such a policy might prohibit a personal finance program from transmitting credit card information over the Internet even though the program needs Internet access to download stock market reports. To prevent the finance program from illicitly transmitting the private information (perhaps cleverly encoded), the compiler checks that the information flows in the program are admissible.

There has been much recent work on formulating Denning's original lattice model of information-flow control [9] in terms of type systems for static program verification [1, 16, 20, 21, 25–27, 30, 32]. The desired security property is *non-interference* [14], which states that high-security data is not observable by low-security computation. Nevertheless, secure information flow in the context of higher-order languages with imperative features is not well understood.

This paper proposes the use of continuation-passing style (CPS) translations [8, 12, 28] as a means of ensuring non-interference in imperative, higher-order languages. There are two reasons for using CPS. First, CPS is a vehicle for proving a non-interference result, generalizing previous work by Smith and Volpano [30]. Second, CPS is useful for representing low-level programs [4, 18], which opens up the possibility of verifying the security of compiler output via typed assembly language [18] or proof-carrying code [22].

This research was supported by DARPA Contract F30602-99-1-0533, monitored by USAF Rome Laboratory. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement DARPA, AFRL, or the U.S. Government.

We observe that a naive approach to providing security types for an imperative CPS language yields a system that is too conservative: secure programs (in the non-interference sense) are rejected. To rectify this problem, we introduce *ordered linear continuations*, which allow information flow control in the CPS target language to be made more precise. The ordering property of linear continuations is crucial to the non-interference argument, which is the first such theorem for a higher-order, imperative language.

As with previous non-interference results for call-by-value languages [16, 20], the theorem holds only for programs that halt regardless of high-security data. Consequently, termination channels can arise, but they leak at most one bit per run on average, we consider them acceptable. There are other channels not captured by this notion of non-interference: high-security data can alter the running time of the program or change its memory consumption. Non-interference holds despite these apparent information leaks because the language itself provides no means for observing these resources (for instance, access to the system clock). Recent work attempts to address such covert channels [3].

The next section shows why a naive type system for secure information flow is too restrictive for CPS and motivates the use of ordered linear continuations. Section 3 presents the target language, its operational semantics, and the novel features of its type system. The non-interference theorem is proved in Section 4, and Section 5 demonstrates the viability of this language as a low-level calculus by showing how to CPS translate a higher-order, imperative language. We conclude with some discussion and related work in Section 6.

2 CPS and Security

Type systems for secrecy or integrity are concerned with tracking dependencies in a program [1]. One difficulty is *implicit flows*, which arise from the control flow of the program. Consider the code fragment `A` in Figure 1. There is an implicit flow between the value stored in `x` and the value stored in `y`, because examining the contents of `y` after the program has run gives information about the value in `x`. There is no information flow between `x` and `z`, however. This code is secure even when `x` and `y` are high-security variables and `z` is low-security. (In this paper, *high security* means "high secrecy" or "low integrity." Dually, *low security* means "low secrecy" or "high integrity.")

Fragment `B` illustrates the problem with CPS translation. It shows the code from `A` after control transfer has been made explicit. The variable `k` is bound to the continuation of the `if`, and the jump is indicated by the application `k` $\langle\rangle$. Because the invocation of `k` has been lifted into the branches of the conditional, a naive type system for information flow will conservatively require that the body of `k` not write to low-security memory locations: The value of `x` would apparently be observable by low-security code. Program `B` is rejected because `k` writes to a low-security variable, `z`.

However, this code *is* secure: There is no information flow between `x` and `z` in `B` because the continuation `k` is invoked in both branches. As example `C` shows, if

```
(A)   if x then { y := 1; } else { y := 2; }
      z := 3; halt;

(B)   let k = (λ⟨⟩. z := 3; halt) in
      if x then { y := 1; k ⟨⟩; } else { y := 2; k ⟨⟩; }

(C)   let k = (λ⟨⟩. z := 3; halt) in
      if x then { y := 1; k ⟨⟩; } else { y:= 2; halt; }

(D)   letlin k = (λ⟨⟩. z := 3; halt) in
      if x then { y := 1; k ⟨⟩; } else { y:= 2; k ⟨⟩; }

(E)   letlin k0 = (λ⟨⟩. halt) in
      letlin k1 = (λk. z := 1; k ⟨⟩) in
      letlin k2 = (λk. z := 2; k ⟨⟩) in
      if x then { letlin k = (λ⟨⟩. k1 k0) in k2 k }
           else { letlin k = (λ⟨⟩. k2 k0) in k1 k }
```

Fig. 1. Examples of Information Flow in CPS

`k` is *not* used in one of the branches, then information about `x` can be learned by observing `z`. Linear type systems [2, 13, 33, 34] can express exactly the constraint that `k` is used in both branches. By making `k`'s linearity explicit, the type system can use the additional information to recover the precision of the source program analysis. Fragment `D` illustrates our simple approach: In addition to a normal `let` construct, we include `letlin` for introducing linear continuations. The program `D` certifies as secure even when `z` is a low-security variable, whereas `C` does not.

Although linearity allows for more precise reasoning about information flow, linearity alone is unsafe in the presence of first-class linear continuations. In example `E`, continuations `k0`, `k1`, and `k2` are all linear, but there is an implicit flow from `x` to `z` because `z` lets us observe the *order* in which `k1` and `k2` are invoked. It is thus necessary to regulate the ordering of linear continuations.

It is simpler to make information flow analysis precise for the source language because the structure of the language limits control flow. For example, it is known that both branches of a conditional return to a common merge point. This knowledge can be exploited to obtain less conservative analysis of implicit flows, but the standard CPS transformation loses this information by unifying all forms of control to a single mechanism. In our approach, the target language still has a single underlying control transfer mechanism (examples `B` and `D` execute exactly the same code), but information flow can be analyzed with the same precision as in the source.

3 The Secure CPS Calculus

The target is a call-by-value, imperative language similar to those found in the work on Typed Assembly Language [6, 18], although its type system is inspired by previous language-based security research [16, 20, 32]. This section describes the secure CPS language, its operational behavior, and its static semantics.

Types

$$
\begin{array}{rcl}
\ell, \mathsf{pc} & \in & \mathcal{L} \\
\tau & ::= & \mathsf{int} \mid \mathsf{1} \mid \sigma\ \mathsf{ref} \mid [\mathsf{pc}](\sigma, \kappa) \to \mathsf{0} \\
\sigma & ::= & \tau_\ell \\
\kappa & ::= & \mathsf{1} \mid (\sigma, \kappa) \to \mathsf{0}
\end{array}
$$

Values and Primitive Operations

$$
\begin{array}{rcl}
bv & ::= & n \mid \langle\rangle \mid L^\sigma \mid \lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\, e \\
v & ::= & x \mid bv_\ell \\
lv & ::= & \langle\rangle \mid y \mid \lambda\langle\mathsf{pc}\rangle(x\!:\!\sigma, y\!:\!\kappa).\, e \\
prim & ::= & v \mid v \oplus v \mid \mathtt{deref}(v)
\end{array}
$$

Expressions

$$
\begin{array}{rl}
e ::= & \mathtt{let}\ x = prim\ \mathtt{in}\ e \\
\mid & \mathtt{let}\ x = \mathtt{ref}_\ell^\sigma\ v\ \mathtt{in}\ e \\
\mid & \mathtt{set}\ v := v\ \mathtt{in}\ e \\
\mid & \mathtt{letlin}\ y = lv\ \mathtt{in}\ e \\
\mid & \mathtt{let}\ \langle\rangle = lv\ \mathtt{in}\ e \\
\mid & \mathtt{if0}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e \\
\mid & \mathtt{goto}\ v\ v\ lv \\
\mid & \mathtt{lgoto}\ lv\ v\ lv \\
\mid & \mathtt{halt}^\sigma\ v
\end{array}
$$

Fig. 2. Syntax for the Secure CPS Language

3.1 Syntax

The syntax for the secure CPS language is given in Figure 2. Elements of the lattice of security labels, $\mathcal{L}$, are ranged over by meta-variables ℓ and pc. We reserve the meta-variable pc to suggest that the security label corresponds to information learned by observing the program counter. The $\sqsubseteq$ operator denotes the lattice ordering, with the join operation given by $\sqcup$, and least element $\bot$.

Types fall into two syntactic classes: security types, σ, and linear types, κ. Security types are the types of ordinary values and consist of a base-type component, τ, annotated with a security label, ℓ. Base types consist of integers, unit, references, and continuations (written $[\mathsf{pc}](\sigma, \kappa) \to \mathsf{0}$). Correspondingly, base values, bv, include integers, n, a unit, $\langle\rangle$, type-annotated memory locations, L^σ, and continuations, $\lambda[\mathsf{pc}]f(x : \sigma, y : \kappa).\, e$. All computation occurs over secure values, v, which are base values annotated with a security label. Variables, x, range over values. We adopt the notation $\mathsf{label}(\tau_\ell) = \ell$, and extend the join operation to security types: $\tau_\ell \sqcup \ell' = \tau_{(\ell \sqcup \ell')}$.

An ordinary continuation $\lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\, e$ is a piece of code (the expression e) that accepts a non-linear argument of type σ and a linear argument of type κ. Continuations may recursively invoke themselves using the variable f. The notation $[\mathsf{pc}]$ indicates that this continuation may be called only from a context in which the program counter carries information of security at most pc. To avoid unsafe implicit flows, the body of the continuation may create effects only observable by principals able to read data with label pc.

Linear values are either unit, variables, or linear continuations, which contain code expressions parameterized by non-linear and linear arguments just like ordinary continuations. Unlike ordinary continuations, linear continuations may not be recursive[1], but they may be invoked from any calling context; hence linear types do not require any pc annotation. The syntax $\langle\mathsf{pc}\rangle$ serves to distinguish

[1] A linear continuation `k` may be discarded by a recursive ordinary continuation that loops infinitely, passing itself `k`. Precise terminology for our “linear” continuations would be “affine” to indicate that they may, in fact, never be invoked.

linear continuation values from non-linear ones. As for ordinary continuations, the label pc restricts the continuation's effects.

The primitive operations include binary arithmetic ($\oplus$), dereference, and a means of copying secure values. Program expressions consist of a sequence of `let` bindings for primitive operations, reference creation, and imperative updates (via `set`). The `letlin` construct introduces a linear continuation, and the expression `let` $\langle\rangle = lv$ `in` e, necessary for type-checking but operationally a no-op, eliminates a linear unit before executing e. Straight-line code sequences are terminated by conditional statements, non-local transfers of control via `goto` (for ordinary continuations) or `lgoto` (for linear continuations), or `halt`.

3.2 Operational Semantics

The operational semantics (Figure 3) are given by a transition relation between machine configurations of the form $\langle M, \mathsf{pc}, e\rangle$. Memories, M, are finite partial maps from typed locations to closed values. The notation $M[L^\sigma \leftarrow v]$ denotes the memory obtained from M by updating the location L^σ to contain the value v of type σ. A memory is *well-formed* if it is closed under the dereference operation and each value stored in the memory has the correct type. The notation $e\{v/x\}$ indicates capture-avoiding substitution of value v for variable x in expression e.

The label pc in a machine configuration represents the security level of information that could be learned by observing the location of the program counter. Instructions executed with a program-counter label of pc are restricted so that they update only to memory locations with labels more secure than pc. For example, [*E3*] shows that it is valid to store a value to a memory location of type σ only if the security label of the data joined with the security labels of the program counter and the reference itself is lower than $\mathsf{label}(\sigma)$, the security clearance needed to read the data stored at that location. Rules [*E6*] and [*E7*] show how the program-counter label changes after branching on data of security level ℓ. Observing which branch is taken reveals information about the condition variable, and so the program counter must have the higher security label $\mathsf{pc} \sqcup \ell$.

As shown in rules [*P1*]–[*P3*], computed values are stamped with the pc label. Checks like the one on [*E3*] prevent illegal information flows. The two `let` rules ([*E1*] and [*E4*]) substitute the bound value in the rest of the program.

Operationally, the rules for `goto` and `lgoto` are very similar—each causes control to be transfered to the target continuation. They differ in their treatment of the program-counter label, as seen in rules [*E8*] and [*E9*]. Ordinary continuations require that the pc before the jump be bounded above by the label associated with the body of the continuation, preventing implicit flows. Linear continuations instead cause the program-counter label to be restored (potentially lowered) to that of the context in which they were declared.

3.3 Static Semantics

The type system for the secure CPS language enforces the linearity and ordering constraints on continuations and guarantees that security labels on values are re-

$$[P1]\quad \langle M,\, \mathsf{pc},\, bv_\ell\rangle \Downarrow bv_{\ell\sqcup\mathsf{pc}} \qquad [P2]\quad \langle M,\, \mathsf{pc},\, n_\ell \oplus n'_{\ell'}\rangle \Downarrow (n[\![\oplus]\!]n')_{\ell\sqcup\ell'\sqcup\mathsf{pc}}$$

$$[P3]\quad \frac{M(L^\sigma) = bv_{\ell'}}{\langle M,\, \mathsf{pc},\, \mathtt{deref}(L^\sigma_\ell)\rangle \Downarrow bv_{\ell\sqcup\ell'\sqcup\mathsf{pc}}}$$

$$[E1]\quad \frac{\langle M,\, \mathsf{pc},\, prim\rangle \Downarrow v}{\langle M,\, \mathsf{pc},\, \mathtt{let}\ x = prim\ \mathtt{in}\ e\rangle \longmapsto \langle M,\, \mathsf{pc},\, e\{v/x\}\rangle}$$

$$[E2]\quad \frac{\ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad L^\sigma \notin Dom(M)}{\langle M,\, \mathsf{pc},\, \mathtt{let}\ x = \mathtt{ref}^\sigma_{\ell'}\ bv_\ell\ \mathtt{in}\ e\rangle \longmapsto \langle M[L^\sigma \leftarrow bv_{\ell\sqcup\mathsf{pc}}],\, \mathsf{pc},\, e\{L^\sigma_{\ell'\sqcup\mathsf{pc}}/x\}\rangle}$$

$$[E3]\quad \frac{\ell \sqcup \ell' \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma) \quad L^\sigma \in Dom(M)}{\langle M,\, \mathsf{pc},\, \mathtt{set}\ L^\sigma_\ell := bv_{\ell'}\ \mathtt{in}\ e\rangle \longmapsto \langle M[L^\sigma \leftarrow bv_{\ell\sqcup\ell'\sqcup\mathsf{pc}}],\, \mathsf{pc},\, e\rangle}$$

$$[E4]\quad \langle M,\, \mathsf{pc},\, \mathtt{letlin}\ y = lv\ \mathtt{in}\ e\rangle \longmapsto \langle M,\, \mathsf{pc},\, e\{lv/y\}\rangle$$

$$[E5]\quad \langle M,\, \mathsf{pc},\, \mathtt{let}\ \langle\rangle = \langle\rangle\ \mathtt{in}\ e\rangle \longmapsto \langle M,\, \mathsf{pc},\, e\rangle$$

$$[E6]\quad \langle M,\, \mathsf{pc},\, \mathtt{if0}\ 0_\ell\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \longmapsto \langle M,\, \mathsf{pc} \sqcup \ell,\, e_1\rangle$$

$$[E7]\quad \langle M,\, \mathsf{pc},\, \mathtt{if0}\ n_\ell\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \longmapsto \langle M,\, \mathsf{pc} \sqcup \ell,\, e_2\rangle \qquad (n \neq 0)$$

$$[E8]\quad \frac{\mathsf{pc} \sqsubseteq \mathsf{pc}' \quad v = (\lambda[\mathsf{pc}']f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell \quad e' = e\{v/f\}\{bv_{\ell'\sqcup\mathsf{pc}}/x\}\{lv/y\}}{\langle M,\, \mathsf{pc},\, \mathtt{goto}\ (\lambda[\mathsf{pc}']f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell\ bv_{\ell'}\ lv\rangle \longmapsto \langle M,\, \mathsf{pc}' \sqcup \ell,\, e'\rangle}$$

$$[E9]\ \langle M,\, \mathsf{pc},\, \mathtt{lgoto}\ (\lambda\langle\mathsf{pc}'\rangle(x\!:\!\sigma, y\!:\!\kappa).\, e)\ bv_\ell\ lv\rangle \longmapsto \langle M,\, \mathsf{pc}',\, e\{bv_{\ell\sqcup\mathsf{pc}}/x\}\{lv/y\}\rangle$$

Fig. 3. Expression Evaluation

spected. Together, these restrictions rule out illegal information flows and impose enough structure on the language for us to prove a non-interference property.

As in other mixed linear–non-linear type systems [31], two separate type contexts are used. Γ is a finite partial map from non-linear variables to security types, whereas K is an *ordered* list (with concatenation denoted by ",") mapping linear variables to their types. The order in which continuations appear in K defines the order in which they are invoked: Given $\mathrm{K} = \bullet, (y_n\!:\!\kappa_n), \ldots, (y_1\!:\!\kappa_1)$, the continuations will be executed in the order $y_1 \ldots y_n$. The context Γ admits the usual weakening and exchange rules (which we omit), but K does not. The two contexts are separated by $\|$ in the judgments to make them more distinct, and $\bullet$ denotes an empty context.

Figures 4 and 5 show the rules for typechecking. The judgment form $\Gamma \vdash v : \sigma$ says that ordinary value v has security type σ in context Γ. Linear values may mention linear variables and so have judgments of the form $\Gamma \parallel \mathrm{K} \vdash lv : \kappa$. Primitive operations may not contain linear variables, but the security of the value produced depends on the program-counter: $\Gamma\ [\mathsf{pc}] \vdash prim : \sigma$ says that in context Γ where the program-counter label is bounded above by pc, $prim$ computes a value of type σ. Similarly, $\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e$ means that expression e is

$$[TV1]\quad \overline{\Gamma \vdash n_\ell : \mathsf{int}_\ell}$$

$$[TV2]\quad \overline{\Gamma \vdash \langle\rangle_\ell : \mathsf{1}_\ell}$$

$$[TV3]\quad \overline{\Gamma \vdash L^\sigma_\ell : \sigma\ \mathsf{ref}_\ell}$$

$$[TV4]\quad \frac{}{\Gamma \vdash x : \sigma}\ \Gamma(x) = \sigma$$

$$[TV5]\quad \frac{f, x \notin \mathit{Dom}(\Gamma) \quad \sigma' = ([\mathsf{pc}](\sigma,\kappa) \to \mathsf{0})_\ell \quad \Gamma, f\!:\!\sigma', x\!:\!\sigma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e}{\Gamma \vdash (\lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell : \sigma'}$$

$$[TV6]\quad \frac{\Gamma \vdash v : \sigma \quad \vdash \sigma \leq \sigma'}{\Gamma \vdash v : \sigma'}$$

$$[S1]\quad \frac{\mathsf{pc}' \sqsubseteq \mathsf{pc} \quad \vdash \sigma' \leq \sigma \quad \vdash \kappa' \leq \kappa}{\vdash [\mathsf{pc}](\sigma,\kappa) \to \mathsf{0} \quad \leq \quad [\mathsf{pc}'](\sigma',\kappa') \to \mathsf{0}} \qquad [S2]\quad \frac{\vdash \tau \leq \tau' \quad \ell \sqsubseteq \ell'}{\vdash \tau_\ell \leq \tau'_{\ell'}}$$

$$[TL1]\quad \overline{\Gamma \parallel \bullet \vdash \langle\rangle : \mathsf{1}}$$

$$[TL2]\quad \overline{\Gamma \parallel y\!:\!\kappa \vdash y : \kappa}$$

$$[TL3]\quad \frac{x \notin \mathit{Dom}(\Gamma), y \notin \mathit{Dom}(\mathrm{K}) \quad \kappa' = (\sigma,\kappa) \to \mathsf{0} \quad \Gamma, x\!:\!\sigma \parallel y\!:\!\kappa, \mathrm{K}\ [\mathsf{pc}] \vdash e}{\Gamma \parallel \mathrm{K} \vdash \lambda\langle\mathsf{pc}\rangle(x\!:\!\sigma, y\!:\!\kappa).\, e : \kappa'}$$

Fig. 4. Value and Linear Value Typing

type-safe and contains no illegal information flows in the type context $\Gamma \parallel \mathrm{K}$, when the program-counter label is at most pc. In the latter two forms, pc is a conservative approximation to the information affecting the program counter.

The rules for checking ordinary values, [*TV1*]–[*TV6*] shown in Figure 4, are, for the most part, standard. A value cannot contain free linear variables because discarding (or copying) it would break linearity. A continuation type contains the pc label used to check its body (rule [*TV5*]). The lattice ordering on security labels lifts to a subtyping relationship on values (rule [*S2*]). Continuations exhibit the expected contravariance (rule [*S1*]). We omit the obvious reflexivity and transitivity rules. Reference types are invariant, as usual.

Linear values are checked using rules [*TL1*]–[*TL3*]. They may safely mention free linear variables, but the variables must not be discarded or reordered. Thus, unit checks only in the empty linear context (rule [*TL1*]), and a linear variable checks only when it is alone in the context (rule [*TL2*]). In a linear continuation (rule [*TL3*]), the linear argument, y, is the tail of the stack of continuations yet to be invoked. Intuitively, this judgment says that the continuation body e must invoke the continuations in K before jumping to y.

The rules for primitive operations ([*TP1*]–[*TP3*] in Figure 5) require that the calculated value have security label at least as restrictive as the current pc, reflecting the "label stamping" behavior of the operational semantics. Values read through `deref` (rule [*TP3*]) pick up the label of the reference as well, which prevents illegal information flows due to aliasing.

Rule [*TE4*] illustrates how the conservative bound on the security level of the program-counter is propagated: The label used to check the branches is the label before the test, pc, joined with the label on the data being tested, ℓ. The rule for `goto`, [*TE8*], restricts the program-counter label of the calling context, pc, joined with the label on the continuation itself, ℓ, to be less than the program-counter label under which the body was checked, pc'. This prevents

$$[TP1]\ \frac{\Gamma \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma\ [\mathsf{pc}] \vdash v : \sigma} \qquad [TP2]\ \frac{\Gamma \vdash v : \mathsf{int}_\ell \quad \Gamma \vdash v' : \mathsf{int}_\ell \quad \mathsf{pc} \sqsubseteq \ell}{\Gamma\ [\mathsf{pc}] \vdash v \oplus v' : \mathsf{int}_\ell}$$

$$[TP3]\ \frac{\Gamma \vdash v : \sigma\ \mathsf{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma \sqcup \ell)}{\Gamma\ [\mathsf{pc}] \vdash \mathtt{deref}(v) : \sigma \sqcup \ell}$$

$$[TE1]\ \frac{\Gamma\ [\mathsf{pc}] \vdash \mathit{prim} : \sigma \qquad \Gamma, x{:}\sigma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e}{\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathit{prim}\ \mathtt{in}\ e} \qquad [TE2]\ \frac{\Gamma \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \ell \sqcup \mathsf{label}(\sigma) \qquad \Gamma, x{:}\sigma\ \mathsf{ref}_\ell \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e}{\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{ref}^{\sigma}_{\ell}\ v\ \mathtt{in}\ e}$$

$$[TE3]\ \frac{\Gamma \vdash v : \sigma\ \mathsf{ref}_\ell \quad \Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e \qquad \Gamma \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma)}{\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash \mathtt{set}\ v := v'\ \mathtt{in}\ e} \qquad [TE4]\ \frac{\Gamma \vdash v : \mathsf{int}_\ell \quad \Gamma \parallel \mathrm{K}\ [\mathsf{pc} \sqcup \ell] \vdash e_i}{\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash \mathtt{if0}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}$$

$$[TE5]\ \frac{\Gamma \parallel \mathrm{K}_2 \vdash \lambda\langle \mathsf{pc}' \rangle (x{:}\sigma, y{:}\kappa).\, e' : (\sigma, \kappa) \to \mathsf{0} \qquad \mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad \Gamma \parallel \mathrm{K}_1, y{:}(\sigma,\kappa) \to \mathsf{0}\ [\mathsf{pc}] \vdash e}{\Gamma \parallel \mathrm{K}_1, \mathrm{K}_2\ [\mathsf{pc}] \vdash \mathtt{letlin}\ y = \lambda\langle \mathsf{pc}' \rangle (x{:}\sigma, y{:}\kappa).\, e'\ \mathtt{in}\ e}$$

$$[TE6]\ \frac{\Gamma \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma \parallel \bullet\ [\mathsf{pc}] \vdash \mathtt{halt}^{\sigma}\ v} \qquad [TE7]\ \frac{\Gamma \parallel \mathrm{K}_1 \vdash \mathit{lv} : \mathsf{1} \quad \Gamma \parallel \mathrm{K}_2\ [\mathsf{pc}] \vdash e}{\Gamma \parallel \mathrm{K}_1, \mathrm{K}_2\ [\mathsf{pc}] \vdash \mathtt{let}\ \langle\rangle = \mathit{lv}\ \mathtt{in}\ e}$$

$$[TE8]\ \frac{\Gamma \vdash v : ([\mathsf{pc}'](\sigma,\kappa) \to \mathsf{0})_\ell \qquad \Gamma \vdash v' : \sigma \quad \Gamma \parallel \mathrm{K} \vdash \mathit{lv} : \kappa \qquad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma \parallel \mathrm{K}\ [\mathsf{pc}] \vdash \mathtt{goto}\ v\ v'\ \mathit{lv}} \qquad [TE9]\ \frac{\Gamma \parallel \mathrm{K}_2 \vdash \mathit{lv} : (\sigma,\kappa) \to \mathsf{0} \qquad \Gamma \vdash v : \sigma \quad \Gamma \parallel \mathrm{K}_1 \vdash \mathit{lv}' : \kappa \qquad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma \parallel \mathrm{K}_1, \mathrm{K}_2\ [\mathsf{pc}] \vdash \mathtt{lgoto}\ \mathit{lv}\ v\ \mathit{lv}'}$$

Fig. 5. Primitive Operation and Expression Typing

implicit information flows from propagating into function bodies. Likewise, the values passed to a continuation (linear or not) must pick up the calling context's pc (via the constraint $\mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)$) because they carry information about the context in which the continuation was invoked.

The rule for `halt`, [*TE6*], requires an empty linear context, indicating that the program consumes all linear continuations before stopping. The σ annotating `halt` is the type of the final output of the program; its label should be constrained by the security clearance of the user of the program.

The rules for `letlin`, [*TE5*], and `lgoto`, [*TE9*], manipulate the linear context to enforce the ordering property on continuations. For `letlin`, the linear context is split into K_1 and K_2. The body e is checked under the assumption that the new continuation, y, is invoked before any continuation in K_1. Because y invokes the continuations in K_2 before its linear argument (as described above for rule [*TL3*]), the ordering $\mathrm{K}_1, \mathrm{K}_2$ in subsequent computation will be respected. The rule for `lgoto` works similarly.

Linear continuations capture the pc (or a more restrictive label) of the context in which they are introduced, as shown in rule [*TE5*]. Unlike the rule for `goto`, the rule for `lgoto` does not constrain the pc, because the linear continuation *restores* the program-counter label to the one it captured. Because linear continuations capture the pc of their introduction context, we make the mild assumption that *initial programs* introduce all linear continuation values (not variables) via `letlin`. During execution this constraint is not required, and programs in the image of the translation satisfy this property.

This type system is sound with respect to the operational semantics [36]. The proof is, for the most part, standard, following in the style of Wright and Felleisen [35]. We simply state the lemmas necessary for the discussion of the non-interference result of the next section.

Lemma 1 (Subject Reduction). *If* $\bullet \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e$ *and* M *is a well-formed memory such that* $Loc(e) \subseteq Dom(M)$ *and* $\langle M, \mathsf{pc}, e\rangle \longmapsto \langle M', \mathsf{pc}', e'\rangle$*, then* $\bullet \parallel \mathrm{K}\ [\mathsf{pc}'] \vdash e'$ *and* M' *is a well-formed memory such that* $Loc(e') \subseteq Dom(M')$*.*

Lemma 2 (Progress). *If* $\bullet \parallel \bullet\ [\mathsf{pc}] \vdash e$ *and* M *is well-formed and* $Loc(e) \subseteq Dom(M)$*, then either* e *is of the form* $\mathtt{halt}^{\sigma}\ v$ *or there exist* M'*,* pc'*, and* e' *such that* $\langle M, \mathsf{pc}, e\rangle \longmapsto \langle M', \mathsf{pc}', e'\rangle$

Note that Subject Reduction holds for terms containing free occurrences of linear variables. This fact is important for proving that the ordering on linear continuations is respected. The Progress lemma (and hence Soundness) applies only to closed programs, as usual.

4 Non-Interference

This section proves a non-interference result for the secure CPS language, generalizing Smith and Volpano's preservation-style argument [30]. A technical report [36] gives a detailed account of our approach in a more expressive language.

Informally, the non-interference result shows that low-security computations are not able to observe high-security data. Here, "low-security" refers to the set of security labels $\sqsubseteq \zeta$, where ζ is an arbitrary point in $\mathcal{L}$, and "high-security" refers to labels $\not\sqsubseteq \zeta$. The proof shows that high-security data and computation can be arbitrarily changed without affecting the value of any computed low-security result. Furthermore, memory locations visible to low-security observers (locations storing data labeled $\sqsubseteq \zeta$) are also unaffected by high-security values.

Non-interference reduces to showing that two programs are equivalent from the low-security perspective. Given a program e_1 that operates on high- and low-security data, it suffices to show that e_1 is low-equivalent to the program e_2 that differs from e_1 in its high-security computations.

How do we show that e_1 and e_2 behave the same from the low-security point of view? If $\mathsf{pc} \sqsubseteq \zeta$, meaning that e_1 and e_2 may perform actions visible to low observers, they necessarily must perform the same computation on low-security values. Yet e_1 and e_2 may differ in their behavior on high-security data and still be equivalent from the low perspective. To show their equivalence, we should find substitutions γ_1 and γ_2 containing the relevant high-security data such that $e = \gamma_1(e)$ and $e_2 = \gamma_2(e)$—both e_1 and e_2 look the same after factoring out the high-security data.

On the other hand, when $\mathsf{pc} \not\sqsubseteq \zeta$, no matter what e_1 and e_2 do their actions should not be visible from the low point of view; their computations are irrelevant. The operational semantics guarantee that the program-counter label is monotonically increasing *except* when a linear continuation is invoked. If e_1 invokes a linear continuation causing pc to fall below ζ, e_2 must follow suit; otherwise the low-security observer can distinguish them. The ordering on linear continuations forces e_2 to invoke the same low-security continuation as e_1.

The crucial invariant maintained by well-typed programs is that it is possible to factor out (via substitutions) the relevant high-security values and those linear continuations that reset the program-counter label to be $\sqsubseteq \zeta$.

Definition 1 (Substitutions). *For context Γ, let $\gamma \models \Gamma$ mean that γ is a finite map from variables to closed values such that $Dom(\gamma) = Dom(\Gamma)$ and for every $x \in Dom(\gamma)$ it is the case that $\bullet \vdash \gamma(x) : \Gamma(x)$.*

For linear context K, write $\Gamma \vdash k \models \mathrm{K}$ to indicate that k is a finite map of variables to linear values (with free variables from Γ) with the same domain as K and such that for every $y \in Dom(k)$ we have $\Gamma \parallel \bullet \vdash k(y) : \mathrm{K}(y)$.

Substitution application, written $\gamma(e)$, indicates the capture-avoiding substitution of the value $\gamma(x)$ for free occurrences of x in e, for each x in the domain of γ ($k(e)$ is defined similarly).

Linear continuations that set the pc label $\not\sqsubseteq \zeta$ may appear in low-equivalent programs, because, from the low-security point of view, they are not relevant.

Definition 2 (`letlin` Invariant). *A term satisfies the `letlin` invariant if every linear continuation expression $\lambda\langle\mathsf{pc}\rangle(x\,{:}\,\sigma, y\,{:}\,\kappa).\, e$ appearing in the term is either in the binding position of a `letlin` or satisfies $\mathsf{pc} \not\sqsubseteq \zeta$.*

If substitution k contains only low-security linear continuations and $k(e)$ is a closed term such that e satisfies the `letlin` invariant, then all the low-security continuations not `letlin`-bound in e must be obtained from k. This invariant ensures that k factors out all of the relevant continuations from $k(e)$.

Extending these ideas to values, memories, and machine configurations we obtain the definitions below:

Definition 3 (ζ-Equivalence).

$\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$	*If $\gamma_1, \gamma_2 \models \Gamma$ and for every $x \in Dom(\Gamma)$ it is the case that $\mathsf{label}(\gamma_i(x)) \not\sqsubseteq \zeta$ and $\gamma_i(x)$ satisfies the `letlin` invariant.*
$\Gamma \parallel \mathrm{K} \vdash k_1 \approx_\zeta k_2$	*If $\Gamma \vdash k_1, k_2 \models \mathrm{K}$ and for every $y \in Dom(\mathrm{K})$ it is the case that $k_1(y) \equiv_\alpha k_2(y) = \lambda\langle\mathsf{pc}\rangle(x\,{:}\,\sigma, y'\,{:}\,\kappa).\, e$ such that $\mathsf{pc} \sqsubseteq \zeta$ and e satisfies the `letlin` invariant.*
$v_1 \approx_\zeta v_2 : \sigma$	*If there exist Γ, γ_1, and γ_2 plus terms $v_1' \equiv_\alpha v_2'$ such that $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$, and $\Gamma \vdash v_i' : \sigma$ and $v_i = \gamma_i(v_i')$ and each v_i' satisfies the `letlin` invariant.*
$M_1 \approx_\zeta M_2$	*If for all $L^\sigma \in Dom(M_1) \cup Dom(M_2)$ if $\mathsf{label}(\sigma) \sqsubseteq \zeta$, then $L^\sigma \in Dom(M_1) \cap Dom(M_2)$ and $M_1(L^\sigma) \approx_\zeta M_2(L^\sigma) : \sigma$.*

Definition 4 (Non-Interference Invariant). *The non-interference invariant is a predicate on machine configurations, written* $\Gamma \parallel \mathrm{K} \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \approx_\zeta \langle M_1,\, \mathsf{pc}_2,\, e_2\rangle$ *that holds if the following conditions are all met:*

- **(i)** *There exist substitutions* $\gamma_1, \gamma_2, k_1, k_2$ *and terms* e_1' *and* e_2' *such that* $e_1 = \gamma_1(k_1(e_1'))$ *and* $e_2 = \gamma_2(k_2(e_2'))$.
- **(ii)** *Either* (*a*) $\mathsf{pc}_1 = \mathsf{pc}_2 \sqsubseteq \zeta$ *and* $e_1' \equiv_\alpha e_2'$ *or* (*b*) $\Gamma \parallel \mathrm{K}\ [\mathsf{pc}_1] \vdash e_1'$ *and* $\Gamma \parallel \mathrm{K}\ [\mathsf{pc}_2] \vdash e_2'$ *and* $\mathsf{pc}_i \not\sqsubseteq \zeta$.
- **(iii)** $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ *and* $\Gamma \parallel \mathrm{K} \vdash k_1 \approx_\zeta k_2$
- **(iv)** $Loc(e_1) \subseteq Dom(M_1)$ *and* $Loc(e_2) \subseteq Dom(M_2)$ *and* $M_1 \approx_\zeta M_2$.
- **(v)** *Both* e_1' *and* e_2' *satisfy the* `letlin` *invariant.*

Our proof is a preservation argument showing that the Non-Interference Invariant holds after each transition. When the pc is low, equivalent configurations execute in lock step (modulo high-security data). After the program branches on high-security information (or jumps to a high-security continuation), the two programs may temporarily get out of sync, but during that time they may affect only high-security data. If the program counter drops low again (via a linear continuation), both computations return to lock-step execution.

We first show that ζ-equivalent configuration evaluate in lock step as long as the program counter has low security.

Lemma 3 (Low-pc Step). *Suppose* $\Gamma \parallel \mathrm{K} \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$, $\mathsf{pc}_1 \sqsubseteq \zeta$ *and* $\mathsf{pc}_2 \sqsubseteq \zeta$. *If* $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \longmapsto \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$, *then* $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$ *and there exist* Γ' *and* K' *such that* $\Gamma' \parallel \mathrm{K}' \vdash \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle \approx_\zeta \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$.

Proof. (Sketch) We omit the details due to space constraints. Reason by cases on the security of the value used in the transition—if it's label is $\sqsubseteq \zeta$, α-equivalence implies both programs behave identically, otherwise, we extend the substitutions corresponding to Γ' to contain the differing high-security data. □

Next, we prove that linear continuations do indeed get called in the order described by the linear context.

Lemma 4 (Linear Continuation Ordering). *Assume* $\mathrm{K} = y_n : \kappa_n, \ldots, y_1 : \kappa_1$, *each* κ_i *is a linear continuation type, and* $\bullet \parallel \mathrm{K}\ [\mathsf{pc}] \vdash e$. *If* $\bullet \vdash k \models \mathrm{K}$, *then in the evaluation starting from any well-formed configuration* $\langle M,\, \mathsf{pc},\, k(e)\rangle$, *the continuation* $k(y_1)$ *will be invoked before any other* $k(y_i)$.

Proof. The operational semantics and Subject Reduction are valid for open terms. Progress, however, does not hold for open terms. Evaluate the open term e in the configuration $\langle M,\, \mathsf{pc},\, e\rangle$. If the computation diverges, none of the y_i's ever reach an active position, and hence are not invoked. Otherwise, the computation must get stuck (it can't halt because Subject Reduction implies that all configurations are well-typed; the `halt` expression requires an empty linear context). The stuck term must be of the form `lgoto` $y_i\ v\ lv$, and because it is well-typed, rule [*TE9*] implies that $y_i = y_1$. □

We use the ordering lemma to prove that equivalent high-security configurations eventually return to equivalent low-security configurations.

Lemma 5 (High-pc Step). *If* $\Gamma \parallel \mathrm{K} \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$ *and* $\mathsf{pc}_i \not\sqsubseteq \zeta$*, then* $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \longmapsto \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$ *implies that either* e_2 *diverges or* $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto^* \langle M_2',\, \mathsf{pc}_2',\, e_2'\rangle$ *and there exist* Γ' *and* K' *such that* $\Gamma' \parallel \mathrm{K}' \vdash \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle \approx_\zeta \langle M_2',\, \mathsf{pc}_2',\, e_2\rangle$.

Proof. (Sketch) By cases on the transition step of the first configuration. Because $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and all rules except [*E9*] increase the program-counter label, we may choose zero steps for e_2 and still show that $\approx_\zeta$ is preserved. Condition (ii) holds via part(b). The other invariants follow because all values computed and memory locations written to must have labels higher than pc_1 (and hence $\not\sqsubseteq \zeta$). Thus, the only memory locations affected are high-security: $M_1' \approx_\zeta M_2 = M_2'$. Similarly, [*TE5*] forces linear continuations introduced by e_1 to have $\mathsf{pc} \not\sqsubseteq \zeta$. Substituting them in e_1 maintains clause (vi) of the invariant.

Now consider the case for [*E9*]. Let $e_1 = \gamma_1(k_1(e_1''))$, then $e_1'' = \mathtt{lgoto}\; lv\; v_1\; lv_1$ for some lv. If lv is not a variable, clause (vi) ensures that the program counter in lv's body is $\not\sqsubseteq \zeta$. Pick 0 steps for the second configuration as above. Otherwise, if lv is a variable, y, then [*TE9*] guarantees that $\mathrm{K} = \mathrm{K}', y\,{:}\,\kappa$. By assumption, $k_1(y) = \lambda\langle\mathsf{pc}\rangle(x\,{:}\,\sigma, y'\,{:}\,\kappa').\, e$, where $\mathsf{pc} \sqsubseteq \zeta$. Assume e_2 does not diverge. By the ordering lemma, $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto^* \langle M_2',\, \mathsf{pc}_2',\, \mathtt{lgoto}\; k_2(y)\; v_2\; lv_2\rangle$. Simple induction on the length of this transition sequence shows that $M_2 \approx_\zeta M_2'$, because the program counter may not become $\sqsubseteq \zeta$. Thus, $M_1' = M_1 \approx_\zeta M_2 \approx_\zeta M_2'$. By invariant (iii), $k_2(y) \equiv_\alpha k_1(y)$. Furthermore, [*TE9*] requires that $\mathsf{label}(\sigma) \not\sqsubseteq \zeta$. Let $\Gamma' = \Gamma, x\,{:}\,\sigma$, $\gamma_1' = \gamma_1\{x \mapsto \gamma_1(v_1) \sqcup \mathsf{pc}_1\}$, $\gamma_2' = \gamma_2\{x \mapsto \gamma_2(v_2) \sqcup \mathsf{pc}_2\}$; take k_1' and k_2' to be the restrictions of k_1 and k_2 to the domain of K', and choose $e_1' = \gamma_1'(k_1'(e))$ and $e_2' = \gamma_2'(k_2'(e))$. All of the necessary conditions are satisfied as is easily verified via the operational semantics. $\square$

Finally, we use the above lemmas to prove non-interference. Assume a program that computes a low-security value has access to high-security data. Arbitrarily changing the high-security data does not affect the program's result.

First, some convenient notation for the initial continuation: Let $\mathit{stop}(\tau_\ell) : \kappa_{\mathit{stop}} = \lambda\langle\bot\rangle(x\,{:}\,\tau_\ell, y\,{:}\,\mathsf{1}).\, \mathtt{let}\ \langle\rangle = y\ \mathtt{in}\ \mathtt{halt}^{\tau_\ell}\ x$ where $\kappa_{\mathit{stop}} = (\tau_\ell, \mathsf{1}) \to \mathsf{0}$.

Theorem 1 (Non-interference). *Suppose* $x : \sigma \parallel y : \kappa_{\mathit{stop}}\ [\bot] \vdash e$ *for some initial program* e*. Further suppose that* $\mathsf{label}(\sigma) \not\sqsubseteq \zeta$ *and* $\bullet \vdash v_1, v_2 : \sigma$*. Then*

$$\langle\emptyset,\, \bot,\, e\{v_1/x\}\{\mathit{stop}(\mathsf{int}_\zeta)/y\}\rangle \longmapsto^* \langle M_1,\, \zeta,\, \mathtt{halt}^{\mathsf{int}_\zeta}\ n_{\ell_1}\rangle$$
$$\textit{and}$$
$$\langle\emptyset,\, \bot,\, e\{v_2/x\}\{\mathit{stop}(\mathsf{int}_\zeta)/y\}\rangle \longmapsto^* \langle M_2,\, \zeta,\, \mathtt{halt}^{\mathsf{int}_\zeta}\ m_{\ell_2}\rangle$$

implies that $M_1 \approx_\zeta M_2$ *and* $n = m$.

Proof. It is easy to verify that

$$x\,{:}\,\sigma \parallel y\,{:}\,\kappa_{\mathit{stop}} \vdash \langle\emptyset,\, \bot,\, e\{v_1/x\}\{\mathit{stop}(\mathsf{int}_\zeta)/y\}\rangle \approx_\zeta \langle\emptyset,\, \bot,\, e\{v_2/x\}\{\mathit{stop}(\mathsf{int}_\zeta)/y\}\rangle$$

by letting $\gamma_1 = \{x \mapsto v_1\}$, $\gamma_2 = \{x \mapsto v_2\}$, and $k_1 = k_2 = \{y \mapsto \mathit{stop}(\mathsf{int}_\zeta)\}$. Induction on the length of the first expression's evaluation sequence, using the Low- and High-pc Step lemmas plus the fact that the second evaluation sequence terminates implies that $\Gamma \parallel K \vdash \langle M_1, \zeta, \mathtt{halt}^{\mathsf{int}_\zeta}\ n_{\ell_1}\rangle \approx_\zeta \langle M_2, \zeta, \mathtt{halt}^{\mathsf{int}_\zeta}\ m_{\ell_2}\rangle$. Clause (iv) of the Non-interference Invariant implies that $M_1 \approx_\zeta M_2$. Soundness implies that $\ell_1 \sqsubseteq \zeta$ and $\ell_2 \sqsubseteq \zeta$. This means, because of clause (iii), that neither n_{ℓ_1} nor m_{ℓ_2} are in the range of γ'_i. Thus, the integers present in the `halt` expressions do not arise from substitution. Because $\zeta \sqsubseteq \zeta$, clause (ii) implies that $\mathtt{halt}^{\mathsf{int}_\zeta}\ n_{\ell_1} \equiv_\alpha \mathtt{halt}^{\mathsf{int}_\zeta}\ m_{\ell_2}$, from which we obtain $n = m$ as desired. □

5 Translation

This section presents a CPS translation for a secure, imperative, higher-order language that includes only the features essential to demonstrating the translation. Its type system is adapted from the SLam calculus [16] to follow our "label stamping" operational semantics. The judgment $\Gamma \vdash_{\mathsf{pc}} e : s$ shows that expression e has source type s under type context Γ, assuming the program-counter label is bounded above by pc.

Source types are similar to those of the target, except that instead of continuations there are functions. Function types are labeled with their *latent effect*, a lower bound on the security level of memory locations that will be written to by that function. The type translation, following previous work on typed CPS conversion [15], is given in terms of three mutually recursive functions: $(-)^*$, for base types, $(-)^+$ for security types, and $(-)^-$ to linear continuation types:

$$\begin{array}{lll} \mathsf{int}^* = \mathsf{int} & (s\ \mathsf{ref})^* = s^+\ \mathsf{ref} & (s_1 \xrightarrow{\ell} s_2)^* = [\ell](s_1^+, s_2^-) \to \mathsf{0} \\ t_\ell^+ = (t^*)_\ell & s^- = (s^+, \mathsf{1}) \to \mathsf{0} & \end{array}$$

Figure 6 shows the term translation as a type-directed map from source typing derivations to target terms. For simplicity, we present an un-optimizing CPS translation, although we expect that first-class linear continuations will support more sophisticated translations, such as tail-call optimization [8]. To obtain the full translation of a closed term e of type s, we use the initial continuation from Section 4: `letlin stop =` $\mathit{stop}(s^+)$ `in` $[\![\emptyset \vdash_\ell e : s]\!]\mathtt{stop}$.

The basic lemma for establishing correctness of the translation is proved by induction on the typing derivation of the source term. This result also shows that the CPS language is at least as precise as the source.

Lemma 6 (Type Translation). $\Gamma \vdash_\ell e : s \Rightarrow \Gamma^+ \parallel y{:}s^-\ [\ell] \vdash [\![\Gamma \vdash_\ell e : s]\!]y$.

6 Related Work

The constraints imposed by linearity can be seen as a form of resource management [13], in this case limiting the set of possible future computations. Linear continuations have been studied in terms of their category theoretic semantics [11] and also as a computational interpretation of classical logic [5]. Polakow and Pfenning have investigated the connections between ordered linear-logic, stack-based abstract machines, and CPS [24]. Linearity also plays a role in security types for process calculi such as the π-calculus [17]. Because the usual translation of the λ-calculus into the π-calculus can be seen as a form of CPS

$$[\![\Gamma, x{:}s' \vdash_{\mathsf{pc}} x : s' \sqcup \mathsf{pc}]\!]y \Rightarrow \mathtt{lgoto}\ y\ x\ \langle\rangle$$

$$\left[\!\!\left[\frac{\Gamma, f{:}s, x{:}s_1 \vdash_{\mathsf{pc}'} e : s_2}{\Gamma \vdash_{\mathsf{pc}} (\mu f(x{:}s_1).\, e)_\ell : s \sqcup \mathsf{pc}}\right]\!\!\right]y \Rightarrow \begin{cases} \mathtt{lgoto}\ y\ (\lambda[\mathsf{pc}']f(x{:}s_1^+, y'{:}s_2^-). \\ \qquad\qquad [\![\Gamma, f{:}s, x{:}s_1 \vdash_{\mathsf{pc}'} e : s_2]\!]y')_\ell\ \langle\rangle \end{cases}$$

$$\left[\!\!\left[\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{pc}} e : s \\ \Gamma \vdash_{\mathsf{pc}} e' : s_1 \\ \ell \sqsubseteq \mathsf{pc}' \sqcap \mathsf{label}(s_1)\end{array}}{\Gamma \vdash_{\mathsf{pc}} (e\ e') : s_2}\right]\!\!\right]y \Rightarrow \begin{cases} \mathtt{letlin}\ k_1\ \mathtt{=}\ \lambda\langle\mathsf{pc}\rangle(f{:}s^+, y_1{:}1). \\ \qquad \mathtt{let}\ \langle\rangle\ \mathtt{=}\ y_1\ \mathtt{in} \\ \qquad \mathtt{letlin}\ k_2\mathtt{=}\lambda\langle\mathsf{pc}\rangle(x{:}s_1^+, y_2{:}1). \\ \qquad\qquad \mathtt{let}\ \langle\rangle\mathtt{=}\ y_2\ \mathtt{in} \\ \qquad\qquad \mathtt{goto}\ f\ x\ y \\ \qquad \mathtt{in}\ [\![\Gamma \vdash_{\mathsf{pc}} e' : s_1]\!]k_2 \\ \mathtt{in}\ [\![\Gamma \vdash_{\mathsf{pc}} e : s]\!]k_1 \end{cases}$$

$$\left[\!\!\left[\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{pc}} e : \mathsf{int}_\ell \\ \Gamma \vdash_{\mathsf{pc}'} e_i : s' \\ \ell \sqsubseteq \mathsf{pc}'\end{array}}{\begin{array}{l}\Gamma \vdash_{\mathsf{pc}} \mathtt{if0}\ e\ \mathtt{then} \\ \qquad e_1\ \mathtt{else}\ e_2 : s'\end{array}}\right]\!\!\right]y \Rightarrow \begin{cases} \mathtt{letlin}\ k_1\ \mathtt{=}\ \lambda\langle\mathsf{pc}\rangle(x{:}\mathsf{int}_\ell^+, y_1{:}1). \\ \qquad \mathtt{let}\ \langle\rangle\ \mathtt{=}\ y_1\ \mathtt{in} \\ \qquad \mathtt{if0}\ x\ \mathtt{then}\ [\![\Gamma \vdash_{\mathsf{pc}'} e_1 : s']\!]y \\ \qquad\qquad \mathtt{else}\ [\![\Gamma \vdash_{\mathsf{pc}'} e_2 : s']\!]y \\ \mathtt{in}\ [\![\Gamma \vdash_{\mathsf{pc}} e : \mathsf{int}_\ell]\!]k_1 \end{cases}$$

$$\left[\!\!\left[\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{pc}} e : s'\ \mathsf{ref}_\ell \\ \Gamma \vdash_{\mathsf{pc}} e' : s' \\ \ell \sqsubseteq \mathsf{label}(s')\end{array}}{\Gamma \vdash_{\mathsf{pc}} e := e' : s'}\right]\!\!\right]y \Rightarrow \begin{cases} \mathtt{letlin}\ k_1\ \mathtt{=}\ \lambda\langle\mathsf{pc}\rangle(x_1{:}s'\ \mathsf{ref}_\ell^+, y_1{:}1). \\ \qquad \mathtt{let}\ \langle\rangle\ \mathtt{=}\ y_1\ \mathtt{in} \\ \qquad \mathtt{letlin}\ k_2\mathtt{=}\lambda\langle\mathsf{pc}\rangle(x_2{:}s'^+, y_2{:}1). \\ \qquad\qquad \mathtt{let}\ \langle\rangle\ \mathtt{=}\ y_2\ \mathtt{in} \\ \qquad\qquad \mathtt{set}\ x_1\ \mathtt{:=}\ x_2\ \mathtt{in} \\ \qquad\qquad \mathtt{lgoto}\ y\ x_2\ \langle\rangle \\ \qquad \mathtt{in}\ [\![\Gamma \vdash_{\mathsf{pc}} e' : s']\!]k_2 \\ \mathtt{in}\ [\![\Gamma \vdash_{\mathsf{pc}} e : s'\ \mathsf{ref}_\ell]\!]k_1 \end{cases}$$

Fig. 6. CPS Translation (Here $s = (s_1 \xrightarrow{\mathsf{pc}'} s_2)_\ell$, and the k_i's and y_i's are fresh.)

translation, it might be enlightening to investigate the connections between security in process calculi and low-level code.

CPS translation has been studied in the context of program analysis [10, 23]. Sabry and Felleisen observed that increased precision in some CPS data flow analyses is due to duplication of analysis along different execution paths [29]. They also note that some analyses "confuse continuations" when applied to CPS programs. Our type system distinguishes linear from non-linear continuations to avoid confusing "calls" with "returns." More recently, Damian and Danvy showed that CPS translation can improve binding-time analysis in the λ-calculus [7], suggesting that the connection between binding-time analysis and security [1] warrants more investigation.

Linear continuations appear to be a higher-order analog to *post-dominators* in a control-flow graph. Algorithms for determining post-dominators (see Muchnick's text [19]) might yield inference techniques for linear continuation types. Conversely, linear continuations might yield a type-theoretic basis for correctness proofs of optimizations based on post-dominators.

Understanding secure information flow in low-level programs is essential to providing secrecy of private data. We have shown that explicit ordering of continuations can improve the precision of security types. Ordered linear continuations constrain the uses of continuations so that implicit flows of information can be controlled more accurately. These constraints also make possible our noninterference proof, the first of its kind for a higher-order, imperative language.

Many thanks to James Cheney, Dan Grossman, François Pottier, Stephanie Weirich, and Lantian Zheng for their comments on drafts of this paper. Thanks also to Jon Riecke for many interesting discussions about the SLam calculus.

References

1. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, 1999.
2. Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.
3. Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, January 2000.
4. Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
5. Gavin Bierman. A classical linear lambda calculus. *Theoretical Computer Science*, 227(1–2):43–78, 1999.
6. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 262–275, 1999.
7. Daniel Damian and Olivier Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 209–220, 2000.
8. Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2:361–391, 1992.
9. Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
10. J. Mylaert Filho and G. Burn. Continuation passing transformations and abstract interpretation. In *Proc. First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*, 1993.
11. Andrzej Filinski. Linear continuations. In *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*, 1992.
12. Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM '93 Conference on Programming Language Design and Implementation*, 1993.
13. Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
14. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, April 1982.
15. Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proc. 20th ACM Symp. on Principles of Programming Languages (POPL)*, 1993.
16. Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.

17. Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer, 2000.
18. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.
19. Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
20. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, San Antonio, TX, USA, January 1999.
21. Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*, pages 129–142, Saint-Malo, France, 1997.
22. George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.
23. Flemming Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265–287, 1982.
24. Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In J. Despeyroux, editor, *2nd Workshop on Logical Frameworks and Meta-languages*, Santa Barbara, California, June 2000.
25. François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, 2000.
26. Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. In *Proceedings of the European Symposium on Programming.* Springer-Verlag, March 1999. LNCS volume 1576.
27. Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop.* IEEE Computer Society Press, July 2000.
28. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation: An International Journal*, 1993.
29. Amr Sabry and Matthias Felleisen. Is continuation-passing useful for data flow analysis? In *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 1–12, 1994.
30. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, San Diego, California, January 1998.
31. David N. Turner and Philip Wadler. Operational interpretations of linear logic. *Theoretical Computer Science*, 2000. To Appear.
32. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
33. Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods.* North Holland, 1990.
34. Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science.* Springer-Verlag, 1993.
35. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.
36. Steve Zdancewic and Andrew C. Myers. Confidentiality and integrity with untrusted hosts. Technical Report 2000-1810, Cornell University, 2000.