Secure Information Flow via Linear Continuations*

Steve Zdancewic (zdance@cs.cornell.edu)
Cornell University†

Andrew C. Myers (andru@cs.cornell.edu)‡
Cornell University

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

Linear continuations impose a stack discipline on the control flow of programs. This additional structure in the type system lets us establish a strong information-flow security property called *noninterference.* We prove that our CPS target language enjoys the noninterference property and we show how to translate secure high-level programs to this low-level language. This noninterference proof is the first of its kind for a language with higher-order functions and state.

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 information 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.

* This is an extended and revised version of a conference paper presented at the European Symposium on Programming, April 2001 [40].

† Department of Computer Science, Cornell University Ithaca NY 14853, USA

‡ 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 of DARPA, AFRL, or the U.S. Government.

There has been much recent work on formulating Denning's original lattice model of information-flow control [12, 13] in terms of type systems for static program verification [1, 20, 24, 25, 29–31, 34, 36]. The desired security property is *noninterference* [18], which states that high-security data is not observable by low-security computation. Continuing the example from above, the high-security data is the credit card information and a low-security computation might be the downloading of stock market reports. Noninterference says that the information transmitted to the insecure network cannot depend on the credit card data—hence, the program cannot improperly transmit the confidential information.

The problem of secure information flow has been well studied in the context of simple imperative languages. Nevertheless, secure information flow in the context of higher-order languages with imperative features is not well understood. Furthermore, all previous work has considered information flow properties in the source language. However, source-level analysis is not enough: Compiler transformations (or bugs!) may introduce new security holes. One appealing option is to verify the *output* of the compiler, for instance via typed assembly language [22] or proof-carrying code [26].

This paper proposes the use of continuation-passing style (CPS) translations [11, 16, 32] as a means of studying noninterference in imperative, higher-order languages. This approach has two benefits. First, CPS expresses higher-order programs in a form that is amenable to proving noninterference results. Our proof of security can be seen as a generalization of previous work by Smith and Volpano [34]. Second, CPS is useful for representing low-level programs [4, 22], which opens up the possibility of verifying the security of compiler output.

We observe in the next section that a naive approach to providing security types for an imperative CPS language yields a system that is too conservative: many secure programs (in the noninterference sense) are rejected. To rectify this problem, we introduce *linear continuations*, which allow information flow control in the CPS target language to be made more precise. Our secure CPS language makes explicit the "folk-theorem" present in the CPS literature that "one continuation is enough," a feature of CPS translation that has only recently begun to be studied [10, 28, 9, 5]. The stack ordering property of linear continuations is crucial to the noninterference argument.

As with previous noninterference results for call-by-value languages [20, 24], the theorem holds only for programs that halt regardless of high-security data. Consequently, termination channels can arise, but because they leak at most one bit per run on average, we consider them acceptable. There are other channels not captured by this notion of noninterference: high-security data can alter the running time of the program or change its memory consumption. Noninterference 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 linear continuations. Section 3 presents the target language, its operational semantics, and the novel features of its type system. The noninterference theorem is discussed in Section 4 (proofs of the interesting lemmas are given in Appendix A), 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 programs [1]. One difficulty is *implicit flows*, which arise from the control flow of the program. Consider the code fragment `A` in Figure 1.[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.")

A programmer using a type system for enforcing information flow policies might assign `x` the type bool_H (high-security boolean) and `z` the type int_L (low-security integer). If `y` were given the type int_H, program fragment `A` would type check, but if `y` were given a low-security type `A` would not type check due to the implicit flow from `x` to `y`. Security-typed languages deal with these implicit flows by associating a security annotation with the *program counter* (which we will usually indicate by pc). In example `A`, the program counter at the point before the `if` statement might be labeled with L to indicate that it does not depend on high-security data. Within the branches of the conditional, however, the program counter depends on the value of `x`, and hence the

[1] We have chosen to give the examples in an imperative pseudo-code in which continuations can be introduced explicitly (as in `k = (λ⟨⟩. halt)`) and invoked (as in `k ⟨⟩`). The actual syntax of the secure CPS language is given in Section 3.1.

```
(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 }
```

Figure 1. Examples of information flow in CPS

pc must be H—the security label of `x`. Values (such as the constants `1` and `2` of the example) pick up the security annotation of the program counter, and consequently when `y` has type int_L the assignment `y := 1` is illegal—the (implicitly) high-security value `1` is being assigned to a low-security memory location.

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 ⟨⟩`. 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. On the other hand, as example `C` shows, if `k` is *not* used in one of the branches, then information about `x` can be learned by observing `z`. Linear type systems [2, 17, 37, 38] 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 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 insufficient for security in the presence of first-class 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. The type system presented in Section 3.4 requires that exactly one linear continuation be available at any point—thus eliminating the possibility of writing code like example `E`. We show in Section 4 that these constraints are sufficient to prove a noninterference result.

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 the type system statically distinguishes between different kinds of continuations, allowing information flow to be analyzed with the same precision as the source.

2.1. Linear Continuations

Before diving into the formal definition of the secure CPS language, it is helpful to have some intuition about what a linear continuation is. Ordinary continuations represent a possible future computation of a program. As such, how they are manipulated encapsulates the control flow aspects of a piece of code. Powerful language constructs such as `call-cc` expose the continuations to the programmer in a first-class way, allowing the user to manipulate the control flow of the program directly. However, such use of continuations is far from the common case. As observed by Berdine *et al.* [5], many control-flow constructs use continuations linearly (exactly once). This linearity arises from the restrictions of the source language: functions return exactly once, merge-points of conditional statements are reachable in exactly one way from each branch, *etc.* The fact that `call-cc` and other nonstandard control-flow operators discard or duplicate continuations is part of what makes them difficult to reason about.

Combining linearity with an ordering[2] on continuations restricts their manipulation even further. In fact, ordered linear continuations essentially enforce a stack-discipline on control [28]. Introducing a linear continuation is analogous to pushing an activation record onto a stack; invoking a linear continuation corresponds to popping that activation record. Because many constructs (function call/return, nested blocks, and merge-points of conditionals) of high-level structured programs can be implemented via a stack of activation records, ordered linear continuations are a natural fit to describing their control flow behavior.

Using ordered linear continuations in a type system divorces the description of the stack-like control constructs of a programming language from its syntax (block structure). This separation is essential for preserving control-flow information across compilation steps such as CPS transformation, because the syntactic structure of the program is altered. The main insight is that we can push information implicitly found in the structure of a program into explicit descriptions (the types) of the program.

3. The Secure CPS Calculus

The target of our secure CPS translation is a call-by-value, imperative language similar to those found in the work on Typed Assembly Language [7, 22], although its type system is inspired by previous language-based security research [20, 24, 36]. This section describes the secure CPS language, its operational behavior, and its static semantics.

3.1. Syntax

The syntax for the secure CPS language is given in Figure 2.

Following standard practice in information-flow systems, we generalize high- and low-security labels into a lattice, $\mathcal{L}$, of possible security annotations. Elements of $\mathcal{L}$ 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 $\sqsubseteq$ symbol denotes the lattice ordering. The lattice join and meet operations are given by $\sqcup$ and $\sqcap$, respectively, and the least and greatest elements are written $\bot$ and $\top$.

[2] The ordering in the type system presented here is trivial because exactly one continuation is allowed in the context. It is possible to generalize these results to account for multiple continuations in the context. [40]

$$
\begin{array}{rrcl}
\text{Security Labels} & \ell, \mathsf{pc} & \in & \mathcal{L} \\
\text{Base Types} & \tau & ::= & \mathsf{int} \;\mid\; 1 \;\mid\; \sigma\ \mathsf{ref} \;\mid\; [\mathsf{pc}](\sigma, \kappa) \to 0 \\
\text{Security Types} & \sigma & ::= & \tau_\ell \\
\text{Linear Types} & \kappa & ::= & \sigma \to 0 \\
\\
\text{Base Values} & bv & ::= & n \;\mid\; \langle\rangle \;\mid\; L^\sigma \;\mid\; \lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\, e \\
\text{Values} & v & ::= & x \;\mid\; bv_\ell \\
\text{Linear Values} & lv & ::= & y \;\mid\; \lambda\langle\mathsf{pc}\rangle(x\!:\!\sigma).\, e \\
\text{Primitives} & prim & ::= & v \;\mid\; v \oplus v \;\mid\; \mathtt{deref}(v) \\
\\
\text{Expressions} & e & ::= & \;\;\mathtt{let}\ x = prim\ \mathtt{in}\ e \\
 & & & \mid \mathtt{let}\ x = \mathtt{ref}^\sigma_\ell\ v\ \mathtt{in}\ e \\
 & & & \mid \mathtt{set}\ v := v\ \mathtt{in}\ e \\
 & & & \mid \mathtt{letlin}\ y = lv\ \mathtt{in}\ e \\
 & & & \mid \mathtt{if0}\ v\ \mathtt{then}\ e\ \mathtt{else}\ e \\
 & & & \mid \mathtt{goto}\ v\ v\ lv \\
 & & & \mid \mathtt{lgoto}\ lv\ v
\end{array}
$$

Figure 2. Syntax for the secure CPS language

Types fall into two main 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 include integers, unit, and references. Continuation types, written $[\mathsf{pc}](\sigma, \kappa) \to 0$, indicate a security level and the types of their arguments. The notation 0 represents the "void" type, indicating that a continuation never returns a value.

Corresponding to these types, base values, bv, include integers, n, a unit value, $\langle\rangle$, 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 $\mathsf{label}(bv_\ell) = \ell$ for obtaining the label of a type or closed value, and extend the join (and meet) operation to security types: $\tau_\ell \sqcup \ell' = \tau_{(\ell \sqcup \ell')}$.

As an example, the value $3_\bot$ represents a low-security integer (one of type $\mathsf{int}_\bot$) that is observable by any computation. On the other hand, the value $4_\top$ represents a (very) high-security integer that should be observable only by computations with top-security clearance. The operational semantics will ensure that labels are propagated correctly. For instance we have $3_\bot + 4_\top = 7_\top$, because low-security computation should be prevented from observing the sum—it contains information about the high-security value $4_\top$.

References contain two security annotations. For example, the type $\mathsf{int}_\top\ \mathsf{ref}_\bot$ represents the type of low-security pointers to high-security integers, which is distinct from $\mathsf{int}_\bot\ \mathsf{ref}_\top$, the type of high-security pointers to low-security integers. The data returned by a dereference operation is protected by the join of the two labels. Thus, integers obtained through pointers of either of these two reference types will receive a security label of $\top$.

An ordinary continuation $\lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\,e$ is a piece of code (the expression e) that accepts a nonlinear argument of type σ and a linear argument of type κ. Continuations may recursively invoke themselves using the name f, which is bound in e. 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 observable only by principals able to read data with label pc.

A linear value, lv, is either a variable (ranged over by y), or a linear continuation, which contains a code expression e parameterized by a nonlinear argument just as for ordinary continuations. Linear continuations may not be recursive, 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 linear continuation values from nonlinear ones. As for ordinary continuations, the label pc restricts the continuation's effects, but unlike ordinary continuations, the pc is constrained by their *introduction contexts* (as opposed to their elimination contexts). Intuitively, linear continuations capture the security context in which they are created and, when invoked, *restore* the program counter label to the one captured.

The primitive operations include binary arithmetic, $\oplus$, dereference, and a means of copying secure values. Primitive operations are side-effect free. 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. Straight-line code sequences are terminated by conditional statements, nonlocal transfers of control via `goto` (for ordinary continuations) or `lgoto` (for linear continuations).

We use a special linear variable $\mathtt{halt}^\sigma$ of type $\sigma \to \mathsf{0}$ to represent the initial continuation. It corresponds to the bottom of the control stack. Thus, well-formed programs terminate with statements of the form $\mathtt{lgoto}\ \mathtt{halt}^\sigma\ v$, where v is the final result of the program.

3.2. Operational Semantics

The operational semantics (Figure 3) is given by a transition relation between machine configurations of the form $\langle M, \mathsf{pc}, e\rangle$. The notation $e\{v/x\}$ indicates capture-avoiding substitution of value v for variable x in expression e.

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, which is 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. We use $\emptyset$ to denote the empty memory, and we write $Loc(e)$ for the set of location names occurring in 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 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 [*E5*] and [*E6*] 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*] through [*P3*], computed values are stamped with the pc label. The notation $[\![\oplus]\!]$ denotes the semantic counterpart to the syntactic operation only $\oplus$. Checks like the one on [*E3*] prevent illegal information flows via direct means such as assignment. We shall show in Section 4 that, for well-typed programs, *all* illegal information flows are ruled out.

Operationally, the rules for `goto` and `lgoto` are very similar—each causes control to be transferred to the target continuation. They differ in their treatment of the program-counter label, as seen in rules [*E7*] and [*E8*]. Ordinary continuations stamp the pc label with program counter label and annotation of the target 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. In accordance with the label-stamping intuition, both `goto` and `lgoto` stamp the pc label of the calling context into the value passed to the continuation.

$$[P1] \qquad \langle M,\, \mathsf{pc},\, bv_\ell\rangle \Downarrow bv_{\ell\sqcup\mathsf{pc}}$$

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

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

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

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

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

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

$$[E5] \qquad \langle M,\, \mathsf{pc},\, \mathtt{if0}\ 0_\ell\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \longmapsto \langle M,\, \mathsf{pc} \sqcup \ell,\, e_1\rangle$$

$$[E6] \qquad \langle M,\, \mathsf{pc},\, \mathtt{if0}\ n_\ell\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle \longmapsto \langle M,\, \mathsf{pc} \sqcup \ell,\, e_2\rangle \qquad (n \neq 0)$$

$$[E7] \qquad \frac{v = (\lambda[\mathsf{pc}']f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell}{\begin{array}{l}\langle M,\, \mathsf{pc},\, \mathtt{goto}\ v\ bv_{\ell'}\ lv\rangle \\ \quad \longmapsto \langle M,\, \mathsf{pc} \sqcup \mathsf{pc}' \sqcup \ell,\, e\{v/f\}\{bv_{\ell'\sqcup\mathsf{pc}}/x\}\{lv/y\}\rangle\end{array}}$$

$$[E8] \qquad \langle M,\, \mathsf{pc},\, \mathtt{lgoto}\ (\lambda\langle\mathsf{pc}'\rangle(x\!:\!\sigma).\, e)\ bv_\ell\rangle \longmapsto \langle M,\, \mathsf{pc}',\, e\{bv_{\ell\sqcup\mathsf{pc}}/x\}\rangle$$

Figure 3. Expression evaluation

As mentioned above, well-formed programs contain a free linear continuation variable $\mathtt{halt}^\sigma$. Consequently the "stuck" term $\mathtt{lgoto}\ \mathtt{halt}^\sigma\ v$ represents a valid terminal state.

3.3. An Example Evaluation

This section gives a concrete example of the operational semantics.

Consider the evaluation shown in Figure 4. It shows the program fragment (D) from Figure 1 of the introduction using the syntax of our secure CPS language. In this instance, the condition variable is the high-security value $\mathtt{0}_\top$, and the program-counter label is initially $\bot$, the

$$
\begin{array}{lllll}
 & & \langle M, & \bot, & \texttt{letlin k = } k_{impl} \texttt{ in} \\
 & & & & \texttt{if0 } 0_\top \texttt{ then set a := } 1_\bot \texttt{ in lgoto k } \langle\rangle \\
 & & & & \quad\quad \texttt{else set a := } 2_\bot \texttt{ in lgoto k } \langle\rangle\,\rangle \\
(1) & \longmapsto & \langle M, & \bot, & \texttt{if0 } 0_\top \texttt{ then set a := } 1_\bot \texttt{ in lgoto } k\ \langle\rangle \\
 & & & & \quad\quad \texttt{else set a := } 2_\bot \texttt{ in lgoto } k\ \langle\rangle\,\rangle \\
(2) & \longmapsto & \langle M, & \top, & \texttt{set a := } 1_\bot \texttt{ in lgoto } k_{impl}\ \langle\rangle \quad \rangle \\
(3) & \longmapsto & \langle M', & \top, & \texttt{lgoto } k_{impl}\ \langle\rangle \quad \rangle \\
(4) & \longmapsto & \langle M', & \bot, & \texttt{set z := } 3_\bot \texttt{ in lgoto halt}^{1_\bot}\ \langle\rangle \quad \rangle \\
(5) & \longmapsto & \langle M'', & \bot, & \texttt{lgoto halt}^{1_\bot}\ \langle\rangle \quad \rangle
\end{array}
$$

Where

$$
\begin{array}{rcl}
M & = & \{\mathsf{a} \mapsto \mathsf{0}_\top, \mathsf{z} \mapsto \mathsf{0}_\bot\} \\
M' & = & \{\mathsf{a} \mapsto \mathsf{1}_\top, \mathsf{z} \mapsto \mathsf{0}_\bot\} \\
M'' & = & \{\mathsf{a} \mapsto \mathsf{1}_\top, \mathsf{z} \mapsto \mathsf{3}_\bot\} \\
\mathsf{a} & : & \mathsf{int}_\top\ \mathsf{ref}_\bot \\
\mathsf{z} & : & \mathsf{int}_\bot\ \mathsf{ref}_\bot \\
k_{impl} & = & \lambda\langle\bot\rangle(\texttt{x:}1_\bot)\texttt{. set z := } 3_\bot \texttt{ in lgoto halt}^{1_\bot}\ \langle\rangle
\end{array}
$$

Figure 4. Example program evaluation

lowest security label. The memory, M, initially maps the high-security location y to the value $\mathsf{0}_\top$ and the low-security location z to the value $\mathsf{0}_\bot$. (This information is summarized in the figure.)

Step (1) is a transition via [*E4*] that introduces the linear continuation, k_{impl} and binds it to the variable k. As indicated by the notation $\langle\bot\rangle$ in k_{impl}'s definition, when invoked, k_{impl} will set the pc label back to $\bot$. In step (2), the program transitions via rule [*E5*], testing the condition variable. In this case, because $\mathsf{0}_\top$ is high-security, the program counter label increases to $\top = \bot \sqcup \top$, and the program takes the first branch. Next, step (3) is a transition by rule [*E3*], which updates the contents of memory location a. The new value stored is high-security, because, instantiating [*E3*], we have: $\ell = \bot, \ell' = \bot, \mathsf{pc} = \top$ and $\ell' \sqcup \ell \sqcup \mathsf{pc} = \bot \sqcup \bot \sqcup \top = \top$. This assignment succeeds because a is a location that stores high-security data; if a were a location of type $\mathsf{int}_\bot\ \mathsf{ref}_\bot$, the check $\ell' \sqcup \ell \sqcup \mathsf{pc} \sqsubseteq \mathsf{label}(\mathsf{int}_\bot) = \bot$ would fail—however, the type system presented in the next section statically rules out such behavior, making such dynamic checks unnecessary.

The fourth step is the linear invocation, via rule [*E8*]. As promised, k_{impl} resets the program counter label to $\bot$, and in addition, we substitute the actual arguments for the formal parameters in the body of the continuation. The last transition is another instance of rule [*E3*], this time updating the contents of z with the low-security value $\mathsf{3}_\bot$.

How would this program differ if an ordinary continuation were used instead of k_{impl}? The crucial difference would appear in step (4), where instead of rule [*E8*], we would be forced to use rule [*E7*]. Note that [*E7*] requires the pc label of the continuation to be *higher* than the one in the machine configuration. In this case, because the calling context has $\mathsf{pc} = \top$, the body of the continuation would be forced to $\top$ as well. It is not possible to write a value to the low-security location z in such circumstances, and hence we cannot write this program using an ordinary continuation in place of k_{impl} without forcing z to be a high-security location.

3.4. 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 respected. Together, these restrictions rule out illegal information flows and impose enough structure on the language for us to prove a noninterference property.

As in other mixed linear–nonlinear type systems [35], the type context is split into an ordinary, nonlinear section and a linear section. Γ is a finite partial map from nonlinear variables to security types; it admits the usual weakening and exchange rules (which we omit). The linear part of the context consists of a single linear variable and its type, $y\!:\!\kappa$. This variable must be used exactly once and so cannot be discarded. The two parts of the context are separated by $\|$ in the judgments to make them more distinct (as in $\Gamma \parallel y\!:\!\kappa$). We use $\bullet$ to denote an empty nonlinear context.

Figures 5 through 9 show the rules for type-checking. 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 $\Gamma \parallel y\!:\!\kappa \vdash lv : \kappa$. Like values, primitive operations may not contain linear variables, but the security of the value produced depends on the program-counter. We thus use the judgment $\Gamma\ [\mathsf{pc}] \vdash prim : \sigma$ to say that in context Γ where the program-counter label is bounded above by pc, $prim$ computes a value of type σ. Similarly, $\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e$ means that expression e is type-safe and contains no illegal information flows in the type context $\Gamma \parallel y\!:\!\kappa$, when the program-counter label is at most pc.[3] In the latter two forms, pc is a conservative approximation to the security label of information affecting the program counter.

[3] Because expressions represent continuations, and hence to not return values, no type is associated with judgments $\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e$. Alternatively, we could write $\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e : \mathsf{0}$ to indicate that e does not return. But, as all expressions have type $\mathsf{0}$, we simply omit the $: \mathsf{0}$.

$$[TV1]\qquad \frac{}{\Gamma \vdash n_\ell : \mathsf{int}_\ell}$$

$$[TV2]\qquad \frac{}{\Gamma \vdash \langle\rangle_\ell : 1_\ell}$$

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

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

$$[TV5]\qquad \frac{\begin{array}{c} f, x \notin \mathrm{dom}(\Gamma) \\ \sigma' = ([\mathsf{pc}](\sigma, \kappa) \to 0)_\ell \\ \Gamma, f\!:\!\sigma', x\!:\!\sigma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e \end{array}}{\Gamma \vdash (\lambda[\mathsf{pc}]f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell : \sigma'}$$

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

Figure 5. Value typing

The rules for checking ordinary values, $[TV1]$–$[TV6]$ shown in Figure 5, are, for the most part, standard. A value cannot contain free linear variables because discarding (or copying) the value would break the linearity constraint on the variable. 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 (shown in Figure 6). Continuations exhibit the expected contravariance (rule $[S2]$). References, are, as usual, invariant with respect to the data being stored, but the security labels of the references themselves obey the usual covariant subtyping. Consequently, $\sigma\ \mathsf{ref}_\bot \leq \sigma\ \mathsf{ref}_\top$ for any σ, but it is never the case that $\sigma\ \mathsf{ref}_\ell \leq \sigma'\ \mathsf{ref}_{\ell'}$ when $\sigma \neq \sigma'$.

Linear values are checked using rules $[TL1]$ and $[TL2]$. They may safely mention free linear variables, but the variables must not be discarded. Thus we may conclude that a linear variable is well-formed exactly when it is the linear context (rule $[TL1]$). In a linear continuation (rule $[TL3]$), the linear context, y, is the top of the stack of continuations yet to be invoked. Intuitively, this judgment says that the continuation body e must exit via a `lgoto` to y. The code e may declare

$$[S1] \qquad \frac{}{\vdash \tau \leq \tau}$$

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

$$[S3] \qquad \frac{\vdash \tau \leq \tau' \quad \ell \sqsubseteq \ell'}{\vdash \tau_\ell \leq \tau'_{\ell'}}$$

$$[S4] \qquad \frac{\vdash \sigma \leq \sigma' \quad \vdash \sigma' \leq \sigma''}{\vdash \sigma \leq \sigma''}$$

Figure 6. Value subtyping

$$[TL1] \qquad \frac{}{\Gamma \parallel y\!:\!\kappa \vdash y : \kappa}$$

$$[TL2] \qquad \frac{\begin{array}{c} x \notin \mathrm{dom}(\Gamma) \\ \Gamma, x\!:\!\sigma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e \end{array}}{\Gamma \parallel y\!:\!\kappa \vdash \lambda\langle \mathsf{pc}\rangle(x\!:\!\sigma).\, e : \sigma \to \mathsf{0}}$$

Figure 7. Linear value typing

its own internal linear continuations, but they must be consumed before y can be used. For simplicity, we disallow subtyping on linear types.[4]

The rules for primitive operations (in Figure 8) 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.

Figure 9 lists the rules for typechecking expressions. Primitive operations are introduced by a `let` expression as shown in [*TE1*]. The rules for creating new references and doing reference update, rules [*TE2*] and [*TE3*], require that the reference protect the security of the program counter. The condition $\mathsf{pc} \sqsubseteq \ell \sqcup \mathsf{label}(\sigma)$ says that any data read through the reference may only be observed by contexts able to observe the current program counter. The condition $\mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)$ in [*TE3*] prevents explicit flows in a similar way.

[4] It is possible to formulate a sound type system that admits subtyping for linear types, and indeed we see hints of subtyping in rule [*TE5*].

$$[TP1] \qquad \frac{\Gamma \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)}{\Gamma\ [\mathsf{pc}] \vdash v : \sigma}$$

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

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

Figure 8. Primitive operation typing

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`, [*TE6*], 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 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 rules for `letlin`, [*TE5*], and `lgoto`, [*TE7*], manipulate the linear context to enforce the ordering property on continuations. For `letlin`, the linear variable y must be used in the body of the continuation being declared. The body of the declaration, e, is checked under the assumption that the new continuation, y', is available. Collectively, these manipulations amount to pushing the continuation y' onto the control stack. The rule for `lgoto` simply requires that the linear continuation being invoked consumes the linear continuation declared in the context, corresponding to popping the control stack.

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 program-counter label of the target continuation, 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 (except variables) via `letlin`. This assumption rules out trivially insecure programs; during execution this

$$[TE1] \quad \frac{\begin{array}{c}\Gamma\ [\mathsf{pc}] \vdash \mathit{prim} : \sigma \\ \Gamma, x\!:\!\sigma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathit{prim}\ \mathtt{in}\ e}$$

$$[TE2] \quad \frac{\begin{array}{c}\Gamma \vdash v : \sigma \quad \mathsf{pc} \sqsubseteq \ell \sqcup \mathsf{label}(\sigma) \\ \Gamma, x\!:\!\sigma\ \mathsf{ref}_\ell \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{let}\ x = \mathtt{ref}^{\sigma}_{\ell}\ v\ \mathtt{in}\ e}$$

$$[TE3] \quad \frac{\begin{array}{c}\Gamma \vdash v : \sigma\ \mathsf{ref}_\ell \quad \Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e \\ \Gamma \vdash v' : \sigma \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(\sigma)\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{set}\ v := v'\ \mathtt{in}\ e}$$

$$[TE4] \quad \frac{\Gamma \vdash v : \mathsf{int}_\ell \quad \Gamma \parallel y\!:\!\kappa\ [\mathsf{pc} \sqcup \ell] \vdash e_i}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{if0}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2}\ (i \in \{1,2\})$$

$$[TE5] \quad \frac{\begin{array}{c}\Gamma \parallel y\!:\!\kappa \vdash \lambda\langle \mathsf{pc}' \rangle (x'\!:\!\sigma').\, e' : \sigma' \to 0 \\ \mathsf{pc} \sqsubseteq \mathsf{pc}' \qquad \Gamma \parallel y'\!:\!\sigma' \to 0\ [\mathsf{pc}] \vdash e\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{letlin}\ y' = \lambda\langle \mathsf{pc}' \rangle (x'\!:\!\sigma').\, e'\ \mathtt{in}\ e}$$

$$[TE6] \quad \frac{\begin{array}{c}\Gamma \vdash v : ([\mathsf{pc}'](\sigma', \kappa') \to 0)_\ell \\ \Gamma \vdash v' : \sigma' \qquad \Gamma \parallel y\!:\!\kappa \vdash lv : \kappa' \\ \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \quad \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma')\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{goto}\ v\ v'\ lv}$$

$$[TE7] \quad \frac{\begin{array}{c}\Gamma \parallel y\!:\!\kappa \vdash lv : \sigma \to 0 \\ \Gamma \vdash v : \sigma \\ \mathsf{pc} \sqsubseteq \mathsf{label}(\sigma)\end{array}}{\Gamma \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash \mathtt{lgoto}\ lv\ v}$$

Figure 9. Expression typing

constraint is not required, and programs in the image of the CPS translation of Section 5 satisfy this property.

This type system is sound with respect to the operational semantics. The proof is, for the most part, standard, following in the style of Wright and Felleisen [39]. We simply state the lemmas necessary for the discussion of the noninterference result of the next section.

LEMMA 3.1 (Subject Reduction). *If* $\bullet \parallel y : \kappa\ [\mathsf{pc}] \vdash e$ *and* M *is a well-formed memory such that* $Loc(e) \subseteq \mathrm{dom}(M)$ *and* $\langle M, \mathsf{pc}, e\rangle \longmapsto \langle M', \mathsf{pc}', e'\rangle$, *then* $\bullet \parallel y\!:\!\kappa\ [\mathsf{pc}'] \vdash e'$ *and* M' *is a well-formed memory such that* $Loc(e') \subseteq \mathrm{dom}(M')$.

LEMMA 3.2 (Progress).
If $\bullet \parallel y\!:\!\kappa\ [\mathsf{pc}] \vdash e$ *and* M *is well-formed and* $Loc(e) \subseteq \mathrm{dom}(M)$, *then either* e *is of the form* `lgoto` $y\ v$ *or there exist* M', pc', *and* e' *such that* $\langle M, \mathsf{pc}, e\rangle \longmapsto \langle M', \mathsf{pc}', e'\rangle$.

Note that these lemmas are proved for terms containing free occurrences of the linear variable. The Progress lemma treats the free linear variable as the *initial continuation* that terminates the program when invoked.

4. Noninterference

This section proves a noninterference result for the secure CPS language, generalizing a previous result from Smith and Volpano [34]. The approach is to use a preservation-style argument that shows a particular invariant related to low-security views of a well-typed program is maintained by each computation step.

Informally, the noninterference result says that low-security computations are not able to observe high-security data. Here, the term "low-security" is relative to an arbitrary point, ζ, in the security lattice $\mathcal{L}$. Thus, ℓ is a low-security label whenever $\ell \sqsubseteq \zeta$. Similarly, "high-security" refers to those labels $\not\sqsubseteq \zeta$. The security level of a computation is indicated by the label of the program counter under which the computation is taking place. Thus, by "low-security computation", we mean a transition step in the operational semantics whose starting configuration (the one before the step) contains a $\mathsf{pc} \sqsubseteq \zeta$.

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 likewise unaffected by high-security values. This characterization reduces noninterference to the problem of showing that a given program e_1 is equivalent (from a low-security observer's point of view) to any program e_2 that differs from e_1 only in its high-security parts.

Key to the argument is a formal definition of "low-equivalence," by which we intend to capture the property that two programs' executions are indistinguishable by an observer only able to see the low-security portions of memory and machine state.

How do we show that configurations $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle$ and $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$ behave identically from the low-security point of view? Clearly, the memories M_1 and M_2 must agree on the values contained in low-security locations. In addition, if $\mathsf{pc}_1, \mathsf{pc}_2 \sqsubseteq \zeta$, meaning that e_1 and e_2 may perform actions (such as modifying a low-security memory location) visible to low observers, the programs necessarily must perform the same computation on low-security values. On the other hand, when $\mathsf{pc} \not\sqsubseteq \zeta$, the actions of e_1 and e_2 should be invisible to the low view.

This intuition guides the formal definition of low-equivalence, which we write $\approx_\zeta$. The definition builds on standard alpha-equivalence (written $\equiv_\alpha$) as a base notion of equality. We use substitutions to factor out the relevant high-security values and those linear continuations that reset the program-counter label to be $\sqsubseteq \zeta$.

DEFINITION 4.1 (Substitutions).

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

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

To show ζ-equivalence between e_1 and e_2, we should find substitutions γ_1 and γ_2 containing the relevant high-security data such that $e_1 \equiv_\alpha \gamma_1(e)$ and $e_2 \equiv_\alpha \gamma_2(e)$—both e_1 and e_2 look the same as e after factoring out the high-security data.

The other important piece of the proof is that we can track the linear continuations that restore the program counter to a label that is $\not\sqsubseteq \zeta$. Here is where the stack ordering on linear continuations comes into play: The operational semantics guarantees that the program-counter label is monotonically increasing *except* when a linear continuation is invoked. If e_1 invokes a linear continuation that causes pc_1 to fall below ζ, e_2 must follow suit and call an equivalent continuation; otherwise the low-security observer may distinguish e_1 from e_2. The stack ordering on linear continuations is exactly the property that forces e_2 to invoke the same low-security continuation as e_1.

Note that only the low-security linear continuations are relevant to the ζ-equivalence of two programs—the high-security linear continuations in the programs may differ. Furthermore, our plan is to establish an invariant with respect to the operational semantics. This means we must be able to keep track of the relevant low-security continuations as they are introduced and consumed by `letlin` and `lgoto`.

There is a slight technical difficulty in doing so in the substitution-style operational semantics we have presented: We want to maintain the invariant that ζ-equivalent programs always have equivalent pending low-security continuations. Statically, the linear variable in the context names these continuations, but dynamically, these variables are substituted away, and so there is no way to name the "next" low-security linear continuation.

To get around this problem, our approach is to introduce auxiliary substitutions that map *stacks* of linear variables to low-security linear continuations. The top of stack corresponds to the next low-security linear continuation that will be invoked.

An alternative would be to use an operational semantics that manipulates the stack of linear continuations directly and then define a notion of ζ-equivalence on these more structured program configurations. Yet another alternative, taken in the conference version of this paper [40], is to allow multiple linear continuations variables in the linear context, which has a natural correspondence with non-commutative linear logic. All three approaches require essentially the same amount of bookkeeping needed for the noninterference proof. The difference is *where* the bookkeeping gets done: The approach in this paper simplifies the type system and operational semantics at the expense of complicating the noninterference proof. The other two options push the additional complexity into either the operational or static semantics, respectively.

DEFINITION 4.2 (Linear Continuation Stack). *Let* K *be an ordered list (a stack) of linear type variables* $y_1\!:\!\kappa_1, \ldots, y_n\!:\!\kappa_n$ *such that* $n \geq 1$. *We write* $\Gamma \vdash k \models \mathrm{K}$ *to indicate that* k *is a substitution that maps each* y_i *to a linear value such that* $\Gamma \parallel \mathtt{halt}^{\sigma}\!:\!\sigma \to \mathsf{0} \vdash k(y_1) : \kappa_1$ *and* $\Gamma \parallel y_{i-1}\!:\!\kappa_{i-1} \vdash k(y_i) : \kappa_i$ *for* $i \in \{2 \ldots n\}$. *We write top*(K) *to indicate the top of the stack, namely* $y_n\!:\!\kappa_n$.

Application of a stack substitution k to a term e is defined as:

$$k(e) = e\{k(y_n)/y_n\}\{k(y_{n-1})/y_{n-1}\} \ldots \{k(y_1)/y_1\}.$$

Note that the order of the substitutions is important because the continuation $k(y_n)$ may refer to the linear variable y_{n-1}.

Two linear continuation stacks k_1 and k_2 are equivalent if they have the same domain and map each variable to equivalent continuations. We must also ensure that the stack contains *all* of the pending low-security continuations.

DEFINITION 4.3 (`letlin` Invariant). *A term satisfies the* `letlin` *invariant if every linear continuation expression* $\lambda\langle \mathsf{pc}\rangle(x\!:\!\sigma).\, e$ *appearing* *in the term is either in the binding position of a* `letlin` *or satisfies* $\mathsf{pc} \not\sqsubseteq \zeta$.

The idea behind the `letlin` invariant is that when $k(e)$ is a closed term such that e satisfies the `letlin` invariant, any invocation of a low-security linear continuation in e must arise from the substitution k—in other words, k contains any pending low-security linear continuations.

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

DEFINITION 4.4 (ζ-Equivalence).

$\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$	*If* $\gamma_1, \gamma_2 \models \Gamma$ *and for every* $x \in \mathrm{dom}(\Gamma)$ *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* $y \in \mathrm{dom}(\mathrm{K})$ *it is the case that* $k_i(y) \equiv_\alpha \lambda\langle\mathsf{pc}\rangle(x\!:\!\sigma).\, e$ *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* $L^\sigma \in \mathrm{dom}(M_1) \cup \mathrm{dom}(M_2)$, $\mathsf{label}(\sigma) \sqsubseteq \zeta$ *implies that* $L^\sigma \in \mathrm{dom}(M_1) \cap \mathrm{dom}(M_2)$ *and* $M_1(L^\sigma) \approx_\zeta M_2(L^\sigma) : \sigma$.

Finally, we can put all of these requirements together to define the ζ-equivalence of two machine configurations, which also gives us the invariant for the noninterference proof.

DEFINITION 4.5 (Noninterference Invariant).

The noninterference invariant is a predicate on machine configurations, written $\Gamma \parallel \mathrm{K} \vdash \langle M_1, \mathsf{pc}_1, e_1\rangle \approx_\zeta \langle M_2, \mathsf{pc}_2, e_2\rangle$, *that holds if there exist substitutions* $\gamma_1, \gamma_2, k_1, k_2$ *and terms* e_1' *and* e_2' *such that the following conditions are all met:*

(i) $e_1 = \gamma_1(k_1(e_1'))$ *and* $e_2 = \gamma_2(k_2(e_2'))$.

(ii) $\Gamma \parallel \mathit{top}(\mathrm{K})\ [\mathsf{pc}_1] \vdash e_1'$ *and* $\Gamma \parallel \mathit{top}(\mathrm{K})\ [\mathsf{pc}_2] \vdash e_2'$

(iii) *Either* (a) $\mathsf{pc}_1 = \mathsf{pc}_2 \sqsubseteq \zeta$ and $e_1' \equiv_\alpha e_2'$ or
(b) $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and $\mathsf{pc}_2 \not\sqsubseteq \zeta$.

(iv) $\Gamma \vdash \gamma_1 \approx_\zeta \gamma_2$ *and* $\Gamma \parallel \mathrm{K} \vdash k_1 \approx_\zeta k_2$

(v) $\mathit{Loc}(e_1) \subseteq \mathrm{dom}(M_1)$ *and* $\mathit{Loc}(e_2) \subseteq \mathrm{dom}(M_2)$ *and* $M_1 \approx_\zeta M_2$.

(vi) *Both* e_1' *and* e_2' *satisfy the* `letlin` *invariant.*

The main technical work of the noninterference proof is a preservation argument showing that the Noninterference 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. The proof of this lemma is given in Appendix A.

LEMMA 4.1 (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* $\mathsf{pc}_2 \sqsubseteq \zeta$

– $\langle M_1, \mathsf{pc}_1, e_1 \rangle \longmapsto \langle M_1', \mathsf{pc}_1', e_1' \rangle$

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* $\Gamma' \parallel \mathrm{K}' \vdash \langle M_1', \mathsf{pc}_1', e_1' \rangle \approx_\zeta \langle M_2', \mathsf{pc}_2', e_2' \rangle$.

We use the stack ordering property of linear continuations, as made explicit in the Progress Lemma, to prove that equivalent high-security configurations eventually return to equivalent low-security configurations. The proof is given in Appendix A.

LEMMA 4.2 (High-pc Step). *Suppose*

– $\Gamma \parallel \mathrm{K} \vdash \langle M_1, \mathsf{pc}_1, e_1 \rangle \approx_\zeta \langle M_2, \mathsf{pc}_2, e_2 \rangle$

– $\mathsf{pc}_1 \not\sqsubseteq \zeta$ *and* $\mathsf{pc}_2 \not\sqsubseteq \zeta$

– $\langle M_1, \mathsf{pc}_1, e_1 \rangle \longmapsto \langle M_1', \mathsf{pc}_1', e_1' \rangle$

then 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* $\Gamma' \parallel \mathrm{K}' \vdash \langle M_1', \mathsf{pc}_1', e_1' \rangle \approx_\zeta \langle M_2', \mathsf{pc}_2', e_2 \rangle$.

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

THEOREM 4.1 (Noninterference). *Suppose*

- $x\!:\!\sigma \parallel \mathtt{halt}^{\mathsf{int}_\zeta}\!:\!\mathsf{int}_\zeta \to \mathsf{0}\ [\bot] \vdash e$ *for some initial program* e.
- $\mathsf{label}(\sigma) \not\sqsubseteq \zeta$
- $\bullet \vdash v_1, v_2 : \sigma$

Then

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

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

Proof. Let e_1 be the term $e\{v_1/x\}$ and let e_2 be the term $e\{v_2/x\}$. It is easy to verify that

$$x\!:\!\sigma \parallel \mathtt{halt}^{\mathsf{int}_\zeta}\!:\!\mathsf{int}_\zeta \to \mathsf{0} \vdash \langle \emptyset,\, \bot,\, e_1\rangle \approx_\zeta \langle \emptyset,\, \bot,\, e_2\rangle$$

by letting $\gamma_1 = \{x \mapsto v_1\}$, $\gamma_2 = \{x \mapsto v_2\}$, and $k_1 = k_2 = \{\mathtt{halt}^{\mathsf{int}_\zeta} \mapsto \mathtt{halt}^{\mathsf{int}_\zeta}$. 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

$$\Gamma \parallel \mathrm{K} \vdash \langle M_1,\, \zeta,\, \mathtt{halt}^{\mathsf{int}_\zeta}\ n_{\ell_1}\rangle \approx_\zeta \langle M_2,\, \zeta,\, \mathtt{halt}^{\mathsf{int}_\zeta}\ m_{\ell_2}\rangle$$

Clause (v) of the Noninterference 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 (iv), that neither n_{ℓ_1} nor m_{ℓ_2} are in the range of γ_i'. Thus, the integers present in the $\mathtt{halt}$ expressions do not arise from substitution. Because $\zeta \sqsubseteq \zeta$, clause (iii) 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. The grammar for this source language is given in Figure 5.

The source type system is adapted from the SLam calculus [20] to follow our "label stamping" operational semantics. Unlike the SLam calculus, which also performs access control checks, the source language type system is concerned only with secure information flow. 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. We omit a full account of this type system, and instead

$$
\begin{array}{rcll}
t & ::= & \mathsf{int} \mid s\ \mathsf{ref} \mid s \xrightarrow{\ell} s & \\
s & ::= & t_\ell & \\
bv & ::= & n \mid \mu f(x\!:\!s).e & \text{Integers and Recursive Functions} \\
v & ::= & x \mid (bv)_\ell & \text{Secure Values} \\
e & ::= & v \mid (e\ e) & \text{Values and Function Application} \\
 & & \mid (\mathtt{ref}\ e) & \text{Reference Creation} \\
 & & \mid\ !\,e & \text{Dereference} \\
 & & \mid (e \mathrel{\mathtt{:=}} e) & \text{Assignment} \\
 & & \mid \mathtt{if0}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & \text{Conditional}
\end{array}
$$

Figure 10. Source language grammar

focus on the interesting examples in the CPS translation. The left-hand side of Figure 11 contains the typing judgments for some of the more interesting source expressions.

Source types are like 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 [19], is given in terms of three mutually recursive functions: $(-)^*$, for base types, $(-)^+$ for security types, and $(-)^-$ to linear continuation types:

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

Figure 11 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 [11]. To obtain the full translation of a closed term e of type s, we pass in the initial continuation variable instantiated at the correct type:

$$[\![\emptyset \vdash_\ell e : s]\!]\mathtt{halt}^{s^+}$$

As expected, linear continuations are introduced by the translation at points that correspond (via the structure of the source program) to pushing an activation record on to the stack, and `lgoto`s are introduced where pops occur. The linear variable y represents the current "top of stack" continuation; invoking it will cause the activation stack to be popped, after executing the body of the continuation y. Note that

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

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

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

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

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

Figure 11. CPS translation (Here $s = (s_1 \xrightarrow{\mathsf{pc}'} s_2)_\ell$, $s' = s \sqcup \mathsf{pc}$, and the k_i's and y_i's are fresh.)

all of the implicit control flow of the source language is expressed by ordered linear continuations; ordinary continuations are used only to express source-level functions, which, because they may be copied or never invoked, are inherently nonlinear. However, the unique return continuation of a function is represented by a linear continuation.

The basic lemma for establishing type 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 5.1 (Type Translation). *If* $\Gamma \vdash_{\mathsf{pc}} e : s$ *then* $\Gamma^+ \parallel y{:}s^- \; [\mathsf{pc}] \vdash [\![\Gamma \vdash_{\mathsf{pc}} e : s]\!]y$.

While proving that this CPS translation is operationally correct is beyond the scope of this paper, the translation is substantially identical to other translations that have been shown to be correct. We expect that the simulation techniques of Danvy and Filinski [11], for example, could be adapted to prove similar correctness results for this transformation.

6. Related Work

The constraints imposed by linearity can be seen as a form of resource management [17], in this case limiting the set of possible future computations. Linearity has been more widely used in the context of memory consumption [2, 7, 37, 38]. Linear continuations have been studied in terms of their category-theoretic semantics [15] and also as a computational interpretation of classical logic [6]. Polakow and Pfenning have investigated the connections between ordered linear-logic, stack-based abstract machines, and CPS [28]. Berdine, *et al.*, have studied a number of situations in which continuations are used linearly [5].

Linearity also plays a role in security types for process calculi such as the π-calculus [21]. Because the usual translation of the λ-calculus into the π-calculus can be seen as a form of CPS 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 [14, 27]. Sabry and Felleisen observed that increased precision in some CPS data flow analyses is due to duplication of analysis along different execution paths [33]. They also note that some analyses "confuse continuations" when applied to CPS programs. Our type system distinguishes linear from nonlinear 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 [8], 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 [23]) 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.

Our system applies the technology of linear continuations to achieve new results in the secure information flow domain. The ultimate goal is to produce compilers that, in addition to transforming code, also transform the accompanying security policies so that low-level code can be verified. Beyond CPS conversion, compilers also use closure conversion, hoisting, and various optimizations such as inlining or constant propagation. These transformations may also affect information flows, yet how to build type systems rich enough to express security policies at such low levels is still an open question.

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, providing a target language suitable for compilation of high-level secure programs. Linear continuations provide the additional structure that is sufficient to prove the strong security property of noninterference.

Acknowledgements

We would like to thank James Cheney, Dan Grossman, Greg Morrisett, 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. Likewise, we much appreciated the reviewers' comments, which led to a cleaner presentation of the type system.

References

1. Abadi, M., A. Banerjee, N. Heintze, and J. Riecke: 1999, 'A Core Calculus of Dependency'. In: *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 147–160.
2. Abramsky, S.: 1993, 'Computational Interpretations of Linear Logic'. *Theoretical Computer Science* **111**, 3–57.
3. Agat, J.: 2000, 'Transforming Out Timing Leaks'. In: *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 40–53.
4. Appel, A.: 1992, *Compiling with Continuations*. Cambridge University Press.
5. Berdine, J., P. W. O'Hearn, U. S. Reddy, and H. Thielecke: 2001, 'Linearly Used Continuations'. In: *Proceedings of the Continuations Workshop*.
6. Bierman, G.: 1999, 'A Classical Linear Lambda Calculus'. *Theoretical Computer Science* **227**(1–2), 43–78.
7. Crary, K., D. Walker, and G. Morrisett: 1999, 'Typed Memory Management in a Calculus of Capabilities'. In: *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 262–275.

8. Damian, D. and O. Danvy: 2000, 'Syntactic Accidents in Program Analysis: On the Impact of the CPS Transformation'. In: *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. pp. 209–220.
9. Danvy, O.: 2000, 'Formalizing Implementation Strategies for First-Class Continuations'. In: *Proc. of the 9th European Symposium on Programming*, Vol. 1792 of *Lecture Notes in Computer Science*. pp. 88–103.
10. Danvy, O., B. Dzafic, and F. Pfenning: 1999, 'On Proving Syntactic Properties of CPS Programs'. In: A. Gordon and A. Pitts (eds.): *Proceedings of the Third International Workshop on Higher Order Operational Techniques in Semantics*, Vol. 20 of *Electronic Notes in Theoretical Computer Science*. pp. 19–31.
11. Danvy, O. and A. Filinski: 1992, 'Representing Control: A Study of the CPS Transformation'. *Mathematical Structures in Computer Science* **2**, 361–391.
12. Denning, D. E.: 1976, 'A Lattice Model of Secure Information Flow'. *Comm. of the ACM* **19**(5), 236–243.
13. Denning, D. E. and P. J. Denning: 1977, 'Certification of Programs for Secure Information Flow'. *Comm. of the ACM* **20**(7), 504–513.
14. Filho, J. M. and G. Burn: 1993, 'Continuation Passing Transformations and Abstract Interpretation'. In: *Proc. First Imperial College, Department of Computing, Workshop on Theory and Formal Methods*.
15. Filinski, A.: 1992, 'Linear Continuations'. In: *Proc. 19th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 27–38.
16. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen: 1993, 'The Essence of Compiling With Continuations'. In: *Proc. of the '93 SIGPLAN Conference on Programming Language Design*. pp. 237–247.
17. Girard, J.-Y.: 1987, 'Linear Logic'. *Theoretical Computer Science* **50**, 1–102.
18. Goguen, J. A. and J. Meseguer: 1982, 'Security Policies and Security Models'. In: *Proc. IEEE Symposium on Security and Privacy*. pp. 11–20.
19. Harper, R. and M. Lillibridge: 1993, 'Explicit Polymorphism and CPS Conversion'. In: *Proc. 20th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 206–219.
20. Heintze, N. and J. G. Riecke: 1998, 'The SLam Calculus: Programming with Secrecy and Integrity'. In: *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 365–377.
21. Honda, K., V. Vasconcelos, and N. Yoshida: 2000, 'Secure Information Flow as Typed Process Behaviour'. In: *Proc. of the 9th European Symposium on Programming*, Vol. 1782 of *Lecture Notes in Computer Science*. pp. 180–199.
22. Morrisett, G., D. Walker, K. Crary, and N. Glew: 1999, 'From System F to Typed Assembly Language'. *ACM Transactions on Programming Languages and Systems* **21**(3), 528–569.
23. Muchnick, S. S.: 1997, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers.
24. Myers, A. C.: 1999, 'JFlow: Practical Mostly-Static Information Flow Control'. In: *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 228–241.
25. Myers, A. C. and B. Liskov: 1997, 'A Decentralized Model for Information Flow Control'. In: *Proc. 17th ACM Symp. on Operating System Principles (SOSP)*. Saint-Malo, France, pp. 129–142.
26. Necula, G. C.: 1997, 'Proof-Carrying Code'. In: *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 106–119.
27. Nielson, F.: 1982, 'A Denotational Framework for Data Flow Analysis'. *Acta Informatica* **18**, 265–287.

28. Polakow, J. and F. Pfenning: 2000, 'Properties of Terms in Continuation-Passing Style in an Ordered Logical Framework'. In: J. Despeyroux (ed.): *2nd Workshop on Logical Frameworks and Meta-languages*. Santa Barbara, California.
29. Pottier, F. and S. Conchon: 2000, 'Information Flow Inference for Free'. In: *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. pp. 46–57.
30. Sabelfeld, A. and D. Sands: 2000, 'Probabilistic noninterference for multi-threaded programs'. In: *Proc. of the 13th IEEE Computer Security Foundations Workshop*. pp. 200–214.
31. Sabelfeld, A. and D. Sands: 2001, 'A PER model of secure information flow in sequential programs'. *Higher-Order and Symbolic Computation* **14**(1), 59–91.
32. Sabry, A. and M. Felleisen: 1993, 'Reasoning About Programs in Continuation-Passing Style'. *Lisp and Symbolic Computation* **6**(3/4), 289–360.
33. Sabry, A. and M. Felleisen: 1994, 'Is Continuation-Passing Useful for Data Flow Analysis?'. In: *Proc. SIGPLAN '94 Conference on Programming Language Design and Implementation*. pp. 1–12.
34. Smith, G. and D. Volpano: 1998, 'Secure Information Flow in a Multi-Threaded Imperative Language'. In: *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*. pp. 355–364.
35. Turner, D. N. and P. Wadler: 1999, 'Operational Interpretations of Linear Logic'. *Theoretical Computer Science* **227**(1-2), 231–248.
36. Volpano, D., G. Smith, and C. Irvine: 1996, 'A Sound Type System for Secure Flow Analysis'. *Journal of Computer Security* **4**(3), 167–187.
37. Wadler, P.: 1990, 'Linear types can change the world!'. In: M. Broy and C. Jones (eds.): *Programming Concepts and Methods*.
38. Wadler, P.: 1993, 'A Taste of Linear Logic'. In: *Mathematical Foundations of Computer Science*, Vol. 711 of *Lecture Notes in Computer Science*. pp. 185–210.
39. Wright, A. K. and M. Felleisen: 1994, 'A Syntactic Approach to Type Soundness'. *Information and Computation* **115**(1), 38–94.
40. Zdancewic, S. and A. C. Myers: 2001, 'Secure Information Flow and CPS'. In: *Proc. of the 10th European Symposium on Programming*, Vol. 2028 of *Lecture Notes in Computer Science*. pp. 46–61.

Appendix

A. Proofs

This appendix proves the Low- and High-pc Step Lemmas from Section 4. We make use of, but do not show, standard lemmas such as Substitution.

LEMMA A.1 (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* $\mathsf{pc}_2 \sqsubseteq \zeta$
- $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \longmapsto \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$

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* KK' *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. Let $e_1 = \gamma_1(k_1(e_1''))$ and $e_2 = \gamma_2(k_2(e_2''))$ where the substitutions are as described by the conditions of the Noninterference Invariant. Because $\mathsf{pc}_i \sqsubseteq \zeta$, clause (iii) implies that e_1'' and e_2'' must be α-equivalent expressions and $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$. Hence the only difference in their behavior arises due to the substitutions or the different memories. We proceed by cases on the transition step taken by the first program. The main technique is to reason by cases on the security level of the value used in the step—if it's low-security, by α-equivalence, both programs compute the same values, otherwise we extend the substitutions γ_1 and γ_2 to contain the high-security data. We show a few representative cases in detail to give the flavor of the argument, the remainder follow in a similar fashion.

[*E1*]

$$\frac{\langle M, \mathsf{pc}, \mathit{prim}\rangle \Downarrow v}{\langle M, \mathsf{pc}, \texttt{let } x = \mathit{prim} \texttt{ in } e\rangle \longmapsto \langle M, \mathsf{pc}, e\{v/x\}\rangle}$$

In this case, e_1'' and e_2'' must be of the form $\texttt{let } x = \mathit{prim} \texttt{ in }$ e, consequently e_2 must also transition via rule [*E1*]. Because $M_1 = M_1'$ and $M_2 = M_2'$, and the locations found in terms e_1' and e_2' are found in e_1 and e_2 respectively, condition (v) of the Noninterference Invariant holds after the transition.

It suffices to find an e' and γ_i' such that $e_1' = \gamma_1'(k_1(e'))$ and $e_2' = \gamma_2'(k_2(e'))$. If prim is a value, then take $\gamma_i' = \gamma_i$ and let $e' = e\{\mathit{prim}/x\}$. These choices satisfy the conditions. Otherwise, prim is not a value. Consider the evaluation $\langle M_1, \mathsf{pc}, \gamma_1(\mathit{prim})\rangle \Downarrow bv_\ell$. There are two cases.

If $\ell \sqsubseteq \zeta$ then prim cannot contain any free variables, for otherwise condition (iv) would be violated—evaluation rules [*P2*] and [*P3*] imply that the label of the resulting value be higher than the label of any constituent, and all the values of γ_1 have label higher than ζ. Thus, $\gamma_1(\mathit{prim}) = \mathit{prim} = \gamma_2(\mathit{prim})$. $\langle M_2, \mathsf{pc}, \gamma_2(\mathit{prim})\rangle \Downarrow bv'_{\ell'}$ and because $M_1 \approx_\zeta M_2$ we have $bv_\ell \approx_\zeta bv'_{\ell'} : \sigma$. Thus, there exist Γ'', γ_1'', γ_2'' and values $v_1 \equiv_\alpha v_2$ such that $\Gamma'' \vdash \gamma_1'' \approx_\zeta \gamma_2''$ and $bv_\ell = \gamma_1''(v_1)$ and $bv'_{\ell'} = \gamma_2''(v_2)$. Thus, we take $\gamma_1' = \gamma_1 \cup \gamma_1''$, $\gamma_2' = \gamma_2 \cup \gamma_2''$ and $e_i'' = e\{v_i/x\}$. Conditions (iv), (v), and (vi) hold trivially; conditions (i), (ii), and (iii) are easily verified based on the operational semantics and the fact that $\mathsf{pc}_1 = \mathsf{pc}_2 = \mathsf{pc}$.

If $\ell \not\sqsubseteq \zeta$ then $\langle M_2, \mathsf{pc}, \gamma_2(\mathit{prim})\rangle \Downarrow bv'_{\ell'}$ where it is also the case that $\ell' \not\sqsubseteq \zeta$. ($\mathit{prim}$ either contains a variable, which forces ℓ' to be high, or *prim* contains a value explicitly labeled with a high-label.) It follows that $bv_\ell \approx_\zeta bv'_{\ell'} : \sigma$ and so we take $\gamma'_1 = \gamma_1\{x \mapsto bv_\ell\}$ and $\gamma'_2 = \gamma_2\{x \mapsto bv'_{\ell'}\}$, and $e''_i = e$, which are easily seen to satisfy the conditions.

[*E2*]

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

In this case, e''_1 and e''_2 must be of the form $\mathtt{let}\ x = \mathtt{ref}^\sigma_\ell\ v\ \mathtt{in}$ e where $v = bv_{\ell'}$. Note that $\gamma_1(v) \approx_\zeta \gamma_2(v) : \sigma$ and so it follows that $M'_1 = M_1[L^\sigma \leftarrow \gamma_1(v) \sqcup \mathsf{pc}]$ is ζ-equivalent to $M'_2 = M_2[L^\sigma \leftarrow \gamma_2(v) \sqcup \mathsf{pc}]$, satisfying invariant (v). Now if $\ell \sqsubseteq \zeta$, we simply take $\gamma'_i = \gamma_i$, and note that $e'_1 = \gamma_1(e\{L^\sigma_{\ell \sqcup \mathsf{pc}}/x\})$ and $e'_2 = \gamma_2(e\{L^\sigma_{\ell \sqcup \mathsf{pc}}/x\})$ satisfy the required invariants. Otherwise, $\ell \not\sqsubseteq \zeta$, and we leave K, and the k_i's unchanged and let $\Gamma' = \Gamma, x : \sigma\ \mathsf{ref}_\ell$ take $\gamma'_i = \gamma_i \cup \{x \mapsto L^\sigma_{\ell \sqcup \mathsf{pc}}\}$.

[*E4*]

$$\langle M,\, \mathsf{pc},\, \mathtt{letlin}\ y' = lv\ \mathtt{in}\ e\rangle \longmapsto \langle M,\, \mathsf{pc},\, e\{lv/y'\}\rangle$$

If $lv = \mathtt{halt}^\sigma$, then the Noninterference Invariant holds trivially after the transition. Otherwise, $lv = \lambda\langle \mathsf{pc}'\rangle(x : \sigma).\, e'$. In this case, e''_1 and e''_2 are $\mathtt{letlin}\ y' = \lambda\langle \mathsf{pc}'\rangle(x : \sigma).\, e'\ \mathtt{in}\ e$. If $\mathsf{pc}' \sqsubseteq \zeta$, simply take $\mathrm{K}' = \mathrm{K}, y' : \sigma \to \mathsf{0}$ and choose $k'_i = k_i \cup \{y' \mapsto \lambda\langle \mathsf{pc}'\rangle(x : \sigma).\, e'\}$, which satisfies invariant (iv) because $k_1 \approx_\zeta k_2$ and the terms e''_1 and e''_2 are well-typed. In the case that $\mathsf{pc}' \not\sqsubseteq \zeta$, we take $k'_i = k_i$ and choose each e'_i to be $e\{\lambda\langle \mathsf{pc}'\rangle(x : \sigma).\, e'/y'\}$ which again satisfies invariant (iv) and the $\mathtt{letlin}$-invariant, (vi). The remaining invariants are easily seen to hold because the memories and ordinary value substitutions do not change.

[*E5*]

$$\langle M,\, \mathsf{pc},\, \mathtt{if0}\ 0_\ell\ \mathtt{then}\ e'_a\ \mathtt{else}\ e'_b\rangle \longmapsto \langle M,\, \mathsf{pc} \sqcup \ell,\, e'_a\rangle$$

In this case, e''_1 and e''_2 must be of the form $\mathtt{if0}\ v\ \mathtt{then}\ e_a\ \mathtt{else}\ e_b$. If v is not a variable, then by α-equivalence, e_2 must also transition via rule [*E5*]. Because M_1 and M_2 don't change, it is easy to establish that all of the invariants hold. When v is not a variable, $\gamma_1(v) = 0_\ell$ for $\ell \not\sqsubseteq \zeta$. Similarly, $\gamma_2(v) = n_{\ell'}$ for $\ell' \not\sqsubseteq \zeta$. We don't know whether the second program transitions via [*E5*] or [*E6*], but in either case it is easy to establish that the resulting configurations are $\approx_\zeta$. Clause (i) holds via the original substitutions; clause (ii) follows from the fact that the configurations are well-typed; clause (iii) holds because part (b) lets us relate *any* high-security programs; clauses (iv) through (vi) are a simple consequence of ζ-equivalence of e_1 and e_2.

[*E7*]

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

In this case, each $e_i'' = \texttt{goto}\ v\ v'\ lv$. It must be the case that $\gamma_1(v) = (\lambda[\mathsf{pc}']f(x\!:\!\sigma, y\!:\!\kappa).\, e)_\ell$. If $\ell \sqsubseteq \zeta$, then $v = (\lambda[\mathsf{pc}']f(x\!:\!\sigma, y\!:\!\kappa).\, e')_\ell$ where $e' = \gamma_1(e)$ because, by invariant (iii), the continuation could not be found in γ_1. Note that $\gamma_1(v') \approx_\zeta \gamma_2(v') : \sigma$. There are two sub-cases, depending on whether $\gamma_1(v')$ has label $\sqsubseteq \zeta$. If so, it suffices to take $\Gamma' = \Gamma$, $\mathrm{K}' = \mathrm{K}$, and leave the substitutions unchanged, for we have $e_i' = \gamma_i(k_i(e\{v/f\}\{\gamma_i(v') \sqcup \mathsf{pc}/x\}\{lv/y\}))$. Otherwise, if the label of $\gamma_1(v') \not\sqsubseteq \zeta$, we take $\Gamma' = \Gamma, x : \sigma$ and $\gamma_i' = \gamma_i\{x \mapsto \gamma_i(v') \sqcup \mathsf{pc}\}$. The necessary constraints are then met by $e_i' = \gamma_i'(k_i(e\{v/f\}\{lv/y\}))$.

The other case is that $\ell \not\sqsubseteq \zeta$, and hence the label of $\gamma_2(v)$ is also $\not\sqsubseteq \zeta$. Thus, $\mathsf{pc}_1' = \mathsf{pc} \sqcup \ell \not\sqsubseteq \zeta$ and $\mathsf{pc}_2' \not\sqsubseteq \zeta$. The resulting configurations satisfy part (b) of clause (iii). The bodies of the continuations are irrelevant, as long as the other invariants are satisfied, but this follows if we build the new value substitutions as in the previous paragraph

LEMMA A.2 (High-pc Step). *Suppose*

- $\Gamma \parallel \mathrm{K} \vdash \langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \approx_\zeta \langle M_2,\, \mathsf{pc}_2,\, e_2\rangle$
- $\mathsf{pc}_1 \not\sqsubseteq \zeta$ *and* $\mathsf{pc}_2 \not\sqsubseteq \zeta$
- $\langle M_1,\, \mathsf{pc}_1,\, e_1\rangle \longmapsto \langle M_1',\, \mathsf{pc}_1',\, e_1'\rangle$

then 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. By cases on the transition step of the first configuration. Because $\mathsf{pc}_1 \not\sqsubseteq \zeta$ and all rules except [*E8*] increase the program-counter label, we may choose zero steps for e_2 and still show that $\approx_\zeta$ is preserved. Condition (iii) 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 (v) of the invariant.

Now consider the case for [*E8*]. Let $e_1 = \gamma_1(k_1(e_1''))$, then $e_1'' = \mathtt{lgoto}\ lv\ v_1$ 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, and it easily follows that the resulting configurations are $\approx_\zeta$ under Γ and K. Otherwise, lv is the variable y. By assumption, $k_1(y) = \lambda\langle\mathsf{pc}\rangle(x\!:\!\sigma).\,e$, where $\mathsf{pc} \sqsubseteq \zeta$. Assume e_2 does not diverge. By the Progress Lemma $\langle M_2,\, \mathsf{pc}_2,\, e_2\rangle \longmapsto^* \langle M_2',\, \mathsf{pc}_2',\, \mathtt{lgoto}\ k_2(y)\ v_2\rangle$ (by assumption, it can't diverge). 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 (iv), $k_2(y) \equiv_\alpha k_1(y)$. Furthermore, [*TE7*] 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 the tail of K, which we choose for K$'$. Finally, let $e_1' = \gamma_1'(k_1'(e))$ and $e_2' = \gamma_2'(k_2'(e))$. All of the necessary conditions are satisfied as is easily verified via the operational semantics. $\square$