A Typed Assembly Language for Confidentiality

Dachuan Yu and Nayeem Islam

DoCoMo Communications Laboratories, USA
{yu, nayeem}@docomolabs-usa.com

Abstract. Language-based information-flow analysis is promising in protecting data confidentiality. Although much work has been carried out in this area, relatively little has been done for assembly code. Source-level techniques do not easily generalize to assembly code, because assembly code does not readily present certain abstraction about the program structure that is crucial to information-flow analysis. Nonetheless, low-level information-flow analysis is desirable, because it yields a small trusted computing base. Furthermore, many (untrusted) applications are distributed in native code; their verification should not be overlooked.

We present a simple yet effective solution for this problem. Our observation is that the missing abstraction in assembly code can be restored using annotations. Following the philosophy of certifying compilation, these annotations are generated by a compiler, used for static validation, and erased before execution. In particular, we propose a type system for low-level information-flow analysis. Our system is compatible with Typed Assembly Language, and models key features including a call stack, memory tuples and first-class code pointers. A noninterference theorem articulates that well-typed programs respect confidentiality.

1 Introduction

With the growing reliance on networked information systems, the protection of confidential data becomes increasingly important. The problem is especially subtle for a computing system which both manipulates sensitive data and requires access to public information channels. Simple policies that restrict the access to either the sensitive data or the public channels (or a combination) often prove too restrictive. A more desirable policy is that no information about the sensitive data can be inferred from observing the public channels, even though a computing system is granted access to both. Such a regulation of the flow of information is often referred to as *information flow*, and the policy that sensitive data should not affect public data is often called *noninterference*.

Whereas it is relatively easy to detect and prevent naive violations that directly give out sensitive data, it is much more difficult to prevent applications from sending out information that is sophisticatedly encoded. Conventional security mechanisms such as access control, firewalls, encryption and anti-virus fall short on enforcing the noninterference policy [14]. On the one hand, noninterference posts seemingly conflicting requirements: it allows the access to sensitive information, but restricts the flow of it. On the other hand, the violation of noninterference cannot be observed from monitoring a single execution of the program [16], yet such execution monitoring is the basis of many conventional mechanisms.

P. Sestoft (Ed.): ESOP 2006, LNCS 3924, pp. 162–179, 2006.

In recent years, much effort has been put on enforcing noninterference using techniques based on programming language theory and implementation. These techniques are promising, because they directly inspect or instrument the program code, and hence have the potential of learning all possible run-time behavior of the program. Unfortunately, the vast amount of language-based research on information flow [14] does not address well the problem for assembly code. The challenge there, as we will elaborate later, largely lies in working with the lack of high-level abstractions and managing the extreme flexibility offered by assembly code.

Nonetheless, it is desirable to enforce noninterference directly at a low-level. On the one hand, high-level programs must be translated into low-level code before executed on a real machine; compilation or optimization bugs may invalidate the security guarantee established for the source programs. On the other hand, some applications are distributed (*e.g.,* native code for mobile computation) or even directly written (*e.g.,* core libraries for embedded systems) in assembly; enforcement at a low-level is a must.

This paper presents some important steps of a project tackling information flow at the assembly level. The contributions are:

- We propose a Typed Assembly Language for Confidentiality (TAL_C) and present its proof of noninterference. Our abstract machine is generic and close to real architectures. To reuse existing results on low-level verification, our system is designed to be compatible with Typed Assembly Language (TAL) [11]. It thus approaches a unified framework for conventional type safety and security.
- Our system models key features of an assembly language, including heap, call stack and register file, memory tuples (aliasing), and first-class code pointers (higher-order functions). Because assembly code is often arduous to work with, we present our formal result with a core language supporting the above features for ease of understanding, but also informally discuss other extensions.
- Although desirable to directly verify at an assembly level, it is more practical to develop programs in high-level languages. We briefly discuss a translation from a system of linear continuations [24] to TAL_C. A companion technical report [23] presents a translation from a security-typed imperative source language with first-order procedures to TAL_C, illustrating certifying compilation for noninterference.

This paper does not address covert channels (*e.g.,* termination [19, 1] and timing [21, 2]) or abstract-violation attacks (*e.g.,* cache [3]). Section 2 provides background on language-based approaches for information flow, places our work in the context of existing researches, and points out the extra difficulties for noninterference at an assembly level. An informal overview of our approach is given in Section 3. Section 4 presents the TAL_C system, focusing on core features that illustrate ideas pertinent to information flow. Section 5 helps better understand TAL_C in comparison with work on linear continuations. Orthogonal issues and practical extensions are discussed in Section 6.

2 Background

2.1 Information Flow

The problem of information flow can be abstracted as a program that operates on data of different security levels, *e.g., low* and *high*. Low data are public data that may be

observed by all principals; high data are secret data whose access is restricted. An information-flow policy requires that no information about high inputs can be inferred from observing low outputs. The security levels can also be generalized to a lattice [20].

Such a policy concerns tracking the flow of information inside a target system. Whereas it is easy to detect explicit flows (*e.g.,* through an assignment from a secret `h` to a public `l` with `l=h`), it is much harder to detect various forms of implicit flow. For example, the statement `l=0; if h then l=1` involves an implicit flow from `h` to `l`. At run-time, if the `then` branch is not taken, a conventional security mechanism based on execution monitoring will not detect any violation. However, information about `h` can indeed be inferred from the result of `l`.

Instead of observing a single execution, language-based techniques derive assurance about a program's behavior by examining, and possibly instrumenting, the program code. In the above example, the information essentially leaks through the program counter (often referred to as *pc*)—the fact that a branch is taken reflects information about the guard of the conditional. In response, a security-type system typically tags the pc with a security label. If the guard of a conditional concerns high data, then the branches are verified under a pc with a high security label. Furthermore, assignments to low variables are prohibited under such a high pc.

2.2 Related Work

Although there has been much work applying language-based techniques to information flow [14], most of it focused on high-level languages. Many high-level abstractions have been formally studied, including functions [8], exceptions [13], objects [5], and concurrency [18, 1], and practical implementation is within reach [12]. Nonetheless, enforcing information flow at only a high level puts the compiler into the trusted computing base (TCB) [15]. Furthermore, we should not overlook the verification of software distributed (or written) directly in low-level code.

Barthe *et al.* [6] presented a security-type system for a bytecode language and a translation that preserves security types. Their stack-based language is much different from the RISC architecture that we model. More importantly, their verification circumvents a main difficulty—the lack of program structures at a low-level—by introducing a trusted component that computes the dependence regions and postdominators [4] for conditionals. This component is inside the TCB and trusted.

Zdancewic and Myers [24] used linear continuations to enforce noninterference at a low-level. Their language is based on variables and still much different from assembly language. In particular, linear continuations, although useful in enforcing a stack discipline that helps information-flow analysis, are absent from conventional assembly code. Hence further (trusted) compilation to native code is required. Nonetheless, we borrowed some ideas from Zdancewic and Myers, including the handling of memory aliasing and code pointers, although these features are modeled as different constructs in our system. A more thorough discussion of the connection between linear continuations and our solution is given in Section 5, after presenting our system.

Bonelli *et al.* [7] explored the realization of linear continuations in an assembly language SIFTAL. Two new instructions are introduced in correspondence with the operations on linear continuations as proposed by Zdancewic and Myers [24]. These

two instructions enforce structured control flows that are missing from normal assembly code with the help of a continuation stack (this stack is different from the one for function calls). One instruction pushes a linear continuation onto the stack, the other pops a linear continuation off the stack and transfers the control to it. Such a mechanism maintains structured control flow in assembly code, thus helps information-flow analysis. Unfortunately, conventional assembly programming and machine models do not contain such a special continuation stack and the instructions manipulating it.

Recently, Medel *et al.* [9] improved SIFTAL to SIF, using a stack of labels to simplify the above approximation of linear continuations. Unlike SIFTAL, SIF resorts to static type annotations to enforce noninterference, and no longer requires a stack of linear continuations during execution. This is in spirit similar to our solution of TAL_C. However, SIF supports only a minimal set of language features (arithmetic, memory update, branching and direct jumps), and does not address how the type annotations can be produced. In contrast, our language TAL_C further supports code pointers and a call stack. The companion technical report [23] presents a type-preserving translation to TAL_C from a security-typed source language, where the support for procedure calls introduces extra subtleties for noninterference. Section 5 provides more details.

This paper targets RISC-style assembly code. We introduce a type system to verify the unstructured control flow, which in turn helps information-flow analysis. Type annotations are used to recover information about high-level program structures, and no trusted component is required for computing postdominators. This contrasts with the above work on bytecode languages. Furthermore, we do not rely on extra constructs such as linear continuations or a continuation stack. An erasure semantics trivially reduces programs in our language to normal assembly code.

2.3 Assembly Code

There are several challenges for enforcing information flow for assembly code.

First, high-level languages make use of a virtually infinite number of variables, each of which can be assigned a fixed security label. In assembly code, the use of memory cells is similar. However, a finite number of registers are reused for different source-level variables. As a result, one cannot assign a fixed security label to a register.

Second, the control flow of an assembly program is not as structured. The body of a conditional is often not obvious, and generally undecidable, from the program code. Hence the idea of using a security context to prevent implicit flow through conditionals cannot be easily carried out.

Third, assembly languages are very expressive. Aliasing between memory cells are difficult to reason about [17]. The support for first-class code pointers (the reflection of higher-order functions at the assembly level) is very subtle. A code pointer may direct a program to different execution paths, even though no branching instruction is present.

Fourth, since it is not practical to always directly program in an assembly language, a low-level type system must be designed so that the type annotations can be generated automatically, *e.g.,* through certifying compilation. The type system must be at least as expressive as a high-level type system, so that any well-typed source program can be translated into well-typed assembly code.

Finally, it is desirable to achieve an erasure semantics where type annotations have no effect at runtime. A security mechanism can not be generally applied in practice if it incurs too much overhead. Similarly, it is also undesirable to change the programming model for accommodating the verification needs. Such a model change indicates either a trusted compilation process or a different target machine.

3 Our Approach

3.1 Explicit Assignment

An obvious kind of information flow is through assignment. In a high-level language, variables can be "tagged" with security labels; the security-type system prevents label mismatch for assignments. At an assembly level, memory cells can be tagged similarly. When storing into a memory cell, a typing rule ensures that the security label of the source matches that of the target.

Registers need to be regulated differently, because they can be reused for different variables with different security labels (registers cannot be aliased, which makes it safe to update their types). Since variable and liveness information is not available at an assembly level, one can not easily base the enforcement upon that.

In fact, a similar problem arises even for normal type safety. A register in TAL can have different types at different program points. These types are essentially inferred from the computation itself. For instance, in an addition instruction $\texttt{add}\, r_d, r_s, r_t$, the register r_d is given the type `int`, because only `int` can be valid here. Similarly, when loading from a memory cell, the target register is given the type of the source memory cell. Adapting such inference for security labels is straightforward. In the addition $\texttt{add}\, r_d, r_s, r_t$, the label of r_d is obtained by joining the labels of r_s and r_t, because the result in r_d reflects information from both r_s and r_t. Moving and memory reading instructions are handled similarly.

3.2 Program Structure

A conditional statement in a high-level program can be verified so that both subcommands respect the security level of the guard expression. Such verification becomes difficult in assembly code, where the "flattened" control flow provides little help in identifying the program structure. A conditional is typically translated into a branching instruction and some code blocks, where the postdominator of the two branches are no longer apparent.

We use annotations to restore the program structure by pointing out the postdominators whenever they are needed. Note that high-level programs provide sufficient information for deciding the postdominators, and these postdominators can always be statically determined. For instance, the end of a conditional command is the postdominator of the two branches. Hence a compiler can generate the annotations automatically based on a securely typed source program. In our system, our postdominator annotation is essentially a static code label paired with a security label.

Since branching instructions ($\texttt{bnz}\, r, l$) are the only instructions that could directly result in different execution paths, it would appear that one should enhance branching

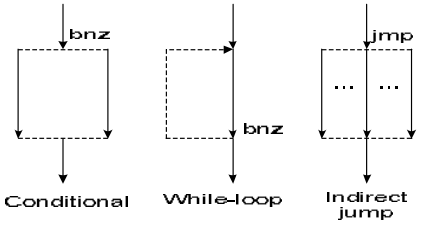

Fig. 1. Flow through program structure

instructions with postdominators. The typing rule then checks both branches under a proper security context that takes into account the guard expression. Such a security context terminates when the postdominator is reached.

Although plausible, this approach is awkward. Figure 1 demonstrates three scenarios. Besides the conditional scenario, branching instructions are also used to implement while-loops, where the postdominator is exactly the beginning of one of the branches. In this case, only the other branch should be checked under a new security context. If we directly annotate the branching instruction, the corresponding typing rule would be "overloaded." More importantly, an assembly program may contain "implicit branches" where no branching instruction is present. The third scenario illustrates that an indirect jump may lead the program to different paths based on the value of its operand register. A concrete example will appear in Section 3.5.

Inspiration of a better solution lies in high-level security-type systems [23], which typically use a subsumption rule to outline a region of computation where the security level is raised from low to high. The end of the region is exactly a postdominator. Following this, our approach is to mimic the high-level subsumption rule with two low-level *raising* and *lowering* operations that explicitly manipulate the security context and mark the beginning and the end of the secured region.

3.3 Memory Aliasing

Aliasing of memory cells present another channel for information transfer. In Figure 2, a low pointer `p_l` and a high pointer `p_h` are aliases of the same cell. This is useful if a high principal wishes to observe a low computation. The code in this figure may change the aliasing relation based on some high variable `h` by letting `p_h` point to another cell. Further modification through `p_h` may or may not change the value in the original cell. As a result, observing through the low pointer `p_l` reveals information about the high variable `h`.

The problem lies in the assignment through the high pointer `p_h`, because it reveals information about the aliasing relation. The solution, following Zdancewic and Myers [24], is to tag pointers with two security labels. One is for the pointer itself, and the other is for the data being referenced. Assignments to low data through high pointers are not allowed. This is a conservative approach—all pointers are considered as potential aliases.

```
(* suppose p_h alias p_l *)
if (h=0) then p_h=new cell;
*p_h=1;
(* now *p_l reveals h *)
```

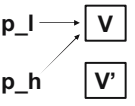

Fig. 2. Flow through aliasing

```
fun f0 () = (l:=0; ())
fun f1 () = (l:=1; ())
let f = (if h then f1 else f0) in f()
```

Fig. 3. Flow through code pointer

3.4 Code Pointers

Code pointers further complicate information flow. Figure 3 shows a piece of functional code where `f` represents different functions based on a high variable `h`. In its reflection at an assembly level, different code blocks will be executed based on the value of `h`. Naturally, `f` contains sensitive information and should be labeled high. However, the actual functions `f0` and `f1` can only be executed under a low context, because they modify a low variable `l`. In this case, the invocation to `f` should be prohibited.

In our system, similar to data pointers, code pointers are also given two security labels. The typing rules ensure that no low function is called through a high code pointer.

3.5 Security Context Coercion

Finally, Figure 4 shows a piece of code where a mutable code pointer complicates the flow analysis. Functions `f0` and `f1` only modify high data. A reference cell `f` is assigned different code pointers within a high conditional. Later, `f` is dereferenced and invoked in a low context.

```
fun f0 () = (h':=0; ())
fun f1 () = (h':=1; ()) ...
if h then f := f1 else f := f0;
l:=1; !f(); l:=l*2; ...
```

Fig. 4. Context coercion without branching

This code is safe with respect to information flow. At a high level, a subsumption rule allows calling the high function `!f()` in a low context. However, in its assembly counterparts, both the calling to `f` and the returning from `f` are implemented as indirect jumps. The calling sequence transfers the control from a low context to a high context, whereas the returning sequence does the opposite. Since the function invocation is no longer atomic at an assembly level, one cannot directly devise a subsumption rule. Furthermore, there is no explicit branching instruction present when `f` is dereferenced and invoked (the third scenario of Figure 1).

In our system, the raising and lowering operations explicitly mark the boundary of the subsumption rule. During certifying compilation, the source-level typing and program structure provide sufficient information for generating the target-level annotations. When a subsumption rule is applied in the source code, the corresponding target code is generated within a pair of raising and lowering operations.

4 TAL$_C$

4.1 Abstract Machine

Our language TAL$_C$ is designed to resemble TAL [11] for ease of understanding. We introduce some new constructs for confidentiality, and accommodate a stack following STAL [10] for supporting procedure calls. For simplicity, we omitted from TAL and STAL some features (*e.g.,* polymorphism, existential types and heap allocation) that are orthogonal to our proposed security operations.

We assume that security labels form a lattice $\mathcal{L}$. We use θ to range over elements of $\mathcal{L}$. We use $\bot$ and $\top$ as the bottom and top of the lattice, $\cup$ and $\cap$ as the lattice join and meet operations, and $\subseteq$ as the lattice ordering. The syntactic constructs of TAL$_C$ can be understood in three steps as follows.

Type Constructs. The top portion of Figure 5 presents the type constructs. Security contexts κ follow the idea of Section 3.2. An empty security context ($\bullet$) represents an program counter with the lowest security label. A concrete context ($\theta \triangleright w$) is made up of a security label θ (the current security level) and a postdominator w. The postdominator w has the syntax of a word value, but its use is restricted by the semantics to be

$$
\begin{array}{rll}
\textit{(contexts)} & \kappa & ::= \bullet \mid \theta \triangleright w \\
\textit{(pre-type)} & \tau & ::= \mathtt{int} \mid \langle \sigma_1, \ldots, \sigma_n \rangle \mid \forall[\Delta].\langle \kappa \rangle\, \Gamma \\
\textit{(types)} & \sigma & ::= \tau_\theta \mid ns \\
\textit{(stack ty)} & \Sigma & ::= \rho \mid nil \mid \sigma :: \Sigma \\
\textit{(var env)} & \Delta & ::= \circ \mid \rho\Delta \mid \alpha\Delta \\
\textit{(type arg)} & \psi & ::= \Sigma \mid w \\
\textit{(heap ty)} & \Psi & ::= \{l_1 : \sigma_1, \ldots l_n : \sigma_n\} \\
\textit{(reg file ty)} & \Gamma & ::= \{r_1 : \sigma_1, \ldots r_n : \sigma_n, \mathtt{sp} : \Sigma\} \\
\textit{(registers)} & r & ::= r_1 \mid r_2 \mid \ldots \\
\textit{(word val)} & w & ::= \alpha \mid l \mid i \mid ns \mid w[\psi] \\
\textit{(small val)} & v & ::= r \mid w \mid v[\psi] \\
\textit{(heap val)} & h & ::= \langle w_1, \ldots, w_n \rangle \mid \mathtt{code}[\Delta]\langle \kappa \rangle \Gamma.I \\
\textit{(heaps)} & H & ::= \{l_1 \mapsto h_1, \ldots, l_n \mapsto h_n\} \\
\textit{(reg files)} & R & ::= \{r_1 \mapsto w_1, \ldots, r_n \mapsto w_n, \mathtt{sp} \mapsto S\} \\
\textit{(stacks)} & S & ::= nil \mid w :: S \\
\textit{(instr)} & \iota & ::= \mathtt{add}\, r_d, r_s, v \mid \mathtt{ld}\, r_d, r_s(i) \mid \mathtt{st}\, r_d(i), r_s \mid \mathtt{mov}\, r_d, v \mid \mathtt{bnz}\, r, v \\
 & & \quad \mid \mathtt{salloc}\, i \mid \mathtt{sfree}\, i \mid \mathtt{sld}\, r_d, \mathtt{sp}(i) \mid \mathtt{sst}\, \mathtt{sp}(i), r_s \mid \mathtt{raise}\, \kappa \\
\textit{(instr seq)} & I & ::= \iota; I \mid \mathtt{lower}\, w \mid \mathtt{jmp}\, v \mid \mathtt{halt}\, [\sigma] \\
\textit{(prog)} & P & ::= (H, R, I)_\kappa
\end{array}
$$

Fig. 5. Syntax of TAL$_C$

eventually an instantiated code label, *i.e.,* the ending point of the current security level. The postdominator w could also be a variable α; this is useful for compiling procedures, which can be called in different contexts with different postdominators.

Pre-types (τ) reflect the normal types as seen in TAL, including integer types, tuple types, and code types. In comparison with TAL, our code type requires an extra security context (κ) as part of the interface. A type (σ) is either a pre-type tagged with a security label or a nonsense type (ns) for uninitialized stack slots. A stack type (Σ) is either a variable (ρ), or a (possibly empty) sequence of types. We sometimes use nature numbers to refer to stack slots; slot 0 refers to the bottom-most element. The variable context (Δ) is used for typing polymorphic code; it documents stack type variables (ρ) and postdominator variables (α). Stack types and postdominators are also generally referred to as type arguments ψ. Finally, heap types (Ψ) or register file types (Γ) are mappings from heap labels or registers to types; the `sp` in the register file represents the stack.

Note that the type constructs provide two layers of security labels for a data pointer (*e.g.,* $\langle \mathtt{int}_{\theta_2} \rangle_{\theta_1}$; Section 3.3) or a code pointer (*e.g.,* $(\forall[\circ].\langle \theta_2 \triangleright l \rangle\ \Gamma)_{\theta_1}$; Section 3.4)—one ($\theta_1$) for the pointer itself, the other (θ_2) for the data or code being referenced.

Value Constructs. The middle portion of Figure 5 shows the value constructs. A word value w is either a variable, a heap label l, an immediate integer i, a nonsense value for an uninitialized stack slot, or another word value instantiated with a type argument. Small values v serve as the operands of some instructions; they are either registers r, word values w, or instantiated small values. Heap values h are either tuples or typed code sequences; they are the building blocks of the heap H. Note that a value does not carry a security label. This is consistent with the philosophy that a value is never intrinsically sensitive—it is sensitive only if it comes from a sensitive location [20], which is documented in the corresponding types (Ψ and Γ). Finally, a register file R stores the contents of all registers and the stack, where the stack is a (possibly empty) sequence of word values.

Code Constructs. Code constructs are given in the bottom portion of Figure 5. We retain a minimal set of instructions from TAL and STAL, and introduce two new instructions (`raise` κ and `lower` l) for manipulating the security context as discussed in Section 3. A program is the usual triple tagged with a security context. The security context facilitates the formal soundness proof, but does not affect the computation.

In the operational semantics (available in the technical report [23]), there are only two cases that modify the security context: `raise` κ' updates the security context to κ', and `lower` w picks up a new security context from the interface of the target code w. In all other cases, the security context remains the same, and the semantics is standard. It is easy to see that this operational semantics mimics the behavior of a real machine, and does not prohibit bad flows. One can obtain a conventional machine by removing the security contexts and `raise` κ instructions, and replacing `lower` w with `jmp` w.

4.2 Typing Rules

The static semantics consists of judgment forms summarized in Figure 6. A security context appears in the judgment of a valid instruction sequence. Heap and register file types are made explicit in the judgment of a valid program for facilitating the noninterference theorem. All other judgment forms closely resemble those of TAL and STAL.

Judgment	Meaning
$\Delta \vdash \kappa$	κ is a valid context
$\Delta \vdash \tau$	τ is a valid pre-type
$\Delta \vdash \sigma$	σ is a valid type
$\Delta \vdash \Sigma$	Σ is a valid stack type
$\vdash \Psi$	Ψ is a valid heap type
$\Delta \vdash \Gamma$	Γ is a valid register file type
$\Delta \vdash \Gamma_1 \subseteq \Gamma_2$	Register file type Γ_1 weakens Γ_2
$\vdash H : \Psi$	Heap H has type Ψ
$\Psi \vdash S : \Sigma$	Stack S has type Σ
$\Psi \vdash R : \Gamma$	Register file R has type Γ
$\Psi \vdash h : \sigma$	Heap value h has type σ
$\Psi; \Delta \vdash w : \sigma$	Word value w has type σ
$\Psi; \Delta; \Gamma \vdash v : \sigma$	Small value v has type σ
$\Psi; \Delta; \Gamma; \kappa \vdash I$	I is a valid sequence of instructions
$\Psi; \Gamma \vdash P$	P is a valid program

Fig. 6. TAL_C typing judgments

The typing rules are available in the technical report [23]. Selected typing rules are given in Figure 7. A type construct is valid (top six judgment forms in Figure 6) if all free type variables are documented in the type environment. Heap values and integers may have any security label. The types of heap labels and registers are as described in the heap type and the register file type respectively. All other rules for non-instructions are straightforward extensions of those in TAL and STAL.

We use $SL(\kappa)$ to refer to the security label component of κ. $SL(\bullet)$ is defined to be $\bot$. The typing rules for `add`, `ld` and `mov` instructions infer the security labels for the destination registers; they take into account the security labels of the source and target operands and the current security context.

The rule for `bnz` first checks that the guard register r is an integer and the target value v is a code label. It then checks that the current security context is high enough to cover the security levels of the guard (preventing flows through program structures; Section 3.2) and the target code (preventing flows through code pointers; Section 3.4). Lastly, the checks on the register file and the remainder instruction sequence make sure that both branches are secure to execute.

The rule for `st` concerns four security labels. The label of the target cell must be higher than or equal to those of the context (Section 3.2), the containing tuple (Section 3.3), and the source value (Section 3.1).

The rules for the stack instructions follow similar ideas. In essence, the stack can be viewed as an infinite number of registers. Instruction `salloc` or `sfree` add new slots to or remove existing slots from the slot, so the rules check the remainder instruction sequence under an updated stack type. The rule for instruction `sld` or `sst` can be understood following that of the `mov` instruction.

The rule for `raise` checks that the new security context is higher than the current one. Moreover, it looks at the postdominator w' of the new context, and makes sure that

$$\frac{\circ \vdash \kappa \qquad \vdash H : \Psi \qquad \Psi \vdash R : \Gamma \qquad \Psi; \circ; \Gamma; \kappa \vdash I}{\Psi; \Gamma \vdash (H, R, I)_\kappa}$$

$$\frac{SL(\kappa) = \theta \qquad \Gamma(r_s) = \langle \sigma_1, \ldots, \sigma_n \rangle_{\theta_1} \qquad \sigma_i = \tau_{\theta_2} \\ \Psi; \Delta; \Gamma\{r_d : \tau_{\theta \cup \theta_1 \cup \theta_2}\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{ld}\ r_d, r_s(i); I}$$

$$\frac{SL(\kappa) = \theta \qquad \Gamma(r) = \texttt{int}_{\theta_1} \qquad \Psi; \Delta; \Gamma \vdash v : (\forall[\circ].\langle \kappa \rangle\ \Gamma')_{\theta_2} \\ \theta_1 \cup \theta_2 \subseteq \theta \qquad \Delta \vdash \Gamma' \subseteq \Gamma \qquad \Psi; \Delta; \Gamma; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{bnz}\ r, v; I}$$

$$\frac{SL(\kappa) = \theta \qquad \Gamma(r_d) = \langle \sigma_1, \ldots, \sigma_n \rangle_{\theta_1} \qquad \sigma_i = \tau_{\theta'} \\ \Gamma(r_s) = \tau_{\theta_2} \qquad \theta \cup \theta_1 \cup \theta_2 \subseteq \theta' \qquad \Psi; \Delta; \Gamma; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{st}\ r_d(i), r_s; I}$$

$$\frac{\Gamma(\texttt{sp}) = \Sigma \qquad \Psi; \Delta; \Gamma\{\texttt{sp} : \overbrace{ns :: \ldots :: ns}^{i} :: \Sigma\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{salloc}\ i; I}$$

$$\frac{SL(\kappa) = \theta \qquad \Gamma(\texttt{sp}) = \sigma_0 :: \ldots :: \sigma_i :: \Sigma \qquad \Gamma(r_s) = \tau_{\theta'} \\ \Psi; \Delta; \Gamma\{\texttt{sp} : \sigma_0 :: \ldots :: \sigma_{i-1} :: \tau_{\theta \cup \theta'} :: \Sigma\}; \kappa \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{sst}\ \texttt{sp}(i), r_s; I}$$

$$\frac{\kappa = \theta \triangleright w \qquad \kappa' = \theta' \triangleright w' \qquad \theta \subseteq \theta' \\ \Psi; \Delta \vdash w' : (\forall[\circ].\langle \kappa \rangle\ \Gamma')_{\theta_1} \qquad \Psi; \Delta; \Gamma; \kappa' \vdash I}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{raise}\ \kappa'; I}$$

$$\frac{\kappa = \theta \triangleright w \qquad \Psi; \Delta \vdash w : (\forall[\circ].\langle \kappa' \rangle\ \Gamma')_{\theta_1} \\ \theta_1 \subseteq SL(\kappa') \qquad \Delta \vdash \Gamma' \subseteq \Gamma}{\Psi; \Delta; \Gamma; \kappa \vdash \texttt{lower}\ w}$$

Fig. 7. Selected typing rules of TAL$_C$

the security context at w' matches the current one. The remainder instruction sequence is checked under the new context.

Since the rule for `raise` already checked the validity of the ending label of a secured region, the task for ending the region is relatively simple. The rule for `lower` checks that its operand label matches that dictated by the security context. This guarantees that a secured region be enclosed within a `raise`-`lower` pair. The rule also makes sure that the code at w is safe to execute, which involves checking the security labels (Section 3.4) and the register file types.

The rule for `jmp` checks that the target code is safe to execute. Similar checks also appeared in the rule for `bnz`. In these two rules, the security context of the target code is always the same as the current one. This is because context changes are separated from conventional instructions in our system. For example, one may enclose high target code within `raise` and `lower` before calling it in a low context.

Finally, halting is valid only if the security context is empty, and the value in r_1 has the expected type σ.

We note that supporting explicit heap allocations does not introduce new difficulties. Although different program branches may exhibit different allocation behaviors under

a high security context, they must agree at a common heap type when joining at a corresponding `lower` instruction. Furthermore, we define the equivalence of heaps (stacks, register files) based only on their elements of low security types (the next section).

4.3 Soundness

TAL$_C$ enjoys conventional type safety (memory and control-flow safety), which can be established following the preservation and progress lemmas.

Before presenting the noninterference theorem, we define the equivalence of two programs with respect to a security level θ. Intuitively, two programs (heaps, stacks, register files) are equivalent if and only if they agree on their low-security contents.

Definition 1 (Heap Equivalence). $\Psi \vdash H_1 \approx_\theta H_2 \iff$ *for every* $l \in \mathrm{dom}(\Psi)$, *if* $\Psi(l) = \tau_{\theta'}$ *and* $\theta' \subseteq \theta$ *then* $H_1(l) = H_2(l)$.

Definition 2 (Stack Equivalence). $\Sigma \vdash S_1 \approx_\theta S_2 \iff$ *for every stack slot* i, *if* $\Sigma(i) = \tau_{\theta'}$ *and* $\theta' \subseteq \theta$ *then* $S_1(i) = S_2(i)$.

Definition 3 (Register File Equivalence). $\Gamma \vdash R_1 \approx_\theta R_2 \iff$ *(1)* $\Gamma(sp) \vdash R_1(sp) \approx_\theta R_2(sp)$, *and (2) for every* $r \in \mathrm{dom}(\Gamma)$, *if* $\Gamma(r) = \tau_{\theta'}$ *and* $\theta' \subseteq \theta$, *then* $R_1(r) = R_2(r)$.

Definition 4 (Program Equivalence). $\Psi; \Gamma \vdash P_1 \approx_\theta P_2 \iff P_1 = (H_1, R_1, I_1)_{\kappa_1}$, $P_2 = (H_2, R_2, I_2)_{\kappa_2}$, $\Psi \vdash H_1 \approx_\theta H_2$, $\Gamma \vdash R_1 \approx_\theta R_2$, *and either:*
(1) $\kappa_1 = \kappa_2$, $SL(\kappa_1) \subseteq \theta$, *and* $I_1 = I_2$, *or (2)* $SL(\kappa_1) \not\subseteq \theta$, $SL(\kappa_2) \not\subseteq \theta$.

It is easy to see that the above relations are all reflexive, symmetrical, and transitive. Our noninterference theorem relates the executions of two equivalent programs that both start in a low security context (relative to the security level of concern). If both executions terminate, then the result programs must also be equivalent.

The idea of the proof is intuitive. Given a security level of concern, the executions can be phased into "low steps" and "high steps." It is easy to relate the two executions under a low step, because they involve the same instructions. Under a high step, the two executions are no longer in lock step. Recall that `raise` and `lower` mark the beginning and the end of a secured region. We relate the program states before the `raise` and after the `lower`, circumventing directly relating two executions under high steps. Interested readers are referred to the technical report [23] for more details.

Theorem 1 (Noninterference). *If* $P = (H, R, I)_\kappa$, $SL(\kappa) \subseteq \theta$, $\Psi; \Gamma \vdash P$, $\Psi; \Gamma \vdash Q$, $\Psi; \Gamma \vdash P \approx_\theta Q$, $P \longmapsto^* (H_p, R_p, \texttt{halt}\ [\sigma_p])_\bullet$, *and* $Q \longmapsto^* (H_q, R_q, \texttt{halt}\ [\sigma_q])_\bullet$, *then exists* Γ' *such that* $\Psi; \Gamma' \vdash (H_p, R_p, \texttt{halt}\ [\sigma_p])_\bullet \approx_\theta (H_q, R_q, \texttt{halt}\ [\sigma_q])_\bullet$.

4.4 Example

Figure 8 gives a simple example to demonstrate the use of security labels and contexts. The high-level pseudo-code program involves a low variable `a` and two high variables `b` and `c`. In a corresponding TAL$_C$ program, we use heap cells labeled l_a, l_b and l_c to represent these variables. The TAL$_C$ program starts from the code labeled l_0 in a low security context. After the initial setup, it raises the security context to $\top \triangleright l_3$. The

A pseudo-code program: $\texttt{a} = 0; \texttt{if } (\texttt{b} <> 0) \texttt{ then c} = 1 \texttt{ else c} = 0; \texttt{a} = 1$

A corresponding TAL$_C$ program: $(H, \{\texttt{sp} : nil\}, \texttt{jmp } l_0)_\bullet$ where $H = \{(l_a, l_b, l_c \text{ omitted})$

$l_0 \mapsto$ $\texttt{code}[\circ]\langle\bullet\rangle\{\texttt{sp} : nil\}.$
$\texttt{mov } r_0, 0;$ % $r_0 \leftarrow 0$
$\texttt{mov } r_1, l_a;$ % $r_1 \leftarrow l_a$
$\texttt{mov } r_2, l_b;$ % $r_2 \leftarrow l_b$
$\texttt{mov } r_3, l_c;$ % $r_3 \leftarrow l_c$
$\texttt{st } r_1(0), r_0;$ % $l_a \leftarrow 0$
$\texttt{raise } \top \triangleright l_3;$ % raise security context
$\texttt{jmp } l_1$

$l_1 \mapsto$ $\texttt{code}[\circ]\langle\top \triangleright l_3\rangle\{r_0 : \langle\texttt{int}_\bot\rangle_\bot, r_1 : \langle\texttt{int}_\bot\rangle_\bot, r_2 : \langle\texttt{int}_\top\rangle_\bot, r_3 : \langle\texttt{int}_\top\rangle_\bot, \texttt{sp} : nil\}.$
$\texttt{ld } r_4, r_2(0);$
$\texttt{bnz } r_4, l_2;$ % go to l_2 if content of l_b is not zero
$\texttt{st } r_3(0), r_0;$ % the `else` branch: $l_c \leftarrow 0$
$\texttt{lower } l_3$ % restore security context and go to l_3

$l_2 \mapsto$ $\texttt{code}[\circ]\langle\top \triangleright l_3\rangle\{r_0 : \langle\texttt{int}_\bot\rangle_\bot, r_1 : \langle\texttt{int}_\bot\rangle_\bot, r_2 : \langle\texttt{int}_\top\rangle_\bot, r_3 : \langle\texttt{int}_\top\rangle_\bot, \texttt{sp} : nil\}.$
$\texttt{mov } r_0, 1;$
$\texttt{st } r_3(0), r_0;$ % the `then` branch: $l_c \leftarrow 1$
$\texttt{lower } l_3$ % restore security context and go to l_3

$l_3 \mapsto$ $\texttt{code}[\circ]\langle\bullet\rangle\{r_1 : \texttt{int}_\bot, \texttt{sp} : nil\}.$
$\texttt{mov } r_0, 1;$
$\texttt{st } r_1(0), r_0;$ % $l_a \leftarrow 1$
$\texttt{halt } [\texttt{int}_\bot]$ }

Fig. 8. TAL$_C$ example

control is then transferred to the code labeled l_1, which contains a test on the high variable `b` and directs the execution to two separate branches. In either branch of the conditional, the high variable `c` is updated, and the security context is restored with `lower` l_3. The code at l_3 is then free to update the low variable `a` again.

A closer look at the code labeled l_1 reveals several interesting issues. When checking the first load instruction (`ld` $r_4, r_2(0)$), the security level for r_4 is inferred to be high ($\top$). The following branching instruction (`bnz` r_4, l_2) type-checks because the current security context ($\top \triangleright l_3$) is high enough to cover the security level of r_4. The next store instruction (`st` $r_3(0), r_0$) is also valid, because it is ok to update a high variable in a high context. In comparison, the store instruction would fail to type-check if `c` was a low variable. Finally, the high security context is ended with a lower instruction (`lower` l_3) that directs the control flow to the postdominator of the conditional.

5 Discussions

Linear Continuations. Zdancewic and Myers [24] introduced a notion of ordered linear continuations to facilitate the information-flow analysis at a low level (we use ZM to refer to their system). An important requirement of such analysis is that one needs to allow a high-security conditional to be surrounded by low-security computation. In ZM, before the conditional statement, a linear continuation is created to capture the

computation after the conditional. Such a linear continuation must be called exactly once at the end of either branch of the conditional. Furthermore, the linear continuation records the security context in which it is created, allowing the security context to be reset properly when the branches meet.

As a higher-order analog to postdominators in a control-flow graph, ordered linear continuations enforce a stack discipline that allows security contexts to be reset at the join points of program branches. The static semantics ensures that the linear continuations are properly nested, and at any time only the top continuation on the (virtual) continuation stack is available. The linearity is enforced because the continuation is essentially popped off the stack when used. In particular, every value in ZM is tagged with a security label. The operational semantics keeps track of the security context during the execution, and ensures that security labels of the values are propagated correctly.

It may help to view our solution as an adaptation of linear continuations for the RISC architecture. A postdominator of program branches is essentially expressed as a static code label. The security operations `raise` and `lower` correspond to the creation and elimination of linear continuations. At any program point, our static semantics keeps track of only the top element of the (virtual) continuation stack. The typing rule for `raise` ensures that the security context at the postdominator matches the current one, thus enforcing the stack discipline.

We wish to point out, nonetheless, that such an adaptation yields a simple, practical and well-grounded solution to the identified problem of information-flow analysis for assembly code. In particular, it bridges the gap between the functional abstraction of linear continuations and the raw assembly code running on actual machines. In comparison with ZM, our system TAL_C models the use of registers and assembly instructions, and hence is closer to the actual RISC architecture. We do not attach security labels to values; this makes it trivial to see that security annotations do not affect computation. In fact, the enforcement of noninterference in TAL_C is cleanly separated from normal program execution. It is also obvious that security operations in TAL_C are orthogonal from conventional instructions (*e.g.,* branching and jumping) and mechanisms (*e.g.,* call stack), which allows our approach to be carried further with other language extensions. Consequently, we consider TAL_C as a good first step toward a scaled-up typed assembly language for noninterference.

Translating Linear Continuations. It may appear that TAL_C is not as expressive as the language of Zdancewic and Myers' [24] (ZM), because the security context of TAL_C uses static labels. Nonetheless, these static labels are only used to refer to code (e.g., that of linear continuations in ZM) whose locations can be statically determined. Indeed, their source level counterparts are the ending points of conditional structures, which are always statically known. Therefore, there is not a loss of expressiveness. We demonstrate this by speculating a translation from ZM to TAL_C.

In ZM, there are two expressions manipulating linear continuations: creation and elimination. The creation of a linear continuation essentially has the form $\texttt{letlin}\ y = \lambda\langle\texttt{pc}\rangle(x{:}\sigma).e\ \texttt{in}\ e'$. A corresponding elimination has the form $\texttt{lgoto}\ y\ v$.

The translation can be carried out following Morrisett *et al.* [11]. The step of CPS conversion is not needed because ZM is already in CPS. During closure conversion, the abstraction $\lambda\langle\texttt{pc}\rangle(x:\sigma).e$ (which corresponds to the code at a postdominator) will be

assigned a static code label. This code label is exactly the static postdominator needed for raising the security context in TAL$_C$. In a formal translation, this label can be used to generate a `raise` instruction when a corresponding branching point is reached. The typing (in particular, the security labels) of a ZM program is sufficient for identifying the branching point.

The elimination of linear continuation (`lgoto` y v) is relatively straightforward. Suppose the code of y (the lambda abstraction) declared using `letlin` is assigned the heap label l_y during closure conversion, the elimination expression `lgoto` y v can be translated as a `lower` l_y preceded with appropriate code computing the argument v.

For better understanding the relationship between linear continuations in ZM and security contexts in TAL$_C$, we further look into an example of nested `letlin` declarations: `letlin` $y_1 = lv_1$ `in letlin` $y_2 = lv_2$ `in` e. Once the second `letlin` is declared, the first linear continuation y_1 should be accessible only from inside lv_2. Therefore, ZM requires that e type checks under y_2, and lv_2 type checks under y_1. This essentially enforces a stack discipline.

TAL$_C$ has a similar mechanism. Suppose the current security context is $\theta_1 \triangleright l_1$ and the current instruction sequence is `raise` $\theta_2 \triangleright l_2; I$. The type system of TAL$_C$ checks I under $\theta_2 \triangleright l_2$, and checks that the code type at l_2 respects $\theta_1 \triangleright l_1$. This enforces a similar stack discipline as in ZM; note that only the top stack element is apparent at any time.

SIF. SIF [9] is developed independently from TAL$_C$. These two systems are similar in spirit—both use static types for information-flow analysis. However, SIF is based on a minimal language where relatively simple annotations, namely a stack of static code labels, suffice. In a more realistic language, a single function (even if monomorphic with respect to security levels) can be called at different program points. The security contexts of these program points may be different with respect to (1) the postdominator of the current context (SIF tracks this with the top stack element), and (2) the "enclosing contexts" (SIF tracks these with the stack tail). Since the label stack of SIF is made up of static code labels, one cannot reuse the same code at different program points with different contexts.

TAL$_C$ only maintains the current security context at any program point, and we show that it suffices for establishing noninterference. With such a treatment, the code types are naturally polymorphic with respect to enclosing contexts. We also allow postdominators to be polymorphic. The certifying compilation scheme in the technical report [23] further demonstrates the expressiveness of TAL$_C$.

6 Extensions and Future Work

Orthogonal Features. For ease of understanding, TAL$_C$ focuses on a minimal set of language features. Nonetheless, polymorphic and existential types, as seen in TAL, are orthogonal and can be introduced with little difficulty. Furthermore, since TAL$_C$ is compatible with TAL, it is also possible to accommodate other features of the TAL family. For instance, alias types [17] may provide a more accurate alias analysis, improving the current conservative approach that considers every pointer as a potential alias. In the following, we will also discuss the use of singleton types [22].

Security Polymorphism. TAL_C relies on a security context $\theta \triangleright w$ to identify the current security level θ and its ending point w. It is monomorphic with respect to security, because the security level of a code block is fixed. In practice, security-polymorphic code can also be useful.

```
int double(x:int) { x=x*2; } ...
if h<10 then double(h);
double(l); ...
```

Fig. 9. Security-polymorphic function

Figure 9 gives an example. The function `double` can be invoked with either low or high input. It is safe to invoke `double` in a context if only the security level of the input matches that of the context. In a security polymorphic TAL_C-like type system, `double` can be given the type $(\forall[\theta,\alpha].\langle\theta \triangleright \alpha\rangle\{r_1 : \texttt{int}_\theta, r_0 : (\forall[].\langle\theta \triangleright \alpha\rangle\{r_1 : \texttt{int}_\theta\})_\bot\})_\bot$. Here r_1 is the argument register, r_0 stores the return pointer, and the meta-variable θ is reused as a variable.

It is straightforward to support this kind of polymorphism. In fact, most of the required constructs are already present in TAL_C. We omitted such polymorphism simply because it complicates the presentation without providing additional insights. Nonetheless, the expressiveness of such polymorphism is still limited. Since the label α is not known until instantiated, the code of `double` has no knowledge about α. Hence the security context $\theta \triangleright \alpha$ cannot be discharged within the body of `double`.

It is not obvious why one would wish to discharge the security context within a polymorphic function. Indeed, it is always possible to wrap a function call inside a secured region by symmetric `raise` and `lower` operations from the caller's side. However, the asymmetric discharging of security context may be desirable for *certifying optimization*. For instance, in Figure 9, `double` is called as the last statement of the body of a high conditional. In this case, directly discharging the security context when `double` returns would remove a superfluous `lower` from the caller's side. Such a discharging requires `lower` to operate on small values—since the return label is not statically fixed, it must be passed in through a register.

It may require singleton and intersection types to support such a `lower` operation. For example, a `double` function that discharges its security context can have type

$$\left(\forall[\theta,\alpha].\langle\theta \triangleright \alpha\rangle \left\{\begin{array}{l} r_1 : \texttt{int}_\theta, \\ r_0 : \texttt{sint}(\alpha)_\bot \wedge (\forall[].\langle\bullet\rangle\{r_1 : \texttt{int}_\theta\})_\bot \end{array}\right\}\right)_\bot .$$

At the end of the function, `lower` r_0 discharges the security context and transfers the control to the return code. For type checking, the singleton integer type $\texttt{sint}(\alpha)$ matches the register r_0 with the label in the security context, and the code type ensures that the control flow to the return point is safe.

Full Erasure. With the powerful type constructs above, one can achieve a full erasure for the `lower` operation. Instead of treating `lower` as an instruction, one can treat it as a transformation on small values. This is in spirit similar to the *pack* operation of

existential types in TAL. Such a `lower` transformation bridges the gap between the current security context and the security level of the target label. The actual control flow transfer is then completed with a conventional jump instruction (*e.g.,* `jmp` (`lower` r_0)).

One can also achieve a full erasure for `lower` even without singleton types. The idea is to separate the jump instruction into direct jump and indirect jump. This is also consistent with real machine architectures. The `lower` operation transforms word values (eventually, direct labels). Lowered labels, similar to packed values, may serve as the operand of direct jump. Indirect jump, on the other hand, takes normal small values. This is expressive enough for certifying compilation, yet may not be sufficient for certifying optimization as discussed above.

7 Conclusion

We have presented a language TAL_C for enforcing data confidentiality in assembly code. The main idea is to use type annotations to restore high-level abstractions that are crucial to information-flow analysis. In TAL_C, operations related to security are kept orthogonal from other language features. As a result, it is possible to accommodate existing results on low-level verification, such as the TAL family. Our technical report presents a translation from a high-level security language with first-order procedures to TAL_C. A soundness theorem shows that the translation preserves security types. We consider this as a useful step toward a certifying compiler for noninterference.

Acknowledgments. We thank Eduardo Bonelli, Adriana Compagnoni, Ricardo Medel, Greg Morrisett, Steve Zdancewic and the anonymous referees for helpful comments on previous drafts.

References

1. M. Abadi, A. Banerjee, H. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Prin. of Prog. Lang.*, pages 147–160, San Antonio, TX, Jan. 1999.
2. J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.
3. J. Agat. *Type Based Techniques for Covert Channel Elimination and Register Allocation*. PhD thesis, Chalmers Univ. of Tech. and Gothenburg Univ., Gothenburg, Sweden, Dec. 2000.
4. T. Ball. What's in a region? Or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Prog. Lang. and Syst.*, 2(1-4):1–16, Mar.-Dec. 1993.
5. A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proc. 15th IEEE CSFW Workshop*, pages 253–267, June 2002.
6. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proc. 5th International Conf. on VMCAI*, volume 2937 of *LNCS*, pages 2–15, Venice, Italy, Jan. 2004.
7. E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. Technical report, Stevens Inst. of Tech., Hoboken, NJ, July 2004.
8. N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In *Proc. 25th ACM Symp. on Prin. of Prog. Lang.*, pages 365–377, San Diego, CA, Jan. 1998.
9. R. Medel, A. Compagnoni, and E. Bonelli. Non-interference for a typed assembly language. In *Proc. 2005 Workshop on Foundations of Computer Security*, Chicago, IL, June 2005.

10. G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, Jan. 2002.
11. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, Nov. 1999.
12. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Prin. of Prog. Lang.*, pages 228–241, San Antonio, TX, 1999.
13. F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003.
14. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
15. J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc. of the IEEE*, 63(9), Sept. 1975.
16. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101, 2001.
17. F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. 9th European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381, Berlin, Germany, Apr. 2000.
18. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Prin. of Prog. Lang.*, pages 355–364, San Diego, CA, Jan. 1998.
19. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *10th IEEE Computer Security Foundations Workshop*, pages 156–169, Washington, DC, June 1997.
20. D. Volpano and G. Smith. A type-based approach to program security. In *Proc. 7th Inter. Joint Conf. CAAP/FASE TAPSOFT*, LNCS, pages 607–621, Lille, France, Apr. 1997.
21. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. 11th IEEE CSFW Workshop*, pages 34–43, Washington, DC, June 1998.
22. H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 6th ACM International Conference on Functional Programming*, pages 169–180, Florence, Italy, Sept. 2001.
23. D. Yu and N. Islam. A typed assembly language for confidentiality. Technical Report DCL-TR-2005-0002, DoCoMo Communications Laboratories USA, San Jose, CA, Mar. 2005. `http://www.docomolabsresearchers-usa.com/~dyu/talc-tr.pdf`.
24. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002.