# A Typed Assembly Language for Non-Interference

Ricardo Medel[1], Adriana Compagnoni[1], and Eduardo Bonelli[2]

[1] Stevens Institute of Technology, Hoboken NJ 07030, USA,
[2] LIFIA, Fac. de Informática, Univ. Nac. de La Plata, Argentina
{rmedel,abc}@cs.stevens.edu
eduardo@sol.info.unlp.edu.ar

**Abstract.** Non-interference is a desirable property of systems in a multilevel security architecture, stating that confidential information is not disclosed in public output. The challenge of studying information flow for assembly languages is that the control flow constructs that guide the analysis in high-level languages are not present. To address this problem, we define a typed assembly language that uses pseudo-instructions to impose a stack discipline on the control flow of programs. We develop a type system for checking that assembly programs enjoy non-interference and its proof of soundness.

## 1 Introduction

The confidentiality of information handled by computing systems is of paramount importance. However, standard perimeter security mechanisms such as access control or digital signatures fail to address the enforcement of information-flow policies. On the other hand, language-based strategies offer a promising approach to information flow security. In this paper, we study confidentiality for an assembly language using a language-based approach to security via type-theory.

In a multilevel security architecture information can range from having low (public) to high (confidential) security level. Information flow analysis studies whether an attacker can obtain information about the confidential data by observing the output of the system. The non-interference property states that any two executions of the same program, where only the high-level inputs differ in both executions, does not exhibit any observable difference in the program's output.

In this paper we define SIF, a typed assembly language for secure information flow analysis with security types. This language contains two pseudo-instructions, `cpush` $L$ and `cjmp` $L$, for handling a stack of code labels indicating the program points where different branches of code converge, and the type system enforces a stack policy on those code labels. Our development culminates with a proof that well-typed SIF programs are assembled to untyped machine code that satisfy non-interference.

The type system of SIF detects explicit illegal flows as well as implicit illegal flows arising from the control structure of a program. Other covert channels such as those based on termination, timing, and power consumption, are outside the scope of this paper.

## 2 SIF, A Typed Assembly Language

In information fbw analysis, a security level is associated with the program counter (`pc`) at each program execution point. This security level is used to detect implicit information fbw from high-level values to low-level values. Moreover, control fbw analysis is crucial in allowing this security level to decrease where there is no risk of illicit fbw of information.

Consider the example in Figure 1(a), where `x` has high security level and `z` has low security level. Notice that `y` cannot have low security level, since information about `x` can be retrieved from `y`, violating the non-interference property. Since the execution path depends on the value stored in the high-security variable `x`, entering the branches of the `if-then-else` changes the security level of the `pc` to high, indicating that only high-level variables can be updated. On the other hand, since `z` is modified after both branches, there is no leaking of information from either `y` or `x` to `z`. Therefore, the security level of the `pc` can be safely lowered.

| Sec. level of `pc` | |
|---|---|
| *low* | `if x=0` |
| *high* | `   then y:=1` |
| *high* | `   else y:=2` |
| *low* | `z:=3` |

(a) High-level program

$L1$ : `bnz` $r_1, L2$ % `if x≠0 goto` $L2$
`move` $r_2 \leftarrow 1$ % `y:= 1`
`jmp` $L3$
$L2$ : `move` $r_2 \leftarrow 2$ % `y:= 2`
$L3$ : `move` $r_3 \leftarrow 3$ % `z:= 3`

(b) Assembly program

**Fig. 1.** Example of implicit illegal information flow.

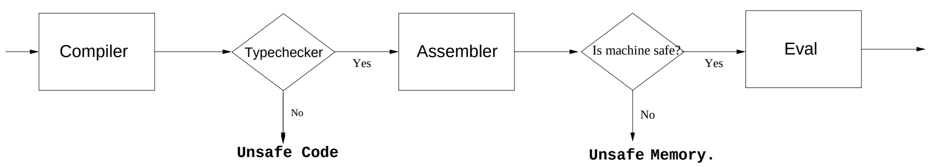
A standard compilation of this example to assembly language may produce the code shown in Figure 1(b). Note that the block structure of the `if-then-else` is lost, and it is not clear where it is safe to lower the security level of the `pc`. We address this problem by including in our assembly language a stack of code labels accessed by two pseudo-instructions, `cpush` $L$ and `cjmp` $L$, to simulate the block structure of high-level languages.

The instruction `cpush` $L$ pushes $L$ onto the stack while `cjmp` $L$ first pops $L$ from the stack if $L$ is already at the top, and then jumps to the instruction labelled by $L$. The extra label information in `cjmp` $L$ allows us to statically control that the intended label is removed, thereby preventing ill structured code.

The SIF code for the example in Figure 1(a) is shown below. The code at $L1$ pushes the label $L3$ onto the stack. The code at $L3$ corresponds to the instructions following the `if-then-else` in the source code. Observe that the code at $L3$ can only be executed once, because the instruction `cjmp` $L3$ at the end of the code pointed to by $L1$ (`then` branch), or at the end of $L2$ (`else` branch), removes the top of the stack and jumps to the code pointed to by $L3$. At this point it is safe to lower the security level of the `pc`, since updating the low-security register $r_3$ does not leak any information about $r_1$.

```
L1 : {r0 : int^⊥, r1 : int^⊤, r2 : int^⊤, r3 : int^⊥, pc : ⊥} || ε
    cpush L3                    % set junction point L3
    bnz r1, L2                  % if x≠0 goto L2
    arithi r2 ← r0 + 1          % y:=1, with r0=0
    cjmp L3
L2 : {r0 : int^⊥, r2 : int^⊤, r3 : int^⊥, pc : ⊤} || L3 · ε
    arithi r2 ← r0 + 2          % y:=2
    cjmp L3
L3 : {r0 : int^⊥, r3 : int^⊥, pc : ⊥} || ε
    arithi r3 ← r0 + 3          % z:=3
    halt
    eof
```

Moreover, as in HBAL [1], the type-checking of the program is separated from the verification of the safety of the machine configuration where the program is assembled. Thus, following the schema shown below, a type-checker can verify if a program is safe for execution on *any safe* memory configuration, and the runtime environment only needs to check that the initial machine configuration is safe before each run.




The assembler removes `cpush` $L$ and translates `cjmp` $L$ into `jmp` $L$, an ordinary unconditional jump, leaving no trace of these pseudo-instructions in the executable code (See the definition of the assembly function $\mathsf{Asm}(-)$ in section 2.4).

### 2.1 The Type System

We assume given a lattice $\mathfrak{L}_{\textbf{sec}}$ of *security labels* [8], with an ordering relation $\sqsubseteq$, least ($\bot$) and greatest ($\top$) elements, and join ($\sqcup$) and meet ($\sqcap$) operations. These labels assign security levels to elements of the language through types. The type expressions of SIF are given by the following grammar:

$$
\begin{array}{ll}
\textit{security labels} & l \ \in \ \mathfrak{L}_{\textbf{sec}} \\
\textit{security types} & \sigma ::= \omega^{l} \\
\textit{word types} & \omega ::= int \quad | \quad [\tau] \\
\textit{memory location types} & \tau ::= \sigma \times \ldots \times \sigma \quad | \quad \texttt{code}
\end{array}
$$

*Security types* ($\sigma$) are word types annotated with a security label. The expression $\textsc{Labl}(\sigma)$ returns the security label of a security type $\sigma$. A *word type* ($\omega$) is either an integer type ($int$) or a pointer to a memory location type ($[\tau]$). *Memory location types* ($\tau$) are tuples of security types, or a special type `code`. We use $\tau[c]$, with $c$ a positive integer, to refer to the $c^{th}$ word type of the product type $\tau$. Since the type `code` indicates the type of an assembly instruction, our system distinguishes code from data.

A *context* $(\Gamma \parallel \Lambda)$ contains a register context $\Gamma$ and a junction points stack $\Lambda$. A *junction points stack* $(\Lambda)$ is a stack of code labels, each representing the convergence point of a fork in the control flow of a program. The empty stack is denoted by $\epsilon$. A *register context* $\Gamma$ contains type information about registers, mapping them to security types. We assume a finite set of registers $\{r_0, \ldots, r_n\}$, with two dedicated registers: $r_0$, that always holds zero, and pc, the program counter.

We write $Dom(\Gamma)$ for the domain of the register context $\Gamma$. The empty context is denoted by $\{\}$. The register context obtained by eliminating from $\Gamma$ all pairs with $r$ as first component be denoted by $\Gamma_{/r}$, while $\Gamma, \Gamma'$ denotes the union of register contexts with disjoint domains. We use $\Gamma, r : \sigma$ as a shorthand for $\Gamma, \{r : \sigma\}$, and $\Gamma[r := \sigma]$ as a shorthand for $\Gamma_{/r}, \{r : \sigma\}$.

Since the program counter is always a pointer to code, we usually write $\mathsf{pc} : l$ instead of $\mathsf{pc} : [\mathtt{code}]^l$, and $\Gamma(\mathsf{pc}) = l$ if $\mathsf{pc} : l \in \Gamma$.

### 2.2 Syntax of SIF Programs

A *program* $(P)$ is a sequence of instructions and code labels ended by the directive `eof`. SIF has standard assembly language instructions such as arithmetic operations, conditional branching, load, and store, plus pseudo-instructions `cpush` and `cjmp` to handle the stack of code labels.

$$
\begin{array}{lll}
\mathit{program} & P ::= & \mathtt{eof} \;\mid\; L; P \;\mid\; p; P \\
\mathit{instructions} & p ::= & \mathtt{halt} \;\mid\; \mathtt{jmp}\ L \;\mid\; \mathtt{bnz}\ r, L \\
 & & \mid\; \mathtt{load}\ r \leftarrow r[c] \;\mid\; \mathtt{store}\ r[c] \leftarrow r \\
 & & \mid\; \mathtt{arith}\ r \leftarrow r \odot r \;\mid\; \mathtt{arithi}\ r \leftarrow r \odot i \\
 & & \mid\; \mathtt{cpush}\ L \;\mid\; \mathtt{cjmp}\ L \\
\mathit{operations} & \odot ::= & + \;\mid\; - \;\mid\; * \;\mid\; /
\end{array}
$$

We use $c$ to indicate an offset, and $i$ to indicate integer literals. We assume an infinite enumerable set of code labels. Intuitively, the instruction `cpush` $L$ pushes the junction point represented by the code label $L$ onto the stack, while the instruction `cjmp` $L$ behaves as a pop and a jump. If $L$ is at the top of the stack, it pops $L$ and then jumps to the instruction labeled $L$.

### 2.3 Typing Rules

A *signature* $(\Sigma)$ is a mapping assigning contexts to labels. The context $\Sigma(L)$ contains the typing assumptions for the registers in the program point pointed to by the label $L$. The judgment $\Gamma \parallel \Lambda \vdash_\Sigma P$ is a typing judgment for a SIF program $P$, with signature $\Sigma$, in a context $\Gamma \parallel \Lambda$. We say that a program $P$ is *well-typed* if $\mathsf{Ctxt}(P) \vdash_\Sigma P$, where $\mathsf{Ctxt}(P)$ is the partial function defined as: $\mathsf{Ctxt}(L; P) = \Sigma(L)$, $\mathsf{Ctxt}(\mathtt{eof}) = \{\} \parallel \epsilon$.

The typing rules for SIF programs, shown in Figures 2 and 3, are designed to prevent illegal flows of information. The directive `eof` is treated as a `halt` instruction. So, rules T_Eof and T_Halt ensure that the stack is empty.

$$\frac{\Gamma' \subseteq \Gamma \quad l \sqsubseteq l'}{(\Gamma, \mathsf{pc} : l \parallel \Lambda) \leq (\Gamma', \mathsf{pc} : l' \parallel \Lambda)}\ \mathsf{ST\_RegBank}$$

$$\frac{\mathsf{Ctxt}(P)\ \vdash_{\Sigma}\ P}{\Gamma \parallel \epsilon\ \vdash_{\Sigma}\ \mathtt{halt}\,; P}\ \mathsf{T\_Halt} \qquad \frac{}{\Gamma \parallel \epsilon\ \vdash_{\Sigma}\ \mathtt{eof}}\ \mathsf{T\_Eof}$$

$$\frac{(\Gamma \parallel \Lambda) \leq \Sigma(L) \quad \Sigma(L)\ \vdash_{\Sigma}\ P}{\Gamma \parallel \Lambda\ \vdash_{\Sigma}\ L; P}\ \mathsf{T\_Label}$$

$$\frac{(\Gamma \parallel \Lambda) \leq \Sigma(L) \quad \mathsf{Ctxt}(P)\ \vdash_{\Sigma}\ P}{\Gamma \parallel \Lambda\ \vdash_{\Sigma}\ \mathtt{jmp}\ L; P}\ \mathsf{T\_Jmp}$$

$$\frac{(\Gamma, r : int^{l'}, \mathsf{pc} : l \sqcup l' \parallel \Lambda) \leq \Sigma(L) \quad \Gamma, r : int^{l'}, \mathsf{pc} : l \sqcup l' \parallel \Lambda\ \vdash_{\Sigma}\ P}{\Gamma, r : int^{l'}, \mathsf{pc} : l \parallel \Lambda\ \vdash_{\Sigma}\ \mathtt{bnz}\ r, L; P}\ \mathsf{T\_CondBrnch}$$

**Fig. 2.** Subtyping for contexts and typing rules for programs ($1^{st}$ part).

Rule T_Label requires that the current context be compatible with the context expected at the position of the label, as defined in the signature ($\Sigma$) of the program. Jumps and conditional jumps are typed by rules T_Jmp and T_CondBrnch. In both rules the current context has to be compatible with the context expected at the destination code. In T_CondBrnch, both the code pointed to by $L$ and the remaining program $P$ are considered destinations of the jump included in this operation. In order to avoid implicit flows of information, the security level of the pc in the destination code should not be lower than the current security level and the security level of the register ($r$) that controls the branching.

In T_Arith the security level of the source registers and the pc should not exceed the security level of the target register to avoid explicit flows of information. The security level of $r_d$ can actually be lowered to reflect its new contents, but, to avoid implicit information flows, it cannot be lowered beyond the level of the pc. Similarly for T Arithi, T_Load and T_Store. In T_Load, an additional condition establishes that the security level of the pointer to the heap has to be lower than or equal to the security level of the word to be read.

The rule T_Cpush controls whether `cpush` $L$ can add the code label $L$ to the stack. Since $L$ is going to be consumed by a `cjmp` $L$ instruction, its security level should not be lower than the current level of the pc. The `cjmp` $L$ instruction jumps to the junction point pointed to by label $L$. Furthermore, to prevent ill structured programs the rule T_Cjmp forces the code label $L$ to be at the top of the stack, and the current context has to be compatible with the one expected at the destination code. However, since a `cjmp` instruction allows the security level to be lowered, there are no conditions on its security level.

$$\frac{\begin{array}{ll}\Gamma(r_d)=\omega^{l_d} & r_d,r_s,r_t\neq \mathsf{pc}\\ \Gamma(r_s)=int^{l_s} & l\sqcup l_s\sqcup l_t\sqsubseteq l_d\\ \Gamma(r_t)=int^{l_t} & \Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma P\end{array}}{\Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma \texttt{arith}\ r_d\leftarrow r_s\odot r_t;P}\ \mathsf{T\_Arith}$$

$$\frac{\begin{array}{ll} & r_d,r_s\neq \mathsf{pc}\\ \Gamma(r_d)=\omega^{l_d} & l\sqcup l_s\sqsubseteq l_d\\ \Gamma(r_s)=int^{l_s} & \Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma P\end{array}}{\Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma \texttt{arithi}\ r_d\leftarrow r_s\odot i;P}\ \mathsf{T\_Arithi}$$

$$\frac{\begin{array}{ll}\Gamma(r_s)=[\tau]^{l_s} & r_d,r_s\neq \mathsf{pc}\\ \Gamma(r_d)=\omega^{l_d} & l\sqcup l_s\sqsubseteq l_c\sqsubseteq l_d\\ \tau[c]=\omega_c^{l_c} & \Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma P\end{array}}{\Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma \texttt{load}\ r_d\leftarrow r_s[c];P}\ \mathsf{T\_Load}$$

$$\frac{\begin{array}{ll}\Gamma(r_d)=[\tau]^{l_d} & r_d,r_s\neq \mathsf{pc}\\ \Gamma(r_s)=\tau[c]=\omega^{l_s} & l\sqcup l_d\sqsubseteq l_s\\ \tau\text{ is }\texttt{code}\text{-free} & \Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma P\end{array}}{\Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma \texttt{store}\ r_d[c]\leftarrow r_s;P}\ \mathsf{T\_Store}$$

$$\frac{l\sqsubseteq\Sigma(L)(\mathsf{pc})\quad \Gamma,\mathsf{pc}:l\parallel L\cdot\Lambda\vdash_\Sigma P}{\Gamma,\mathsf{pc}:l\parallel\Lambda\vdash_\Sigma \texttt{cpush}\ L;P}\ \mathsf{T\_Cpush}$$

$$\frac{\Sigma(L)=\Gamma'\parallel\Lambda\quad {\Gamma'}_{/\mathsf{pc}}\subseteq\Gamma_{/\mathsf{pc}}\quad \mathsf{Ctxt}(P)\vdash_\Sigma P}{\Gamma\parallel L\cdot\Lambda\vdash_\Sigma \texttt{cjmp}\ L;P}\ \mathsf{T\_Cjmp}$$

**Fig. 3.** Typing rules for programs ($2^{nd}$ part).

### 2.4 Type Soundness of SIF

In this section we define a semantics for the untyped assembly instructions operating on a machine model, we give an interpretation for SIF types which captures the way types are implemented in memory, and finally we prove that the execution of a well-typed SIF program modifies a type-safe configuration into another type-safe configuration.

Let $\mathsf{Reg}=\{0,1,\ldots,\mathsf{R}_{\max}\}$ be the register indices, with two dedicated registers: $R(0)=0$, and $R(\mathsf{pc})$ is the program counter. Let $\mathsf{Loc}\subseteq\mathbf{Z}$ be the set of memory locations on our machine, Wrd be the set of machine words that can stand for integers or locations, and Code be the set of machine words which can stand for machine instructions. To simplify the presentation, we assume that Wrd is disjoint from Code; so, our model keeps code separate from data.

A *machine configuration* $M$ is a pair $(H,R)$ where $H:\mathsf{Loc}\rightharpoonup\mathsf{Wrd}\uplus\mathsf{Code}$ is a partial function defining a heap configuration, and $R:\mathsf{Reg}\rightarrow\mathsf{Wrd}$ is a register configuration.

Given a program $P$, a *machine assembled for* $P$ is a machine configuration which contains a representation of the assembly program, with machine instructions stored in some designated contiguous portion of the heap. Supposing $P = p_1; \ldots ; p_n$, the assembly process defines a function $\mathsf{PAdr} : 1, \ldots , n \rightarrow \mathsf{Loc}$ which gives the destination location for the code when assembling the typed instruction $p_u$, where $1 \leq u \leq n$. For each of the locations $\ell$ where $P$ is stored, $H(\ell) \in \mathsf{Code}$. The assembly process also defines the function $\mathsf{LAdr}(L)$, which assigns to each label in $P$ the heap location where the code pointed to by the label was assembled.

Given a machine configuration $M = (H, R)$, we define a *machine transition* relation $M \longrightarrow M'$, as follows: First, $M'$ differs from $M$ by incrementing $R(\mathsf{pc})$ according to the length of the instruction in $H(R(\mathsf{pc}))$; then, the transformation given in the table below is applied to obtain the new heap $H'$, or register bank $R'$. The operations on $r_0$ have no effect.

$$\begin{array}{ll}
\texttt{jmp}\ L & R' = R[\mathsf{pc} := \mathsf{LAdr}(L)] \\
\texttt{bnz}\ r, L & R' = \begin{cases} R, \text{ if } R(r) = 0 \\ R[\mathsf{pc} := \mathsf{LAdr}(L)], \text{ otherwise} \end{cases} \\
\texttt{arith}\ r_d \leftarrow r_s \odot r_t & R' = R[r_d := R(r_s) \odot R(r_t)] \\
\texttt{arithi}\ r_d \leftarrow r_s \odot i & R' = R[r_d := R(r_s) \odot i] \\
\texttt{load}\ r_d \leftarrow r_s[c] & R' = R[r_d := H(R(r_s) + c)] \\
\texttt{store}\ r_d[c] \leftarrow r_s & H' = H[R(r_d) + c := R(r_s)]
\end{array}$$

$\mathsf{Asm}(p_u)$ stands for the sequence of untyped machine instructions which is the result of assembling a typed assembly instruction $p_u$:

$$\begin{array}{ll}
\mathsf{Asm}(L) = \epsilon & \mathsf{Asm}(\texttt{eof}\,) = \texttt{halt} \\
\mathsf{Asm}(\texttt{cpush}\ L) = \epsilon & \mathsf{Asm}(\texttt{cjmp}\ L) = \texttt{jmp}\ L \\
\mathsf{Asm}(p_u) = p_u, \text{ otherwise} &
\end{array}$$

Notice that the sequence has at most one instruction. We write $M \overset{\mathsf{Asm}(p_u)}{\longrightarrow} M'$, if $M$ executes to $M'$ through the instructions in $\mathsf{Asm}(p_u)$, by zero or one transitions in $M$. The reflexive and transitive closure of this relation is defined by the following rules.

$$\frac{}{M \Longrightarrow M}\ \mathsf{Refl} \qquad \frac{M_1 \overset{\mathsf{Asm}(p_u)}{\longrightarrow} M_2}{M_1 \Longrightarrow M_2}\ \mathsf{Incl} \qquad \frac{M_1 \Longrightarrow M_2 \quad M_2 \Longrightarrow M_3}{M_1 \Longrightarrow M_3}\ \mathsf{Trans}$$

### 2.5 Imposing Types on the Model

A *heap context* $\psi$ is a function that maps heap locations to security types. A heap context contains type information about the heap locations required to type the registers. $Dom(\psi)$ denotes the domain of the heap context $\psi$. The empty context is denoted by $\{\}$. We write $\psi[\ell := \tau]$ for the heap context resulting from updating $\psi$ with $\ell : \tau$. Two heap contexts $\psi$ and $\psi'$ are *compatible*, denoted $\mathsf{compat}(\psi, \psi')$, if for all $\ell \in Dom(\psi) \cap Dom(\psi')$, $\psi(\ell) = \psi'(\ell)$. The following rules assign types to heap locations:

$$\frac{H(\ell) \in \mathsf{Code}}{H; \{\ell : \mathtt{code}\} \models \ell : \mathtt{code}\ \mathsf{hloc}}\ \mathsf{T\_HLocCode} \qquad \frac{H(\ell) \in \mathsf{Wrd}}{H; \{\ell : int^l\} \models \ell : int^l\ \mathsf{hloc}}\ \mathsf{T\_HLocInt}$$

$$\frac{H(\ell) \in \mathsf{Wrd} \quad \mathsf{compat}(\psi, \{\ell : [\tau]^l\}) \quad H; \psi \models H(\ell) : \tau\ \mathsf{hloc}}{H; \psi \cup \{\ell : [\tau]^l\} \models \ell : [\tau]^l\ \mathsf{hloc}}\ \mathsf{T\_HLocPtr}$$

$$\frac{\mathsf{compat}(\psi, \psi') \quad H; \psi \models \ell : \tau\ \mathsf{hloc}}{H; \psi \cup \psi' \models \ell : \tau\ \mathsf{hloc}}\ \mathsf{W\_HLoc}$$

$$\frac{\begin{array}{l} m_i = \mathit{size}(\sigma_0) + \ldots + \mathit{size}(\sigma_{i-1}) \\ H; \psi \models \ell + m_i : \sigma_i\ \mathsf{hloc} \qquad \text{for all } 0 \leq i \leq n \end{array}}{H; \psi \models \ell : \sigma_0 \times \ldots \times \sigma_n\ \mathsf{hloc}}\ \mathsf{T\_HLocProd}$$

In order to define the notion of satisfiability of contexts by machine configurations, we need to define a satisfiability relation for registers.

$$\frac{r \neq \mathsf{pc}}{M \models_{\{\}} r : int^l\ \mathsf{reg}}\ \mathsf{T\_RegInt} \qquad \frac{H; \psi \models R(r) : \tau\ \mathsf{hloc}}{(H, R) \models_\psi r : [\tau]^l\ \mathsf{reg}}\ \mathsf{T\_RegPtr}$$

$$\frac{(H, R) \models_\psi r : \sigma\ \mathsf{reg} \quad \mathsf{compat}(\psi, \psi')}{(H, R) \models_{\psi \cup \psi'} r : \sigma\ \mathsf{reg}}\ \mathsf{W\_Reg}$$

A machine configuration $M$ *satisfies* a typing assignment $\Gamma$ with a heap typing context $\psi$ (written $M \models_\psi \Gamma$) if and only if for each register $r_i \in Dom(\Gamma)$, $M$ satisfies the typing statement $M \models_{\psi_i} r_i : \Gamma(r_i)\ \mathsf{reg}$, the heap contexts $\psi_i$ are pairwise compatible, and $\psi = \cup_{\forall i} \psi_i$.

A machine configuration $M = (H, R)$ is in *final state* if $H(R(\mathsf{pc})) = \mathtt{halt}$. We define an approximation to the execution of a typed program $P = p_1; \ldots; p_n$ by relating the execution of the code locations in the machine $M$ with the control paths in the program by means of the relation $p_u \rightsquigarrow p_v$, which holds between pairs of instructions indexed by the set:

$$\begin{array}{l} \{(i, i+1) \mid p_i \neq \mathtt{jmp}, \mathtt{cjmp}, \text{ and } i < n\} \\ \cup \\ \{(i, j+1) \mid p_i = \mathtt{jmp}\ L, \mathtt{bnz}\ r, L, \text{ or } \mathtt{cjmp}\ L, \text{ and } p_j = L\}. \end{array}$$

We use $p_u \overset{*}{\rightsquigarrow} p_v$ to denote the reflexive and transitive closure of $p_u \rightsquigarrow p_v$.

### 2.6 Type Soundness

In this section we show that our type system ensures that the reduction rules preserve type safety. The soundness results imply that if the initial memory satisfies the initial typing assumptions of the program, then each memory configuration reachable from the initial memory satisfies the typing assumptions of its current instruction.

The typing assumptions of each instruction of a program can be obtained from the initial context by the typechecking process. The derivation $\mathsf{Ctxt}(P) \vdash_\Sigma P$

of a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$ determines a sequence of contexts $\Gamma_1 \parallel \Lambda_1, \ldots, \Gamma_n \parallel \Lambda_n$ from sub-derivations of the form $\Gamma_u \parallel \Lambda_u \vdash_\Sigma p_u; p_{u+1}; \ldots; p_n$.

A machine configuration is considered type-safe if it satisfies the typing assumptions of its current instruction. Given a well-typed program $P = p_1; \ldots p_u; \ldots; p_n$ and a heap context $\psi$, we say $M = (H, R)$ is *type safe at* $u$ *for* $P$ *with* $\psi$ if $M$ is assembled for $P$; $R(\mathsf{pc}) = \mathsf{PAdr}(u)$; and $M \models_\psi \Gamma_u$.

We prove two meta-theoretic results Progress and Subject Reduction. Progress (Theorem 1) establishes that a non-final-state type safe machine can always progress to a new machine by executing a well-typed instruction, and Subject Reduction (Theorem 2) establishes that if a type safe machine progresses to another machine, the resulting machine is also type safe.

**Theorem 1 (Progress).** *Suppose a well-typed program* $P = p_1; \ldots p_u; \ldots; p_n$ *and a machine configuration* $M$ *type safe at* $u$*. Then there exists* $M'$ *such that* $M \xrightarrow{\mathsf{Asm}(p_u)} M'$*, or* $M$ *is in* final state.

**Theorem 2 (Subject Reduction).** *Suppose* $P = p_1; \ldots p_u; \ldots; p_n$ *is a well-typed program and* $(H, R)$ *is a machine configuration type safe at* $u$*, and* $(H, R) \xrightarrow{\mathsf{Asm}(p_u)} M'$*. Then there exists* $p_v \in P$ *such that* $p_u \rightsquigarrow p_v$ *and* $M'$ *is type safe at* $v$*.*

The proof of this theorem proceeds by case analysis on the current instruction $p_u$, analyzing each of the possible instructions that follow $p_u$, based on the definition of program transitions. See the companion technical report [13] for details.

## 3 Non-Interference

Given an arbitrary (but fixed) security level $\zeta$ of an *observer*, non-interference states that computed low-security values ($\sqsubseteq \zeta$) should not be affected by high-security input values ($\not\sqsubseteq \zeta$). In order to prove that a program $P$ satisfies non-interference one must show that any two terminating executions fired from indistinguishable (from the point of view of the observer) machine configurations yield indistinguishable configurations of the same security observation level.

In order to establish what it means for machine configurations to be indistinguishable from an observer's point of view whose security level is $\zeta$, we define a $\zeta$-indistinguishability relation for machine configurations.

The following definitions assume a given security level $\zeta$, two machine configurations $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$, two heap contexts $\psi_1$ and $\psi_2$, and two register contexts $\Gamma_1$ and $\Gamma_2$, such that $M_1 \models_{\psi_1} \Gamma_1$ and $M_2 \models_{\psi_2} \Gamma_2$.

Two register banks are $\zeta$-indistinguishable if the observable registers in one bank are also observable in the other, and the contents of these registers are also $\zeta$-indistinguishable.

**Definition 1 ($\zeta$-indistinguishability of register banks).** *Two register banks $R_1$ and $R_2$ are $\zeta$-indistinguishable, written $\triangleright_{H_1:\psi_1,H_2:\psi_2} R_1 : \Gamma_1 \approx_\zeta R_2 : \Gamma_2$ $\mathsf{regBank}$, if for all $r \in Dom_\cup(\Gamma_1, \Gamma_2)$[3], with $r \neq \mathsf{pc}$:*

$$\textsc{Labl}(\Gamma_1(r)) \sqsubseteq \zeta \textit{ or } \textsc{Labl}(\Gamma_2(r)) \sqsubseteq \zeta \textit{ implies} \begin{cases} r \in Dom_\cap(R_1, R_2, \Gamma_1, \Gamma_2), \\ \Gamma_1(r) = \Gamma_2(r), \textit{ and} \\ \triangleright_{H_1:\psi_1,H_2:\psi_2} R_1(r) \approx_\zeta R_2(r) : \Gamma_1(r) \ \mathsf{val} \end{cases}$$

Two word values $v_1$ and $v_2$ of type $\omega^l$ are considered $\zeta$-indistinguishable, written $\triangleright_{H_1:\psi_1,H_2:\psi_2} v_1 \approx_\zeta v_2 : \omega^l$ $\mathsf{val}$, if $l \sqsubseteq \zeta$ implies that both values are equal. In case of pointers to heap locations, the locations have to be also $\zeta$-indistinguishable.

Two heap values $\ell_1$ and $\ell_2$ of type $\tau$ are considered $\zeta$-indistinguishable, written $\triangleright_{H_1:\psi_1,H_2:\psi_2} \ell_1 \approx_\zeta \ell_2 : \tau$ $\mathsf{hval}$, if $\ell_1 \in H_1$, $\ell_2 \in H_2$, and either the type $\tau$ is $\mathsf{code}$ and $\ell_1 = \ell_2$, or $\tau = \sigma_1 \times \ldots \times \sigma_n$ and each pair of offset locations $\ell_1 + m_i$ and $\ell_2 + m_i$ (with $m_i$ as in rule T_HLocProd) are $\zeta$-indistinguishable, or $\tau$ is a word type with a security label $l$ and $l \sqsubseteq \zeta$ implies that both values are equal.

$$\frac{}{\triangleright_\Sigma \epsilon \approx_\zeta \epsilon \ \mathsf{Low}}\ \mathsf{LowAxiom} \qquad \frac{\Sigma(L)(\mathsf{pc}) \sqsubseteq \zeta \quad \triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{Low}}{\triangleright_\Sigma L \cdot \Lambda_1 \approx_\zeta L \cdot \Lambda_2 \ \mathsf{Low}}\ \mathsf{LowLow}$$

$$\frac{\Sigma(L_1)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \Sigma(L_2)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{Low}}{\triangleright_\Sigma L_1 \cdot \Lambda_1 \approx_\zeta L_2 \cdot \Lambda_2 \ \mathsf{cstackLow}}\ \mathsf{LowHigh}$$

$$\frac{\triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{Low}}{\triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{High}}\ \mathsf{HighAxiom} \qquad \frac{\Sigma(L)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{High}}{\triangleright_\Sigma L \cdot \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{High}}\ \mathsf{HighLeft}$$

$$\frac{\Sigma(L)(\mathsf{pc}) \not\sqsubseteq \zeta \quad \triangleright_\Sigma \Lambda_1 \approx_\zeta \Lambda_2 \ \mathsf{High}}{\triangleright_\Sigma \Lambda_1 \approx_\zeta L \cdot \Lambda_2 \ \mathsf{High}}\ \mathsf{HighRight}$$

**Fig. 4.** $\zeta$-indistinguishability of junction points stacks.

The proof of our main result, the Non-Interference Theorem 3, requires two notions of indistinguishability of stacks (Low and High). If one execution of a program branches on a condition while the other does not, the junction points stacks may differ in each of the paths followed by the executions. If the security level of the pc is low in one execution, then it has to be low in the other execution as well, and the executions must be identical. The first three rules of Figure 4 define the relation of low-indistinguishability for stacks. In low-security executions the associated stacks mus be of the same size, and each code label in the stack of the first execution must be indistinguishable from that of the corresponding element in the second one.

If the security level of the pc of one of the two executions is high, then the other one must be high too. The executions are likely to be running different instructions, and

[3] We use $Dom_\oplus(A_1, \ldots, A_n)$ as an abbreviation for $Dom(A_1) \oplus \ldots \oplus Dom(A_n)$.

thus the associated stacks may have different sizes. However, we need to ensure that both executions follow branches of the same condition. This is done by requiring that both associated stacks have a common (low-indistinguishable) sub-stack. The second three rules of Figure 4 define the relation of high-indistinguishability for stacks. Also note that, as imposed by the typing rules, the code labels added to the stack associated to high-security branches are of high-security level.

Finally, we define the relation of indistinguishability of two machine con from the point of view of an observer of level $\zeta$.

**Definition 2.** *Two machine configurations $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ are* $\zeta$-indistinguishable*, denoted by the judgment*

$$\rhd_P M_1 : \Gamma_1, \Lambda_1, \psi_1 \approx_\zeta M_2 : \Gamma_2, \Lambda_2, \psi_2 \ \mathsf{mConfig},$$

*if and only if*

1. $M_1 \models_{\psi_1} \Gamma_1$ *and* $M_2 \models_{\psi_2} \Gamma_2$,
2. $M_1$ *and* $M_2$ *are assembled for* $P$ *at the same addresses,*
3. $\rhd_{H_1:\psi_1, H_2:\psi_2} R_1 : \Gamma_1 \approx_\zeta R_2 : \Gamma_2$ $\mathsf{regBank}$*, and*
4. *either*
   (a) $\Gamma_1(\mathsf{pc}) = \Gamma_2(\mathsf{pc}) \sqsubseteq \zeta$ *and* $R_1(\mathsf{pc}) = R_2(\mathsf{pc})$ *and* $\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2$ $\mathsf{Low}$*, or*
   (b) $\Gamma_1(\mathsf{pc}) \not\sqsubseteq \zeta$ *and* $\Gamma_2(\mathsf{pc}) \not\sqsubseteq \zeta$ *and* $\rhd_\Sigma \Lambda_1 \approx_\zeta \Lambda_2$ $\mathsf{High}$.

Note that both machine configurations must be consistent with their corresponding typing assignments, and they must be executing the code resulting from assembling $P$.

We may now state the non-interference theorem establishing that starting from two indistinguishable machine configurations assembled for the same program $P$, if each execution terminates, the resulting machine configurations remain indistinguishable.

In the following theorem and lemmas, for any instruction $p_i$ in a well-typed program $P = p_1; \ldots; p_n$, the context $\Gamma_i \parallel \Lambda_i$ is obtained from the judment $\Gamma_i \parallel \Lambda_i \vdash_\Sigma p_i; p_n$, which is derived by a sub-derivation of $\mathsf{Ctxt}(P) \vdash_\Sigma P$.

**Theorem 3 (Non-Interference).** *Let $P = p_1; \ldots; p_n$ be a well-typed program, $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations such that both are* type safe *at 1 for $P$ with $\psi$ and*

$$\rhd_P M_1 : \Gamma_1, \epsilon, \psi \approx_\zeta M_2 : \Gamma_1, \epsilon, \psi \ \mathsf{mConfig}.$$

*If $M_1 \Longrightarrow M_1'$ and $M_2 \Longrightarrow M_2'$, with $M_1'$ and $M_2'$ in* final state*, then*

$$\rhd_P M_1' : \Gamma_v, \epsilon, \psi_1 \approx_\zeta M_2' : \Gamma_w, \epsilon, \psi_2 \ \mathsf{mConfig}.$$

The technical challenge that lies in the proof of this theorem is that the $\zeta$-indistinguishability of configurations holds after each transition step. The proof is developed in two stages. First it is proved that two $\zeta$-indistinguishable configurations that have a low (and identical) level for the $\mathsf{pc}$ can reduce in a *lock step fashion* in a manner invariant to the $\zeta$-indistinguishability property. This is stated by the following lemma.

**Lemma 1 (Low-PC Step).** *Let $P = p_1; \dots; p_n$ be a well-typed program, such that $p_{v_1}$ and $p_{v_2}$ are in $P$, $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations. Suppose*

1. *$M_1$ is type safe at $v_1$ and $M_2$ is type safe at $v_2$, for $P$ with $\psi_1$ and $\psi_2$, respectively,*
2. *$\rhd_P M_1 : \Gamma_{v_1}, \Lambda_{v_1}, \psi_1 \approx_\zeta M_2 : \Gamma_{v_2}, \Lambda_{v_2}, \psi_2$ mConfig,*
3. *$\Gamma_{v_1}(\mathsf{pc}) \sqsubseteq \zeta$ and $\Gamma_{v_2}(\mathsf{pc}) \sqsubseteq \zeta$,*
4. *$M_1 \xrightarrow{\mathsf{Asm}(p_{v_1})} M_1'$, and*
5. *there exists $p_{w_1}$ in $P$ such that $p_{v_1} \rightsquigarrow p_{w_1}$, and $M_1'$ is type safe at $w_1$ with $\psi_3$.*

*Then, there exists a configuration $M_2'$ such that:*

$(a)$ $M_2 \xrightarrow{\mathsf{Asm}(p_{v_2})} M_2'$,
$(b)$ *there exists $p_{w_2}$ in $P$ such that $p_{v_2} \rightsquigarrow p_{w_2}$, and $M_2'$ is type safe at $w_2$ with $\psi_4$, and*
$(c)$ $\rhd_P M_1' : \Gamma_{w_1}, \Lambda_{w_2}, \psi_3 \approx_\zeta M_2' : \Gamma_{w_2}, \Lambda_{w_2}, \psi_4$ mConfig.

When the level of the pc is low, the programs execute the same instructions (with possibly different heap and register bank). They may be seen to be *synchronized* and each reduction step made by one is emulated with a reduction of the same instruction by the other. The resulting machines must be $\zeta$-indistinguishable.

However, a conditional branch (bnz) may cause the execution to fork on a high value. As a consequence, both of their pc become high and we must provide proof that there are some $\zeta$-indistinguishable machines to which they reduce. Then, the second stage of the proof consists of showing that every reduction step of one execution whose pc has a high-security level can be met with a number of reduction steps (possibly none) from the other execution such that they reach indistinguishable configurations. The High-PC Step Lemma states such result.

**Lemma 2 (High-PC Step).** *Let $P = p_1; \dots; p_n$ be a well-typed program, such that $p_{v_1}$ and $p_{v_2}$ are in $P$, and $M_1 = (H_1, R_1)$ and $M_2 = (H_2, R_2)$ be machine configurations. Suppose*

1. *$M_1$ is type safe at $v_1$ and $M_2$ is type safe at $v_2$, for $P$ with $\psi_1$ and $\psi_2$, respectively.*
2. *$\rhd_P M_1 : \Gamma_{v_1}, \Lambda_{v_1}, \psi_1 \approx_\zeta M_2 : \Gamma_{v_2}, \Lambda_{v_2}, \psi_2$ mConfig,*
3. *$\Gamma_{v_1}(\mathsf{pc}) \not\sqsubseteq \zeta$ and $\Gamma_{v_2}(\mathsf{pc}) \not\sqsubseteq \zeta$,*
4. *$M_1 \xrightarrow{\mathsf{Asm}(p_{v_1})} M_1'$, and*
5. *there exists $p_{w_1}$ in $P$ such that $p_{v_1} \rightsquigarrow p_{w_1}$ and $M_1'$ is type safe at $w_1$ with $\psi_3$.*

*Then, either the configuration $M_2$ diverges or there exists a machine configuration $M_2'$ such that*

$(a)$ $M_2 \Longrightarrow M_2'$,
$(b)$ *there exists $p_{w_2}$ in $P$ such that $p_{v_2} \overset{*}{\rightsquigarrow} p_{w_2}$ and $M_2'$ is type safe at $w_2$ with $\psi_4$, and*
$(c)$ $\rhd_P M_1' : \Gamma_{w_1}, \Lambda_{w_1}, \psi_3 \approx_\zeta M_2' : \Gamma_{w_2}, \Lambda_{w_2}, \psi_4$ mConfig.

The main technical difficulty here is the proof of the case when one execution does a `cjmp` instruction that lowers the pc level. In this case, the other execution should, in a number of steps, also reduce its pc level accordingly. This is guaranteed by two facts. First, high-indistinguishable stacks share a sub-stack whose top is the label to the junction point where the pc level is reduced and both executions converge. Second, well-typed programs reach final states only with an empty stack, having visited all the labels indicated by the junction point stack.

## 4 Related Work

Information flow analysis has been an active research area in the past three decades [18]. Pioneering work by Bell and LaPadula [4], Feiertag et al. [9], Denning and Denning [8, 7], Neumann et al. [17], and Biba [5] set the basis of multilevel security by defining a model of information flow where subjects and objects have a security level from a lattice of security levels. Such a lattice is instrumental in representing a security policy where a subject cannot read objects of level higher than its level, and it cannot write objects at levels lower than its own level.

The notion of *non-interference* was first introduced by Goguen and Meseguer [10], and there has been a significant amount of research on type systems for confidentiality for high-level languages including Volpano and Smith [20], and Banerjee and Naumann [2]. Type systems for low-level languages have been an active subject of study for several years now, including TAL [14], STAL [15], DTAL [21], Alias Types [19], and HBAL [1].

In his PhD thesis [16], Necula already suggests information flow analysis as an open research area at the assembly language level. Zdancewic and Myers [22] present a low-level, secure calculus with ordered linear continuations. An earlier version of our type system was inspired by that work. However, we discovered that in a typed assembly language it is enough to have a junction point stack instead of mimicking ordered linear continuations. Moreover, their language has an `if-then-else` constructor that guides the information flow analysis, while SIF has pseudo-instructions (`cpush` $L$ and `cjmp` $L$) for the same purpose. However, while the `if-then-else` constructor remains part of their language after typechecking, `cpush` and `cjmp` are eliminated.

Barthe et al. [3] define a JVM-like low-level language with a heap and an operand stack. The type system is parameterized by control dependence regions, and it is assumed that there exist functions that obtain such regions. In contrast, SIF allows such regions to be expressed in the language by using code labels and its well-formedness to be verified during type-checking. Crary et al. [6] define a low-level calculus for information flow analysis, however, their calculus has the structuring construct `if-then-else`, unlike SIF that uses typed pseudo-instructions that are assembled to standard machine instructions.

## 5 Conclusions and Future Work

We defined SIF, a typed assembly language for secure information flow analysis. Besides the standard features, such as heap and register bank, SIF introduces a stack of code labels in order to simulate at the assembly level the block structure of high-level languages. The type system guarantees that well-typed programs assembled on type-safe machine configurations satisfy the non-interference property: for a security level $\zeta$, if two type-safe machine configuration are $\zeta$-indistinguishable, then the resulting machine configurations after execution are also $\zeta$-indistinguishable.

An alternative to our approach is to have a list of the program points where the security level of the `pc` can be lowered safely. This option delegates the security analysis of where the `pc` level can be safely lowered to a previous step (that may use, for example, a function to calculate control dependence regions [12]). This delegation introduces a new trusted structure into the type system. Our type system, however, does not need to trust the well-formation of such a list. Moreover, even the signature ($\Sigma$) attached to SIF programs is untrusted in our setting, since, as we explained in section 2.3, its information about the security level of the `pc` is checked in the rules for `cpush` and `cjmp` in order to prevent illegal information flows.

Currently we are implementing the type system proposed in this paper. We already developed a compiling function from a very simple high-level imperative programming language to SIF and the typechecker for SIF programs. We intend to make the software available upon completion of the system.

We are also developing a version of our language that includes a runtime stack, in order to define a stack-based compilation function from a more complex high-level language to SIF.

**Acknowledgments:** We are grateful to Pablo Garralda, Healfdene Goguen, David Naumann, and Alejandro Russo for enlightening discussions and comments on previous drafts. We want to thank Nicholas Egebo and Haifan Lu, two undergraduate students at Stevens, for their help with the implementation of SIF. This work was partially supported by the *NSF* project *CAREER: A formally verified environment for the production of secure software* – #0093362 and the Stevens Technogenesis Fund.

## References

1. David Aspinall and Adriana B. Compagnoni. Heap bounded assembly language. *Journal of Automated Reasoning, Special Issue on Proof-Carrying Code*, 31(3-4):261–302, 2003.
2. A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of Fifteenth IEEE Computer Security Foundations - CSFW*, pages 253–267, June 2002.
3. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proceedings of VMCAI'04*, volume 2937 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.
4. D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report Technical Report MTR 2547 v2, MITRE, November 1973.
5. K. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.
6. Karl Crary, Aleksey Kliger, and Frank Pfenning. A monadic analysis of information flow security with mutable state. Technical Report CMU-CS-03-164, Carnegie Mellon University, September 2003.
7. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

8. Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, May 1976.
9. R. J. Feiertag, K. N. Levitt, and L. Robinson. Proving multilevel security of a system design. In *6th ACM Symp. Operating System Principles*, pages 57–65, November 1977.
10. J. A. Goguen and J. Meseguer. Security policy and security models. In *Proceedings of the Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1982.
11. Daniel Hedin and David Sands. Timing aware information flow security for a javacard-like bytecode. In *Proceedings of BYTECODE, ETAPS'05, to appear*, 2005.
12. Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Proceedings of CAV'01*, volume 2102, pages 265–285. Springer-Verlag, 2001.
13. Ricardo Medel, Adriana Compagnoni, and Eduardo Bonelli. A typed assembly language for secure information flow analysis. `http://www.cs.stevens.edu/~rmedel/hbal/publications/sifTechReport.ps`, 2005.
14. 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):528–569, May 1999.
15. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*, pages 95–117, Kyoto, March 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28-52. Springer-Verlag, 1998.
16. George Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998.
17. Peter G. Neumman, Richard J. Feiertag, Karl N. Levitt, and Lawrence Robinson. Software development and proofs of multi-level security. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 421–428. IEEE Computer Society, October 1976.
18. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
19. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, April 2000.
20. Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
21. Hongwei Xi and Robert Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology, July 1999.
22. S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3), 2002.