Information Flow Analysis for a Typed Assembly Language with Polymorphic Stacks

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

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

Abstract. We study secure information flow in a stack based Typed Assembly Language (TAL). We define a TAL with an execution stack and establish the soundness of its type system by proving *non-interference*. One of the problems of studying information flow for a low-level language is the absence of high-level control flow constructs that guide information flow analysis in high-level languages. Furthermore, in the presence of an execution stack, code that frees space on the stack must be constrained in order to avoid illegal flows. Finally, in the presence of stack polymorphism, we must ensure that type variables are instantiated without observable differences. These issues are addressed by introducing *junction points* into the type system, ensuring that they behave as ordered linear continuations, and that they interact safely with the execution stack. We also discuss several limitations of our approach and point out some remaining open issues.

1 Introduction and Motivation

The increasing need to guarantee the confidentiality of electronically stored information has prompted the academic community to study confidentiality from different points of view. Although access control regulates who can access information, it does not regulate the proper manipulation of sensitive data, i.e. the flow of information. In contrast, the theory of programming languages provides powerful techniques that have proven successful in studying information flow security.

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

The example in Fig. 1(a) shows a program that has the non-interference property only if the variable `y` is secret. Otherwise, information about the secret (labelled $\top$) value stored in `x` is revealed by the different values that `y` can have, depending on the branch of the `if-then-else` instruction executed. This breach of security is called *implicit* information flow. In order to statically detect

G. Barthe et al. (Eds.): CASSIS 2005, LNCS 3956, pp. 37–56, 2006.

pc level	$\texttt{x} : \texttt{int}^{\top}; \texttt{z} : \texttt{int}^{\bot}$
low	`if x = 0`
high	`  then y := 1`
high	`  else y := 2;`
low	`z := 3`

(a) High-level language

```
L1 : bnz r1,L2
     mov r2,1
     jmp L3
L2 : mov r2,2
L3 : mov r3,3
```

(b) Assembly language

```
L1 : pushJP L3
     bnz r1,L2
     mov r2,1
     jmpJP L3
L2 : mov r2,2
     jmpJP L3
L3 : mov r3,3
```

(c) SIFTAL

Fig. 1. Example of implicit information flow

this kind of information flow, a security level is associated with the program counter at each program point. This association is shown in the left column of the example, and it can be verified that at each program point no variable with a lower security level than the program counter is updated. Notice that the assignment in the last line of code does not depend on the value of `x`. Therefore, the level of the program counter becomes low again and the public (labelled $\bot$) variable `z` can be updated without compromising confidentiality.

Motivated by the desire to obtain secure information flow results for low-level code without trusting the compiler, and the fact that most mobile programs are distributed in some low-level format, we study confidentiality for assembly programs using a language-based approach to security via type theory.

Information flow analysis in low-level languages presents a number of difficulties typically not present in high-level ones. As already noted in several articles on information flow for low-level languages [3, 14], the absence of high-level control flow constructs dictates the need for some alternative mechanism for retrieving high-level program structure. For example, the program in Fig. 1(b) is a standard translation of the high-level program of Fig. 1(a) to assembly language. Notice that the security level of the program counter can be raised after the execution of the `bnz` instruction, but there is no way of knowing where it can be lowered again, in order to allow the update of the public variable `z` represented by register `r3`.

In [15] we introduced the notion of *junction point*, which represents a program point where execution of different branches of computation converge. Our *Typed Assembly Language* SIFTAL uses junction points and typing directives to manipulate a stack of junction points and reflect the control flow structure of the programs. These junction points are instrumental during typechecking to prove non-interference; however, they bear no meaning during execution, and are therefore removed after typechecking.

The code in Fig. 1(c) is a translation to SIFTAL of the high-level program in Fig. 1(a). The SIFTAL program uses the code label *L3* as a junction point, in order to signal where the security level can be lowered. During typechecking, the typing directive `pushJP` *L3* pushes the label *L3* onto a stack, introducing a *linear obligation* that has to be met by using a `jmpJP` *L3* instruction. Moreover, the program is well-typed only if there are no modifications of public registers inside the branches of the `bnz` instruction. Note that `pushJP` has no effect at run-time and it is discarded after typechecking, while `jmpJP` will be replaced by a `jmp` instruction.

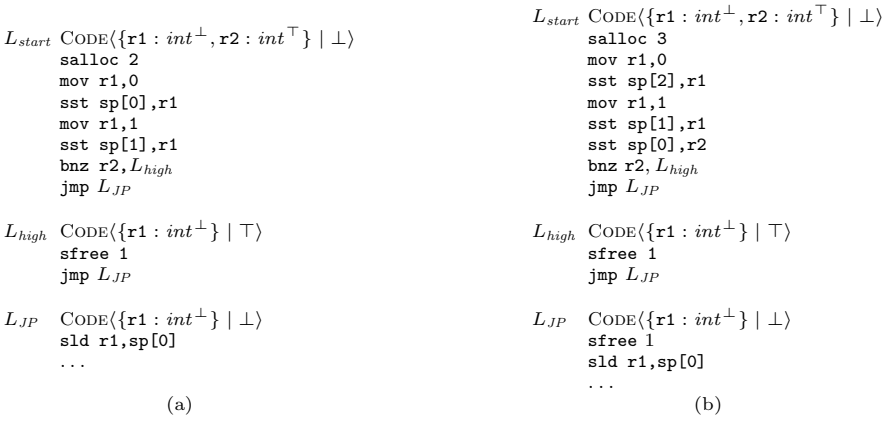

Fig. 2. Insecure information flow in the presence of stacks

SIFTAL is a RISC *load-store* assembly language, and it includes instructions to allocate (`salloc`) and deallocate (`sfree`) space on the stack, and instructions to load (`sld`) and store (`sst`) words from and onto the stack. In order to verify the non-interference of a program that manipulates the stack, we must ensure the absence of explicit illegal flows via the stack: elements that are pushed onto the stack while the **pc** has a security level l must be popped while the **pc** has at least security level l.

However, other more subtle forms of information leaks may arise. Consider the code in Fig. 2. The expression $\textsc{Code}\langle\Gamma \mid \mathbf{pc}\rangle$ that appears alongside a code label is the type of the code block, where Γ is the type of the registers before execution starts, and **pc** indicates the initial security level of the program counter. For simplicity, in these programs we do not include the type of the stack pointer.

Fig. 2(a) shows that it is possible to leak confidential information by allowing to pop a public stack component during the execution of a high-security branch. The first five lines of the program push a 0 and a 1 onto the stack using the auxiliary register `r1`. Then, in line 6 a branching operation based on the secret value stored in `r2` is performed. One branch (line 7) is empty, and only jumps to the junction point L_{JP}. The second branch pointed to by L_{high} eliminates one element from the stack with `sfree`. Thus, the top of the stack is erased, leaving 1 as the new top, and the branch ends with a jump to L_{JP}. At the junction point L_{JP}, the security level of the **pc** is low again, allowing the public register `r1` to learn information about the secret value in the register `r2` by reading the top of the stack. Now, if `r1` contains 1 the attacker knows that `r2` is not 0, if `r1` contains 0 the attacker knows that `r2` contains 0.

The previous problem stems from the fact that a high-security branch has freed the public stack component on the top of the stack. However, as the next example shows, it is not sufficient to restrict the free operation to components whose security level is at least that of the program counter at that point. The code in Fig. 2(b) pushes two public values on the stack and then a secret one.

It then branches on the secret value of `r2`. One branch does nothing and the other frees the top of the stack. It is legal to do it because at the top of the stack there is a secret value and the program counter is high-security at that point. Once the junction point L_{JP} has been reached, the topmost item of the stack is freed. Again, this is legal because the security level of the program counter is low. Therefore, any public or secret value can be erased from the stack. However, depending on the public value of the top of the resulting stack, information may be inferred about the secret value of `r2`. The conclusion is that the type system must guarantee that high branches free the same amount of items from the stack.

Stack types in SIFTAL are polymorphic. That is, type variables can be included in a stack type to abstract part of the type. This is required to implement multiple calls to code sections, as in procedure calls. The new format of the type for a piece of code will be $\textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle$, with Θ being the list of variables (usually denoted by the capital letters X, Y, etc.), Γ being the type of the registers, including a stack pointer register named `sp`, and **pc** being the expected security level of the program counter in such code.

In order to jump to a piece of code that has a polymorphic stack type, the type variables must be instantiated. For example, code of type $\textsc{Code}\langle\forall[X]\{\mathtt{sp} : int^{\bot} \cdot X\} \mid \mathbf{pc}\rangle$ requires a stack with type $int^{\bot} \cdot X$ to be executed safely. If that code is at label L then the jump instruction `jmp` $L[int^{\top} \cdot \epsilon]$ instantiates, for this particular call, the type of the stack to $int^{\bot} \cdot int^{\top} \cdot \epsilon$.

```
L_start  CODE⟨∀[X]{r1 : int^⊥, r2 : int^⊤, sp : X} | ⊥⟩
         salloc 2
         mov r1,0
         sst sp[0],r1
         mov r1,1
         sst sp[1],r1
         pushJP L_JP
         bnz r2,L_high[X]
         jmpJP L_JP[int^⊥ · X] % requires L_JP on top

L_high   CODE⟨∀[Y]{r1 : int^⊥, sp : int^⊥ · L_JP · int^⊥ · Y} | ⊤⟩
         sfree 1
         jmpJP L_JP[Y]       % requires L_JP in the middle

L_JP     CODE⟨∀[Z]{r1 : int^⊥, sp : int^⊥ · Z} | ⊥⟩
         sld r1,sp[0]
         ...
```

Fig. 3. Implicit information flow detected in SIFTAL

By translating the code in Fig. 2(a) to SIFTAL, we obtain the program in Fig. 3. This code fails to typecheck since the occurrence of `jmpJP` in L_{start} requires that L_{JP} be at the top of the stack, but the one in L_{high} requires that it be in the middle, since L_{high} frees the topmost component.

This paper presents an extension of our previous work on information flow for assembly languages [7, 15] to address the issues that arise by the inclusion of an execution stack. The examples in this section illustrate that stack constructs are *not* orthogonal to junction point constructs. Indeed, the main technical contribution of this work is the explicit treatment of junction points as a tool to address the problems that arise in the presence of polymorphic stacks.

Additional benefits of dealing explicitly with junction points in the type system are: 1) well-typed programs must "consume" all of their junction points, which, in the vision of junction points as ordered linear continuations [28], is equivalent to requiring that all linear obligations are met; and 2) whenever a jump to a code block is performed, the type system ensures that the current pending junction points are passed on as obligations to the destination code block. Complete definitions and detailed proofs of the results in this paper appear in the preliminary version of the technical report [6].

2 An Overview of SIFTAL

2.1 Syntax of Terms and Type Expressions

SIFTAL is a Typed Assembly Language (TAL) [17] based on STAL [16]. It has an execution stack and constructs that support stack allocation. The syntax of the types for SIFTAL is given in Fig. 4, and the syntax of SIFTAL programs is given in Fig. 5. We assume the following pairwise disjoint sets: an infinite enumerable set of code labels $L_1, L_2, \ldots$, an infinite enumerable set of memory tuple labels $p_1, p_2, \ldots$, an infinite enumerable set of stack variables $X_1, X_2, \ldots$ and a finite set of registers $\{r_0, \ldots, r_n, \mathtt{sp}\}$.

security labels	$l, \mathbf{pc} \in \mathfrak{L}_{\mathbf{sec}}$
security types	$\sigma ::= \tau^l$
types	$\tau ::= int \mid \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle \mid \langle\sigma_0, \ldots, \sigma_n\rangle$
register bank types	$\Gamma ::= \{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \mathtt{sp} : \Sigma\}$
stack types	$\Sigma ::= X \mid \epsilon \mid \hat{\sigma} \cdot \Sigma \mid L \cdot \Sigma$
stack component types	$\hat{\sigma} ::= \sigma \mid ns$
type assignment	$\Theta ::= \cdot \mid X, \Theta$
heap types	$\Psi ::= \{\ell_1 : \sigma_1, \ldots, \ell_n : \sigma_n\}$
machine configuration types	$\Omega ::= [\Psi, \Gamma, \mathbf{pc}]$

Fig. 4. Types in SIFTAL

Since type expressions in SIFTAL may include annotations for security levels, we assume given a lattice $\mathfrak{L}_{\mathbf{sec}}$ of *security labels*. The least and greatest elements of this lattice are $\bot$ and $\top$, respectively. We use $\sqsubseteq$ for the lattice ordering and $\sqcup$ for the lattice join operation. *Security types* are types annotated with a security label (τ^l). The type of integer constants is *int*, while $\textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle$ is the type of code blocks. The type of tuple labels is denoted by $\langle\sigma_0, \ldots, \sigma_n\rangle$ or $\langle\overline{\sigma}\rangle$. The type of a code block $\textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle$ consists of the register context Γ, a *register bank type* that maps registers to types; the security label of the program counter ($\mathbf{pc}$); and a type assignment (Θ) of stack type variables that binds all free stack variables in Γ. *Execution stack types* (Σ) are sequences of stack component types: security types, nonsense types, and code labels. The nonsense type is used when allocating space on the stack. Letters α, α_i are used for stack component types. We use $FV(\Sigma)$ for the free variables in Σ (and similarly for the free variables in security types, code blocks, etc.). A *heap type* (Ψ) is a mapping from

machine configuration	$\Pi ::= (H, R, B)$
heap	$H ::= \{\ell_1 : h_1, \ldots, \ell_n : h_n\}$
heap labels	$\ell ::= p \mid L$
heap values	$h ::= \langle w, \ldots, w\rangle \mid \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l.B$
register bank	$R ::= \{r_1 : w_1, \ldots, r_n : w_n, \mathtt{sp} : S\}$
code blocks	$B ::= \mathtt{halt} \mid \mathtt{jmp}\ v \mid \mathtt{jmpJP}\ L[\Sigma] \mid \iota; B$
instructions	$\iota ::= aop\ r_d, r_s, v \mid \mathtt{bnz}\ r, v \mid \mathtt{mov}\ r, v \mid \mathtt{ld}\ r_d, r_s[i]$ $\mathtt{st}\ r_d[i], r_s \mid \mathtt{pushJP}\ L \mid \mathtt{salloc}\ i \mid \mathtt{sfree}\ i$ $\mathtt{sld}\ r_d, \mathtt{sp}[i] \mid \mathtt{sst}\ \mathtt{sp}[i], r_s$
arithmetic operations	$aop ::= \mathtt{add} \mid \mathtt{sub} \mid \mathtt{mul}$
operands	$v ::= r \mid w \mid v[\Sigma]$
word values	$w ::= i \mid p \mid L \mid w[\Sigma]$
stack component values	$\hat{w} ::= w \mid ns$
stack	$S ::= \epsilon \mid \hat{w} \cdot S$

Fig. 5. Syntax of SIFTAL

heap addresses to security types. We assume that Ψ maps code block security types to code labels L and tuple security types to tuple pointers p.

A *machine configuration* is a tuple (H, R, B), where H is the *heap* (mapping heap labels to heap values), R is a *register bank* (mapping registers to word values), and B is the currently executing *code block*. A heap label ℓ is either a code label L or a tuple label p. A heap value h is either a code block (annotated with a security type) $\textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l.B$ or a tuple of word values $\langle w_1, \ldots, w_n\rangle$, also denoted $\langle\overline{w}\rangle$.

A word value is either an integer constant (i), a heap label (p or L) or a heap label followed by a series of stack types of the form $L[\Sigma_1]\ldots[\Sigma_n]$, (expressions of the form $p[\Sigma_1]\ldots[\Sigma_n]$ and $i[\Sigma_1]\ldots[\Sigma_n]$ are ruled out by the type system). The register bank is a finite set of registers r_i, including a designated stack pointer `sp` which points to the top of the stack. A stack is modeled as a sequence of stack components: either word values or the special "nonsense" value ns, used for newly allocated stack space.

Besides standard assembly instructions, SIFTAL has instructions to manipulate the execution stack (`salloc`, `sfree`, `sst`, and `sld`) and to handle the junction points (`pushJP` and `jmpJP`). Note that both `jmp` and `jmpJP` may instantiate the stack variables of the destination code. For example, `jmp` $L[\Sigma]$ instantiates the stack variable of code block L with Σ before jumping to it (cf. Sec. 2.3).

2.2 Type System

In order to present the type system of SIFTAL, we need to define some notation first: if $\Gamma = \{r_1 : \sigma_1, \ldots, r_n : \sigma_n, \mathtt{sp} : \Sigma\}$, then $Dom(\Gamma)$ is the set $\{r_1, \ldots, r_n, \mathtt{sp}\}$, and $\Gamma[r := \sigma]$ is the register bank type resulting from updating Γ with $r : \sigma$. We define $\mathtt{label}(\tau^l) = l$. We use $\Gamma\{X \leftarrow \Sigma\}$ for the result of substituting all free occurrences of the stack type variable X in Γ with Σ. A similar notation is used for substituting inside word values, operands, types, code blocks, etc. If Θ is a sequence of stack type variables $X_1, \ldots, X_n$ and $\overline{\Sigma}$ is a sequence of stack types

$$\boxed{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \ \mathsf{blk}}$$

$$\frac{\Theta \rhd \Gamma \ \mathsf{ok} \quad \Gamma(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{halt}\ \mathsf{blk}}\ \mathsf{T\_Halt}$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi v : \textsc{Code}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle^{l'}\ \mathsf{opnd} \qquad \Theta \rhd \textsc{Code}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle \leq \textsc{Code}\langle\forall[\cdot]\Gamma \mid \mathbf{pc} \sqcup l'\rangle}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{jmp}\ v\ \mathsf{blk}}\ \mathsf{T\_Jmp}$$

$$\frac{\begin{array}{c}\Theta \rhd \textsc{Code}\langle\forall[\cdot]\Gamma'\{X \leftarrow \Sigma\} \mid \mathbf{pc}'\rangle \leq \textsc{Code}\langle\forall[\cdot]\Gamma[\mathtt{sp} := \Sigma'] \mid l\rangle \\ \Gamma'(\mathtt{sp}) = \alpha_1 \cdot \ldots \alpha_n \cdot X \qquad \Theta \rhd \Sigma\ \mathsf{ok} \qquad \Gamma(\mathtt{sp}) = L \cdot \Sigma' \qquad \Psi(L) = \textsc{Code}\langle\forall[X]\Gamma' \mid \mathbf{pc}'\rangle^{l}\end{array}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{jmpJP}L[\Sigma]\ \mathsf{blk}}\ \mathsf{T\_Jmpcc}$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi r_s : \mathit{int}^{l_1}\ \mathsf{opnd} \qquad \Theta \mid \Gamma \rhd_\Psi v : \mathit{int}^{l_2}\ \mathsf{opnd} \qquad \mathbf{pc} \sqcup l_1 \sqcup l_2 \sqsubseteq \mathtt{label}(\Gamma(r_d))\ \Theta \mid \Gamma[r_d := \mathit{int}^{\mathtt{label}(\Gamma(r_d))}] \mid \mathbf{pc} \rhd_\Psi B\ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathit{aop}\ r_d, r_s, v; B\ \mathsf{blk}}\ \mathsf{T\_Arith}$$

$$\frac{\Theta \rhd \textsc{Code}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle \leq \textsc{Code}\langle\forall[\cdot]\Gamma \mid \mathbf{pc} \sqcup l_1 \sqcup l_2\rangle \qquad \Theta \mid \Gamma \rhd_\Psi r : \mathit{int}^{l_1}\ \mathsf{opnd} \qquad \Theta \mid \Gamma \rhd_\Psi v : \textsc{Code}\langle\forall[\cdot]\Gamma' \mid \mathbf{pc}'\rangle^{l_2}\ \mathsf{opnd} \qquad \Theta \mid \Gamma \mid \mathbf{pc} \sqcup l_1 \rhd_\Psi B\ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{bnz}\ r, v; B\ \mathsf{blk}}\ \mathsf{T\_CondBrnch}$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi v : \tau_1^{l_1}\ \mathsf{opnd} \quad \mathbf{pc} \sqcup l_1 \sqsubseteq \mathtt{label}(\Gamma(r)) \quad \Theta \mid \Gamma[r := \tau_1^{\mathtt{label}(\Gamma(r))}] \mid \mathbf{pc} \rhd_\Psi B\ \mathsf{blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{mov}\ r, v; B\ \mathsf{blk}}\ \mathsf{T\_Mov}$$

Fig. 6. Typing rules for code blocks (part I)

$\Sigma_1, \ldots, \Sigma_n$, then we write $\Gamma\{\Theta \leftarrow \overline{\Sigma}\}$ for $\Gamma\{X_1 \leftarrow \Sigma_1\}\{X_2 \leftarrow \Sigma_2\} \ldots \{X_n \leftarrow \Sigma_n\}$, if $\{X_i, \ldots, X_n\} \cap FV(\Sigma_{i-1}) = \emptyset$, for $2 \leq i \leq n$.

The typing judgments that determine when a code block B is well-typed under type assignment Θ, register type Γ, program counter security level $\mathbf{pc}$ and heap type Ψ, are given in Figs. 6 and 7. A `halt` instruction is treated as a jump to a special junction point that halts the program execution. As a consequence, the stack type must have the label *Halt* at the top. The judgment $\Theta \rhd \Gamma$ ok verifies that the register bank type is well-formed under type assignment Θ, that is, that the free variables in Γ are declared in the type assignment Θ (see [6] for a formal statement). In order for a `jmp` v instruction to be well-typed, the current register bank type must be compatible with that which is expected at the destination label denoted by v. This is enforced by means of the subtyping judgment. The subtyping relation is omitted here for reasons of space, but it includes the standard requirements that it be a partial order (on types, security types and register bank types) and uses width subtyping for register bank types. Moreover, in order to avoid illegal flows, a jump instruction can only jump to a code block with the same or higher security level. Therefore, the security level of the destination code label must be higher than or equal to the current level $\mathbf{pc}$ of the program counter together with the security label of the destination code label l'.

The `jmpJP` $L[\Sigma]$ instruction is similar, although not identical, to the `jmp` case. First of all, in order to jump to the next junction point L, it must appear at

$$\frac{\Theta \mid \Gamma \rhd_\Psi r_s : \langle \sigma_0, \ldots, \sigma_i, \ldots, \sigma_{n-1} \rangle^{l_1} \text{ opnd } \mathbf{pc} \sqcup l_1 \sqsubseteq \mathtt{label}(\sigma_i) \quad \mathtt{label}(\sigma_i) \sqsubseteq \mathtt{label}(\Gamma(r_d)) \quad \Theta \mid \Gamma[r_d := \sigma_i] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{ld}\ r_d, r_s[i]; B \text{ blk}}\ \mathsf{T\_Ld}$$

$$\frac{\Theta \mid \Gamma \rhd_\Psi r_d : \langle \sigma_0, \ldots, \sigma_i, \ldots, \sigma_{n-1} \rangle^{l} \text{ opnd } \Theta \mid \Gamma \rhd_\Psi r_s : \sigma_i \text{ opnd} \quad \mathbf{pc} \sqcup l \sqsubseteq \mathtt{label}(\sigma_i) \quad \Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{st}\ r_d[i], r_s; B \text{ blk}}\ \mathsf{T\_St}$$

$$\frac{\Psi(L) = \textsc{Code}\langle \forall [X] \Gamma' \mid \mathbf{pc}' \rangle^{l'} \quad \mathbf{pc} \sqsubseteq \mathbf{pc}' \quad \Gamma(\mathtt{sp}) = \Sigma \quad \Theta \mid \Gamma[\mathtt{sp} := L \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{pushJP}\ L; B \text{ blk}}\ \mathsf{T\_Push}$$

$$\frac{\Gamma(\mathtt{sp}) = \Sigma \quad \Theta \mid \Gamma[\mathtt{sp} := \overbrace{ns \cdot \ldots \cdot ns}^{i} \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{salloc}\ i; B \text{ blk}}\ \mathsf{T\_Salloc}$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \hat{\sigma}_{i-1} \cdot \Sigma \quad \Theta \mid \Gamma[\mathtt{sp} := \Sigma] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sfree}\ i; B \text{ blk}}\ \mathsf{T\_Sfree}$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_i \cdot \Sigma \quad \mathbf{pc} \sqcup l \sqsubseteq \mathtt{label}(\Gamma(r_d)) \quad \hat{\sigma}_i = \tau^l \quad \Theta \mid \Gamma[r_d := \tau^{\mathtt{label}(\Gamma(r_d))}] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sld}\ r_d, \mathtt{sp}[i]; B \text{ blk}}\ \mathsf{T\_Sld}$$

$$\frac{\Gamma(\mathtt{sp}) = \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_i \cdot \Sigma \quad \mathbf{pc} \sqsubseteq l \quad \Theta \mid \Gamma \rhd_\Psi r_s : \tau^l \text{ opnd} \quad \Theta \mid \Gamma[\mathtt{sp} := \hat{\sigma}_0 \cdot \ldots \cdot \hat{\sigma}_{i-1} \cdot \tau^l \cdot \Sigma] \mid \mathbf{pc} \rhd_\Psi B \text{ blk}}{\Theta \mid \Gamma \mid \mathbf{pc} \rhd_\Psi \mathtt{sst}\ \mathtt{sp}[i], r_s; B \text{ blk}}\ \mathsf{T\_Sst}$$

Fig. 7. Typing rules for code blocks (part II)

the top of the current stack type $\Gamma(\mathtt{sp})$. Second, the current register bank type must be compatible with the one expected at the destination code. In particular, this includes passing on the pending junction points which appear in the type of the current execution stack $\Gamma(\mathtt{sp})$. The register bank type expected at the destination label L is given by $\Psi(L)$, where all occurrences of the stack type X have been instantiated with Σ. In order to deal with the problem mentioned in the introduction related to stack polymorphism, we assume that junction points have only one free stack variable (which may of course occur any number of times in Γ', the register bank type required by the destination code block) and that the type of the stack $\Gamma'(\mathtt{sp})$ has an occurrence of X at the end (cf. condition $\Gamma'(\mathtt{sp}) = \alpha_1 \cdot \ldots \cdot \alpha_n \cdot X$). This allows us to relate the type instantiated for X, namely Σ, to the type of the current execution stack (Σ') and this gives us a handle on Σ when dealing with non-interference. Finally, since the program counter level of the junction point is to be "reset" to $\mathbf{pc}'$, the label l' of the type of L becomes irrelevant. This is addressed by requiring $l' \sqsubseteq \mathbf{pc}'$, a condition which is easily met by defining l' appropriately.

T_Arith types the arithmetic operators. Since the result of the operation depends on the operands and the current program counter level, the register that holds the result (r_d) is required to have the appropriate security level. The rule for `mov` is similar to T_Arith. T_CondBrnch, T_Ld and T_St are as expected (see [6] for further details).

The `pushJP` L type directive simply adds the code label L to the top of the stack *type* and types the rest of the code block under this new stack type. The condition $\mathbf{pc} \sqsubseteq \mathbf{pc}'$ makes sure that when the junction point L is invoked, the label of the program counter does not drop below the current level **pc**.

Regarding `salloc`, we simply add i nonsense types to the stack type and then typecheck the rest of the code under this new stack type. The instruction that frees the top i components of the stack, namely `sfree` i, simply drops the top i component types of the stack type and then types the rest of the program. Two comments are in order here. First, the stack components that are freed must not be code labels. This would interfere with the linear nature of the junction points (the only directives that may manipulate junction points are `jmpJP` and `pushJP`). Second, no condition on the security labels of the freed components is required. This is a consequence of our approach to junction points which, for example, guarantees that if components are freed before jumping to a low-level junction point, then the freed components must have been secret (and hence not observable to the low-level user). See the High-Step Invariant Lemma (Lemma 2) for details.

The typing rules for `sld` and `sst` follow similar patterns to those discussed above. T_Sld requires that the stack component to be loaded be initialized (i.e. not have nonsense type). We remark that it should be straightforward to extend both T_Sld and T_Sst so as to allow loading stack components that are under junction point labels, although the details remain to be verified.

Due to lack of space, we do not give here the typing rules for word values and operands. In the case of word values, they assign $int^{\perp}$ to integer constants and look up the type in Ψ for code and tuple pointers. Also, there is a subsumption rule that allows subtyping reuse: a word value of type σ may always be used at any supertype σ'. Finally, a rule allows word values of the form $w[\Sigma]$ to be typed. Similar comments apply to the typing rules for operands. Two typing rules are presented for typing heap values: one for typing a tuple $\langle w_1, \ldots, w_n \rangle$ with a tuple security type $\langle \sigma_1, \ldots, \sigma_n \rangle^l$ componentwise, and another for typing annotated code blocks.

The typing rules for execution stacks (Fig. 8) need no further comment except for T_ExeSLbl. This rule states that junction points in the execution stack type may be ignored at run-time. Regarding heaps and register banks, the former is well-typed if each of the labels in its domain is mapped to well-typed heap values and the latter if each register different from `sp` is mapped to a well-typed word value. Finally, a machine configuration (H, R, B) is well-typed if the heap, register bank, current code block and execution stack are all well-typed.

2.3 Operational Semantics

The operational semantics of SIFTAL is shown in Fig. 9. Each rule establishes the semantics of an instruction on the current machine configuration. We say that Π *reduces* to Π' if $\Pi \longrightarrow \Pi'$. We use $\twoheadrightarrow$ for the reflexive, transitive closure of $\longrightarrow$. The expression $\hat{R}(v)$ is defined as: $R(r)$ if $v = r$, w if $v = w$, and $\hat{R}(v_1)[\Sigma]$ if $v = v_1[\Sigma]$. The `jmp` v instruction is executed by first looking up the destination

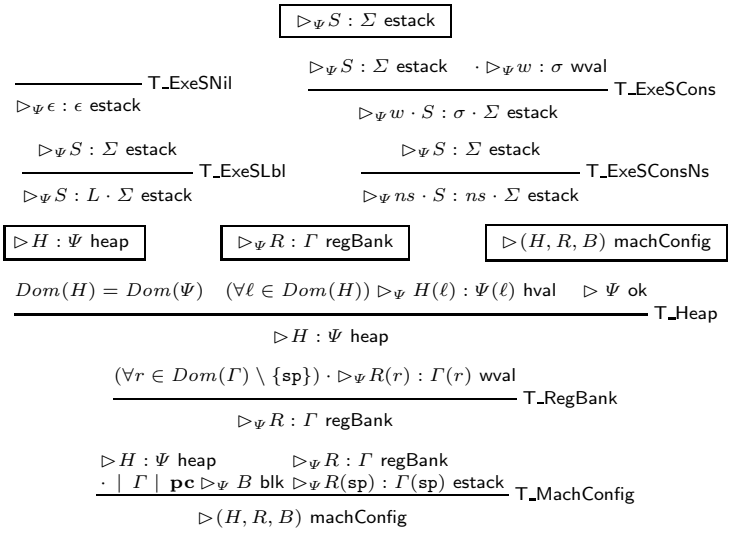

Fig. 8. Type rules for execution stacks, heaps, register banks and machine configurations

$$(H, R, B) \longrightarrow \Pi$$

where if $B =$	then $\Pi =$	
`jmp` v	$(H, R, B\{\Theta \leftarrow \overline{\Sigma}\})$ where $\hat{R}(v) = L[\overline{\Sigma}]$ and $H(L) = \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}'\rangle^l.B$	OS_Jmp
`jmpJP` $L[\Sigma]$	$(H, R, B\{X \leftarrow \Sigma\})$ where $H(L) = \textsc{Code}\langle\forall[X]\Gamma \mid \mathbf{pc}'\rangle^l.B$	OS_Jmpcc
$aop\ r_d, r_s, v; B$	$(H, R[r_d := n], B)$ where $n = \hat{R}(v) \oplus \hat{R}(r_s)$	OS_Arith
`bnz` $r, v; B$	(H, R, B) where $\hat{R}(r) = 0$	OS_Bnz1
`bnz` $r, v; B$	$(H, R, B'\{\Theta \leftarrow \overline{\Sigma}\})$ where $\hat{R}(r) \neq 0$, $\hat{R}(v) = L[\overline{\Sigma}]$ and $H(L) = \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}''\rangle^l.B'$	OS_Bnz2
`mov` $r, v; B$	$(H, R[r := \hat{R}(v)], B)$	OS_Mov
`ld` $r_d, r_s[i]; B$	$(H, R[r_d := w_i], B)$ where $\hat{R}(r_s) = p$ and $H(p) = \langle w_0, \ldots, w_i, \ldots, w_{n-1}\rangle$	OS_Load
`st` $r_d[i], r_s; B$	$(H[p := \langle w_0, \ldots, R(r_s), \ldots, w_{n-1}\rangle], R, B)$ where $\hat{R}(r_d) = p$ and $H(p) = \langle w_0, \ldots, w_i, \ldots, w_{n-1}\rangle$	OS_Store
`pushJP` $L; B$	(H, R, B)	OS_Push
`salloc` $i; B$	$(H, R[\mathtt{sp} := \underbrace{ns \cdot \ldots \cdot ns}_{i} \cdot S], B)$ where $R(\mathtt{sp}) = S$	OS_Salloc
`sfree` $i; B$	$(H, R[\mathtt{sp} := S], B)$ where $R(\mathtt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_{i-1} \cdot S$	OS_Sfree
`sld` $r_d, \mathtt{sp}[i]; B$	$(H, R[r_d := w_i], B)$ where $R(\mathtt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_i \cdot S$	OS_Sld
`sst` $\mathtt{sp}[i], r_s; B$	$(H, R[\mathtt{sp} := \hat{w}_0 \cdot \ldots \cdot \hat{w}_{i-1} \cdot R(r_s) \cdot S], B)$ where $R(\mathtt{sp}) = \hat{w}_0 \cdot \ldots \cdot \hat{w}_i \cdot S$	OS_Sst

Fig. 9. Operational semantics

code label L in v, obtaining the destination code block from the heap and finally instantiating this code with the vector of stack types given in the operand. The rule for `jmpJP` $L[\Sigma]$ is the same as that of `jmp` v for $v = L[\Sigma]$. The remaining rules are self explanatory. Note that `pushJP` is a type directive: it has no effect at run-time. As a consequence, it could be erased once type checking has been completed. Finally, we would like to point out that the semantics of SIFTAL is exactly that of STAL, disregarding the `malloc` and `pack` instructions that are not treated in this work.

The operational semantics is sound with respect to the type system. If a typed machine configuration is not in a valid final state, then it can always progress towards one, as formalized by the Progress and Subject Reduction propositions. In order to formalize this, we use the notation $\rhd(H, R, B) : [\Psi, \Gamma, \mathbf{pc}]$ machConfig to mean that all of the judgments $\rhd H : \Psi$ heap, $\rhd_\Psi R : \Gamma$ regBank, $\rhd_\Psi R(\mathtt{sp}) : \Gamma(\mathtt{sp})$ estack and $\cdot \mid \Gamma \mid \mathbf{pc} \rhd_\Psi B$ blk hold. A machine configuration Π is said to be *stuck* at type $[\Psi, \Gamma, \mathbf{pc}]$ if $\rhd \Pi : [\Psi, \Gamma, \mathbf{pc}]$ machConfig and Π is not of the form $(H, R, \mathtt{halt})$ with $\Gamma(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma$ and there does not exist a machine configuration Π' such that $\Pi \longrightarrow \Pi'$.

Proposition 1 (Progress). *If* $\rhd \Pi : [\Psi, \Gamma, \mathbf{pc}]$ machConfig *then either there exists* Π' *such that* $\Pi \longrightarrow \Pi'$*, or* Π *is of the form* $(H, R, \mathtt{halt})$ *with* $\Gamma(\mathtt{sp}) = \mathit{Halt} \cdot \Sigma$*.*

Proposition 2 (Subject Reduction). *If* $\rhd \Pi$ machConfig *and* $\Pi \longrightarrow \Pi'$ *then* $\rhd \Pi'$ machConfig*.*

3 Non-interference

Non-interference is a semantic property that states that computed low security level values should not be affected by high security ones. Here, "low security" and "high security" are relative to an arbitrary, but fixed a priori, security level ζ that determines what an observer can see (*low security values*) and what he cannot (*high security values*). A low security value or typ) is thus one whose security level is less than or equal to ζ; a high security value or type is one whose security level is *not* less than or equal to ζ. The formalization of non-interference proceeds by defining a notion of indistinguishable machine configuration with respect to the observer (we call this ζ-indistinguishability) and then showing that given any two runs of a program that start at indistinguishable machine configurations, if they terminate[1] then they both reach indistinguishable machine configurations. These two issues are studied in this section.

3.1 ζ-Indistinguishability

An appropriate notion of ζ-indistinguishability for machine configurations requires taking into account each of its components, namely heap, register bank

[1] Termination sensitive information flow analysis is an active topic of research. However, it is not considered in this paper.

$$\boxed{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow}}$$

$$\frac{}{\rhd_{\Psi_1,\Psi_2} \epsilon \approx_\zeta \epsilon : \epsilon \wedge \epsilon \ \mathsf{estackLow}}\ \mathsf{EqES\_LAxiom}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow} \quad \rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2 \ \mathsf{wval}}{\rhd_{\Psi_1,\Psi_2} w_1 \cdot S_1 \approx_\zeta w_2 \cdot S_2 : \sigma_1 \cdot \Sigma_1 \wedge \sigma_2 \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LLLHH}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow}}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta ns \cdot S_2 : ns \cdot \Sigma_1 \wedge ns \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LNonsense}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w_1 \cdot S_1 \approx_\zeta ns \cdot S_2 : \tau^l \cdot \Sigma_1 \wedge ns \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LHighNs}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta w_2 \cdot S_2 : ns \cdot \Sigma_1 \wedge \tau^l \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LNsHigh}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow} \quad \Psi_1(L) = \Psi_2(L) = \textsc{Code}\langle \forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \quad \mathbf{pc} \sqcup l \sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge L \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LSynch}$$

$$\frac{\begin{array}{l}\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \ \mathsf{estackLow}\ \Psi_1(L_1) = \textsc{Code}\langle \forall[\Theta_1]\Gamma_1 \mid \mathbf{pc}_1\rangle^{l_1} \\ \Psi_2(L_2) = \textsc{Code}\langle \forall[\Theta_1]\Gamma_2 \mid \mathbf{pc}_2\rangle^{l_2} \qquad \mathbf{pc}_1 \sqcup l_1 \not\sqsubseteq \zeta \quad \mathbf{pc}_2 \sqcup l_2 \not\sqsubseteq \zeta\end{array}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L_1 \cdot \Sigma_1 \wedge L_2 \cdot \Sigma_2 \ \mathsf{estackLow}}\ \mathsf{EqES\_LHighSynch}$$

Fig. 10. Indistinguishable execution stacks at low level program counter

(including the execution stack) and currently executing code block. We begin our discussion with the execution stack. Clearly, when two runs of the same program are considered, they are seen to execute in lock-step fashion as long as no branching instruction appears. Moreover, the execution stack of each run is seen to have the same size and contain either the same low level values or high level ones. In this case we say that the stacks are *low level indistinguishable* and formalize this notion in Fig. 10. Once a branching instruction appears, say `bnz r,v`, the register `r` may either contain a low level value (in which case both programs are, once again, seen to execute in lock-step fashion) or it may be a high level value. In this last case, each run may take a different path (we talk about different *high level branches*) since this high level value may not coincide in both machine configurations. As a consequence, the execution stacks may begin to vary as a result of the execution of the subsequent instructions. In this case we say that the stacks are *high level indistinguishable* and formalize this notion in Fig. 11. However, note that before any further instruction is executed, the execution stacks of the two machine configurations are low level indistinguishable (cf. EqES_HAxiom in Fig. 11).

We have yet to clarify the meaning of $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2$ wval, which relates to the indistinguishability of word values, used when defining low level indistinguishability of stacks. The naïve approach would be to state that two word values are low indistinguishable if σ_1, σ_2 are low security, $\sigma_1 = \sigma_2$ and $w_1 = w_2$, and high indistinguishable if σ_1 and σ_2 are high security. However, the presence

$$\boxed{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackLow}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HAxiom}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w \cdot S_1 \approx_\zeta S_2 : \tau^l \cdot \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HLeft}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh} \quad l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta w \cdot S_2 : \Sigma_1 \wedge \tau^l \cdot \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HRight}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}{\rhd_{\Psi_1,\Psi_2} ns \cdot S_1 \approx_\zeta S_2 : ns \cdot \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HLeftNs}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta ns \cdot S_2 : \Sigma_1 \wedge ns \cdot \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HRightNs}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh} \quad \Psi_1(L) = \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \quad \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HLeftSynch}$$

$$\frac{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge \Sigma_2 \text{ estackHigh} \quad \Psi_2(L) = \textsc{Code}\langle\forall[\Theta]\Gamma \mid \mathbf{pc}\rangle^l \quad \mathbf{pc} \sqcup l \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} S_1 \approx_\zeta S_2 : \Sigma_1 \wedge L \cdot \Sigma_2 \text{ estackHigh}}\ \text{EqES\_HRightSynch}$$

Fig. 11. Indistinguishable execution stacks at high level program counter

of stack polymorphism complicates matters. Consider the following SIFTAL program, where B is the current code block and $\sigma_X = \textsc{Code}\langle\forall[\Theta]\{\mathtt{sp} : X\} \mid \mathbf{pc}\rangle^\bot$.

$$\begin{array}{lll} & & L1 \quad \textsc{Code}\langle\forall[X]\{\mathtt{r1} : \sigma_X, \mathtt{sp} : X\} \mid \top\rangle^\top \\ B = \mathtt{pushJP}\ L_{JP} & & \quad\ \ \mathtt{jmpJP}\ L_{JP}[\Sigma_2] \\ \quad \mathtt{bnz}\ \mathtt{r}, L1 & & \\ \quad \mathtt{jmpJP}\ L_{JP}[\Sigma_1] & & L_{JP} \ \textsc{Code}\langle\forall[Y]\{\mathtt{r1} : \sigma_Y, \mathtt{sp} : Y\} \mid \bot\rangle^\bot \\ & & \quad\ \ \ldots \end{array}$$

Suppose that we have two runs of this program. Moreover, suppose that the initial machine configuration of each run satisfies the following conditions:

1. they both assign the program counter some (one) low level value,
2. they assign low level indistinguishable execution stacks to `sp`, and
3. the register bank of the first configuration assigns 0 to the register `r` while the register bank of the other machine configuration assigns 1 to `r`. Note that these values are high indistinguishable.

At the `bnz` instruction one run shall jump to *L1* while the other shall continue with the following instruction. At some point, both runs shall "synchronize" once they reach the junction point L_{JP}. Moreover, this junction point resets the program counter to a low security level, namely $\bot$. Note, however, that at this point in time the current machine configuration of each run differs in the type of `r1`, since X has been instantiated with different types (Σ_1 and Σ_2). Hence they cannot be low indistinguishable according to the aforementioned naïve definition.

As a result, when defining low indistinguishability of word values we must allow the types of these values (σ_1 and σ_2 in $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2$ wval) to differ by instantiation of stack variables with different execution stack types. Furthermore, these execution stack types may not be arbitrary, they should also be low indistinguishable. For this reason, in order to formalize the definition of the judgment $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2$ wval we first introduce the notion of low-indistinguishable security and stack types

$$\rhd_{\Psi_1,\Psi_2} \sigma_1 \approx_\zeta \sigma_2 \ \mathsf{secTypeEq} \text{ and } \rhd_{\Psi_1,\Psi_2} \Sigma_1 \approx_\zeta \Sigma_2 \ \mathsf{stackTypeEq}$$

These notions are defined by simultaneous induction. Informally, types σ_1 and σ_2 are low indistinguishable if there is some security type σ and substitutions s_1, s_2 on stack variables such that $\sigma_1 = s_1(\sigma)$, $\sigma_2 = s_2(\sigma)$ and s_1 and s_2 assign low indistinguishable stacks to the same variables. The same applies to the notion of low indistinguishable stack types.

With this in place we can now complete our development of indistinguishability of machine configurations by defining the notion for word values, heap values, heaps, register banks and code blocks (Fig. 12). In the case of word values the judgment $\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2$ wvalEq holds iff there exist substitutions s_1, s_2 and word value w such that:

1. $Dom(s_1) = Dom(s_2) = FV(w)$,
2. $w_1 = s_1(w)$ and $w_2 = s_2(w)$, and
3. for every $X \in Dom(s_1)$, $\Psi_1, \Psi_2 \rhd s_1(X) \approx_\zeta s_2(X)$ stackTypeEq.

In Fig. 12 we write $Dom_\cup(\Psi_1, \Psi_2)$ as an abbreviation for $Dom(\Psi_1) \cup Dom(\Psi_2)$. Likewise, $Dom_\cap(H_1, H_2, \Psi_1, \Psi_2)$ abbreviates $Dom(H_1) \cap Dom(H_2) \cap Dom(\Psi_1) \cap Dom(\Psi_2)$.

Finally, we address ζ-indistinguishability of machine configurations. Both are required to be well-typed and their heaps and register banks ζ-indistinguishable. Furthermore, if their program counters are low level, then we are in the case that both programs are executing in lock-step fashion and, as a consequence, their program counters should have identical security levels, and both their currently executing code blocks and their stacks should be low indistinguishable. If their program counters are high level, then no condition applies to their currently executing code blocks, but their stacks must be high indistinguishable.

Definition 1 (ζ-indistinguishability of machine configurations). *Assume machine configurations $\Pi_i = (H_i, R_i, B_i)$ and machine types $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$, $i \in 1..2$. Then the judgment $\rhd \Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$* machConfig *holds iff*

1. $\rhd \Pi_1 : \Omega_1$ machConfig *and* $\rhd \Pi_2 : \Omega_2$ machConfig,
2. $\rhd H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_2$ heap,
3. $\rhd_{\Psi_1,\Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2$ regBank,
4. (a) *either* $\mathbf{pc}_1 = \mathbf{pc}_2 \sqsubseteq \zeta$ *and* $\rhd_{\Psi_1,\Psi_2} B_1 \approx_\zeta B_2$ code *and* $\rhd_{\Psi_1,\Psi_2} R_1(\mathtt{sp}) \approx_\zeta R_2(\mathtt{sp}) : \Gamma_1(\mathtt{sp}) \wedge \Gamma_2(\mathtt{sp})$ estackLow,
   (b) *or* $\mathbf{pc}_1 \not\sqsubseteq \zeta$ *and* $\mathbf{pc}_2 \not\sqsubseteq \zeta$ *and* $\rhd_{\Psi_1,\Psi_2} R_1(\mathtt{sp}) \approx_\zeta R_2(\mathtt{sp}) : \Gamma_1(\mathtt{sp}) \wedge \Gamma_2(\mathtt{sp})$ estackHigh.

$$\boxed{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \sigma_1 \wedge \sigma_2 \text{ wval}}$$

$$\frac{l_1 \not\sqsubseteq \zeta \quad l_2 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \tau_1^{l_1} \wedge \tau_2^{l_2} \text{ wval}}\ \text{Eq\_wval\_L} \qquad \frac{\begin{array}{c} l_1 = l_2 \sqsubseteq \zeta \\ \rhd_{\Psi_1,\Psi_2} \tau_1^{l_1} \approx_\zeta \tau_2^{l_2} \text{ secTypeEq} \\ \rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 \text{ wvalEq} \end{array}}{\rhd_{\Psi_1,\Psi_2} w_1 \approx_\zeta w_2 : \tau_1^{l_1} \wedge \tau_2^{l_2} \text{ wval}}\ \text{Eq\_wval\_H}$$

$$\boxed{\rhd_{\Psi_1,\Psi_2} h_1 \approx_\zeta h_2 : \sigma_1 \wedge \sigma_2 \text{ hval}}$$

$$\frac{l_1 \not\sqsubseteq \zeta \quad l_2 \not\sqsubseteq \zeta}{\rhd_{\Psi_1,\Psi_2} h_1 \approx_\zeta h_2 : \tau_1^{l_1} \wedge \tau_2^{l_2} \text{ hval}}\ \text{Eq\_hval\_H} \qquad \frac{l_1 = l_2 \sqsubseteq \zeta \quad \rhd_{\Psi_1,\Psi_2} w_i \approx_\zeta w_i' : \sigma_i \wedge \sigma_i' \text{ wval}}{\rhd_{\Psi_1,\Psi_2} \langle \overline{w} \rangle \approx_\zeta \langle \overline{w}' \rangle : \langle \overline{\sigma} \rangle^{l_1} \wedge \langle \overline{\sigma'} \rangle^{l_2} \text{ hval}}\ \text{Eq\_hval\_tpl\_L}$$

$$\frac{l_1 = l_2 \sqsubseteq \zeta \quad \kappa_1^{l_1}.B_1 = \kappa_2^{l_2}.B_2}{\rhd_{\Psi_1,\Psi_2} \kappa_1^{l_1}.B_1 \approx_\zeta \kappa_2^{l_2}.B_2 : \kappa_1^{l_1} \wedge \kappa_2^{l_2} \text{ hval}}\ \text{Eq\_hval\_blk\_L}$$

$\boxed{\rhd H_1 \approx_\zeta H_2 : \Psi_1 \wedge \Psi_2 \text{ heap}}$ holds iff for all $\ell \in Dom_\cup(\Psi_1, \Psi_2)$:

$\texttt{label}(\Psi_1(\ell)) \sqsubseteq \zeta$ or $\texttt{label}(\Psi_2(\ell)) \sqsubseteq \zeta$, implies $\begin{cases} \ell \in Dom_\cap(H_1, H_2, \Psi_1, \Psi_2), \\ \rhd_{\Psi_1,\Psi_2} H_1(\ell) \approx_\zeta H_2(\ell) : \Psi_1(\ell) \wedge \Psi_2(\ell) \text{ hval}. \end{cases}$

$\boxed{\rhd_{\Psi_1,\Psi_2} R_1 \approx_\zeta R_2 : \Gamma_1 \wedge \Gamma_2 \text{ regBank}}$ holds iff for all $r \in Dom_\cup(\Gamma_1, \Gamma_2) \setminus \{\texttt{sp}\}$:

$\texttt{label}(\Gamma_1(r)) \sqsubseteq \zeta$ or $\texttt{label}(\Gamma_2(r)) \sqsubseteq \zeta$, implies $\begin{cases} r \in Dom_\cap(R_1, R_2, \Gamma_1, \Gamma_2), \\ \rhd_{\Psi_1,\Psi_2} R_1(r) \approx_\zeta R_2(r) : \Gamma_1(r) \wedge \Gamma_2(r) \text{ wval}. \end{cases}$

$\boxed{\rhd_{\Psi_1,\Psi_2} B_1 \approx_\zeta B_2 \text{ code}}$ holds iff $\exists s_1, s_2, B$ such that :

1. $Dom(s_1) = Dom(s_2) = FV(B)$
2. $B_1 = s_1(B)$ and $B_2 = s_2(B)$ and
3. for every $X \in Dom(s_1)$, $\Psi_1, \Psi_2 \rhd s_1(X) \approx_\zeta s_2(X)$ stackTypeEq.

Fig. 12. ζ-indistinguishability of word, heap values, heaps, register banks, code blocks

3.2 Noninterference Theorem

This section addresses the formulation and proof of the non-interference theorem, the main result of this work. As mentioned, we consider two runs of the same program that start off from indistinguishable machine configurations. Moreover, we assume that the initial security level of the program counter is $\bot$ and that the execution stack has the *Halt* code label at the top.

Theorem 1 (Non-Interference). *For $i \in \{1, 2\}$, given machine configurations $\Pi_i = (H_i, R_i, B)$ and machine types $\Omega_i = [\Psi_i, \Gamma_i, \bot]$, if these conditions hold:*

- $\Gamma_1(\mathtt{sp}) = Halt \cdot \Sigma_1$ *and* $\Gamma_2(\mathtt{sp}) = Halt \cdot \Sigma_2$,
- $\rhd \Pi_1 \approx_\zeta \Pi_2 : [\Psi_1, \Gamma_1, \bot] \wedge [\Psi_2, \Gamma_2, \bot]$ machConfig,
- $\Pi_1' = (H_1', R_1', \texttt{halt})$ *and* $\Pi_1 \twoheadrightarrow \Pi_1'$, *and*
- $\Pi_2' = (H_2', R_2', \texttt{halt})$ *and* $\Pi_2 \twoheadrightarrow \Pi_2'$,

then there exist machine types $[\Psi_1, \Gamma_1', \mathbf{pc}_1']$ and $[\Psi_2, \Gamma_2', \mathbf{pc}_2']$ such that:

$$\rhd \Pi_1' \approx_\zeta \Pi_2' : [\Psi_1, \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2, \Gamma_2', \mathbf{pc}_2'] \text{ machConfig}.$$

The proof first considers one step reduction sequences and then weaves these together by means of an inductive argument on the length of the reduction sequences. Moreover, two kinds of one step reduction steps are considered, one where the program counter is low (*Low PC Lemma*) and one where the program counter is high (*High PC Lemma*). The proof of the Low PC Lemma does not present difficulties. It consists of showing that each step of the first machine configuration Π_1 can be mimicked by one step of the second machine configuration Π_2 such that ζ-indistinguishable machine configurations are reached.

Lemma 1 (Low PC Lemma). *Given machine configurations* $\Pi_i = (H_i, R_i, B_i)$ *and machine types* $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$, $i \in 1..2$. *Suppose* $\rhd\Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ $\mathsf{machConfig}$, $\mathbf{pc}_1 \sqsubseteq \zeta$ *and* $\mathbf{pc}_2 \sqsubseteq \zeta$, *and* $\Pi_1 \longrightarrow \Pi_1'$. *Then there exists a machine configuration* Π_2' *and machine configuration types* $\Omega_1' = [\Psi_1, \Gamma_1', \mathbf{pc}_1']$ *and* $\Omega_2' = [\Psi_2, \Gamma_2', \mathbf{pc}_2']$ *such that* $\Pi_2 \longrightarrow \Pi_2'$ *and* $\rhd\Pi_1' \approx_\zeta \Pi_2' : [\Psi_1, \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2, \Gamma_2', \mathbf{pc}_2']$ $\mathsf{machConfig}$.

On the other hand, the key case in the proof of the High PC Lemma is when the reduction step $\Pi_1 \longrightarrow \Pi_1'$ lowers the level of the program counter by jumping to a junction point with low level program counter. A machine configuration Π_2' must be found such that $\Pi_2 \twoheadrightarrow \Pi_2'$ and such that Π_1' and Π_2' are ζ-indistinguishable. The main obstacle is how to guarantee that the execution stacks of Π_1 and Π_2, previously *high* indistinguishable and possibly of different sizes, are now *low* indistinguishable and of the same size. Since we started off with a low security program counter (cf. statement of Non-Interference Theorem) we know that the stacks of Π_1 and Π_2 have a common, low indistinguishable substack. The point is that we must make sure that this substack becomes the current stack when the junction point is jumped to. This is possible because junction points are part of the execution stack types. More precisely, when $\Pi_1 \longrightarrow \Pi_1'$ jumps to a junction point L, it must be the case that the type of the execution stack of Π_1 is of the form $L \cdot \Sigma_1$. Furthermore, from the fact that the program counter in the type of $\Psi(L)$ is low and $\rhd_{\Psi_1,\Psi_k} S_1 \approx_\zeta S_2 : L \cdot \Sigma_1 \wedge \Sigma_2$ $\mathsf{estackHigh}$, we deduce that, $\Sigma_2 = ? \cdot \ldots \cdot ? \cdot L \cdot \Sigma_2'$ and $S_2 = w_1 \cdot \ldots \cdot w_m \cdot S_2'$, $m \leq n$, where the question marks "?" may either be junction points, nonsense types or security types. Moreover,

$$\rhd_{\Psi_1,\Psi_k} S_1 \approx_\zeta S_2' : \Sigma_1 \wedge \Sigma_2' \ \mathsf{estackLow} \tag{1}$$

must hold by the definition of the **estackHigh** judgment. The fact that these question marks are high level types is necessary and is guaranteed by the following definition and result:

Definition 2. *An execution stack type* Σ *is said to be* ζ*-topped in* Σ', *if there exist labels* $L_1, \ldots, L_n$ *(possibly none) and stack component types* $\hat{\sigma}_{i,1}, \ldots, \hat{\sigma}_{i,k_i}$, $i \in 1..n$ *(possibly none) such that:*

- $\Sigma' = \hat{\sigma}_{1,1} \cdot \ldots \cdot \hat{\sigma}_{1,k_1} \cdot L_1 \cdot \hat{\sigma}_{2,1} \cdot \ldots \cdot \hat{\sigma}_{2,k_2} \cdot L_2 \cdot \ldots \cdot \hat{\sigma}_{n,1} \cdot \ldots \cdot \hat{\sigma}_{n,k_n} \cdot L_n \cdot \Sigma$,
- $\Psi(L_i) = \textsc{Code}\langle \forall [X_i] \Gamma_i \mid \mathbf{pc}_i \rangle^{l_i}$ *implies* $\mathbf{pc}_i \sqcup l_i \not\sqsubseteq \zeta$, *for all* $1 \leq i \leq n$, *and*
- $\mathtt{label}(\hat{\sigma}_{ij}) \not\sqsubseteq \zeta$, *for all* $1 \leq i \leq n$ *and* $1 \leq j \leq k_i$.

Lemma 2 (High-Step Invariant). *For $i \in 1..k$, assume that machine configurations $\Pi_i = (H_i, R_i, B_i)$ and machine types $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$ are such that:*

1. $\Pi_1 \twoheadrightarrow \Pi_k$,
2. $\triangleright \Pi_i : \Omega_i$ machConfig, *$i \in 1..k$, and Ω_i is given by the Subject Reduction Theorem, for $i \in 2..k$,*
3. $\mathbf{pc}_1 \not\sqsubseteq \zeta$, *and*
4. $\Gamma_k(\mathtt{sp}) = L \cdot \Sigma_k$, *for some L and Σ_k, is ζ-topped in $\Gamma_i(\mathtt{sp})$, for each $i \in 1..k$.*

Then all of the following hold: $\triangleright H_1 \approx_\zeta H_k : \Psi_1 \wedge \Psi_k$ heap, $\triangleright_{\Psi_1,\Psi_k} R_1 \approx_\zeta R_k : \Gamma_1 \wedge \Gamma_k$ regBank, $\triangleright_{\Psi_1,\Psi_k} R_1(\mathtt{sp}) \approx_\zeta R_k(\mathtt{sp}) : \Gamma_1(\mathtt{sp}) \wedge \Gamma_k(\mathtt{sp})$ estackHigh, *and* $\mathbf{pc}_i \not\sqsubseteq \zeta$, *for all* $1 \leq i \leq k$.

Thus, if we know that the reduction starting from Π_2 terminates, we obtain the desired result that at some point the junction point L is invoked by a machine state reachable from Π_2. At this point, according to (1), the machine configurations "synchronize" at a low security level program counter. The proof of the High PC Lemma proceeds by case analysis on the definition of $\Pi_1 \longrightarrow \Pi_1'$, using the High Step Invariant Lemma in the case that this reduction step is a jump to a junction point that resets the program counter to low security level.

Lemma 3 (High PC Lemma). *For $i \in 1..2$, consider machine configurations $\Pi_i = (H_i, R_i, B_i)$ and machine types $\Omega_i = [\Psi_i, \Gamma_i, \mathbf{pc}_i]$. Suppose $\triangleright \Pi_1 \approx_\zeta \Pi_2 : \Omega_1 \wedge \Omega_2$ machConfig, $\mathbf{pc}_1 \not\sqsubseteq \zeta$ and $\mathbf{pc}_2 \not\sqsubseteq \zeta$, $\Pi_1 \longrightarrow \Pi_1'$, and Π_2 terminates. Then there exist a machine configuration Π_2' and machine configuration types $\Omega_1' = [\Psi_1, \Gamma_1', \mathbf{pc}_1']$ and $\Omega_2' : [\Psi_2, \Gamma_2', \mathbf{pc}_2']$ such that $\Pi_2 \twoheadrightarrow \Pi_2'$ and $\triangleright \Pi_1' \approx_\zeta \Pi_2' : [\Psi_1, \Gamma_1', \mathbf{pc}_1'] \wedge [\Psi_2, \Gamma_2', \mathbf{pc}_2']$ machConfig.*

Finally, the proof of the Non-interference Theorem (Theorem 1) follows by weaving reduction steps whose departing machine configurations have a low or high security level program counter using the Low PC or the High PC Lemmas (Lemmas 1 and 3), respectively.

4 Conclusions, Related Work and Future Research

We present a TAL with a polymorphic execution stack, a type system enforcing secure information flow, and a proof of non-interference. The problems stemming from the absence of control flow constructs and the challenges raised by polymorphic execution stacks are addressed by including explicit junction points in types and introducing appropriate type directives (`pushJP` and `jmpJP`) that manipulate them. As an added benefit, we are able to ensure that junction points are treated as linear continuations and that pending junction points are passed on as obligations. Since `pushJP` is a type directive, it may be eliminated during execution, while `jmpJP` may be replaced by a standard jump instruction at runtime. The type system keeps track of two stacks: the execution stack type and the junction points stack. In general, two separate stacks cannot be combined into one; however, in this case the type system enforces a discipline that allows

this combination, where a jump to a junction point L can only be done if L is at the top of the execution stack type.

Information flow analysis has been an active research area in the past three decades [22]. Pioneering work by Bell and LaPadula [4], Feiertag et al. [12], Denning and Denning [10, 11], Neumann et al. [21], 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. A subject cannot read objects of level higher than its level, and it cannot write objects at levels lower than its level.

Non-interference was first introduced by Goguen and Meseguer [13], and there has been a significant amount of research on type systems for confidentiality for high-level languages, including Volpano and Smith [24] and Banerjee and Naumann [2]. Type systems for low-level languages have been an active subject of study for several years now, including TAL [17], STAL [16], DTAL [25], Alias Types [23], and HBAL [1].

In his PhD thesis [20], Necula already suggests information flow analysis as an open research area at the assembly language level. Zdancewic and Myers [28] present a low-level, secure calculus with ordered linear continuations. This low-level calculus, like the calculus developed by Crary et al. [9], possesses high-level control flow structures (such as `if-then-else`) that simplify the analysis but require an extra, unanalyzed, translation to obtain a real low-level executable program. Moreover, none of these calculi includes a register bank or an execution stack. Barthe et al. [3] define a JVM-like low-level language with a heap and an operand stack. Instead of expressing the control dependence regions in the language, as in SIFTAL, this work assumes the existence of trusted functions that obtain such regions. Moreover, when a high branch is executed, the security level of all the elements on the stack is raised and is never lowered back, even when the execution returns to a low-security region.

We have recently learned from personal communication with Dachuan Yu about independent work on information flow analysis for `TAL-c` [26], a calculus similar in spirit to SIFTAL. Based on a preliminary manuscript we can identify differences in the definitions of equivalence of machine configurations, where, for example, their definition forces stacks to be of equal length, preventing the stack from being manipulated in a high branch. `TAL-c` has primitives to raise and lower the security level of the **pc** that delimit security regions, similar to our `pushJP` and `jmpJP`. However, the interaction of these primitives with stack type variables may potentially pull such variables beyond their scope, unless some stringent closure condition is required on typing contexts.

We are currently developing a type preserving compilation scheme from a high-level imperative language to SIFTAL, and studying unrestricted *register reuse*. The community's opinion is divided on whether registers are observable or not. If they are, then the reuse of a register to store data of lower security level may be seen as a leak of information, even if the data itself is not accessible. Although SIFTAL allows the reuse of registers, the security level of a register

remains fixed throughout execution. Lifting this restriction is the subject of current research.

Recent developments [27, 8, 18, 19] argue that mechanisms enforcing the absence of illegal information flows are too drastic to be practical. They study high-level languages with *declassification*, a controlled form of sidestepping of confidentiality policies. A notion of declassification for TALs is required for type preserving compilation of such languages. However, this area remains unexplored.

Acknowledgments. We are grateful to Pablo Garralda, Healfdene Goguen, David Naumann, and Alejandro Russo for enlightening discussions. We also thank Joëlle Despeyroux and Dachuan Yu for comments on earlier drafts. 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. D. Aspinall and A. 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. E. Bonelli, A. Compagnoni, and R. Medel. Information flow analysis for a typed assembly language with polymorphic stacks. `http://www.cs.stevens.edu/~rmedel/siftalTechReport.ps`, 2005.
7. E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A typed assembly language for secure information flow analysis. `http://www.cs.stevens.edu/~rmedel/techReport.ps`, 2005.
8. T. Chothia, D. Duggan, and J. Vitek. Type-based distributed access control. In *Proc. of IEEE Computer Security Foundations Workshop*, Asilomar, California, 2003.
9. K. Crary, A. Kliger, and F. Pfenning. A monadic analysis of information flow security with mutable state. Technical Report CMU-CS-03-164, Carnegie Mellon University, September 2003.
10. D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–242, May 1976.
11. D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

12. 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.
13. 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.
14. D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. In *Proceedings of the First Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005)*, volume 141(1) of *Electronic Notes in Theoretical Computer Science*, pages 163–182, December 2005.
15. R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for noninterference. In M. Coppo, E. Lodi, and G. M. Pinna, editors, *Ninth Italian Conference on Theoretical Computer Science - ICTCS 2005*, volume 3701 of *LNCS*, pages 360–374, Certosa di Pontignano, Siena (Italy), October 2005. Springer.
16. G. Morrisett, K. Crary, N. Glew, and D. 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.
17. 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. This is the expanded version of a paper that appeared in Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 85-97, San Diego, CA, USA, January 1998.
18. A. Myers and A. Sabelfeld. A model for delimited information release. In *International Symposium on Software Security*, volume 3233 of *LNCS*, Tokyo, Japan, 2003.
19. A. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification. 7th IEEE Computer Security Foundations Workshop, 2004.
20. G. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998.
21. P. G. Neumman, R. J. Feiertag, K. N. Levitt, and L. 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.
22. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.
23. F. Smith, D. Walker, and G. Morrisett. Alias types. In Gert Smolka, editor, *Ninth European Symposium on Programming*, volume 1782 of *LNCS*, pages 366–381. Springer-Verlag, April 2000.
24. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT*, pages 607–621, 1997.
25. H. Xi and R. Harper. A dependently typed assembly language. Technical Report OGI-CSE-99-008, Oregon Graduate Institute of Science and Technology, July 1999.
26. D. Yu and N. Islam. A typed assembly language for confidentiality. Personal Communication, July 2005.
27. S. Zdancewic and A. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.
28. S. Zdancewic and A. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3), 2002.