

# More Typed Assembly Languages for Confidentiality

DoCoMo USA Labs Technical Report DCL-TR-2006-0021

Dachuan Yu

DoCoMo Communications Laboratories USA  
yu@docomolabs-usa.com

**Abstract.** We propose a series of type systems for the information-flow security of assembly code. These systems extend previous work  $TAL_C$  with some timing annotations and associated judgments and rules. By using different timing rules, these systems are applicable to different practical settings. In particular, they can be used to prevent illicit information flow through the termination and timing channels in sequential programs as well as the possibilistic and probabilistic channels in multi-threaded programs. We present the formal details of these as a generic type system  $TAL_C^+$  and prove its noninterference.  $TAL_C^+$  is designed as a core target language for certifying compilation. We illustrate its use with a formal scheme of type-preserving translation.

## 1 Introduction

Language-based techniques are promising in enforcing information-flow security [19]. An information-flow problem typically concerns a program which operates on data of different security levels, *e.g.*, *low* and *high*. Low data are public data that may be disclosed to all principals; high data are secret data whose access must be restricted. An information-flow policy, such as noninterference, typically requires that no information about high input data can be inferred from observing low output data. In practice, the security levels can be generalized to a lattice [26].

Whereas most existing work on information flow has focused on high-level languages, low-level languages are gradually receiving more attention, especially from the perspective of typed assembly languages [13] and for the purpose of certifying compilation [10, 21]. In recent work [12, 30], type annotations are used to restore the missing abstraction in assembly code, and type-preserving translation is used to preserve security evidence from the source to the target. By security-type checking directly at the target level, the compiler is lifted out of the trusted computing base, yielding higher confidence on the code behavior. In addition, in the context of mobile code security where code providers are potentially malicious and source code is typically not available for analysis, certifying compilation allows code producers to provide security evidence in the form of types and other annotations to accompany the target code for separate verification on the code consumer's side.

Although a good start, such low-level information-flow analysis only investigated relatively simple settings. In particular, only sequential programs were considered, and attackers were assumed to be unable to observe beyond the regular machine states (*e.g.*,

heaps and register files). In practice, however, attackers may also be able to observe some timing behaviors, either by observing physically the execution time of a program (external timing) or by exploiting thread interaction (internal timing). Such timing behaviors, unfortunately, provide some covert channels of information flow that could be easily exploited.

This paper presents solutions that address information-flow security for low-level languages in more practical settings. For sequential programs, we propose type systems for closing up some covert channels, namely termination [25] and timing [1] channels. For multi-threaded programs, we propose type systems that guarantee possibilistic [23] and probabilistic [20] noninterference. Our solutions are inspired by source-level security-type systems which account for program timing behaviors. Based on our previous work [30] on a typed assembly language for confidentiality ( $TAL_C$ ), we introduce additional annotations to document the timing of program execution. These timing annotations are used to express various source-level constraints, such as the absence of loops in high security contexts.

We observed that different timing annotations were needed to address different information-flow channels. Nonetheless, there was much similarity between the underlying type systems. We have formulated a generic type system  $TAL_C^+$ , which addresses different information-flow channels when given different “timing rules.” The formal results of this paper is given in terms of the generic  $TAL_C^+$ . By giving  $TAL_C^+$  different “parameters,” we obtain specialized systems for different situations. As expected, the trivial “empty” parameter reduces the generic  $TAL_C^+$  to the original  $TAL_C$ .

An advantage of the generic treatment, and of  $TAL_C^+$ , lies in simplicity. The extensions we propose are self-contained and easy to understand, yet they suffice in addressing all the above mentioned practical settings. They also interact naturally with advanced features of typed assembly languages. These enable the potential use of  $TAL_C^+$  in a more sophisticated typed assembly language, which in turn may serve as a general target for certifying compilation. The smooth transition from  $TAL_C$  to  $TAL_C^+$  also serves as a validation that  $TAL_C$  is a good platform for low-level information-flow analysis.

The remainder of this paper is organized as follows. Section 2 provides background on security-type systems at a source level and at a target level. Section 3 illustrates the problems introduced by termination channels and outlines the ideas behind our solution. A rigorous treatment is given in Section 4, where we present our type system for closing up termination channels, prove its noninterference, and detail a certifying compilation scheme targeting it. Although focusing on termination channels for ease of understanding, the formal details in Section 4 are organized generically so that later sections simply provide different “parameters” to obtain type systems and certifying compilation schemes for addressing other information-flow channels. Following this generic treatment, Section 5 discusses timing channels, and Section 6 discusses possibilistic and probabilistic noninterference in multi-threaded programs. Finally, Section 7 discusses related and future work, and Section 8 concludes.

*(Types)*  $\tau, \text{pc} ::= \text{low} \mid \text{high}$   
*(Env's)*  $\Phi ::= \circ \mid V : \tau, \Phi$   
*(Exp's)*  $E ::= i \mid V \mid E_1 + E_2$   
*(Cmd's)*  $C ::= \text{skip} \mid V := E \mid C_1; C_2 \mid \text{if } E \text{ then } C_1 \text{ else } C_2 \mid \text{while } E \text{ do } C$

**Fig. 1.** A simple imperative language

## 2 Background

### 2.1 Security-Type System

Consider the simple language in Figure 1. We use two security levels in this language: low and high. An environment ( $\Phi$ ) specifies the security levels ( $\tau$ ) of variables ( $V$ ). For convenience, we sometimes use  $h$  (and  $h_1, h_2, \dots$ ) and  $l$  (and  $l_1, l_2, \dots$ ) to denote high and low variables respectively. Expressions ( $E$ ) are either constants ( $i$ ), variables ( $V$ ) or additions. Commands ( $C$ ) are all common. We wish to enforce a policy that there should be no flow of information from high variables to low variables. More specifically, we must prevent several kinds of illicit information flow:

1. Explicit flow through assignments, such as  $l := h$ .
2. Implicit flow through program structures (conditionals and while-loops), such as  $\text{if } h \text{ then } l := 1 \text{ else } l := 0$ .
3. If extended with first-class objects and/or functions, there will be additional implicit flow through data and/or code pointers.

A security-type system typically addresses these requirements by:

1. To track security levels in the types of expressions. This helps preventing information leak through explicit flow—mismatch of security levels in assignments will be disallowed.
2. To mark the program counters (PC) with security levels. If a conditional expression has a high guard, its branches will be type-checked under a high PC. This essentially marks the branches as falling in a “sensitive region,” under which updates to low variables are disallowed. This idea applies also to other program structures such as while-loops.
3. In languages with pointers, pointers are given two security levels—one for the pointer, the other for the data or code being referenced. Illicit flow through pointers can be prevented with appropriate checks on the two security levels when the pointers are accessed.

Figure 2 shows a simple security-type system that reflects the first two of these ideas. The rules there are self-explanatory. Note that Rule [C4] (subsumption) helps to allow the inclusion of high computations in low security contexts—if it is secure to execute some code in a sensitive context, then it is also secure to execute the same code in an insensitive context.

[E1–2]	$\Phi \vdash E : \text{high}$	$\Phi \vdash i : t$
[E3–4]	$\frac{\Phi(V) = \text{low}}{\Phi \vdash V : \text{low}}$	$\frac{\vdash E_1 : \text{low} \vdash E_2 : \text{low}}{\vdash E_1 + E_2 : \text{low}}$
[C1–2]	$\frac{\Phi(V) = \text{high}}{\Phi; [\text{pc}] \vdash V := E}$	$\frac{\Phi(V) = \text{low} \quad \Phi \vdash E : \text{low}}{\Phi; [\text{low}] \vdash V := E}$
[C3]	$\frac{\Phi \vdash E : \text{pc} \quad \Phi; [\text{pc}] \vdash C_1 \quad \Phi; [\text{pc}] \vdash C_2}{\Phi; [\text{pc}] \vdash \text{if } E \text{ then } C_1 \text{ else } C_2}$	
[C4–5]	$\frac{\Phi; [\text{high}] \vdash C}{\Phi; [\text{low}] \vdash C}$	$\frac{\Phi; [\text{pc}] \vdash C_1 \quad \Phi; [\text{pc}] \vdash C_2}{\Phi; [\text{pc}] \vdash C_1; C_2}$
[C6]	$\frac{\Phi \vdash E : \text{pc} \quad \Phi; [\text{pc}] \vdash C}{\Phi; [\text{pc}] \vdash \text{while } E \text{ do } C}$	

**Fig. 2.** A simple security-type system

## 2.2 $TAL_C$

$TAL_C$  [30] adapts source-level information-flow analysis for assembly code. The main challenge there is that assembly code does not present as much abstraction as does source code. Whereas the program structures of source code help to determine the security levels of various program points, such structures are not available in assembly code.

For example, a conditional statement in a source program can be type-checked so that both branches respect the security level of the guard expression. Such checks become difficult in assembly code, where the “flattened” control flow provides little help in identifying the program structure. A conditional is typically translated into a branching instruction and some code blocks, where the ending points of the two branches are no longer apparent. Although control-flow analysis can be used to compute such information [3], it expands the trusted computing base substantially—the analysis itself would be trusted.

$TAL_C$  recovers the missing structure using type annotations. These annotations provide explicit information on the security levels of instructions as well as the ending points of the security levels. Two new security operations are introduced to manipulate the annotations, and appropriate typing rules are used to make sure that the annotations faithfully reflect the actual structure of the code.

The syntax of a simplified version of  $TAL_C$  is given in Figure 3. Compared with the original  $TAL_C$ , we have omitted stacks and polymorphism because they are mainly for supporting functions and procedures in source programs and their handling is orthogonal to the  $TAL_C$  extensions in this paper. We use  $\theta$  to represent security levels. A security context  $\theta \triangleright l$  indicates that the security level of the PC will be no less than  $\theta$  until the program point  $l$  is reached.

A `raise  $\kappa$`  operation is used to increase the security level, which corresponds to the beginning of a sensitive region such as those introduced by a high conditional. A `lower  $l$`  is used to restore the security level and transfer the control to label  $l$ , which

(contexts)	$\kappa ::= \bullet \mid \theta \triangleright l$
(pre-types)	$\tau ::= \mathbf{int} \mid \langle \sigma_1, \dots, \sigma_n \rangle \mid \forall [] . \langle \kappa \rangle I$
(types)	$\sigma ::= \tau_\theta$
(heap ty)	$\Psi ::= \{l_1 : \sigma_1, \dots, l_n : \sigma_n\}$
(reg file ty)	$\Gamma ::= \{r_1 : \sigma_1, \dots, r_n : \sigma_n\}$
(registers)	$r ::= r_1 \mid r_2 \mid \dots$
(word val)	$w ::= l \mid i$
(small val)	$v ::= r \mid w$
(heap val)	$h ::= \langle w_1, \dots, w_n \rangle \mid \mathbf{code}[] \langle \kappa \rangle \Gamma . I$
(heaps)	$H ::= \{l_1 \mapsto h_1, \dots, l_n \mapsto h_n\}$
(reg files)	$R ::= \{r_1 \mapsto w_1, \dots, r_n \mapsto w_n\}$
(instr)	$\iota ::= \mathbf{add} \ r_d, r_s, v \mid \mathbf{ld} \ r_d, r_s(i) \mid \mathbf{st} \ r_d(i), r_s \mid \mathbf{mov} \ r_d, v \mid \mathbf{bnz} \ r, v \mid \mathbf{raise} \ \kappa$
(instr seq)	$I ::= \iota; I \mid \mathbf{lower} \ l \mid \mathbf{jmp} \ v \mid \mathbf{halt} \ [\sigma]$
(prog)	$P ::= (H, R, I)_\kappa$

**Fig. 3.** Syntax of a simplified  $TAL_C$

marks the end of a sensitive region. These two security operations can be placed into the target code by a certifying compilation process based on the structures and the typing of source programs.

Given these, the security levels of all program points become apparent through the annotations. The adaptation of the source level solutions is then straightforward. High branching must happen in high contexts, and updates to low data in high contexts are disallowed. In addition, every pointer is given two security levels, following the same treatment for source code.

In Figure 4, we give an example  $TAL_C$  program, which is the target code translated from a security-typed source program. The source program involves a low variable  $a$  and two high variables  $b$  and  $c$ . In the corresponding  $TAL_C$  program, we use heap cells labeled  $l_a$ ,  $l_b$  and  $l_c$  to represent these variables. The  $TAL_C$  program starts from the code labeled  $l_0$  in a low security context  $\bullet$ . After the initial setup, it raises the security context to  $\top \triangleright l_3$ , where  $\top$  represents the security level high. The control is then transferred to the code labeled  $l_1$ , which contains a test on the high variable  $b$  and directs the execution to two separate branches. In either branch of the conditional, the high variable  $c$  is updated, and the security context is restored with lower  $l_3$ . The code at  $l_3$  is then free to update the low variable  $a$  again.

This concludes our introduction to  $TAL_C$ . Interested readers are referred to previous work [29, 30] for more details, including the support for functions and pointers, the exact semantics and noninterference proof, and a certifying compilation scheme targeting  $TAL_C$ .

### 3 Termination Channels

$TAL_C$  assumes that the only means for observing program behaviors is by inspecting the content of certain program variables (or heap and register file for assembly code). In

A security-typed source program:	<pre> a := 0; if b then c := 1 else c := 0; a := 1 </pre>
A corresponding $TAL_C$ program:	<pre> (<math>H, \{\}, \text{jmp } l_0</math>)•   where <math>H = \{(l_a, l_b, l_c \text{ omitted})</math> </pre>
$l_0 \mapsto$	<pre> code[](<math>\bullet</math>){}.   mov r0, 0;           % r0 ← 0   mov r1, l_a;        % r1 ← l_a   mov r2, l_b;        % r2 ← l_b   mov r3, l_c;        % r3 ← l_c   st r1(0), r0;       % a ← 0   raise <math>\top \triangleright l_3</math>; % raise security context   jmp l1 </pre>
$l_1 \mapsto$	<pre> code[](<math>\top \triangleright l_3</math>){<math>r_0 : \text{int}_\perp, r_1 : \langle \text{int}_\perp \rangle_\perp, r_2 : \langle \text{int}_\top \rangle_\perp, r_3 : \langle \text{int}_\top \rangle_\perp</math>}.   ld r4, r2(0);   bnz r4, l2;         % go to l2 if b is not 0   st r3(0), r0;       % the else branch: c ← 0   lower l3            % restore security context and go to l3 </pre>
$l_2 \mapsto$	<pre> code[](<math>\top \triangleright l_3</math>){<math>r_0 : \text{int}_\perp, r_1 : \langle \text{int}_\perp \rangle_\perp, r_2 : \langle \text{int}_\top \rangle_\perp, r_3 : \langle \text{int}_\top \rangle_\perp</math>}.   mov r0, 1;   st r3(0), r0;       % the then branch: c ← 1   lower l3            % restore security context and go to l3 </pre>
$l_3 \mapsto$	<pre> code[](<math>\bullet</math>){<math>r_1 : \langle \text{int}_\perp \rangle_\perp</math>}.   mov r0, 1;   st r1(0), r0;       % a ← 1   halt [<math>\langle \text{int}_\perp \rangle_\perp</math>] </pre>

**Fig. 4.**  $TAL_C$  example

practice, there are some covert channels beyond variables that deserve attention. This section focuses on the handling of termination channels. More specifically, we assume that an attacker could observe whether a program terminates.

Termination channels enable some attacks not prevented by  $TAL_C$ . In the following two examples, no low variable is updated. However, the values of the high variable  $h$  can be learned by observing whether the programs terminate.

1. Nonterminating high loop: `while h do skip`;
2. Nonterminating loop in a high branch: `if h then {while true do skip} else skip`.

### 3.1 Solution at a Source Level

Volpano and Smith [25] proposed to use “minimum typings” to close termination channels. They put restrictions on program constructs that could potentially lead to nontermination. In essence, if a program construct is potentially nonterminating, then it must have the minimum typing (one that corresponds to the lowest security level). In the language of Figure 1, the only such construct is while-loop. Therefore, all while-loops are restricted to have the type `low`. Note that this is a conservative approach, because even terminating loops are considered potentially nonterminating.

This is reflected in the type system as allowing a while-loop only under a low PC. More specifically, we need to change only one typing rule in the type system of Figure 2 so that the PC must be `low` for loops:

$$[C6'] \quad \frac{pc = low \quad \phi \vdash E : pc \quad \phi; [pc] \vdash C}{\phi; [pc] \vdash \text{while } E \text{ do } C}$$

This simple change closes termination channels. In the first example earlier, the high loop is now disallowed, because the high loop guard cannot be typed under a low PC. In the second example, the loop in the high conditional is now also disallowed, because the body of the high conditional has a high PC, but a loop is allowed only under a low PC.

The correctness of this approach with respect to termination is intuitive. Consider executing a program twice on different high variables but same low variables. Upon a high conditional, both executions will terminate because of the absence of loops in the conditional body. Upon a low conditional, both executions will follow the same path, resulting in the same behavior on termination.

### 3.2 Idea at an Assembly Level

In a typed assembly language, one cannot easily identify loops, because both loops and conditionals are implemented using branching instructions. Nonetheless, the essence of the source-level approach is that high branches must be terminating. The security annotations of  $TAL_C$  are handy for identifying such branches—a branch is high if and only if it has a high context. On top of this, we attach to high branches some new timing annotations that mark the upper bound of their execution steps before returning to low regions. The solution of  $TAL_C$  guarantees that terminating high branches will meet

eventually in a low region with matching low variables. The new timing annotations introduced here are to guarantee that high branches do terminate.

More specifically, we introduce a timing tag  $t$  for this purpose, where  $t$  is either a natural number (indicating that the execution will leave the current high region in at most this number of steps) or  $\infty$  (indicating “unknown” or “potentially infinite”). Under a high context, the type system allows a branching instruction only if both branches have finite timing tags. This prevents potentially nonterminating “backward jumps” later on. Note that when giving a timing tag to the branching instruction itself (needed for nested branches), one must take the longer branch into account.

The correctness intuition of this approach is similar to that of the minimum typings at a source level—upon a high branching, two executions of a program may split but will meet in a low region in a finite number of steps.

These timing annotations can come from the compilation of security-typed source code. A conditional will be compiled with finite timing if and only if both branches yield finite timing. Following Volpano and Smith [25], a loop can be compiled conservatively with infinite timing. By accepting only finite timing annotations in high regions, a type system would reject the two counter examples given earlier.

Such timing annotations work naturally with non-recursive loop-free functions, because the function bodies can be given finite timing annotations. To support code reuse, it is useful to introduce a simple form of polymorphism, as in the following example:

$$f : \forall m. \langle m + 3 \rangle \{ r_0 : \langle m \rangle \{ \} \} \\ \text{do something that costs } 2 \\ \langle m + 1 \rangle \text{ jmp } r_0$$

Similarly to the code polymorphism of TAL [13], this “post-timing” polymorphism essentially declares multiple copies of the same function for different typing contexts. Which copy to use is decided by a use site with a type instantiation.

In addition to non-termination, minimum typings have also been used to handle abnormal termination [25], such as uncaught exceptions caused by partial operations. The support for this in assembly code is straightforward by disallowing partial operations in high regions.

### 3.3 Allowing Terminating Loops

The key idea above conservatively disallows potentially nonterminating constructions in high contexts. It is sometimes useful to relax this restriction by allowing terminating loops (or well-founded recursions). This can be achieved using singleton types.

$$l : \forall [t : \text{nat} \mid t \geq 1]. \langle 2t \rangle \{ r_1 : \text{int}(t) \} \\ \text{sub } r_1, r_1, 1 \quad \% r_1 \leftarrow r_1 - 1 \\ \forall [t : \text{nat} \mid t \geq 1]. \langle 2t - 1 \rangle \{ r_1 : \text{int}(t - 1) \} \\ \text{bnz } r_1, l \quad \% \text{ to branch if } r_1 \neq 0 \\ \langle 0 \rangle \{ r_1 : \text{int}(0) \} \\ \text{halt } [\text{int}]$$

The above example code demonstrates some simple concepts of singleton types. In the type annotations,  $t$  is a type variable of kind (or sort) `nat` mimicking natural numbers, and  $t \geq 1$  is a constraint. The initial type annotation at label  $l$  essentially means that there exists a natural number  $t$  satisfying  $t \geq 1$  such that the “clock” (timing annotation) is  $2t$  and  $r_1$  has type `int( $t$ )` (an integer of value  $t$ ). Note how the clock depends on the value in  $r_1$ . In the next program point, the annotation reflects the execution of the subtraction instruction (`sub`). Interesting manipulation happens when type-checking the branching instruction (`bnz`), where the result of the comparison is used in establishing the typing of the next program points. In the case where  $r_1$  is not zero, a newly introduced constraint ( $t - 1 \neq 0$ ) would be used in establishing the typing annotation at the target  $l$  of the branching, with the type variable at  $l$  now instantiated with  $(t - 1)$ . In the case where  $r_1$  is zero, a constraint ( $t - 1 = 0$ ) would be used in establishing the typing annotation at `halt`.

In essence, singleton types are used here to connect the clock, a type-level concept, with values. More generally, the applications of singleton types have been detailed both in the context of resource bounds certification [7] and for a typed assembly language [28]. The work is applicable here with little adaptation. Since the use of singleton types is orthogonal to our main contribution (certifying compilation for information-flow analysis in certain practical settings), we omit its formal handling for a simpler exposition, instead briefly pointing out the potential use when introducing the typing rule of the branching instruction. In any case, the formal contents of this paper already suffice in supporting certifying compilation from related source-level systems.

## 4 $\text{TAL}_C^+$

Now we present a typed assembly language  $\text{TAL}_C^+$  following the above idea. In particular, we present  $\text{TAL}_C^+$  as an extension to  $\text{TAL}_C$  (the version in Figure 3 is used, where features orthogonal to timing behaviors are omitted as explained in Section 2.2). For ease of reading, we put the new additions in shaded boxes. By removing the shaded boxes, we get exactly  $\text{TAL}_C$ .

Although we are focusing on termination channels for now,  $\text{TAL}_C^+$  can also be used to prevent illicit flow through other channels. The details will follow in later sections.

### 4.1 Type System

Following  $\text{TAL}_C$ , we assume security labels form a lattice  $\mathcal{L}$ . We use  $\theta$  to range over elements of  $\mathcal{L}$ . We use  $\perp$  and  $\top$  as the bottom and top of the lattice,  $\cup$  and  $\cap$  as the lattice join and meet operations, and  $\subseteq$  as the lattice ordering.

The syntax extension is given in Figure 5. Timing annotations  $t$  are either natural numbers  $n$  or the special  $\infty$ . They accompany security contexts  $\kappa$  in code types and code values.

In Figure 6, typing judgments are extended for the timing annotations. Instruction sequences and programs are further checked with respect to  $t$ . We introduce four new judgment forms on timing annotations. By using different definitions of these four judgment forms, we obtain type systems for closing different information channels. For termination channels, the definitions are in Figure 7.

(*timing anno*)  $t ::= n$  (*natural numbers*)  $|\infty$   
 (*pre-type*)  $\tau ::= \text{int} \mid \langle \sigma_1, \dots, \sigma_n \rangle \mid \forall [] . \langle \kappa; t \rangle \Gamma$   
 (*heap val*)  $h ::= \langle w_1, \dots, w_n \rangle \mid \text{code} [] \langle \kappa; t \rangle \Gamma . I$

**Fig. 5.**  $\text{TAL}_C^+$  syntax

Judgment	Meaning
$\Gamma_1 \subseteq \Gamma_2$	Register file type $\Gamma_1$ weakens $\Gamma_2$
$\vdash H : \Psi$	Heap $H$ has type $\Psi$
$\Psi \vdash R : \Gamma$	Register file $R$ has type $\Gamma$
$\Psi \vdash h : \sigma$	Heap value $h$ has type $\sigma$
$\Psi \vdash w : \sigma$	Word value $w$ has type $\sigma$
$\Psi; \Gamma \vdash v : \sigma$	Small value $v$ has type $\sigma$
$\Psi; \Gamma; \kappa; t \vdash I$	$I$ is a valid sequence of instructions
$\Psi; \Gamma; t \vdash P$	$P$ is a valid program
$ comm  = n$	Command $comm$ requires time $n$
$\theta \vdash t \xrightarrow{t'} t''$	Timing may change from $t$ to $t''$ after $t'$
$\theta \vdash t$	Timing $t$ is OK under security level $\theta$
$\theta \vdash t \sim t'$	$t$ and $t'$ match under security level $\theta$

**Fig. 6.**  $\text{TAL}_C^+$  typing judgments

$$\begin{array}{c}
 comm \in \{\text{add, ld, st, mov, bnz, raise, lower, jmp, halt}\} \\
 \hline
 |comm| = 1 \\
 \frac{}{\perp \vdash t \xrightarrow{t'} t''} \quad \frac{t \geq t' + t''}{\theta \vdash t \xrightarrow{t'} t''} \\
 \frac{}{\perp \vdash t} \quad \frac{t \neq \infty}{\theta \vdash t} \quad \frac{}{\theta \vdash t \sim t'}
 \end{array}$$

**Fig. 7.** Timing rules on termination channels

$|comm|$  is designed to track the progress of time during program execution. With respect to termination, it suffices to consider any assembly command (or security operation) as consuming one unit of time. The time passage  $\theta \vdash t \xrightarrow{t'} t''$  is irrelevant in low security contexts  $\perp$ . In other security contexts, it requires that  $t$  be no less than the sum of  $t'$  and  $t''$ ; it is used in the typing rules to ensure that the timing tag is monotonically decreasing with respect to the control flow. Here the addition  $+$  is extended straightforwardly to work with  $\infty$ . This judgment may seem unwieldy when  $\infty$  is involved. Fortunately, a premise in a rule of well-typed programs, to be shown later, will prevent  $\infty$  from being used in high contexts for well-typed programs.

Two rules are used for establishing the judgment of valid timing  $\theta \vdash t$ . Any timing is valid under a low security context  $\perp$ . In contrast, under high security contexts, a timing is valid if and only if it is finite. Finally, we define matching timing  $\theta \vdash t \sim t'$  to hold trivially, since it does not play a role for termination channels. It is a placeholder for the extensions in later sections.

$$\begin{array}{c}
\frac{m \leq n}{\{r_1 : \sigma_1 \dots r_m : \sigma_m\} \subseteq \{r_1 : \sigma_1 \dots r_n : \sigma_n\}} \\
\frac{\Psi = \{l_1 : \sigma_1, \dots, l_n : \sigma_n\} \quad \Psi \vdash h_i : \sigma_i}{\vdash \{l_1 \mapsto h_1, \dots, l_n \mapsto h_n\} : \Psi} \\
\frac{\Gamma = \{r_1 : \sigma_1, \dots, r_n : \sigma_n\} \quad \Psi \vdash w_i : \sigma_i}{\Psi \vdash \{r_1 \mapsto w_1, \dots, r_n \mapsto w_n\} : \Gamma} \\
\frac{\Psi \vdash w_i : \sigma_i}{\Psi \vdash \langle w_1, \dots, w_n \rangle : \langle \sigma_1, \dots, \sigma_n \rangle_\theta} \\
\frac{\Psi; \Gamma; \kappa; \bar{t} \vdash I}{\Psi \vdash \text{code}[\kappa; \bar{t}] \Gamma.I : (\forall \square. \langle \kappa; \bar{t} \rangle \Gamma)_\theta} \\
\frac{\Psi \vdash i : \text{int}_\theta \quad \frac{\Psi(l) = \sigma}{\Psi \vdash l : \sigma}}{\frac{\Gamma(r) = \sigma}{\Psi; \Gamma \vdash r : \sigma} \quad \frac{\Psi \vdash w : \sigma}{\Psi; \Gamma \vdash w : \sigma}} \\
\frac{\vdash H : \Psi \quad \Psi \vdash R : \Gamma \quad \Psi; \Gamma; \kappa; \bar{t} \vdash I \quad \boxed{SL(\kappa) \vdash t}}{\Psi; \Gamma; \bar{t} \vdash (H, R, I)_\kappa}
\end{array}$$

**Fig. 8.**  $\text{TAL}_C^+$  typing rules: non-instructions

The extended typing rules are in Figures 8 and 9. For understanding the timing related aspects, it suffices to focus on the shaded boxes. Interested readers are referred to previous work [30] for details of the  $\text{TAL}_C$  type system.

In  $\text{TAL}_C^+$ , only a few changes are made to  $\text{TAL}_C$  on the rules for non-instructions. Notably, there is an extra invariant in the rule for checking programs, namely the timing  $t$  must be valid with respect to the security level of the context  $\kappa$ . Here we use  $SL(\kappa)$

$$\frac{SL(\kappa) = \theta \quad \Gamma(r_s) = \text{int}_{\theta_1} \quad \Psi; \Gamma \vdash v : \text{int}_{\theta_2} \quad \Psi; \Gamma\{r_d : \text{int}_{\theta \cup \theta_1 \cup \theta_2}\}; \kappa; t' \vdash I \quad \theta \vdash t \xrightarrow{|\text{add}|} t'}{\Psi; \Gamma; \kappa; t \vdash \text{add } r_d, r_s, v; I}$$

$$\frac{SL(\kappa) = \theta \quad \Gamma(r_s) = \langle \sigma_1, \dots, \sigma_n \rangle_{\theta_1} \quad \sigma_i = \tau_{\theta_2} \quad \Psi; \Gamma\{r_d : \tau_{\theta \cup \theta_1 \cup \theta_2}\}; \kappa; t' \vdash I \quad \theta \vdash t \xrightarrow{|\text{ld}|} t'}{\Psi; \Gamma; \kappa; t \vdash \text{ld } r_d, r_s(i); I}$$

$$\frac{SL(\kappa) = \theta \quad \Psi; \Gamma \vdash v : \tau_{\theta'} \quad \Psi; \Gamma\{r_d : \tau_{\theta \cup \theta'}\}; \kappa; t' \vdash I \quad \theta \vdash t \xrightarrow{|\text{mov}|} t'}{\Psi; \Gamma; \kappa; t \vdash \text{mov } r_d, v; I}$$

$$\frac{SL(\kappa) = \theta \quad \Gamma(r) = \text{int}_{\theta_1} \quad \Psi; \Gamma \vdash v : (\forall[].\langle \kappa; t' \rangle \Gamma')_{\theta_2} \quad \theta_1 \cup \theta_2 \subseteq \theta \quad \Gamma' \subseteq \Gamma \quad \Psi; \Gamma; \kappa; t'' \vdash I \quad \theta \vdash t \xrightarrow{|\text{bnz}|} t' \quad \theta \vdash t \xrightarrow{|\text{bnz}|} t'' \quad \theta \vdash t' \sim t''}{\Psi; \Gamma; \kappa; t \vdash \text{bnz } r, v; I}$$

$$\frac{SL(\kappa) = \theta \quad \Gamma(r_d) = \langle \sigma_1, \dots, \sigma_n \rangle_{\theta_1} \quad \sigma_i = \tau_{\theta'} \quad \Gamma(r_s) = \tau_{\theta_2} \quad \theta \cup \theta_1 \cup \theta_2 \subseteq \theta' \quad \Psi; \Gamma; \kappa; t' \vdash I \quad \theta \vdash t \xrightarrow{|\text{st}|} t'}{\Psi; \Gamma; \kappa; t \vdash \text{st } r_d(i), r_s; I}$$

$$\frac{SL(\kappa) = \theta \quad \kappa' = \theta' \triangleright w' \quad \theta \subseteq \theta' \quad \Psi \vdash w' : (\forall[].\langle \kappa; t_1 \rangle \Gamma')_{\theta_1} \quad \Psi; \Gamma; \kappa'; t' \vdash I \quad \theta \vdash t \xrightarrow{|\text{raise}|} t' \quad \theta' \vdash t'}{\Psi; \Gamma; \kappa; t \vdash \text{raise } \kappa'; I}$$

$$\frac{\kappa = \theta \triangleright w \quad \Psi \vdash w : (\forall[].\langle \kappa'; t' \rangle \Gamma')_{\theta_1} \quad \theta_1 \subseteq SL(\kappa') \quad \Gamma' \subseteq \Gamma \quad \theta \vdash t \xrightarrow{|\text{lower}|} t_1 \quad \text{where } t_1 = \begin{cases} 0 & \text{if } SL(\kappa') = \perp \\ t' & \text{otherwise} \end{cases}}{\Psi; \Gamma; \kappa; t \vdash \text{lower } w}$$

$$\frac{SL(\kappa) = \theta \quad \Psi; \Gamma \vdash v : (\forall[].\langle \kappa; t' \rangle \Gamma')_{\theta_1} \quad \theta_1 \subseteq \theta \quad \Gamma' \subseteq \Gamma \quad \theta \vdash t \xrightarrow{|\text{jmp}|} t'}{\Psi; \Gamma; \kappa; t \vdash \text{jmp } v}$$

$$\frac{\kappa = \bullet \quad \Gamma(r_1) = \sigma}{\Psi; \Gamma; \kappa; t \vdash \text{halt } [\sigma]}$$

**Fig. 9.** TAL<sub>C</sub><sup>+</sup> typing rules: instructions

to refer to the security label component of  $\kappa$ ;  $SL(\bullet)$  is defined to be  $\perp$ . This timing invariant ensures that  $\infty$  cannot be used to type program points in high contexts, even though the previously shown judgment of time passage is permissive in the case of  $\infty$ .

The rules for `add`, `ld`, `mov` and `st` are all extended in the same way. An extra check on the time passage is used to ensure that the timing annotations match the instructions.

Instruction `bzz` has two potential successors. Therefore, both branches must be checked with respect to the time passage. The matching judgment trivially holds for `now`. Note that if singleton types are used to allow terminating loops, an instantiation of the timing annotation at the code label  $v$  would be required, and the comparison result on the register  $r$  (whose type may be dependent on a value related to the timing annotation) would be used together with existing constraints to establish the judgments on time passage.

Upon entering a new security context marked by `raise`, we check that the new timing  $t'$  is valid under the new security level  $\theta'$ . In addition, the time passage is also checked; this is useful in a multi-level security lattice where the current security level  $\theta$  might not be  $\perp$ . There is no need to check the timing  $t_1$  of the end point  $w'$  of the new context  $\kappa'$  directly at this point.

For `lower`, we make sure that the time passage is valid under  $\theta$ . In the case of going to the lowest security level, this trivially holds. For `jmp`, we simply check the time passage. Finally, nothing special is needed for `halt`.  $TAL_C$  allows `halt` only in the empty security context. In such a context, all timing annotations are valid.

## 4.2 Soundness

The noninterference proof of  $TAL_C^+$  extends that of  $TAL_C$ . We first define the equivalence of two programs with respect to a security level  $\theta$ . Intuitively, two programs (heaps and register files) are equivalent if and only if they agree on their low-security contents.

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

**Definition 2 (Register File Equivalence).**  $\Gamma \vdash R_1 \approx_\theta R_2 \iff \forall r \in \text{dom}(\Gamma)$ , if  $\Gamma(r) = \tau_{\theta'}$  and  $\theta' \subseteq \theta$ , then  $R_1(r) = R_2(r)$ .

**Definition 3 (Program Equivalence).**  $\Psi; \Gamma \vdash P_1 \approx_\theta P_2 \iff P_1 = (H_1, R_1, I_1)_{\kappa_1}$ ,  $P_2 = (H_2, R_2, I_2)_{\kappa_2}$ ,  $\Psi \vdash H_1 \approx_\theta H_2$ ,  $\Gamma \vdash R_1 \approx_\theta R_2$ , and either:

1.  $\kappa_1 = \kappa_2$ ,  $SL(\kappa_1) \subseteq \theta$ , and  $I_1 = I_2$ , or
2.  $SL(\kappa_1) \not\subseteq \theta$ ,  $SL(\kappa_2) \not\subseteq \theta$ .

The above relations are all reflexive, symmetrical, and transitive. Our noninterference theorem relates the executions of two equivalent programs that both start in a low security context (relative to the security level of concern). In  $TAL_C$ , we showed that if both executions terminate, then the result programs are equivalent. The extra timing annotations now guarantee that nontermination can only happen in a context at the lowest security level.

The idea of the proof is intuitive. Given a security level of concern, the executions are phased into “low steps” and “high steps.” It is easy to relate the two executions under a low step, because they involve the same instructions. Under a high step, the two executions are no longer in lock step. Recall that `raise` and `lower` mark the beginning and end of a secured region. We relate the program states before the `raise` and after the `lower`, circumventing directly relating two executions under high steps. In addition, there would be no nontermination in the secured region.

We give the formal details in three lemmas and a noninterference theorem. Lemma 1 indicates that a security context in a high step can be changed only with `raise` or `lower`. Lemma 2 says that a program in a high context will eventually reduce to a step that discharges the current security context with a `lower`. Lemma 3 articulates the lock step relation between two equivalent programs in a low step. Theorem 1 of noninterference then follows: given two equivalent programs, if one terminates, then the other terminates in a state equivalent to the first. As a corollary, if one does not terminate, then the other does not either.

In the following,  $\mapsto^*$  represents the reflexive and transitive closure of a single-step relation  $\mapsto$  of the operational semantics.  $\Gamma \succeq_\theta \Gamma'$  means that  $\Gamma(r) = \Gamma'(r)$  for every  $r$  such that  $\Gamma'(r) = \tau_{\theta'}$  and  $\theta' \subseteq \theta$ . We use  $Q$  in addition to  $P$  to denote programs when comparing two executions.

**Lemma 1 (High Step).** *If  $P = (H, R, I)_\kappa$ ,  $SL(\kappa) \not\subseteq \theta$ ,  $\Psi; \Gamma; t \vdash P$ , then either:*

1. *there exists  $\Gamma_1, t_1$  and  $P_1 = (H_1, R_1, I_1)_\kappa$  such that  $P \mapsto P_1$ ,  $\Psi; \Gamma_1; t_1 \vdash P_1$ ,  $\Gamma \succeq_\theta \Gamma_1$ ,  $t_1 < t$ , and  $\Psi; \Gamma_1 \vdash P \approx_\theta P_1$ , or*
2.  *$I$  is of the form  $(\text{raise } \kappa'; I')$  or  $(\text{lower } w)$ .*

**Proof sketch:** By case analysis on the first instruction of  $I$ .  $I$  cannot be `halt`, because the typing rule for `halt` requires the context to be  $\bullet$ . If  $I$  is not `halt`, `raise` or `lower`, by the operational semantics and inversion on the typing rules, one can get  $\Gamma_1, t_1$  and  $P_1$  for the next step.  $t_1 < t$  holds due to the time passage judgment in the typing rules. The typing rules also prohibit writing into a low heap cell, hence low heap cells remain the same after the step. When a register is updated,  $\Gamma_1$  gives it a type whose security label takes  $SL(\kappa)$  into account, hence that register has a high type in  $\Gamma_1$ . As a result,  $\Gamma \succeq_\theta \Gamma_1$  and  $\Psi; \Gamma_1 \vdash P \approx_\theta P_1$ .  $\square$

**Lemma 2 (Context Discharge).** *If  $P = (H, R, I)_{\theta \triangleright w}$ ,  $\theta \not\subseteq \theta'$ ,  $\Psi; \Gamma; t \vdash P$ , then there exists  $\Gamma', t'$  and  $P' = (H', R', \text{lower } w)_{\theta \triangleright w}$  such that  $P \mapsto^* P'$ ,  $\Psi; \Gamma'; t' \vdash P'$ ,  $\Gamma \succeq_{\theta'} \Gamma'$ ,  $t' \leq t$ , and  $\Psi; \Gamma' \vdash P \approx_{\theta'} P'$ .*

**Proof sketch:** By inversion on  $\Psi; \Gamma; t \vdash P$ , we get  $\theta \vdash t$ . By the assumption  $\theta \not\subseteq \theta'$ , we get  $\theta \neq \perp$ . By inversion on  $\theta \vdash t$ , we get  $t \neq \infty$ . The proof then proceeds by generalized induction on  $t$ .

The base case of  $t = 0$  is easily discharged, because by inversion on the well-typedness of  $P$  with  $\theta \neq \perp$ , we obtain a time-passage judgment which implies that  $t > 0$ .

In the inductive case, suppose the proposition holds for any  $t < t_0$ . We show that the proposition also holds for  $t = t_0$ . There are two cases to consider, following Lemma 1.

In the case where the first instruction of  $I$  is not `raise` or `lower`, by Lemma 1, there exists  $\Gamma_1, t_1$  and  $P_1$  such that  $P \mapsto P_1, \Psi; \Gamma_1; t_1 \vdash P_1, \Gamma \succeq_{\theta'} \Gamma_1, t_1 < t, \Psi; \Gamma_1 \vdash P \approx_{\theta'} P_1$ , and the security context of  $P_1$  is the same as that of  $P$ . By induction hypothesis on  $t_1$  and  $P_1$ , there exists  $\Gamma', t'$ , and  $P'$  such that  $P_1 \mapsto^* P', \Psi; \Gamma'; t' \vdash P', \Gamma_1 \succeq_{\theta'} \Gamma', t' \leq t_1$  and  $\Psi; \Gamma' \vdash P_1 \approx_{\theta'} P'$ . Putting the above together,  $P \mapsto^* P', \Gamma \succeq_{\theta'} \Gamma'$  because  $\succeq_{\theta'}$  is transitive by definition,  $t' \leq t$ , and  $\Psi; \Gamma' \vdash P \approx_{\theta'} P'$  by definition and the fact that  $\Gamma_1 \succeq_{\theta'} \Gamma'$ .

Case  $I = \text{raise } \theta_1 \triangleright w_1; I_1$ . By definition of the operational semantics,  $P \mapsto P_1$  where  $P_1 = (H, R, I_1)_{\theta_1 \triangleright w_1}$ . By inversion on  $\Psi; \Gamma; t \vdash P$  and the typing rule of `raise`,  $\theta \subseteq \theta_1$  and there exists  $t_1 < t$  such that  $\Psi; \Gamma; \theta_1 \triangleright w_1; t_1 \vdash I_1$ . By definition of well-typed programs,  $\Psi; \Gamma; t_1 \vdash P_1$ . By induction hypothesis on  $t_1$  and  $P_1$ , there exists  $\Gamma_2, t_2$ , and  $P_2 = (H_2, R_2, I_2)_{\theta_1 \triangleright w_1}$  such that  $P_1 \mapsto^* P_2, \Psi; \Gamma_2; t_2 \vdash P_2, \Gamma \succeq_{\theta'} \Gamma_2, t_2 \leq t_1$ , and  $\Psi; \Gamma_2 \vdash P_1 \approx_{\theta'} P_2$ .  $\Psi; \Gamma_2 \vdash P \approx_{\theta'} P_2$  then follows because the heaps and register files in  $P$  and  $P_1$  are the same.

Suppose  $H(w_1) = \text{code}[\kappa; t_3] I_3$ . By inversion on the typing rule of `lower` (the current instruction of  $P_2$ ),  $t_3 < t_2$ . By the operational semantics,  $P_2 \mapsto P_3$  where  $P_3 = (H_2, R_2, I_3)_{\kappa}$ . By inversion on the well-typedness of  $I$  (i.e., `raise`  $\theta_1 \triangleright w_1; I_1$ ),  $\kappa = \theta \triangleright w$ . By induction hypothesis on  $t_3$  and  $P_3$ , there exists  $\Gamma', t'$ , and  $P' = (H', R', I_3)_{\theta \triangleright w}$  such that  $P_3 \mapsto^* P', \Psi; \Gamma'; t' \vdash P', \Gamma_2 \succeq_{\theta'} \Gamma', t' \leq t_3$ , and  $\Psi; \Gamma' \vdash P_3 \approx_{\theta'} P'$ . Putting the above together, the original proposition holds for case  $I = \text{raise } \theta_1 \triangleright w_1; I_1$ .

Case  $I = \text{lower } w_1$ . By inversion on the typing rule of `lower`,  $w = w_1$ . Choosing  $\Gamma' = \Gamma, t' = t$  and  $P' = P$ , the proposition holds trivially.  $\square$

**Lemma 3 (Low Step).** *If  $P = (H, R, I)_{\kappa}, SL(\kappa) \subseteq \theta, \Psi; \Gamma; t \vdash P, \Psi; \Gamma; t \vdash Q, \Psi; \Gamma \vdash P \approx_{\theta} Q, P \mapsto P_1, Q \mapsto Q_1$ , then exists  $\Gamma_1$  and  $t_1$  such that  $\Psi; \Gamma_1; t_1 \vdash P_1, \Psi; \Gamma_1; t_1 \vdash Q_1$  and  $\Psi; \Gamma_1 \vdash P_1 \approx_{\theta} Q_1$ .*

**Proof sketch:** By case analysis on the first instruction of  $I$ . By  $SL(\kappa) \subseteq \theta$  and the definition of  $\approx_{\theta}$ ,  $P$  and  $Q$  contain the same instruction sequence. The case of `raise` does not change the state, hence trivially maintains the equivalence. All other cases maintain that the security context is lower than  $\theta$ . Inspection on the typing rules shows that low locations in the heap can only be assigned low values. Once a register is given a high value, its type in  $\Gamma_1$  will change to high. In the case of branching, the guard must be low, so both  $P$  and  $Q$  branch to the same code. Hence the two programs remain equivalent after one step.  $\square$

**Theorem 1 (Noninterference).** *If  $P = (H, R, I)_{\kappa}, SL(\kappa) \subseteq \theta, \Psi; \Gamma; t \vdash P, \Psi; \Gamma; t \vdash Q, \Psi; \Gamma \vdash P \approx_{\theta} Q$ , and  $P \mapsto^* (H_p, R_p, \text{halt } [\sigma_p])_{\bullet}$ , then exists  $H_q, R_q, \sigma_q$  and  $\Gamma'$  such that  $Q \mapsto^* (H_q, R_q, \text{halt } [\sigma_q])_{\bullet}$ , and  $\Psi; \Gamma' \vdash (H_p, R_p, \text{halt } [\sigma_p])_{\bullet} \approx_{\theta} (H_q, R_q, \text{halt } [\sigma_q])_{\bullet}$ .*

**Proof sketch:** By generalized induction on the number of steps of the derivation  $P \mapsto^* (H_p, R_p, \text{halt } [\sigma_p])_{\bullet}$ . The base case of zero step is trivial. The inductive case is proven by case analysis on the first instruction of  $I$ .

Consider the case where  $I = \text{raise } \theta_1 \triangleright w_1; I_1$  and  $\theta_1 \not\subseteq \theta$ . By definition of the operational semantics and the typing rules, there exist  $t_1$  such that  $P \mapsto P_1$  where

$P_1 = (H, R, I_1)_{\theta_1 \triangleright w_1}$ ,  $t_1 \leq t$ , and  $\Psi; \Gamma; t_1 \vdash P_1$ . By Lemma 2, there exists  $\Gamma_2$ ,  $t_2$  and  $P_2 = (H_2, R_2, \text{lower } w_1)_{\theta_1 \triangleright w_1}$  such that  $P_1 \mapsto^* P_2$ ,  $\Psi; \Gamma_2; t_2 \vdash P_2$ ,  $\Gamma \succeq_{\theta} \Gamma_2$ ,  $t_2 \leq t_1$  and  $\Psi; \Gamma_2 \vdash P_1 \approx_{\theta} P_2$ . Hence  $\Psi \vdash H \approx_{\theta} H_2$  and  $\Gamma_2 \vdash R \approx_{\theta} R_2$ .

By case `lower` of the operational semantics,  $P_2 \mapsto P_3$  where  $P_3 = (H_2, R_2, I_3)_{\kappa_3}$  and  $H(w_1) = \text{code}[\kappa_3; t_3] \Gamma_3.I_3$ . By inversion on the derivation of  $\Psi; \Gamma_2; t_2 \vdash P_2$ , we get  $\Gamma_3 \subseteq \Gamma_2$ ,  $t_3 + 1 \leq t_2$ , and  $\Psi; \Gamma_3; t_3 \vdash P_3$ . It follows that  $\Gamma_3 \vdash R \approx_{\theta} R_2$ . By inversion on the derivation of  $\Psi; \Gamma; t \vdash P$  where the instruction sequence of  $P$  is `raise`  $\theta_1 \triangleright w_1; I_1$ , we get  $\kappa_3 = \kappa$ .

By similarly reasoning,  $Q \mapsto^* Q_3$  where  $Q_3 = (H'_2, R'_2, I_3)_{\kappa_3}$ ,  $\Psi \vdash H \approx_{\theta} H'_2$ ,  $\Gamma_3 \vdash R \approx_{\theta} R'_2$  and  $\Psi; \Gamma_3; t_3 \vdash Q_3$ . By transitivity of the equivalence relations,  $\Psi \vdash H_2 \approx_{\theta} H'_2$  and  $\Gamma_3 \vdash R_2 \approx_{\theta} R'_2$ . Hence  $\Psi; \Gamma \vdash P_3 \approx_{\theta} Q_3$ . The case then follows by induction hypothesis.

All other cases remain low after a step. By Lemma 3, the two programs in the next step are equivalent and well-typed. The proof then follows by induction hypothesis.  $\square$

### 4.3 Certifying Compilation

$\text{TAL}_C^{\pm}$  is sufficiently expressive for supporting the source-level solution of Section 3. We now present a translation that preserves security types from a minimal source language. This source language is the one in Figure 1 with its type system in Figure 2 adapted with Rule [C6']. This allows a concise presentation, yet suffices in demonstrating the main contribution: the use of timing annotations to close up termination channels in assembly code. Again, we use `shaded boxes` to emphasize the extensions.

The low-high security hierarchy of Figure 1 defines a simple lattice consisting of two elements:  $\perp$  and  $\top$ . We use  $|\mathfrak{t}|$  to denote the translation of source type  $\mathfrak{t}$  in  $\text{TAL}_C^{\pm}$ :  $|\text{low}| \equiv \text{int}_{\perp}$  and  $|\text{high}| \equiv \text{int}_{\top}$ .

We assume that the program translation starts in a heap  $H_0$  and a heap type  $\Psi_0$  which satisfy  $\vdash H_0 : \Psi_0$  and contain entries for all the variables of the source program. For any source variable  $v$  that  $\mathfrak{t}(v) = \mathfrak{t}$ , there exists a location  $l_v$  in the heap such that  $\Psi(l_v) = \langle |\mathfrak{t}| \rangle_{\perp}$ . We use  $\mathfrak{t} \sim \Psi$  to refer to this correspondence.

$$\begin{array}{l}
[\text{TREconst}] \quad |i| = \text{mov } r, i \parallel r; \text{mov} \\
[\text{TREvar}] \quad |v| = \text{mov } r, l_v; \text{ld } r', r(0) \parallel r'; \text{mov} + |\text{ld}| \\
[\text{TREadd}] \quad \frac{\begin{array}{l} |E| = \vec{v} \parallel r; t \quad |E'| = \vec{v}' \parallel r'; t' \\ \vec{v}' \text{ does not use } r \quad t'' = t + t' + |\text{add}| \end{array}}{|E + E'| = \vec{v}; \vec{v}'; \text{add } r'', r, r' \parallel r''; t''}
\end{array}$$

**Fig. 10.** Expression translation

$$\begin{array}{c}
\text{[TRC1]} \quad \frac{|E| = \bar{t} \parallel r; t_1 \quad t' = t + t_1 + |\text{mov}| + |\text{st}| + |\text{jmp}|}{\left| \frac{\hat{\Phi}(\mathbf{V}) = \text{high}}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{V} := \mathbf{E}} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi \{ l : (\forall [] . \langle \kappa; t' \rangle \} \}_\perp \\ H \{ l \mapsto \text{code}[\langle \kappa; t' \rangle \} . \bar{t}; \text{mov } r', l_v; \text{st } r'(0), r; \text{jmp } l' \} \\ t' \end{array} \right]} \\
\text{[TRC2]} \quad \frac{|E| = \bar{t} \parallel r; t_1 \quad t' = \infty}{\left| \frac{\hat{\Phi}(\mathbf{V}) = \text{low} \quad \hat{\Phi} \vdash \mathbf{E} : \text{low}}{\hat{\Phi}; [\text{low}] \vdash \mathbf{V} := \mathbf{E}} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \bullet \end{array} \right] = \left[ \begin{array}{c} \Psi \{ l : (\forall [] . \langle \bullet; t' \rangle \} \}_\perp \\ H \{ l \mapsto \text{code}[\langle \bullet; t' \rangle \} . \bar{t}; \text{mov } r', l_1; \text{st } r'(0), r; \text{jmp } l' \} \\ t' \end{array} \right]} \\
\text{[TRC3]} \quad \frac{\begin{array}{c} |E| = \bar{t} \parallel r; t_0 \quad l_1, l_2 \text{ are fresh labels} \quad t' = t_0 + \max\{(t_1 + |\text{bnz}|), (t_2 + |\text{bnz}| + |\text{jmp}|)\} \\ \left| \frac{\text{TD}_1}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_1} \right| \left[ \begin{array}{c} \Psi \\ H \\ l_1; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_1 \\ H_1 \\ t_1 \end{array} \right] \quad \left| \frac{\text{TD}_2}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_2} \right| \left[ \begin{array}{c} \Psi_1 \\ H_1 \\ l_2; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_2 \\ H_2 \\ t_2 \end{array} \right] \end{array}}{\left| \frac{\hat{\Phi} \vdash \mathbf{E} : \text{pc} \quad \frac{\text{TD}_1}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_1} \quad \frac{\text{TD}_2}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_2}}{\hat{\Phi}; [\text{pc}] \vdash \text{if } \mathbf{E} \text{ then } \mathbf{C}_1 \text{ else } \mathbf{C}_2} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_2 \{ l : (\forall [] . \langle \kappa; t' \rangle \} \}_\perp \\ H_2 \{ l \mapsto \text{code}[\langle \kappa; t' \rangle \} . \bar{t}; \text{bnz } r, l_1; \text{jmp } l_2 \} \\ t' \end{array} \right]} \\
\text{[TRC4]} \quad \frac{\left| \frac{\text{TD}}{\hat{\Phi}; [\text{high}] \vdash \mathbf{C}} \right| \left[ \begin{array}{c} \Psi \{ l_1 : (\forall [] . \langle \top \triangleright l'; t_1 \rangle \} \}_\perp \\ H \{ l_1 \mapsto \text{code}[\langle \top \triangleright l'; t_1 \rangle \} . \text{lower } l' \} \\ l_0; l_1 : t_1; \top \triangleright l' \end{array} \right] \Sigma = \left[ \begin{array}{c} \Psi' \\ H' \\ t_2 \end{array} \right] \quad \begin{array}{c} l_0, l_1 \text{ are fresh labels} \\ t_1 = |\text{lower}| \\ t' = t_2 + |\text{raise}| + |\text{jmp}| \end{array}}{\left| \frac{\text{TD}}{\hat{\Phi}; [\text{high}] \vdash \mathbf{C}} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \bullet \end{array} \right] = \left[ \begin{array}{c} \Psi' \{ l : (\forall [] . \langle \bullet; t' \rangle \} \}_\perp \\ H' \{ l \mapsto \text{code}[\langle \bullet; t' \rangle \} . \text{raise } \top \triangleright l'; \text{jmp } l_0 \} \\ t' \end{array} \right]} \\
\text{[TRC5]} \quad \frac{\begin{array}{c} l_1 \text{ is a fresh label} \\ \left| \frac{\text{TD}_2}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_2} \right| \left[ \begin{array}{c} \Psi \\ H \\ l_1; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_1 \\ H_1 \\ t_1 \end{array} \right] \quad \left| \frac{\text{TD}_1}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_1} \right| \left[ \begin{array}{c} \Psi_1 \\ H_1 \\ l; l_1 : t_1; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_2 \\ H_2 \\ t_2 \end{array} \right] \end{array}}{\left| \frac{\text{TD}_1}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_1} \quad \frac{\text{TD}_2}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_2}}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}_1; \mathbf{C}_2} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_2 \\ H_2 \\ t_2 \end{array} \right]} \\
\text{[TRC6]} \quad \frac{\begin{array}{c} |E| = \bar{t} \parallel r; t_1 \\ l_1 \text{ is a fresh label} \quad \left| \frac{\text{TD}}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}} \right| \left[ \begin{array}{c} \Psi \{ l : (\forall [] . \langle \kappa; \infty \rangle \} \}_\perp \\ H \{ l \mapsto \text{code}[\langle \kappa; \infty \rangle \} . \text{jmp } l \} \\ l_1; l : \infty; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi' \\ H' \\ t_2 \end{array} \right]}{\left| \frac{\text{pc} = \text{low} \quad \hat{\Phi} \vdash \mathbf{E} : \text{pc} \quad \frac{\text{TD}}{\hat{\Phi}; [\text{pc}] \vdash \mathbf{C}}}{\hat{\Phi}; [\text{pc}] \vdash \text{while } \mathbf{E} \text{ do } \mathbf{C}} \right| \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi' \{ l : (\forall [] . \langle \kappa; \infty \rangle \} \}_\perp \\ H' \{ l \mapsto \text{code}[\langle \kappa; \infty \rangle \} . \bar{t}; \text{bnz } r, l_1; \text{jmp } l' \} \\ \infty \end{array} \right]}
\end{array}$$

Fig. 11. Command translation

We define expression translation of the form  $|E| = \vec{l} \parallel r; \vec{t}$  in Figure 10. The instruction vector  $\vec{l}$  computes the value of  $E$ , and the result is put in the register  $r$ . The time needed to complete  $\vec{l}$  is  $t$ .

In Figure 11, we define command translation based on the structure of the typing derivation of the source program. Which translation rule to apply is determined by the last typing rule used to check the source command. We use TD to denote (possibly multiple) typing derivations. In particular, the command translation has the form:

$$\left[ \frac{\text{TD}}{[\text{pc}] \vdash C} \mid \left[ \begin{array}{l} \Psi \\ H \\ l_{start}; l_{end}; \vec{t}; \kappa \end{array} \right] \right] = \left[ \begin{array}{l} \Psi' \\ H' \\ \vec{t}' \end{array} \right].$$

The 6 arguments are: a code heap type  $\Psi$ , a code heap  $H$ , starting and ending labels  $l_{start}$  and  $l_{end}$  for the computation of  $C$ , the timing annotation  $t$  of the code at the ending label  $l_{end}$ , and a security context  $\kappa$ . It generates the extended code heap type  $\Psi'$  and code heap  $H'$ , and produces the timing annotation for the starting label  $l_{start}$ .

The definition of the command translation may appear complex, since it provides a formal model of a certifying compiler. Nonetheless, it is not difficult to follow if we remember some invariants maintained by the translation:

- $H$  is well-typed under  $\Psi$  and contains entries for all source variables and procedures;
- $\Psi$  and  $H$  contain the continuation code labeled  $l_{end}$ ;
- The code at  $l_{end}$  has the timing behavior  $t$ ;
- The new code labeled  $l_{start}$  will be put into  $\Psi'$  and  $H'$ ;
- The produced annotation  $t'$  will reflect the timing behavior of the code starting at  $l_{start}$ ;
- The security context  $\kappa$  must match  $\text{pc}$ .

We now explain the manipulation of the timing annotations. Suppose addition  $+$  is extended in the expected way to work with  $\infty$ . In Rule [TRC1], when computing the timing annotation  $t'$  of the starting label, we add the cost of the assembly instructions for computing the source command on top of the timing annotation  $t$  of the ending label. In Rule [TRC2], the security level is low, so we directly use the special annotation  $\infty$ . Rule [TRC3] dictates that, for a conditional, one must take into account the timing of the “longer” branch. Rule [TRC4] essentially introduces some wrapper code, hence its timing computation. For a sequential command, Rule [TRC5] pieces together the translation of the two sub-commands. Finally, for a while-loop, Rule [TRC6] simply uses the special annotation  $\infty$ .

The unshaded parts of the translation rules are explained in greater detail in a technical report [29]. The correctness of the translation is formulated as two lemmas, which can be proven straightforwardly by structural induction on the derivation of the translation.

**Lemma 4 (Expression Translation).** *If  $\Phi \sim \Psi$ ,  $\Phi \vdash E : \tau$ ,  $|E| = \vec{l} \parallel r; \vec{t}$ , and  $\Psi; \{r : |\tau|\}; \kappa; \vec{t} \vdash I$ , then  $\Psi; \{\}; \kappa; \vec{t} + \vec{t}' \vdash \vec{l}; I$ .*

**Lemma 5 (Command Translation).** *If  $\Phi \sim \Psi$ ,  $\Phi; [\text{pc}] \vdash \mathbf{C}$ ,*

$$\left| \frac{\text{TD}}{\Phi; [\text{pc}] \vdash \mathbf{C}} \right| \left[ \begin{array}{c} \Psi \\ H \\ l_{start}; l_{end}; \bar{t}; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi' \\ H' \\ \bar{t}' \end{array} \right], \Psi(l_{end}) = (\forall []. \langle \kappa; \bar{t} \rangle \{ \})_{\perp},$$

*$SL(\kappa) = |\text{pc}|, \vdash H : \Psi$ , then  $\Phi \sim \Psi', \vdash H' : \Psi'$  and  $\Psi' \vdash l_{start} : (\forall []. \langle \kappa; \bar{t}' \rangle \{ \})_{\perp}$ .*

## 5 Timing Channels

In the previous section, we looked at the effect of program termination on information flow. In this section, we extend the machine model to explicitly specify execution time  $t$  and output actions `output  $n$`  [1], allowing the observation of more exact timing information of the program execution.

On top of  $\text{TAL}_C$ 's small-step state transition in the form of  $P \mapsto P'$ , we further specify the action sequences produced along with the program execution in the form of  $P \xrightarrow{as} P'$ . This means that  $P$  steps to  $P'$  producing observable action sequence  $as$ , where  $as$  is a mixed sequence of output numbers  $n$  and execution times  $t$ .

$$(Action\ Sequences) \quad as ::= \epsilon \mid t\ as \mid n\ as$$

In the operational semantics, the value of  $as$  would be determined by the current instruction. Consider the following sample case:

$$(H, R, \text{mov } r_d, w; I)_{\kappa} \xrightarrow{t_{\text{movi}}} (H, R\{r_d \mapsto w\}, I)_{\kappa}$$

This is the operational semantics case of executing a `mov` instruction on an immediate operand  $w$ . Besides the regular machine state update, the above also specifies the time  $t_{\text{movi}}$  needed for completing this instruction. We omit the straightforward definition of the operational semantics, and use  $t_{\text{comm}}$  to represent the execution time of instruction  $comm$ . Following previous source-level techniques [1], we assume that a primitive operation (*i.e.*, an assembly instruction) should execute in constant time, regardless of the values given as arguments. Note that this does not prevent `mov  $r_d, w$`  (move immediate) and `mov  $r_d, r_s$`  (move register) from being timed differently. The reflexive transitive closure of the step transition is extended accordingly.

This extended machine model exposes information leak through the timing channels. Take the following source-level program as an example:

```
if h then {time-consuming operation} else skip;
output n
```

If  $h \neq 0$ , the program produces observable action sequence  $t_{\text{long}}\ n$ , where  $t_{\text{long}}$  is the execution time of the “*time-consuming operation*.” If  $h = 0$ , the program produces  $t_{\text{skip}}\ n$ , where  $t_{\text{skip}}$  is the execution time of the `skip` command. Obviously, this presents information leak, even if no low data is updated in the program.

The approach of the previous sections for closing termination channels can be extended to account for such information leak through timing channels. Instead of recording an upper bound of execution steps till the end of a high region, we use timing annotations to record the exact observable action sequence  $as$ . Based on the same reasoning

for disallowing low updates in a high region, we also disallow output actions in a high region. In essence, the extended timing annotations reflect execution time. In such an extended system, the typing rule for branching should check that the timing annotations of the two branches match.

$$\begin{array}{c}
\frac{\text{comm is any instruction}}{|comm| = t_{comm}} \\
\frac{}{\perp \vdash t} \\
\frac{t \neq \infty}{\theta \vdash t}
\end{array}
\qquad
\frac{}{\perp \vdash t \xrightarrow{t'} t''}
\qquad
\frac{t = t' + t''}{\theta \vdash t \xrightarrow{t'} t''}$$

$$\frac{}{\perp \vdash t \sim t'}
\qquad
\frac{t = t'}{\theta \vdash t \sim t'}$$

**Fig. 12.** Timing rules on timing channels

The exact adaptation to  $TAL_C^+$  is given in Figure 12. By giving the timing judgments different definitions, we obtain a type system for closing timing channels, and the typing rules of Figures 8 and 9 remain the same. The changes to the timing rules are marked in shaded boxes in Figure 12; they reflect the idea that exact timing is tracked. For time passage  $\theta \vdash t \xrightarrow{t'} t''$  in a high context,  $t$  must reflect the sum of  $t'$  and  $t''$  exactly.  $\infty$  is still only allowed in low contexts. Finally, the matching of timing  $\theta \vdash t \sim t'$  requires the equality of  $t$  and  $t'$  unless in a low context; recall that this is used when type-checking a branching instruction to make sure that the branches exhibit the same timing behavior in high contexts.

The correctness of this approach is intuitive. At the beginning of a high branch, based on the extended timing annotations, the branches will meet at the end of the high region in the same amount of time. In addition, there are no outputs or updates to low variables in a high region. On the technical side, the noninterference proof follows that of Section 4, with minor difference on how to perform the induction. The proof for the termination-based system sometimes (Lemma 2 in particular) conducts induction directly on the timing annotation, which happens to reflect an upper bound of the number of operation steps. For this timing-based system, the timing annotation is different. By introducing a notion of “operation steps” and conducting induction on it, the proof goes through in the same way as before.

In terms of expressiveness, this system has decidable typing and is more restrictive than previous work by Agat [1] where type checking is undecidable. In particular, Agat’s type system allows some low updates in high branches, such as the following:

if  $h$  then  $l := 1$  else  $l := 1$

In general, Agat’s type system allows a high conditional as long as the two branches have the same “externally observable behavior.” This notion is supported with a typing rule based on  $\Gamma$ -bisimulation, which is undecidable. For practical use, Agat uses “padding” commands to equalize the execution times of the branches. For example, a padding command for  $h := 1$  would be a special skip-command `SkipAsn h 1`, which costs the same time but does not perform actual state update. Using these padding com-

mands, Agat proposes a cross-copying transformation that generates  $\Gamma$ -bisimilar conditionals. In this transformation,  $\Gamma$ -bisimulation is used conservatively—a command is  $\Gamma$ -bisimilar to itself and a high command is also  $\Gamma$ -bisimilar to its padding counterpart.

$$\begin{array}{c}
\text{[TRC3']} \\
\hline
\left[ \frac{\begin{array}{c} |E| = \vec{r} \parallel r; t_0 \\ \frac{\text{TD}_1}{\Phi; [\text{pc}] \vdash C_1} \end{array} \quad \begin{array}{c} l_1, l_2 \text{ are fresh labels} \\ \left[ \begin{array}{c} \Psi \\ H \\ l_1; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_1 \\ H_1 \\ t_1 \end{array} \right] \end{array} \quad \frac{\begin{array}{c} t'_1 = t_1 + |\text{jmp}| \\ \frac{\text{TD}_2}{\Phi; [\text{pc}] \vdash C_2} \end{array} \quad \begin{array}{c} t' = t_0 + t'_1 + |\text{bnz}| \\ \left[ \begin{array}{c} \Psi_2 \\ H_2 \\ t_2 \end{array} \right] \end{array} \right. \\
\left. \frac{\Phi \vdash E : \text{pc} \quad \frac{\text{TD}_1}{\Phi; [\text{pc}] \vdash C_1} \quad \frac{\text{TD}_2}{\Phi; [\text{pc}] \vdash C_2}}{\Phi; [\text{pc}] \vdash \text{if } E \text{ then } C_1 \text{ else } C_2} \quad \left[ \begin{array}{c} \Psi \\ H \\ l; l' : t; \kappa \end{array} \right] = \left[ \begin{array}{c} \Psi_2 \left\{ \begin{array}{l} l : (\forall [] . \langle \kappa; t' \rangle \{ \} )_{\perp} \\ l'_1 : (\forall [] . \langle \kappa; t'_1 \rangle \{ \} )_{\perp} \end{array} \right\} \\ H_2 \left\{ \begin{array}{l} l \mapsto \text{code}[] \langle \kappa; t' \rangle \{ \} . \vec{r}; \text{bnz } r, l'_1; \text{jmp } l_2 \\ l'_1 \mapsto \text{code}[] \langle \kappa; t'_1 \rangle \{ \} . \text{jmp } l_1 \end{array} \right\} \\ t' \end{array} \right]
\end{array}$$

**Fig. 13.** Translating conditionals for closing timing channels

This transformation does not accept programs which update low variables in high branches. Therefore, the above example is no longer considered valid. Our certifying compilation scheme in Section 4 can be easily adapted to support such transformed programs: if a program  $P$  is accepted by the cross-copying transformation and transformed into  $P'$ , then the adapted certifying compilation scheme will translate  $P'$  into well-typed assembly code.

For the adaptation, the only change would be in Rule [TRC3], where we ensure that “balanced branches” remain balanced after the compilation. We provide one way as Rule [TRC3'] in Figure 13. Note that in a transformed source program, the two branches of a conditional are balanced. Therefore, Rule [TRC3'] will generate code where  $t_1 = t_2$ . In the previous Rule [TRC3], the target code is not balanced, and “max” is used to take the upper bound. Now Rule [TRC3'] inserts an extra code block labeled  $l'_1$  so that the branches remain balanced in the target code.

## 6 Multi-threading and Internal Timing

### 6.1 Possibilistic Noninterference

To consider the problem of information flow in a multi-threaded setting, we need to extend the notion of noninterference to account for the nondeterministic execution of threads. A straightforward generalization is to define the observable behavior of a program as the set of possible execution results.

The interaction between threads can be exploited as a channel of information flow. Even if an attacker cannot observe the external timing behaviors of a program, some internal timing behaviors may still result in illicit information flow, *e.g.*, by affecting the execution order of interacting threads.

```

Initially:  f = 0
Thread 0:  if h then f := 1 else f := 2
Thread 1:  while f <> 1 do skip;
           l := false;
           f := 2;
Thread 2:  while f <> 2 do skip;
           l := true;
           f := 1;

```

**Fig. 14.** Example on possibilistic channels

Figure 14 gives an example program of three threads. Suppose the value of  $f$  is initially 0. Consider the case where  $h$  is `true`. Thread 1 will be “unlocked” first. Thread 2 will be “unlocked” after the last statement of thread 1. Therefore, the final value of  $l$  will be `true`. Similarly, the final value of  $l$  will be `false` if  $h$  is `false`. Effectively, the above multi-threaded program copies the content of  $h$  into  $l$ . Each of the individual threads is well-typed under a typical sequential information-flow type system (e.g., that of Figure 2), assuming that  $h$  and  $f$  are high variables and  $l$  is a low variable. Nonetheless, the threads together leak high information.

Smith and Volpano [23] proposed a source-level information-flow type system for multi-threaded programs, and showed that it guarantees possibilistic noninterference in the presence of nondeterministic thread scheduling. In this type system, every thread is checked separately to satisfy two requirements in addition to those enforced for noninterference in the sequential setting:

1. The guard of a while-loop must have type `low`;
2. The while-loop itself must also have type `low`.

It is curious that these conditions happen to be the same as those for closing termination channels. Essentially, loops can only happen under low PCs (see Rule [C6'] in Section 3). This is convenient, because we can now reuse the techniques described in Section 3. In particular, we enforce the absence of backward jumps in high regions with monotonically decreasing timing annotations. The introduction of multi-threading does not affect the treatment described earlier, because the type checking is carried out separately for each thread.

## 6.2 Probabilistic Noninterference

The generalization of noninterference in Section 6.1 considers the set of possible execution results. This is sometimes not strong enough to prevent certain exploits in practice. For example, the probability distribution of the possible results may serve as a channel of information flow.

Consider the program in Figure 15. Suppose  $h$  is high,  $l$  is low, and  $C_{long}$  is a long statement sequence with many potential context switches in the middle. This program would be considered secure with the analysis of Section 6.1. However, under most schedulers, the result of  $l$  is likely to be the same as  $h$ , because the thread that executes

```

Thread 0:  if h then  $C_{long}$  else skip;
           l := true;

Thread 1:  if h then skip else  $C_{long}$ ;
           l := false;

```

**Fig. 15.** Example on probabilistic channels

the longer branch will likely to finish later than the other thread. Even though the set of possible results of  $l$  is the same regardless of the value of  $h$ , the probability distributions of the possible results are different.

Such probabilistic attacks exploit the internal timing behaviors of programs. Therefore, it is natural to adapt the techniques for closing timing channels in Section 5 from addressing external timing to addressing internal timing. More specifically, instead of reflecting the external execution time of the program instructions, we let the timing annotations reflect the number of potential context switches—the “internal time” observable by threads. This internal time advances by one unit whenever there is a potential context switch. The analysis of Section 5 would be adapted accordingly, enforcing the following:

1. To disallow low assignments in high regions;
2. To disallow while-loops in high regions;
3. To allow a high conditional only if the two branches have matching internal timing.

This approach is inspired by Sabelfeld and Sands [20], who obtained the desired probabilistic noninterference result for a source language by connecting context switches with the probability distribution of program execution paths in the context of arbitrary schedulers. Similar to Agat’s transformation system [1], the system of Sabelfeld and Sands uses “padding” instructions to equalize program branches. Focusing on internal timing, their system uses skip instructions and dummy forks instead of the “skip-commands” (e.g., `SkipAsn`) used by Agat. Transformed programs of their system will have the same number of atomic commands in high branches. It is natural for  $TAL_C^+$  to support the certifying compilation of such transformed programs, as long as one takes care to implement atomic commands, skip instructions, and dummy forks correctly.

In related work, Volpano and Smith [27] proposed a system that requires high conditionals to be protected so that the branches execute atomically. This can be viewed as a special instance of the above idea, because the atomic branches exhibit the same internal timing behavior (*i.e.*, no context switch).

More recently, Smith [22] proposed a less restrictive system that allows a high variable to appear in the guard of a while-loop as long as no assignment to a low variable follows. The key idea is that a command may be given a type of the form  $\tau_1 \text{ cmd } \tau_2$ , meaning that the command assigns only to variables of level  $\tau_1$  or higher, and has running time that depends only on variables of level  $\tau_2$  or lower. It is conceivable to adapt this idea for assembly code, since both security levels and running times have clear counterparts in  $TAL_C^+$ . We leave the details of this as future work.

We have elided the support for thread synchronization in this paper. The secure support for semaphore-based synchronization at a source level has been studied by Sabelfeld [18], where semaphore variables are restricted to have type `low`.

## 7 Related and Future Work

$TAL_C^+$  is motivated by the need of certifying compilation in the area of information-flow security [19, 10, 21]. On the technical aspects,  $TAL_C^+$  is inspired mainly by two lines of work: language-based information-flow security [19] and typed assembly languages [13].

From the perspective of information-flow security, we are inspired by previous work on covert channels and concurrency in high-level languages. Volpano and Smith [25] studied termination-sensitive noninterference, and proposed a type system that closes termination channels by disallowing loops from occurring at sensitive program points. Agat [1] used program transformation to prevent timing leaks, where the execution times of high branches are equalized by cross-padding with appropriate dummy instructions. Smith and Volpano [23] established possibilistic noninterference for a multi-threaded language by, again, disallowing loops from occurring at sensitive program points. Sabelfeld and Sands [20] proved a probabilistic noninterference result with respect to a scheduler-independent security condition. These systems are closely related to our solutions and have been discussed in the main body of this paper to illustrate the expressiveness of  $TAL_C^+$ .

From the perspective of typed assembly languages, there are some recent efforts toward enforcing information-flow security directly at the target-code level [32, 9, 4, 12, 30, 5]. However, relatively little is done on addressing covert channels or concurrency. Kobayashi and Shirane [9] proposed a JVMML-based type system for timing-sensitive noninterference. The system relies on the computation of control dependency for identifying sensitive regions, and closes timing channels in sequential code by inserting a delay linear with respect to the normal execution time. Hedin and Sands [8] proposed another JVMML-based type system parameterized over an abstract timing model characterizing execution times of instructions and an algorithm enforcing low-observable equivalence. When given a proper timing model and a corresponding algorithm, the system guarantees timing-sensitive noninterference in sequential code. The system relies on the computation of the least merge points of branch instructions to address the lack of program structures in low-level code. Barth *et al.* [6] proposed a framework for enforcing secure information flow for multi-threaded low-level code. Special primitives for interacting with the scheduler are introduced during compilation to prevent internal timing leaks. The system assumes that the low-level code comes with a security environment produced by the compiler for describing the security levels of program points.

In general, the lack of a suitable target for certifying compilation, especially one that handles both covert channels and concurrency uniformly, presents difficulty when applying related solutions to the real world, where covert channels and concurrency are easily exploitable. In this paper, we introduced timing extensions to  $TAL_C$  applicable under various security assumptions. Only simple arithmetic manipulations on the timing annotations are required during type checking. Although the annotations are produced

by a compiler, they are verified separately by the  $TAL_C^+$  type system, thus bad annotations will be caught. As a result, only the type checker is in the trusted computing base, and the system correctness will not be affected by (a buggy implementation of) the computation on dependence regions, least merge points, or security environments. Using the generic  $TAL_C^+$  framework, it is promising to build a common typed assembly language for “customized” noninterference (termination-sensitive, timing-sensitive, possibilistic, probabilistic, and plain). In addition, it is interesting to note that timing extensions of typed assembly languages also have applications beyond noninterference [16, 24].

There are nonetheless still many aspects that deserve further research, especially on the practical implementation of the theoretical ideas. In particular, the interaction between security-type preserving compilation and optimization remains an open question. In this paper, we provided a compilation scheme that preserves security types, where we did not perform any optimizations. Conventional compilers perform sophisticated optimizations to produce efficient code. Some of these optimizations may invalidate the security guarantees established for source programs. A simple example is that an optimization might change the execution time of program branches, affecting timing-sensitive noninterference. Whereas there is likely a trade-off between security and efficiency, optimizations that preserve types and security are worth investigating.

There are also topics which have been studied for high-level languages, but not for low-level code. Some examples include related security policies [14, 31, 11, 17], practical implementation and type inference [15, 2], and abstraction-violating attacks [2].

## 8 Conclusion

We have presented a generic type system  $TAL_C^+$  for information-flow security in assembly code under various practical settings. Since some useful abstractions (*e.g.*, program structures) are missing from assembly code,  $TAL_C^+$  introduces various timing annotations to guide the information-flow analysis. When equipped with different timing annotations,  $TAL_C^+$  can guarantee termination-sensitive, timing-sensitive, possibilistic and probabilistic noninterference. Besides proving the soundness of the type system, we also provide a certifying compilation scheme targeting  $TAL_C^+$ . We consider this as a useful step toward the practical use of security-type systems in assembly code.

## References

1. J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symposium on Principles of Programming Languages*, pages 40–53, Boston, MA, Jan. 2000.
2. J. Agat. *Type-based techniques for covert channel elimination and register allocation*. PhD thesis, Chalmers Univ. of Technology & Gothenburg Univ., Gothenburg, Sweden, Dec. 2000.
3. T. Ball. What’s in a region? Or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Prog. Lang. & Syst.*, 2(1–4):1–16, Mar.–Dec. 1993.
4. G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In *Proc. 5th VMCAI*, volume 2937 of *LNCS*, pages 2–15, Venice, Italy, Jan. 2004.
5. G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Proc. 2006 IEEE Symposium on Security and Privacy*, pages 230–242, Oakland, CA, May 2006.

6. G. Barthe, T. Rezk, A. Russo, and A. Sabelfeld. Security of multithreaded programs by compilation. In *Proc. 12th European Symposium on Research in Computer Security*, Dresden, Germany, Sept. 2007.
7. K. Crary and S. Weirich. Resource bound certification. In *Proc. 27th ACM symposium on Principles of programming languages*, pages 184–198, Boston, MA, USA, Jan. 2000.
8. D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. In *Proc. 1st Workshop on Bytecode Semantics, Verification, Analysis and Transformation*, pages 163–182, Edinburgh, Scotland, UK, Apr. 2005.
9. N. Kobayashi and K. Shirane. Type-based information flow analysis for low-level languages. In *3rd Asian Workshop on Prog. Lang. & Syst.*, pages 302–316, Shanghai, China, Nov. 2002.
10. D. Kozen. Language-based security. In *Proc. 24th MFCS*, volume 1672 of *LNCS*, pages 284–298, Szklarska Poreba, Poland, Sept. 1999.
11. P. Laud. Semantics and program analysis of computationally secure information flow. In *Proc. 10th European Symposium on Programming*, pages 77–91, Genova, Italy, Apr. 2001.
12. R. Medel, A. Compagnoni, and E. Bonelli. A typed assembly language for non-interference. In *Proc. 9th ICTCS*, pages 360–374, Siena, Italy, Oct. 2005.
13. G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, Nov. 1999.
14. A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proc. IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, May 1998.
15. A. C. Myers *et al.*. Jif: Java + information flow. <http://www.cs.cornell.edu/jif/>, 2001.
16. G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In *Special Issue on Mobile Agent Security*, volume 1419 of *LNCS*, pages 61–91, Oct. 1997.
17. A. D. Pierro, C. Hankin, and H. Wiklicky. Approximate non-interference. In *Proc. 15th CSFW*, pages 1–17, Cape Breton, NS, Canada, June 2002.
18. A. Sabelfeld. The impact of synchronisation on secure information flow in concurrent programs. In *Proc. 4th International Conference on Perspectives of System Informatics*, volume 2244 of *LNCS*, pages 227–241, Novosibirsk, Russia, July 2001.
19. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
20. A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-thread programs. In *Proc. 13th CSFW*, pages 200–214, Cambridge, England, July 2000.
21. F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101, 2001.
22. G. Smith. A new type system for secure information flow. In *Proc. 14th CSFW*, pages 115–125, Cape Breton, NS, Canada, June 2001.
23. G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th POPL*, pages 355–364, San Diego, CA, Jan. 1998.
24. J. C. Vanderwaart and K. Crary. Automated and certified conformance to responsiveness policies. In *Proc. 2005 TLDI*, pages 79–90, Long Beach, CA, Jan. 2005.
25. D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *10th CSFW*, pages 156–169, Rockport, MA, June 1997.
26. D. Volpano and G. Smith. A type-based approach to program security. In *Proc. 7th TAP-SOFT*, volume 1214 of *LNCS*, pages 607–621, Lille, France, Apr. 1997.
27. D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. In *Proc. 11th CSFW*, pages 34–43, Rockport, MA, June 1998.
28. H. Xi and R. Harper. A dependently typed assembly language. In *Proc. 6th ACM International Conference on Functional Programming*, pages 169–180, Florence, Italy, Sept. 2001.

29. D. Yu and N. Islam. A typed assembly language for confidentiality. Technical Report DCL-TR-2005-0002, DoCoMo USA Labs, San Jose, CA, Mar. 2005. <http://www.docomolabsresearchers-usa.com/~dyu/talc-tr.pdf>.
30. D. Yu and N. Islam. A typed assembly language for confidentiality. In *Proc. 15th ESOP*, volume 3924 of *LNCS*, pages 162–179, Vienna, Austria, Mar. 2006.
31. S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. 14th CSFW*, pages 15–23, Cape Breton, NS, Canada, June 2001.
32. S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002.