A Type System for Observational Determinism

Tachio Terauchi
Tohoku University
terauchi@ecei.tohoku.ac.jp

Abstract

Zdancewic and Myers introduced observational determinism as a scheduler independent notion of security for concurrent programs. This paper proposes a type system for verifying observational determinism. Our type system verifies observational determinism by itself, and does not require the type checked program to be confluent. A polynomial time type inference algorithm is also presented.

1 Introduction

Non-interference is a property of a program which states that the program's publicly observable (i.e., low security) behavior is independent of its confidential (i.e., high security) inputs. Non-interference has applications in software security. We refer to the survey by Sabelfeld and Myers [11] for a general overview.

Motivated by earlier work by Roscoe [10], Zdancewic and Myers proposed *observational determinism* [14] as a notion of non-interference for concurrent programs which roughly says that a program is secure if and only if its publicly observable behavior is independent of its confidential data *and* independent of process/thread scheduling. As such, unlike other notions of non-interference such as possibilistic security, observational determinism implies security regardless of the scheduler choice.

In the same paper, Zdancewic and Myers proposed a type system for enforcing observational determinism. However, their type system required the type checked program to be confluent in order to be verified secure. Confluence, even if relaxed to confluence up to observational parts of the program, can be a restrictive requirement. (Definitional observational determinism does not imply confluence.) This paper presents, to the best of our knowledge, the first type system that verifies observational determinism by itself for non-trivial concurrent programs. The type system integrates a flow-based security analysis with a fractional capabilities-based determinism checker in a novel way to allow programs to be non-deterministic in non-security-relevant portions of the program. In particular, our type system does not force confluence and can verify some non-confluent programs to be secure. In addition to added flexibility (e.g., being able to check determinism of concurrent reads), fractional capabilities enable efficient (polynomial time) type inference algorithm via linear programming.

e	::=	x	(*variable*)
	|	ℓ	(*location*)
	|	n	(*integer constant*)
	|	e_1 *op* e_2	(*integer operations*)
	|	$!e$	(*location read*)
s	::=	$\texttt{ref}\ x = e\ \texttt{in}\ s$	(*new location*)
	|	$\texttt{chan}\ x\ \texttt{in}\ s$	(*new channel*)
	|	$s_1; s_2$	(*sequence*)
	|	$e := e$	(*location write*)
	|	$\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2$	(*branch*)
	|	$\texttt{while}\ e\ \texttt{do}\ s$	(*loop*)
	|	$\texttt{spawn}\ \{s\}$	(*new thread*)
	|	$\texttt{send}\ e$	(*signal send*)
	|	$\texttt{wait}\ e$	(*signal receive*)
	|	$\texttt{skip}$	(*skip*)

Figure 1. The syntax of the simple concurrent language.

2 Preliminaries

To concretize the presentation, we focus on the simple concurrent language shown in Figure 1. We briefly describe the syntax. Meta variables x, x_1, etc. range over (program) variables. Variables are immutable like in functional languages, whereas *locations* (i.e., first class reference cells) can be used for destructive updates. The construct $\texttt{ref}\ x = e\ \texttt{in}\ s$, creates a new location and initializes it to the result of evaluating e, and binds the location to the variable x. Note that the language allows locations to appear in the source syntax. This is needed so that low se-

curity variables and high security variables in the classical information flow setting can be expressed as *low security locations* and *high security locations*. The language contains branches and loops. The construct `spawn` $\{s\}$ creates a new thread to evaluate s. Threads may synchronize each other by sending signals over channels. Channels are dynamically created by `chan` x `in` s, which allocates a fresh channel, binds it to the variable x, and evaluates s. Signals are "asynchronous" in the sense that the sender does not need to wait for a receiver to send the signal. Finally, the language contains integer constants. Binary (deterministic) operations over integers ($+, -, \times, \leq$, etc.) are ranged over by the symbol *op*.

To describe non-interference, we consider programs with free locations, some of which are classified as *high security locations* or *low security locations*. The essence of information flow is to check if information contained in high security locations could leak to an attacker who can only observe the contents of low security locations. For instance, the program below leaks information in the high security location ℓ_h to the low security location ℓ_l.

$$\ell_l := 0; \texttt{while}\ !\ell_h > 0\ \texttt{do}\ (\ell_l := !\ell_l + 1; \ell_h := !\ell_h - 1)$$

We define the semantics of the language by small-step reductions from states to states. A *state* is a triple (S, B, p) where S is a *store* mapping locations to *values* (v, see below), B is a mapping from channel constants (c) to a non-zero integer indicating the number of pending signals, and p is a *program state* defined as follows.

$$\begin{aligned} v &::= c \mid \ell \mid n \\ e &::= \cdots \mid c \\ p &::= s \mid p_1 \mid\mid p_2 \end{aligned}$$

Here, the meta variable c ranges over channel constants. Expressions are extended to contain channel constants. A program state (p) is a parallel composition of (extended) statements.

We define the following general notations. Given a function f, $f[u \mapsto v]$ denotes the function $\{w \mapsto f(w) \mid w \in \mathit{dom}(f) \setminus \{u\}\} \cup \{u \mapsto v\}$. Given a function f and a set $X \subseteq \mathit{dom}(f)$, $f|_X$ denotes the restriction of f to X, that is, $\{u \mapsto f(u) \mid u \in X\}$. Given strings u and v, we write $u \sim v$ if u and v are stutter equivalent. We write $u \prec v$ if there exists a prefix w of v such that $u \sim w$.

Figure 2 shows the reduction rules. The evaluation contexts are defined as follows.

$$E ::= [\,] \mid E; s \mid E||p \mid p||E$$

The reduction rules are mostly self-explanatory. The reduction rules use the evaluation rules for expressions of the form $(S, e) \rightsquigarrow v$, also shown in Figure 2. In the rule for binary operation, $[\![op]\!]$ denotes the standard semantics of

$$(S, v) \rightsquigarrow v$$

$$\frac{(S, e_1) \rightsquigarrow n_1 \quad (S, e_2) \rightsquigarrow n_2}{(S, e_1\ op\ e_2) \rightsquigarrow n_1 [\![op]\!] n_2} \qquad \frac{(S, e) \rightsquigarrow \ell}{(S, !e) \rightsquigarrow S(\ell)}$$

$$\frac{(S, e) \rightsquigarrow n \quad n \neq 0}{(S, B, E[\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]) \to (S, B, E[s_1])}$$

$$\frac{(S, e) \rightsquigarrow 0}{(S, B, E[\texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2]) \to (S, B, E[s_2])}$$

$$\begin{aligned} &(S, B, E[\texttt{while}\ e\ \texttt{do}\ s]) \\ &\to (S, B, E[\texttt{if}\ e\ \texttt{then}\ (s; \texttt{while}\ e\ \texttt{do}\ s)\ \texttt{else}\ \texttt{skip}]) \end{aligned}$$

$$(S, B, E[\texttt{spawn}\ \{s\}]) \to (S, B, E[\texttt{skip}] \mid\mid s)$$

$$\frac{(S, e) \rightsquigarrow v \quad \ell \notin \mathit{dom}(S)}{(S, B, E[\texttt{ref}\ x = e\ \texttt{in}\ s]) \to (S[\ell \mapsto v], B, E[s[\ell/x]])}$$

$$\frac{(S, e_1) \rightsquigarrow \ell \quad (S, e_2) \rightsquigarrow v}{(S, B, E[e_1 := e_2]) \to (S[\ell \mapsto v], B, E[\texttt{skip}])}$$

$$\frac{c \notin \mathit{dom}(B)}{(S, B, E[\texttt{chan}\ x\ \texttt{in}\ s]) \to (S, B[c \mapsto 0], E[s[c/x]])}$$

$$\frac{(S, e) \rightsquigarrow c}{(S, B, E[\texttt{send}\ e]) \to (S, B[c \mapsto B(c) + 1], E[\texttt{skip}])}$$

$$\frac{(S, e) \rightsquigarrow c \quad B(c) > 0}{(S, B, E[\texttt{wait}\ e]) \to (S, B[c \mapsto B(c) - 1], E[\texttt{skip}])}$$

Figure 2. The reduction rules.

the binary operator *op*. We let the sequential composition operator ; be associative with `skip` as an identity, that is,

$$\begin{aligned} &s_1; s_2; s_3 = (s_1; s_2); s_3 = s_1; (s_2; s_3) \\ &\texttt{skip}; s = s; \texttt{skip} = s \end{aligned}$$

The bottom three rules handle communication of signals over channels. A new channel is created with its number of pending signals set to 0. When a sender sends a signal over the channel, the number is incremented. The receiver blocks on a channel until there is a signal over the channel, and decrements the number by receiving the signal.

A *trace*, T, is a finite sequence of states. For $i \leq |T|$, we write $T(i)$ to denote the ith state of T. We write $(S, B, p) \Downarrow T$ if $(S, B, p) = T(1)$ and $T(i) \to T(i+1)$ for $1 \leq i < |T|$. We write $s \Downarrow T$ if $(S, \emptyset, s) \Downarrow T$ for some S such that $\mathit{dom}(S)$ is the set of free locations in s and $\mathit{ran}(S) \subseteq \mathbb{Z}$ (for simplicity, we assume that the initial store only contains

integers). For a set of locations $K \subseteq dom(S)$, we write $(S, B, p)|_K$ for $S|_K$. We extend the same notation to traces, that is, $T|_K$ is a sequence of stores restricted to K such that the ith element is $T(i)|_K$.

We define observational determinism following [14, 6][1]. Informally, s is observationally deterministic if its trace restricted to the updates to the low security locations is independent of thread scheduling and the initial contents of the high security locations.

Definition 2.1 (Observational Determinism) *Let s be a program with no free variables and K be the set of free locations of s. Let $L \subseteq K$ be low security locations and $H \subseteq K$ be high security locations. Let $\bar{H} = K \setminus H$. We say that s is observationally deterministic w.r.t. (L, H) if for any T, T' such that $s \Downarrow T$, $s \Downarrow T'$, and $T(1)|_{\bar{H}} = T'(1)|_{\bar{H}}$, we have $T|_L \prec T'|_L$ or $T'|_L \prec T|_L$.*

Observational determinism is a quite strong notion of security. For instance, observational determinism implies deterministic updates to the low security locations (because $t|_{\bar{H}} = t|_{\bar{H}}$ for any state t). Therefore, for example, the following program is not observationally deterministic w.r.t. $(\{\ell_l\}, \emptyset)$. (There are no high security locations.)

$$\texttt{spawn}\ \{\ell_l := 0\}; \ell_l := 1$$

For motivation behind the definition of observational determinism, we refer to the paper by Zdancewic and Myers [14] (or the paper by Huisman et al. [6]) which contains excellent examples illustrating why a more relaxed notion of security may be problematic.

3 The Type System

We present a type system that guarantees that a typable program is observationally deterministic. We informally describe the basic idea. In short, the type system combines a flow-based analysis for non-interference [14] with a fractional capability-based determinism checker for concurrent programs [2, 13]. Intuitively, the type system applies a security flow analysis to check that there is no flow (including implicit flow from branches and loops) from high security level to low security level, and also infers the program parts that must be at low security level and therefore need to be deterministic. Then, the type system applies a fractional capability analysis to verify that those parts are indeed deterministic. Only checking determinism for parts of the program allows the type system to accept programs that are not totally deterministic (see Section 3.1 and 4 for some examples.)

[1]The definition given here differs somewhat from those in [14] and [6]. It turns out that there are some subtle issues with their definitions, which are discussed in Appendix A

The following example illustrates the basic idea.

$$\begin{array}{l}\texttt{ref}\ x = 0\ \texttt{in}\\ \quad \texttt{spawn}\ \{x := 1\};\\ \quad \texttt{if}\ !x\ \texttt{then}\ \ell_l := 0\ \texttt{else}\ \ell_l := 1\end{array}$$

Note that the program is not observationally deterministic w.r.t. $(\{\ell_l\}, \emptyset)$ because depending on the thread scheduling, ℓ_l is either assigned 0 or 1. To detect this, the type system first infers that, for ℓ_l to be at low security level, the branch condition $!x$ must be at low security level and thus needs to be deterministic. Then, the type system tries to find a fractional capability assignment that would prove the determinism of $!x$, and fails to do so (because it is non-deterministic), and the program is rejected as possibly not observationally deterministic.

We now formally describe the type system. The types are defined as follows.

$$\begin{array}{lcl}\kappa & \in & \textbf{Abstract Locations}\\ \gamma & \in & \textbf{Abstract Channels}\\ \rho & ::= & \kappa \mid \gamma\\ q & ::= & h \mid l\\ \tau & ::= & \mathit{int}^q \mid \mathit{ref}(\kappa, \tau)^q \mid \mathit{chan}(\gamma, \Psi)^q\end{array}$$

Each type is qualified by a *security level*: h or l. Intuitively, values of the type qualified by h may depend on high security information or thread scheduling, whereas those qualified by l are guaranteed to be independent of them. Security levels form the usual two point lattice such that $l \sqsubseteq h$ and $h \not\sqsubseteq l$. The least upper bound of q_1 and q_2 is written $q_1 \sqcup q_2$.

The accessor function *qual* and *ty* are defined as follows.

$$\begin{array}{rcl}\mathit{qual}(\mathit{int}^q) & = & q\\ \mathit{qual}(\mathit{ref}(\kappa, \tau)^q) & = & q\\ \mathit{qual}(\mathit{chan}(\gamma, \Psi)^q) & = & q\\ \mathit{ty}(\mathit{int}^q) & = & \mathit{int}\\ \mathit{ty}(\mathit{ref}(\kappa, \tau)^q) & = & \mathit{ref}(\kappa, \tau)\\ \mathit{ty}(\mathit{chan}(\gamma, \Psi)^q) & = & \mathit{chan}(\gamma, \Psi)\end{array}$$

Security levels induce the subtyping relation

$$\frac{\mathit{qual}(\tau_1) \sqsubseteq \mathit{qual}(\tau_2) \quad \mathit{ty}(\tau_1) = \mathit{ty}(\tau_2)}{\tau_1 \sqsubseteq \tau_2}$$

The type system represents locations statically as *abstract locations*, ranged over by meta variables κ, κ_1, etc. Each abstract location denotes a set of (concrete) locations so that a location of the type $\mathit{ref}(\kappa, \tau)^q$ is in the set represented by κ. Similarly, channels are represented by as *abstract channels*, ranged over by γ, γ_1, etc., each denoting a set of (concrete) channels. Meta variables ρ, ρ_1, etc. range over *abstract resources*, used to refer to both abstract locations and abstract channels. Meta variables Ψ, Ψ_1, etc. range over *capability mappings*, which are functions from abstract resources to non-negative rational numbers $[0, \infty)$.

There are two reasons for using abstract resources. One is to allow aliasing of resources, because both locations and channels are first class objects and so the static system cannot track their identity precisely. The second reason is to group all low security locations in the same abstract location. The type system ensures that accesses to *l-level locations* (i.e., locations with the type of the form $\mathit{ref}(\kappa, \tau)^q$ such that $\mathit{qual}(\tau) = l$) happen observationally deterministically at the granularity of abstract resources. Therefore, if multiple l-level locations are grouped in the same abstract location, then access to all those locations as a whole happen observationally deterministically.

Recall capability mapping (Ψ, Ψ_1, etc.) is a function from abstract resources to non-negative rational numbers $[0, \infty)$. A capability mapping denotes access capabilities to abstract resources. Each thread holds some amount of capabilities, representing the access capabilities of the thread.

Only the threads holding the capability Ψ such that $\Psi(\kappa) \geq 1$ are allowed to write to an l-level abstract location κ, whereas only the threads holding the capability Ψ such that $\Psi(\kappa) > 0$ are allowed to read low security values. (Therefore, the write capability implies the read capability.) The type system ensures that the total amount of capabilities summed across all live threads is at most 1 for any abstraction location. This guarantees that there are no other threads that can access an l-level location when some thread is writing to the location, guaranteeing deterministic low-security accesses to the location.

For example, this scheme soundly rejects the following program where ℓ_l is a low security location. To see this, note that in order to type check both writes to ℓ_l, both threads must be given the capability 1 to access the abstract location representing ℓ_l, but this violates the "at most 1" criteria.

$$\texttt{spawn}\ \{\ell_l := 0\}; \ell_l := 1$$

Only demanding non-negative rationals for reads allow parallel deterministic reads. For instance, the following program, where ℓ_l is a low security location, can be type checked by giving both the spawned and the spawner threads a fractional amount (e.g., 0.5) of the capability to access x.

$$\begin{array}{l}\texttt{ref}\ x = 0\ \texttt{in}\\ \texttt{ref}\ y = 0\ \texttt{in}\\ \quad \texttt{spawn}\ \{\ell_l := !x\}; y := !x\end{array}$$

The type system allows *capability passing* between threads so that multiple threads may access the same l-level location. Capabilities can be passed when threads synchronize over a channel. For instance, to type check the following program (where ℓ_l is a low security location), we pass the capability to access ℓ_l initially held by the spawned thread (i.e., the one that does $\ell_l := 0$) to the spawner thread (i.e., the one that does $\ell_l := 1$) when sending a signal over the channel c.

$$\begin{array}{l}\texttt{chan}\ c\ \texttt{in}\\ \quad \texttt{spawn}\ \{\ell_l := 0; \texttt{send}\ c\};\\ \quad \texttt{wait}\ c; \ell_l := 1\end{array}$$

To statically reason about capability passing, a channel type contains a capability mapping indicating the amount of capabilities passed from the sender to the receiver when communicating over the channel. That is, a channel of the type type $\mathit{chan}(\gamma, \Psi)^q$ can be used to pass Ψ amount of capabilities from the sender to the receiver.

It is necessary to check that capability passing happens observationally deterministically to disallow cases such as the one below where ℓ_l is a low security location.

$$\begin{array}{l}\texttt{chan}\ c\ \texttt{in}\\ \quad \texttt{spawn}\ \{\texttt{wait}\ c; \ell_l := 0\};\\ \quad \texttt{spawn}\ \{\texttt{wait}\ c; \ell_l := 1\};\\ \quad \texttt{send}\ c\end{array}$$

To this end, the type system only allows the threads holding the capability Ψ such that $\Psi(\gamma) \geq 1$ to receive capabilities from channels in γ. As with abstract locations, the type system ensures that the total amount of capabilities summed across all live threads is at most 1 for an abstract channel. Therefore, at any point in time, there is at most one thread that can receive capabilities from the channel. As with locations, capabilities to access channels can also be passed, allowing multiple threads to use the same channel to pass capabilities.

We define arithmetic operations over capabilities. The addition and subtraction of capability mappings are defined point-wise as $\Psi_1 + \Psi_2 = \lambda\rho.\Psi_1(\rho) + \Psi_2(\rho)$ and $\Psi_1 - \Psi_2 = \lambda\rho.\Psi_1(\rho) - \Psi_2(\rho)$. Because capabilities must be non-negative, $\Psi_1 - \Psi_2$ is undefined if $\Psi_1(\rho) < \Psi_2(\rho)$ for some ρ. The relation $\Psi_1 \geq \Psi_2$ is defined as $\forall\rho.\Psi_1(\rho) \geq \Psi_2(\rho)$, and $\Psi_1 > \Psi_2$ is defined as $\forall\rho.\Psi_1(\rho) > \Psi_2(\rho)$. For convenience, we let $\mathit{0}$ denote a constant capability mapping that maps all abstract resources to 0, that is, $\mathit{0} = \lambda\rho.0$. Therefore, for example, $\mathit{0}[\rho \mapsto 1]$ is a capability mapping that maps ρ to 1 and ρ' to 0 for all $\rho' \neq \rho$.

Figure 3 and Figure 4 show the type checking rules. The judgement for expressions is of the form $\Gamma, \Psi \vdash e : \tau$ where Γ is a type environment mapping variables and locations to their types, Ψ is the *pre-capability* before the evaluation of e (and after, because expressions do not change thread's capabilities), and τ is the type of e. The judgement for statements is of the form $\Gamma, \Psi_1, q \vdash s : \Psi_2$ where the *context security level* q is used to capture implicit flow induced by branches, Ψ_1 is the pre-capability of s, and the *post-capability* Ψ_2 is the capability after the evaluation of s.

We briefly discuss each rule. **VAR**, **INT**, **LOC**, **OP**, **SEQ**, and **SKIP** are self-explanatory. **IF** and **WHILE** increase the security level inside the branch/loop bodies by

$$\frac{}{\Gamma, \Psi \vdash x : \Gamma(x)}\ \textbf{VAR} \qquad \frac{}{\Gamma, \Psi \vdash n : \mathit{int}^{q_1}}\ \textbf{INT} \qquad \frac{}{\Gamma, \Psi \vdash \ell : \Gamma(\ell)}\ \textbf{LOC}$$

$$\frac{\Gamma, \Psi \vdash e_1 : \mathit{int}^{q_1} \quad \Gamma, \Psi_1 \vdash e_2 : \mathit{int}^{q_2} \quad q_1 \sqcup q_2 \sqsubseteq q_3}{\Gamma, \Psi \vdash e_1\ \mathit{op}\ e_2 : \mathit{int}^{q_3}}\ \textbf{OP}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{ref}(\kappa, \tau_1)^{q_1} \quad \tau_1 \sqsubseteq \tau \quad q_1 \sqsubseteq \mathit{qual}(\tau) \quad \mathit{qual}(\tau) = l \Rightarrow \Psi(\kappa) > 0}{\Gamma, \Psi \vdash\ !e : \tau}\ \textbf{READ}$$

Figure 3. The type checking rules for expressions.

$$\frac{\Gamma, \Psi \vdash e : \mathit{int}^{q_1} \quad \Gamma, \Psi, q_2 \vdash s_1 : \Psi_1 \quad \Gamma, \Psi, q_2 \vdash s_2 : \Psi_2 \quad q \sqcup q_1 \sqsubseteq q_2 \quad \Psi_1 \geq \Psi_3 \quad \Psi_2 \geq \Psi_3}{\Gamma, \Psi, q \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2 : \Psi_3}\ \textbf{IF}$$

$$\frac{\Gamma, \Psi_1 \vdash e : \mathit{int}^{q_1} \quad \Gamma, \Psi_1, q_2 \vdash s : \Psi_2 \quad q \sqcup q_1 \sqsubseteq q_2 \quad \Psi_2 \geq \Psi_1 \quad \Psi \geq \Psi_1}{\Gamma, \Psi, q \vdash \texttt{while}\ e\ \texttt{do}\ s : \Psi_1}\ \textbf{WHILE}$$

$$\frac{}{\Gamma, \Psi, q \vdash \texttt{skip} : \Psi}\ \textbf{SKIP} \qquad \frac{\Gamma, \Psi, q \vdash s_1 : \Psi_1 \quad \Gamma, \Psi_1, q \vdash s_2 : \Psi_2}{\Gamma, \Psi, q \vdash s_1; s_2 : \Psi_2}\ \textbf{SEQ}$$

$$\frac{\Gamma, \Psi \vdash e : \tau_1 \quad \tau_1 \sqsubseteq \tau \quad \Gamma[x \mapsto \mathit{ref}(\kappa, \tau)^{q_1}], \Psi, q \vdash s : \Psi_1}{\Gamma, \Psi, q \vdash \texttt{ref}\ x = e\ \texttt{in}\ s : \Psi_1}\ \textbf{REF} \qquad \frac{\Gamma, \Psi \vdash e_1 : \mathit{ref}(\kappa, \tau)^{q_1} \quad \Gamma, \Psi \vdash e_2 : \tau_1 \quad \tau_1 \sqsubseteq \tau \quad q \sqcup q_1 \sqsubseteq \mathit{qual}(\tau) \quad \mathit{qual}(\tau) = l \Rightarrow \Psi(\kappa) \geq 1}{\Gamma, \Psi, q \vdash e_1\ \texttt{:=}\ e_2 : \Psi}\ \textbf{WRITE}$$

$$\frac{\Gamma, \Psi_1, q \vdash s : \Psi_2}{\Gamma, \Psi, q \vdash \texttt{spawn}\ \{s\} : \Psi - \Psi_1}\ \textbf{SPAWN} \qquad \frac{\Gamma[x \mapsto \mathit{chan}(\gamma, \Psi_1)^{q_1}], \Psi, q \vdash s : \Psi_2}{\Gamma, \Psi, q \vdash \texttt{chan}\ x\ \texttt{in}\ s : \Psi_2}\ \textbf{NEWC}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{chan}(\gamma, \Psi_1)^l}{\Gamma, \Psi, l \vdash \texttt{send}\ e : \Psi - \Psi_1}\ \textbf{SEND} \qquad \frac{\Gamma, \Psi \vdash e : \mathit{chan}(\gamma, \Psi_1)^l \quad 0 < \Psi_1 \Rightarrow \Psi(\gamma) \geq 1}{\Gamma, \Psi, l \vdash \texttt{wait}\ e : \Psi + \Psi_1}\ \textbf{RECV}$$

Figure 4. The type checking rules for statements.

the security level of the branch/loop condition to account for the implicit information flow. Note that these rules make sure that capabilities are "conserved" through the sequential flow of computation, that is, the post-capability may not exceed their pre-capability. At **SPAWN**, the parent thread gives a part of its capability (Ψ_1) to the newly created thread, and so $\Psi - \Psi_1$ amount of capability is left for the continuation of the parent thread.

REF, **READ**, and **WRITE** type location accesses. **REF** is self-explanatory. At **WRITE**, the hypothesis $q \sqcup q_1 \sqsubseteq \mathit{qual}(\tau)$ ensures that the context security level (i.e., q) is below that of the location (i.e., $\mathit{qual}(\tau)$). This is needed to disallow implicit information flow such as

$$\texttt{if}\ !\ell_h\ \texttt{then}\ \ell_l\ \texttt{:=}\ 0\ \texttt{else}\ \ell_l\ \texttt{:=}\ 1$$

where ℓ_h is a high security location and ℓ_l is a low security location. Also, the security level of the reference (i.e., q_1) must be below $\mathit{qual}(\tau)$. This is needed because locations are first class, and disallows programs like the one below.

$$\begin{array}{l}\texttt{ref}\ x = \ell_{l1}\ \texttt{in}\\ \quad \texttt{if}\ !\ell_h\ \texttt{then}\ x\ \texttt{:=}\ \ell_{l2}\ \texttt{else skip};\\ \quad \ell_{l1}\ \texttt{:=}\ 1;\end{array}$$

where ℓ_h is a high security location and ℓ_{l1}, ℓ_{l2} are low security locations. As remarked before, the hypothesis $\mathit{qual}(\tau) = \ell \Rightarrow \Psi(\kappa) \geq 1$ asserts that to write to a low security location, the thread must own the full (i.e., 1) capability to access it. Note that **WRITE** allows writing a low security level value to an h-level location. In such a case, no capability constraints are imposed.

As remarked before **READ**'s hypothesis $\mathit{qual}(\tau) = l \Rightarrow \Psi(\kappa) > 0$ ensures that only a thread holding a non-zero capability may read a low security value from the location. Also, **READ** ensures $q_1 \sqsubseteq \mathit{qual}(\tau)$. As in **WRITE**, this is

$$\frac{\Gamma, \Psi \vdash e : \mathit{chan}(\gamma, \Psi_1)^{q_1} \quad 0 < \Psi_1 \Rightarrow q = q_1 = l}{\Gamma, \Psi, q \vdash \texttt{send}\ e, \Psi - \Psi_1}\ \textbf{SEND'}$$

$$\frac{\Gamma, \Psi \vdash e : \mathit{chan}(\gamma, \Psi_1)^{q_1} \quad 0 < \Psi_1 \Rightarrow (\Psi(\gamma) \geq 1 \wedge q = q_1 = l)}{\Gamma, \Psi, q \vdash \texttt{wait}\ e : \Psi + \Psi_1}\ \textbf{RECV'}$$

Figure 5. The relaxed rules for signal send and receive.

needed because locations are first class and disallows programs like the one below.

```
ref x = ℓ_a in
  if !ℓ_h then x := ℓ_b else skip;
  ℓ_l := !!x
```

where ℓ_h is a high security location and ℓ_l is a low security locations. (ℓ_a, ℓ_b are some arbitrary locations.) Note that **READ** allows reading a high security value from an l-level location. In such a case, no capability constraints are imposed.

NEWC, **SEND**, and **RECV** handle signals. **NEWC** checks channel allocation, and is self-explanatory. Note that the post-capabilities in **SEND** and **RECV** reflect the capability passing remarked earlier, that is, the capability Ψ_1 is passed from the sender to the receiver.

At **RECV**, the hypothesis $0 < \Psi_1 \Rightarrow \Psi(\gamma) \geq 1$ ensures that if the passed capability is non-zero, then the thread must have the full capability to access the channel. As remarked before, this is needed to ensure that the capability passing happens deterministically. **SEND** and **RECV** require the context security level and the security level of the channel to be at l. This disallows cases such as the one below where ℓ_h is a high security location and ℓ_l is a low security location.

```
chan c in
chan d in
  spawn {wait c; ℓ_l := 0};
  spawn {wait d; ℓ_l := 1};
  if !ℓ_h then send c; else send d;
```

Note that this program would type check if the hypothesis were missing because it would allow sending the capability to access ℓ_l via both the channel c and d. The type system remains sound if the rule is relaxed so that the security levels need to be l only if the passed capability is non-zero, as shown in Figure 5. Unfortunately, these rules induce cyclic dependencies between capability constraints and security level constraints, and make type inference somewhat more complex.

We now state the soundness of the type system.

Definition 3.1 (Well-typed Program) *Let s be a program with no free variables and L, H be its low security locations and high security locations. We write $\vdash (s, L, H)$ if there exists Γ such that*

(1) For all free location ℓ of s, $\Gamma(\ell) = \mathit{ref}(\kappa, \mathit{int}^{q_1})^{q_2}$ for some κ, q_1, q_2.

(2) For all $\ell \in H$, $\Gamma(\ell) = \mathit{ref}(\kappa, \mathit{int}^{h})^{q}$ for some κ, q.

(3) There exists κ such that for all $\ell \in L$, $\Gamma(\ell) = \mathit{ref}(\kappa, \mathit{int}^{l})^{q}$ for some q.

(4) $\Gamma, \Psi, q \vdash s : \Psi'$ for some Ψ, q, Ψ' such that $\Psi(\rho) \leq 1$ for all ρ.

The condition (1) follows from the fact that initial stores map locations to integers. The condition (2) ensures that high security locations have high security level. The condition (3) ensures that low security locations share the same abstract location, and have low security level. The condition (4) ensures that s can be typed with a pre-capability of at most 1 for any abstract resource.

Theorem 3.2 (Soundness) *If $\vdash (s, L, H)$ then s is observationally deterministic w.r.t. (L, H).*

The proof of the theorem appears in Appendix B.

3.1 Example

Consider the following program.

```
chan c in
  spawn {
    ℓ_a := 1;
    wait c;
    if !ℓ_h then ℓ_b := 1 else 0;
    ℓ_2 := 1;      send c};
  ℓ_a := 0; ℓ_b := 0; ℓ_c := 0;
  send c; wait c;
  ℓ_l := !ℓ_c
```

The program is observationally deterministic w.r.t. $(\{\ell_l\}, \{\ell_h\})$ as the content of ℓ_c at the last line of the program must be 1 because the reads and writes are synchronized properly via communication over the channel c. Note that the contents of ℓ_a at the end of the program is non-deterministic as it depends on the thread scheduling, and the content of ℓ_b at the end of the program depends on the value of the high security location.

The type system can type check this program. The key is to give the channel c the type of the form

```
ref x = 0 in
ref i = 0 in
ref j = 0 in
chan c1 in
chan c2 in
chan d in
  spawn {while !i < 10 do
    (x := !i; i := !i + 1; send c1; wait c2)};
  spawn {while 1; do
    (wait c1; wait d; ℓl := !x; send c2)};
  spawn {while !j < 10 do (j := !j + 1; send d)};
  spawn {while 1 do wait d}
```

Figure 6. Producer-consumer example.

chan$(\gamma, 0[\kappa \mapsto 1])^q$ where ℓ_c is given the type of the form *ref*$(\kappa, \mathit{int}^l)^l$. This allows passing of the (full) capability to access ℓ_c to the spawned thread after the write to ℓ_c by the spawner thread, and back to the spawner thread after the spawned thread is done with its write to ℓ_c. Thus, $!\ell_c$ at the last line can be given a low security type, and the program type checks.

4 Confluence and Observational Determinism

Note that the example above is not confluent as updates to ℓ_a cannot be "undone", though the low security part of the store (i.e., ℓ_l) is confluent. However, in general, observational determinism does not imply confluence even restricted to the low security locations. We show that our type system can actually check some of such "observationally non-confluent" programs to be observationally deterministic.

Let $t_1 \rightarrow^* t_2$ be 0 or more reduction steps from t_1 to t_2. Formally, a state t is said to be *confluent* if $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$ imply that there exist states t_1', t_2' such that $t_1 \rightarrow^* t_1'$ and $t_1 \rightarrow^* t_2'$ and $t_1' \equiv t_2'$ where $\equiv$ is some suitable equivalence relation. A program s is confluent if for all S such that *dom*(S) is the set of free locations in s, $(S, \emptyset, s)$ is confluent. A reasonable choice for $\equiv$ may be equality up to renaming of variables/locations/channels and process reordering. A more relaxed notion can be obtained by only requiring such a relation on the stores, or some portion of the stores.

Consider the simple producer-consumer program shown in Figure 6. The program spawns four threads, the first thread is the producer thread that writes $0, 1, \ldots, 9$ to x. The second thread is the consumer thread that reads from x. Via channels c_1 and c_2, the second thread (the consumer) synchronizes with the first thread (the producer) to read (some prefix of) $0, 1, \ldots 9$. The consumer thread also waits on signals sent via d, which are sent by the third thread. Finally, the fourth thread also receives signals via d.

The program is observationally deterministic w.r.t. $(\{\ell_l\}, \emptyset)$. Note that because there are no high security locations, the only source of information leak is through non-determinism due to the thread scheduling. The program is observationally deterministic because the updates to ℓ_l is some prefix of $0, 1, \ldots, 9$ regardless of the thread scheduling. Our type system is able to type check this program by assigning the location ℓ_l the type *ref*$(\kappa, \mathit{int})^l$, channels c_1, c_2 the type *chan*$(\gamma, 0[\kappa \mapsto 1])^l$ for some γ, and channel d the type *chan*$(\gamma', 0)^q$ for some q, γ'.

This program is not confluent even restricted to the low security location ℓ_l. To see this, consider a series of reductions in which the loop in the fourth thread waits on d until after the consumer thread writes to ℓ_l ten times. In this series, the updates to ℓ_l are exactly $0, 1, \ldots, 9$. Now, consider another series of execution where the fourth thread receives a signal from d at least once before the consumer thread writes to ℓ_l ten times. Then, the sequence of updates to ℓ_l is some strict prefix of $0, 1, \ldots, 9$. It is impossible for the two series to reduce to states that have the same value for ℓ_l, in particular, the latter series cannot reach the 10th update.

In general, even when restricted to differences in low security locations, it would be difficult to guarantee confluence because it would mean that any potential operation preceding an update to a low security location must be deterministically blocking or non-blocking. In particular, this implies that termination/non-termination of the preceding loops must be deterministic.

5 Type Inference

We give a polynomial time type inference algorithm (more precisely, a typability algorithm). That is, given (s, L, H), the algorithm decides whether $\vdash (s, L, H)$. We informally describe the algorithm. The algorithm is separated in three phases. The first phase is a security flow analysis phase that decorates the derivation tree with security levels. We make the first phase find the greatest security level assignments (i.e., preferring h over l wherever possible). If no solution can be found in this phase (e.g., trying to find q such that $h \sqsubseteq q$ and $q \sqsubseteq l$), the algorithm rejects the program as untypable. The first phase ignores capability constraints. The second phase finds capability assignments for channels, which involves resolving the constraints $0 < \Psi_1 \Rightarrow \Psi(\gamma) \geq 1$ from **RECV**, i.e., determines when $\Psi_2(\gamma) < 1$ so that it must be the case that $\Psi = 0$. Finally, the third phase uses the information obtained from the first two phases to find capability assignments for locations. The third phase uses the inferred security levels to determine which $\Psi(\rho) > 0$ and $\Psi(\rho) \geq 1$ constraints to dis-

$$\frac{}{\Delta, \varphi \vdash x : \Delta(x); \emptyset} \quad \frac{\alpha, r \text{ fresh}}{\Delta, \varphi \vdash n : \alpha^r; \{\alpha = \mathit{int}\}} \quad \frac{}{\Delta, \varphi \vdash \ell : \Delta(\ell); \emptyset}$$

$$\frac{\Delta, \varphi_1 \vdash e_1 : \alpha_1^{r_1}; C_1 \quad \Delta, \varphi_2 \vdash e_2 : \alpha_2^{r_2}; C_2 \quad r \text{ fresh}}{\Delta, \varphi_1 \vdash e_1 \ \mathit{op}\ e_2 : \alpha_2^r; C_1 \cup C_2 \cup \{\alpha_1 = \alpha_2 = \mathit{int}, r_1 \sqsubseteq r, r_2 \sqsubseteq r, \varphi_1 = \varphi_2\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha^r; C \quad \alpha_1, r_1, r_2, \varrho \text{ fresh}}{\Delta, \varphi \vdash !e : \alpha_1^{r_1}; C \cup \{\alpha = \mathit{ref}(\varrho, \alpha_1^{r_2}), r \sqsubseteq r_1, r_2 \sqsubseteq r_1, (r_1 = l \Rightarrow \varphi(\varrho) > 0)\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha_1^{r_1}; C_1 \quad \Delta, \varphi_1, r \vdash s : \varphi_2; C_2 \quad \varrho, r_2, r_3 \text{ fresh}}{\Delta, \varphi, r \vdash \texttt{ref}\ x = e\ \texttt{in}\ s : \varphi_2; C_1 \cup C_2 \cup \{\mathit{ref}(\varrho, \alpha_1^{r_3})^{r_2} = \Delta(x), r_1 \sqsubseteq r_3, \varphi = \varphi_1\}}$$

$$\frac{\Delta, \varphi \vdash e : \alpha^{r_1}; C \quad r, \zeta, \varphi_1, \varphi_2 \text{ fresh}}{\Delta, \varphi, r \vdash \texttt{wait}\ e : \varphi_2; C \cup \{\alpha = \mathit{chan}(\zeta, \varphi_1), r = r_1 = l, \varphi_2 = \varphi + \varphi_1, (0 < \varphi_1 \Rightarrow \varphi(\zeta) \geq 1)\}}$$

Figure 7. Representative type inference rules.

charge (i.e., the hypothesis $\mathit{qual}(\tau) = l \Rightarrow \ldots$ at **WRITE** and **READ**). The second and the third phase finds the capability assignments by reducing the capability constraint satisfaction problem to linear programming instances.

5.1 Phase 1

The first phase is mostly a standard unification-based type inference, generating security level constraints and capability constraints on the side. Figure 7 shows a few representative constraint generation rules (corresponding to the type rules **VAR**, **INT**, **LOC**, **OP**, **READ**, **REF**, and **RECV**). Here, r's are security level variables, φ's are capability mapping variables, α's are type variables (actually, types with the outer-most security level stripped), ϱ's are abstract location variables, and ζ's are abstract channel variables. Each rule is a straightforward syntax-directed constraint generation rule for a rule from Figure 3 and Figure 4.

The judgement for expressions, $\Delta, \varphi \vdash e : \alpha^r; C$, is read "given environment Δ, e is inferred to have the type α^r and the pre-capability φ, with the set of constraints C." The judgement for statements is of the form $\Delta, \varphi_1, r \vdash s : \varphi_2; C$, and is read "under given environment Δ, s is inferred to have the pre-capability φ_1, the post-capability φ_2, and the context security level r, with the set of constraints C." For simplicity, we assume that `let`-bound variables are distinct. We initialize Δ to map each variable and location to a fresh type variable qualified by a fresh security level variable (i.e., α^r's where each α and r are distinct). Then, we visit each expression in a bottom up manner to build the set of constraints, applying a syntax-directed constraint generation rule at each AST node.

Analyzing a program s, we get $\Delta, \varphi_s, r \vdash s : \varphi'_s; C$. We use the generated constraints C and the pre-capability of the whole program, φ_s, in the rest of the algorithm.

We add to C the following constraints to enforce the conditions (1), (2), and (3) from Definition 3.1.

- For each ℓ free in the program s, the constraint $\Delta(\ell) = \mathit{ref}(\varrho, \mathit{int}^{r_1})^{r_2}$ where ϱ, r_1, r_2 are fresh.
- For each $\ell \in H$, the constraint $\Delta(\ell) = \mathit{ref}(\varrho, \mathit{int}^h)^r$ where ϱ, r are fresh.
- Let ϱ be a fresh abstract location variable, then, for each $\ell \in L$, the constraint $\Delta(\ell) = \mathit{ref}(\varrho, \mathit{int}^l)^r$ where r is fresh.

The resulting set of constraints contains six kinds of constraints:

(a) Type unification constraints: $\sigma_1 = \sigma_2$.

(b) Abstract resource constraints: $\varrho_1 = \varrho_2$ and $\zeta_1 = \zeta_2$

(c) Security level constraints: $o_1 \sqsubseteq o_2$

(d) Capability inequality constraints: $\phi_1 \geq \phi_2$.

(e) Channel access constraints: $0 < \varphi_1 \Rightarrow \varphi_2(\zeta) \geq 1$.

(f) Location access constraints: $r = l \Rightarrow \varphi(\varrho) \geq 1$ and $r = l \Rightarrow \varphi(\varrho) > 0$.

where

$$\begin{aligned} \sigma &::= \alpha \mid \mathit{ref}(\varrho, \alpha^r) \mid \mathit{chan}(\varrho, \varphi) \mid \mathit{int} \\ \phi &::= \varphi \mid \phi + \phi \mid \phi - \phi \\ o &::= r \mid l \mid h \end{aligned}$$

Note that an equality constraint $o_1 = o_2$ can be expressed by the constraints $o_1 \sqsubseteq o_2$ and $o_2 \sqsubseteq o_1$, and an equality constraint $\phi_1 = \phi_2$ can be expressed by the constraints $\phi_1 \geq \phi_2$ and $\phi_2 \geq \phi_1$.

Input : C and φ_e computed by the first phase
Output : ZC

$NonNeg := \{\varphi \geq 0 \mid \varphi$ appears in $C\}$
$CI :=$ Capability inequality constraints in C
$Consts := \{1 \geq \varphi_e\} \cup CI \cup NonNeg$
$Z := \emptyset$
loop :
  for each $\zeta \notin Z$ do
    if $\neg LPSat(Consts \cup CC(\zeta) \cup \{cz(\zeta) \mid \zeta \in Z\})$
      then $Z := Z \cup \{\zeta\}$
        goto loop
$ZC := \{cz(\zeta) \mid \zeta \in Z\}$

Figure 8. The second phase of the type inference algorithm.

The first phase deals with the constraint kinds (a), (b), and (c), and leaves the rest for the other two phases. Constraints of the kind (a) can be solved by the standard unification algorithm, which may generate more constraints of the kinds (b) and (c) (as types may contain security levels and capability mappings). Constraints of the kind (b) can also be resolved by the standard unification algorithm. The unification may fail to find a solution to the constraints of the kind (a), e.g., if the program uses an integer constant as a channel. If so, the algorithm rejects the program as untypable at this point.

After processing all constraints of the kind (a) and (b), the algorithm tries to find the greatest solution for the constraints of the kind (c). By greatest, we mean the solution that assigns h to the largest set of security level variables. The reason for wanting the greatest solution is to prevent discharging local access constraints (f) as much as possible to reduce capability constraints solved in the third phase. The greatest solution can be obtained by repeatedly selecting a constraint $r \sqsubseteq l$ from the constraint set, and substituting l for r until there are no more constraints of that kind. The constraints are unsatisfiable if $h \sqsubseteq l$ appears in the constraints as a result, in which case, the program is rejected as untypable. Otherwise, all unassigned security level variables are set to h, and phase 1 returns the resulting constraint set C with all the security level variables, type variables, and abstract location/channel variables substituted by their computed assignments.

5.2 Phase 2

The second phase takes care of the channel access constraints, i.e., the constraints of the kind (e) and (d) (constraints (d) are also used in phase 3). We informally describe the process. Note that each channel access constraint, that is, $0 < \varphi_1 \Rightarrow \varphi_2(\zeta) \geq 1$, can be restated as $\varphi_2(\zeta) < 1 \Rightarrow 0 \geq \varphi_1$. For each ζ (that is, its equivalence class obtained by the unification in phase 1), we look for a capabilities assignment that minimizes discharging of the right hand side of $\varphi_2(\zeta) < 1 \Rightarrow 0 \geq \varphi_1$. Note that the constraint resolution in phase 1 guarantees that if $\varphi_2(\zeta) < 1 \Rightarrow 0 \geq \varphi_1$ and $\varphi'_2(\zeta) < 1 \Rightarrow 0 \geq \varphi'_1$ are in C, then $\varphi_1 = \varphi'_1$ (more precisely, it is implied by the channel inequality constraints of the kind (d)) because φ_1, φ'_1 are both denote the capabilities passed when communicating over a channel in ζ. Thus, it suffices to add a single constraint $0 \geq \varphi_1$ to discharge all constraints for ζ if their left hand sides (in the original form) are found to be unsatisfiable.

We now describe the process more formally. For each ζ, let $CC(\zeta)$ be the right hand side of set of all channel access constraints for ζ, and $cz(\zeta)$ is negation of one of the left hand sides. More precisely, $CC(\zeta)$ and $cz(\zeta)$ are defined as follows.

$$CC(\zeta) = \{\varphi_2 \geq 1 \mid (0 < \varphi_1 \Rightarrow \varphi_2(\zeta) \geq 1) \in C\}$$
$$cz(\zeta) = 0 \geq \varphi_2 \text{ such that } (0 < \varphi_1 \Rightarrow \varphi_2(\zeta) \geq 1) \in C$$

The algorithm maintains the set Z of ζ's that are found to have unsatisfiable $CC(\zeta)$, and tries to minimize Z. We start with $Z = \emptyset$, and iterate over each ζ and try to satisfy $CC(\zeta)$ along with the capability inequality constraints and the constraints $\{cz(\zeta) \mid \zeta \in Z\}$. If unsatisfied, we add ζ to Z, and repeat the process all over. Eventually, the algorithm comes to a fixed point where no more elements are added to Z.

Figure 8 shows the pseudo-code implementing phase 2. The algorithm takes the set of constraints C computed by phase 1 as the input. The algorithm instantiates each capability variable as a fresh linear programming variable so that the procedure $LPSat$ checks whether the given set of rational linear inequality constraints is satisfiable. Such a procedure can be implemented by well-known linear programming algorithms such as the simplex algorithm and interior points methods. Recall that φ_e is the pre-capability of the whole program. The constraint $1 \geq \varphi_e$ enforces the condition (3) from Definition 3.1. The constraints $NonNeg$ enforces the non-negativity of capabilities. The algorithm returns ZC which contains the capability variables that must be set to 0.

5.3 Phase 3

The third phase completes the type inference by taking care of the location access constraints (i.e., the constraints of the kind (f)). Like phase 2, we use linear programming to find satisfying assignments for the location access capabilities. For each ϱ, let $LC(\varrho)$ be the set of constraints consisting of the right hand sides of the discharged location access constraints for ϱ. More precisely, $LC(\varrho)$ can be defined as

follows.

$$LC(\varrho) = \{\varphi \geq 1 \mid (l = l \Rightarrow \varphi(\varrho) \geq 1) \in C\} \cup \{\varphi > 0 \mid (l = l \Rightarrow \varphi(\varrho) > 0) \in C\}$$

where C is the constraints computed by phase 1. Note that phase 1 has computed assignments for all the security level variables and so we know which location access constraints to discharge at this point. As in phase 2, let

$$NonNeg := \{\varphi \geq 0 \mid \varphi \text{ appears in } C\}$$
$$CI = \text{Capability inequality constraints in } C$$

Then, we check whether $LPSat(LC(\varrho) \cup NonNeg \cup ZC \cup CI \cup \{1 \geq \varphi_e\})$ where ZC was computed by phase 2. If the constraints are satisfiable for all ϱ, then the program is found to be typable. Otherwise, the program is rejected as untypable.

Handling Strict Inequalities Note that the constraint $\varphi > 0$ is strict, which is often not (directly) supported by linear programming algorithms. To this end, we add a fresh linear programming variable ϵ and replace each $\varphi > 0$ with $\varphi \geq \epsilon$, and set the objective function to be ϵ. We ask the linear programming algorithm to find a solution that maximizes ϵ. If the solver returns a solution such that $\epsilon > 0$, then we have found a satisfying assignment. Otherwise, the constraint set is rejected as unsatisfiable.

5.4 Computational Complexity

We discuss the computational complexity of the type inference algorithm. The size of linear programming instances solved in phase 2 and phase 3 is linear in the size of the capability constraints generated by phase 1. The number of linear programming instances solved in phase 2 is at most quadratic to the number of distinct equivalence classes ζ from phase 1, and the number of linear programming instances solved in phase 3 is at equal to the number of distinct equivalence classes ϱ from phase 1. The linear programming instances from phase 1 and phase 2 can be solved in polynomial time by linear programming algorithms such as interior points methods. Hence, both phase 1 and phase 2 can be solved in time polynomial in the size of the constraints generated by phase 1.

Finally, phase 1 can resolve the type and security levels constraints in time polynomial in the size of the program, and therefore, the size of the constraints passed to phase 2 and 3 is at most polynomial in the size of the program. Hence, the total running time is polynomial in the size of the program.

6 Related Work

Most type systems (e.g., [5, 9, 7]) for non-interference in concurrent programs are designed to check a security notion weaker than or incomparable to observational determinism, such as possibilistic security. As such, they are unsuitable for enforcing observational determinism.

Zdancewic and Myers [14] propose to verify observational determinism by applying security flow analysis and determinism checking (in fact, confluence) separately. In contrast, our type system utilizes information obtained from the security flow analysis to check determinism only for the parts of the program that are inferred to require determinism. Among other things, this scheme allows the type system to verify some non-confluent observationally-deterministic programs.

Huisman et al. [6] propose to check observational determinism via self composition [4, 1]. Because they use a different definition of observational determinism (see Appendix A), their self composition does not reduce to a safety problem. It is easy to reduce observational determinism defined in this paper to a safety problem via self composition. Combining security type systems with self composition has been investigated for sequential programs [12, 8]. We leave for future work to investigate if they can be combined for checking observational determinism.

The use of fractional capabilities for checking determinism is inspired by Terauchi and Aiken's type system for checking determinism of concurrent programs [13], which was in turn inspired by static capability systems for reasoning about program resources [2, 3]. However, their type system enforces confluence and demands determinism over the entire program, whereas the type system in this paper only checks determinism for some parts of the program.

7 Conclusions

We have presented a type system for verifying observational determinism. The type system enforces observational determinism by itself, and does not require the type checked program to be confluent. The type system combines a flow-based analysis for non-interference with a fractional capability-based determinism checker so that only the parts that are inferred to be relevant for security are checked for determinism. We have also presented a polynomial time type inference algorithm that utilizes linear programming.

References

[1] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*, pages 100–114, Pacific Grove, CA, June 2004. IEEE Computer Society.

[2] J. Boyland. Checking interference with fractional permissions. In *Static Analysis, Tenth International Symposium*, pages 55–72, San Diego, CA, June 2003.

[3] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, Texas, Jan. 1999.

[4] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security, WITS*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.

[5] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 81–92, Portland, Oregon, Jan. 2002.

[6] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop, (CSFW-19 2006)*, pages 3–, Venice, Italy, July 2006. IEEE Computer Society.

[7] N. Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[8] D. A. Naumann. From coupling relations to mated invariants for checking information flow. In *Computer Security - ESORICS 2006, 11th European Symposium on Research in Computer Security, Proceedings*, pages 279–296, Hamburg, Germany, Sept. 2006.

[9] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002*, pages 320–330.

[10] A. W. Roscoe. CSP and determinism in security modelling. In *SP'95: Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 114–127, Washington, DC, 1995. IEEE Computer Society.

[11] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[12] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, Eleventh International Symposium*, pages 352–367, Verona, Italy, August 2004.

[13] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. In *Concurrency Theory, 17th International Conference*, volume 4137, pages 218–232, Bonn, Germany, Aug. 2006.

[14] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003)*, pages 29–, Pacific Grove, CA, June 2003. IEEE Computer Society.

A Definitions of Observational Determinism

The definition of observational determinism in this paper differs from the ones given previously in the literature [14, 6]. Zdancewic and Myers' original definition can be obtained by weakening ours such that the traces only need to agree on updates to individual locations in L instead of L as a whole. More precisely, their notion can be stated as follows. (Only the last sentence differs from Definition 2.1.)

Definition A.1 (Observational Determinism [14]) *Let s be a program with no free variables and K be the set of free locations of s. Let $L \subseteq K$ be low security locations and $H \subseteq K$ be high security locations. Let $\bar{H} = K \setminus H$. We say that s is observationally deterministic w.r.t. (L, H) if for any T, T' such that $s \Downarrow T$, $s \Downarrow T'$, and $T(1)|_{\bar{H}} = T'(1)|_{\bar{H}}$, we have $T|_{\{\ell\}} \prec T'|_{\{\ell\}}$ or $T'|_{\{\ell\}} \prec T|_{\{\ell\}}$ for all $\ell \in L$.*

Huisman et al. [6] noted an issue with this definition, citing the following program as an example that leaks information while being observationally deterministic.

$$\ell_{l1} := 0; \ell_{l2} := 0;$$
$$\texttt{while } !\ell_h > 0 \texttt{ do } (\ell_{l1} := !\ell_{l1} + 1; \ell_h := \ell_h - 1);$$
$$\ell_{l2} := 1$$

Here, ℓ_h is a high security location, and ℓ_{l1}, ℓ_{l2} are low security locations. The program satisfies Definition A.1 as a sequence of updates to ℓ_{l1} is always a prefix of $0, 1, 2, \ldots$, and a sequence of updates to ℓ_{l2} is always a prefix of $0, 1$. But the attacker can learn about the content of ℓ_h by checking the value written to ℓ_{l1} just before 1 is written to ℓ_{l2}

Our definition of observational determinism is strong enough to reject this program. For instance, with the initial content of ℓ_h set to 1, the sequence of updates to the low security locations is

$$\ell_{l1} \leftarrow 0, \ell_{l2} \leftarrow 0, \ell_{l1} \leftarrow 1, \ell_{l2} \leftarrow 1$$

but with the initial content of ℓ_h set to 2, the sequence of updates to the low security locations is

$$\ell_{l1} \leftarrow 0, \ell_{l2} \leftarrow 0, \ell_{l1} \leftarrow 1, \ell_{l1} \leftarrow 2, \ell_{l2} \leftarrow 1$$

Huisman et al. [6] proposed yet another definition of observational determinism as a remedy for the above issue. Essentially their definition still enforces observational determinism only per location basis, but requires that the traces to eventually be able to perform all the updates to a low security location made by the other traces. To formally state their definition, we need to introduce infinite traces. For a state t, we write $t \Downarrow_\infty U$ where U is a infinite sequence of states such that either $U(1) = t$ and $U(i) \to U(i+1)$ for all $i \geq 1$, or there exists $n \geq 1$ where $U(1) = t$, $U(i) \to U(i+1)$ for all $1 \leq i < n$, there exists no state t' such that $U(n) \to t'$, and $U(i) = U(i+1)$ for all $i \geq n$. We write $s \Downarrow_\infty U$ if $(S, \emptyset, s) \Downarrow_\infty U$ where *dom*(S) is the set of free locations in e. We extend restriction operation ($|_K$) and stutter equivalence ($\sim$) to infinite strings in a straightforward way. Then, their definition of observational determinism can be stated as follows.

Definition A.2 (Observational Determinism [6]) *Let s be a program with no free variables and K be the set of free locations of s. Let $L \subseteq K$ be low security locations and $H \subseteq K$ be high security locations. Let $\bar{H} = K \setminus H$. We say that s is observationally deterministic w.r.t. (L, H) if for any U, U' such that $s \Downarrow_\infty U$, $s \Downarrow_\infty U'$, and $U(1)|_{\bar{H}} = U'(1)|_{\bar{H}}$, we have $U|_{\{\ell\}} \sim U'|_{\{\ell\}}$ for all $\ell \in L$.*

There are two issues with this definition. One is that it still allows information leak as seen in the program below.

$$\texttt{if } !\ell_h \texttt{ then } \ell_{l1} := 1; \ell_{l2} := 1 \texttt{ else } \ell_{l2} := 1; \ell_{l1} := 1$$

Here, ℓ_h is a high security location and ℓ_{l1}, ℓ_{l2} are low security locations. The program is observationally deterministic according to their definition (and not according to ours). But, the attacker can learn information about the content of ℓ_h by observing if ℓ_{l1} is updated before ℓ_{l2}.

The second issue is that checking observational determinism according to their definition requires the static analysis to be able to answer a liveness[2] question of the form "eventually, the traces agree on each other, forever." For this reason, the self composition reduction in their paper [6] is not a safety problem. It also seems difficult to enforce such a requirement via type-based methods without being overly conservative.

B Proof of Soundness

We give a rough outline of the proof. The idea is to define an instrumented semantics that records updates to the low security locations. Then, confluence with this semantics implies determinism restricted to updates to the low security locations. Unfortunately, our type system does not guarantee confluence, and we also have to take in account of high security locations. To this end, we define an erasure operation that erases all portions of the program that are typable in a high security program context or has a high security level type. In addition to recording updates for each abstract location, we let the reduction rules for erased states ignore updates to h-level locations and the number of pending signals for non-capability passing channels. It is shown that the erased semantics is confluent, and simulates the low security updates of the original. Then, observational determinism follows from these facts.

The individual techniques used in the proof are not new. Erasure technique is often used to prove correctness of non-interference type systems (e.g., [7]), and fractional capabilities-based confluence argument is from [13]. Therefore, we only detail the part of the proof that shows how the existing techniques are integrated.

First, we define the notion of a well-typed state. We extend the type rule to type run-time states by extending Γ to map constant channels to types and introducing the following rule.

$$\frac{}{\Gamma, \Psi \vdash c : \Gamma(c)}$$

Let *cap*(*chan*$(\gamma, \Psi)) = \Psi$. We write $\Gamma \vdash S$ if for each $\ell \in$ *dom*(S), $\Gamma, 0, l \vdash S(\ell) : \tau, 0$ where $\Gamma(\ell) =$ *ref*(κ, τ) for some κ. Recall that a program state p is a parallel composition of finitely many threads, that is, it is of the form $s_1 || s_2 || \ldots || s_n$.

Definition B.1 (Well-typed State) *Suppose*

$$p = s_1 || \ldots || s_n$$

We write $\Gamma \vdash (S, B, p)$ if there exist $\Psi_1, \ldots, \Psi_n$ such that

(1) For all $\ell \in H$, $\Gamma(\ell) =$ ref$(\kappa,$ int$^h)^q$ for some κ, q.

(2) There exists κ such that for all $\ell \in L$, $\Gamma(\ell) =$ ref$(\kappa,$ int$^l)^q$ for some q.

(3) For each s_i, $\Gamma, \Psi_i, q \vdash s_i : \Psi'$ for some Ψ, q, Ψ'.

(4) $\Gamma \vdash S$.

(5) Let

$$\Psi = \sum_{c \in dom(B)} B(c) \times cap(\Gamma(c)) + \sum_{i=1}^{n} \Psi_i$$

Then $\forall \rho . 1 \geq \Psi(\rho)$.

In (5), $a \times \Psi = \lambda \rho . a \times \Psi(\rho)$.

Lemma B.2 *Initial states of a well-typed program are well-typed.*

[2] Definitionally, it is not a liveness property as it is not a property about a single trace.

B.1 Marking

To prepare the source program for erasure, we define a *marking* of the program which marks program parts to be erased. Given a typable program , we mark each statement with its context security level and each expression with the security level of its type. We write *mark*(s) to denote the mark of s and *mark*(e) to denote the mark of e. That is, if $\Gamma, \Psi \vdash e : \tau$ is a subderivation for the program, then *mark*$(e) =$ *qual*(τ), and if $\Gamma, \Psi_1, q \vdash s : \Psi_2$ is a subderivation of the type derivation for the program, then *mark*$(s) = q$. Note that the type rules guarantee that if a subexpression is marked h, then the expression is marked h, and if a statement is marked h, then its substatements are marked h.

To preserve marks across reductions, we introduce the following derived type rule.

$$\frac{\Gamma, \Psi_1, h \vdash s : \Psi_2}{\Gamma, \Psi_1, l \vdash s : \Psi_2}$$

Above, we let *mark*$(s) = h$. Note that adding the rule does not change the power of the type system as the rule can be derived as a lemma. With the additional rule, the following holds.

Lemma B.3 (Preservation) *Suppose* $\Gamma_1 \vdash (S_1, B_1, p_1)$ *and* $(S_1, B_1, p_1) \to (S_2, B_2, p_2)$. *Then there exists* $\Gamma_2 \supseteq \Gamma_1$ *such that* $\Gamma_2 \vdash (S_2, B_2, p_2)$, *preserving the marks. Also,* $\Gamma_2 \supsetneq \Gamma_1$ *only if allocating a new location or a channel.*

Proof: By case analysis on the reduction kind. The derived rule is used to prove the cases for if and while. □

Justified by the above lemma, in the remainder of the proof, we usually carry typing along the reductions, e.g., $\Gamma_1 \vdash (S_1, B_1, p_1) \to \Gamma_2 \vdash (S_2, B_2, p_2)$, and assume that the statements are marked according to the typing.

B.2 Erasure

Let *loclev*(*ref*$(\kappa, \tau)^q$) = *qual*(τ). We extend values for erasure states with a special value $\top$ such that $v \preceq v'$ iff $v = v'$ or $v' = \top$. Given $\Gamma \vdash (S, B, p)$, we define erasure as follows. We type $\top$ by

$$\frac{}{\Gamma, \Psi \vdash \top : \tau}$$

Definition B.4 (Erasure) $\Gamma' \vdash (S', B', p')$ *is an erasure of* $\Gamma \vdash (S, B, p)$, *written*

$$\Gamma \vdash (S, B, p) \preceq \Gamma' \vdash (S', B', p')$$

if the following holds.

- $\Gamma' \vdash (S', B', p')$
- p_2 *is* p_1 *with the statements marked* h *replaced by* `skip` *and expression marked* h *replaced by* $\top$.
- *dom*$(S) \supseteq$ *dom*$(S') \supseteq L$.
- *For all* $\ell \in$ *dom*(S'), $S(\ell) \preceq S'(\ell)$.
- *For all* $\ell \in$ *dom*(S') *such that* *loclev*$(\Gamma'(\ell)) = l$, $S'(\ell) \neq \top$.
- *dom*$(B) \supseteq$ *dom*(B')
- *For all* $c \in$ *dom*(B'), $B(c) \preceq B'(c)$.
- *For all* $c \in$ *dom*(B') *such that* *cap*$(\Gamma'(c)) \neq \emptyset$, $B'(c) \neq \top$.

Note that $\top$ may appear as a counter in the buffer. Essentially, an erasure state agrees on l-marked parts of the program, l-level locations with the original, and the buffers for channels with non-0 capability.

We also define the semantics for erasure states to be equivalent to the original except that writes to a h-level location writes $\top$, and receives from a 0-channel never blocks, and branch on $\top$ is arbitrary (it does not matter as both branch bodies are guaranteed to be `skip`). A write to $\top$ is a no-op. More formally, we use the following modified rules.

$$\frac{(S, e_1) \rightsquigarrow \ell \quad \textit{loclev}(\Gamma(\ell)) = h}{\begin{array}{l}\Gamma \vdash (S, B, E[e_1 := e_2]) \\ \to \Gamma \vdash (S, B, S[\ell \mapsto \top], E[\texttt{skip}])\end{array}}$$

$$\frac{(S, e) \rightsquigarrow c \quad \textit{cap}(\Gamma(c)) = 0}{\begin{array}{l}\Gamma \vdash (S, B, E[\texttt{wait } c]) \\ \to \Gamma \vdash (S, B[c \mapsto \top], E[\texttt{skip}])\end{array}}$$

$$\begin{array}{l}\Gamma \vdash (S, B, E[\texttt{if } \top \texttt{ then skip else skip}]) \\ \to \Gamma \vdash (S_1, B_1, E[\texttt{skip}])\end{array}$$

$$\begin{array}{l}\Gamma \vdash (S, B, E[\top := e_2]) \\ \to \Gamma \vdash (S, B, S, E[\texttt{skip}])\end{array}$$

$\Gamma \vdash$ on the reduced state is justified by preservation lemma essentially equivalent to Lemma B.3.

Note that all expressions whose type is h are erased. Then, the following follows from the type rule **READ**.

Lemma B.5 *Erasure semantics never reads a* h*-level location (which includes high security locations* H*).*

We can construct an erasure from the initial state.

Lemma B.6 *Let* $\Gamma \vdash (S, \emptyset, s)$ *be an initial state of the program. Then there exists an erasure* $\Gamma' \vdash (S', \emptyset, s')$ *of that state. In particular, it suffices to let* $\Gamma' = \Gamma$ *and* $S' = S$.

B.3 Simulation

Erasure semantics simulates the original up to l-level locations. More formally,

Lemma B.7 *Suppose*

$$\Gamma_1 \vdash (S_1, B_1, p_1) \preceq \Gamma_1' \vdash (S_1', B_1', p_1')$$

$$\Gamma_1 \vdash (S_1, B_1, p_) \rightarrow \Gamma_2 \vdash (S_2, B_2, p_2)$$

Then, either

- $\Gamma_2 \vdash (S_2, B_2, p_2) \preceq \Gamma_1' \vdash (S_1', B_1', p_1')$*, or*
- *There exists* $\Gamma_2' \vdash (S_2', B_2', p_2')$ *such that*

$$\Gamma_1' \vdash (S_1', B_1', p_1') \rightarrow \Gamma_2' \vdash (S_2', B_2', p_2')$$

$$\Gamma_2 \vdash (S_2, B_2, p_2) \preceq \Gamma_2' \vdash (S_2', B_2', p_2')$$

Proof: By case analysis on the reduction kind. The key observation is that non-erased parts of the program are equivalent in the both sides of $\preceq$. □

From Definition B.4 and the lemma above, it is immediate that the erasure semantics simulates the updates to l-level locations by the original. More formally,

Lemma B.8 *Suppose* $\Gamma \vdash (S, B, p) \preceq \Gamma' \vdash (S', B', p')$. *Let* $K = \{\ell \in dom(S') \mid loclev(\Gamma'(\ell)) = l\}$. *Then,* $\Gamma \vdash (S, B, p) \Downarrow T$ *implies that there exists* T' *such that* $\Gamma' \vdash (S', B', p') \Downarrow T'$ *and* $T|_K \sim T'|_K$. *In particular,* $T|_L \sim T'|_L$

B.4 Instrumented Semantics

We now instrument the semantics for erased states so that updates to locations are recorded per abstract location basis. Let $absloc(ref(\kappa, \tau)^q) = \kappa$. A state is now $\Gamma \vdash (S, B, p); R$ where R is a mapping from abstract locations such that $R(\kappa)$ contains list of updates for ℓ's such that $absloc(\Gamma(\ell)) = \kappa$. We extend the transition rules in a straightforward manner to record the updates.

B.5 Determinism

Let the states to be equivalent up to process reordering and renaming of locations, channels, and variables. The erased states (with the instrumentation) are confluent.

Lemma B.9 *Suppose* $\Gamma \vdash (S, B, p); R$ *is a erased state. Then for all* t_1, t_2 *such that* $\Gamma \vdash (S, B, p); R \rightarrow^* t_1$ *and* $\Gamma \vdash (S, B, p); R \rightarrow^* t_2$*, there exists a state* t_3 *such that* $t_1 \rightarrow^* t_3$ *and* $t_2 \rightarrow^* t_3$.

The proof is much like the confluence theorem from [13]. The key observation is that capabilities guarantee that there can be at most one thread that can write to an l-level location, and that there can be at most one thread that can be blocked on a channel (recall that *0* channels are non-blocking in the erasure semantics). Thus any possible combination of reductions at a state commutes.

Because all the updates are recorded, the above lemma implies that updates to locations are deterministic as records cannot be "undone."

Lemma B.10 *Suppose* $\Gamma \vdash (S, B, p)$ *is an erased state. Let* $K(\kappa) = \{\ell \mid absloc(\Gamma(\ell)) = \kappa\}$. *Then* $\Gamma \vdash (S, B, p) \Downarrow T$ *and* $\Gamma \vdash (S, B, p) \Downarrow T'$ *imply* $T|_K(\kappa) \sim T'|_K(\kappa)$. *In particular,* $T|_L \sim T'|_L$

The formal proof is much like that for confluence implying per-thread determinism from [13].

Finally, the soundness theorem follows from Lemmas B.2, B.5, B.6 B.8, and B.10, that is, it follows from the fact that updates to low security locations L in the original semantics are simulated (i.e., over-approximated) by the erasure semantics (B.6,B.8), and that the erasure semantics is observationally deterministic (Lemmas B.5,B.10).