Observational Determinism for Concurrent Program Security

Steve Zdancewic
Department of Computer and Information Science
University of Pennsylvania
stevez@cis.upenn.edu

Andrew C. Myers
Computer Science Department
Cornell University
andru@cs.cornell.edu

Abstract

Noninterference is a property of sequential programs that is useful for expressing security policies for data confidentiality and integrity. However, extending noninterference to concurrent programs has proved problematic. In this paper we present a relatively expressive secure concurrent language. This language, based on existing concurrent calculi, provides first-class channels, higher-order functions, and an unbounded number of threads. Well-typed programs obey a generalization of noninterference that ensures immunity to internal timing attacks and to attacks that exploit information about the thread scheduler. Elimination of these refinement attacks is possible because the enforced security property extends noninterference with observational determinism. Although the security property is strong, it also avoids some of the restrictiveness imposed on previous security-typed concurrent languages.

1 Introduction

Type systems for tracking information flow within programs are an attractive way to enforce security properties such as data confidentiality and integrity. Recent work has proposed a number of *security-typed languages* whose type systems statically check information flow, ranging from simple calculi [28, 45, 17, 1, 38, 49, 41, 19, 18] to full-featured languages [27, 50, 30, 4]. Many systems for which information security is important are concurrent—for example, web servers, databases, operating systems—yet the problem of checking information flow in concurrent programming languages has not yet received a satisfactory solution.

This research was supported in part by DARPA Contract F30602-99-1-0533, monitored by USAF Rome Laboratory, by ONR Grant N00014-01-1-0968, by NSF CAREER Award 0133302, by NSF Grant 0208642, and by a Sloan Research Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

This paper makes two contributions. First, it presents a definition of information-flow security that is appropriate for concurrent systems. Second, it describes a simple but expressive concurrent language with a type system that provably enforces security.

Notions of secure information flow are usually based on *noninterference* [15], a property only defined for deterministic systems. Intuitively, noninterference requires that the publicly visible results of a computation do not depend on confidential (or secret) information. Generalizing noninterference to concurrent languages is problematic because these languages are naturally nondeterministic: the order of execution of concurrent threads is not specified by the language semantics. Although this nondeterminism permits a variety of thread scheduler implementations, it also leads to *refinement attacks* in which information is leaked through resolution of nondeterministic choices (in this case, scheduler choices). These attacks often exploit *timing flows*, covert channels that have long been considered difficult to control [20].

Several recent papers have presented type systems for secure information flow in concurrent languages [43, 38, 41, 5, 19, 29, 36]. The type systems of these languages enforce secure information flow; but most of these type systems are so restrictive that programming becomes impractical.

The secure concurrent language presented in this paper, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$, addresses both of these limitations. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ has been proved to enforce a generalization of noninterference for concurrent systems, based on low-security observational determinism. Much of the restrictiveness of prior security type systems for concurrent languages arises from the desire to control timing channels; we show that some restrictiveness can be avoided by distinguishing different kinds of timing channels. This approach opens up a trade-off between the observational powers of an attacker and the restrictiveness of the enforcement mechanism. In some situations, the more restrictive model of the attacker may be warranted, but in other cases, the less restrictive model is acceptable.

The concurrency features in most previous secure languages have been limited. By contrast, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ provides fairly general support for concurrency: it allows an arbitrary number of threads, it supports both message-passing and shared-memory styles of programming, and communication channels are themselves first-class values. $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$ is not intended to serve as a user-level programming language (its syntax and type system are too unwieldy). Instead, it is intended to serve as a model language for studying information flow and concurrency; nevertheless, using these constructs, one can encode (via CPS translation) other security-typed languages [49].

The approach taken here is to factor the information security analysis into two pieces: a type system that eliminates both explicit and implicit storage channels, and a race-freedom analysis that eliminates timing channels. Factoring the security analysis makes both the information-flow analysis and its proof of correctness more modular. In addition, any improvements in the accuracy of alias analysis (used to detect races) lead directly to improvements in the security analysis. This paper focuses mainly on the type system for explicit information flows, but race freedom is discussed in Section 4.3.

The remainder of this paper is structured as follows. Sections 2 and 3 present and argue for our definition of information security and also informally describes our concurrent programming model and its security implications. Section 4 gives the formal definition of $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$. Section 5 states our soundness results, including the security theorem. (Full proofs are available in the dissertation of the first author [48].) Section 6 covers related work, and the paper concludes in Section 7.

2 Security model

Information security is fundamentally connected to the ability of an observer to distinguish different program executions. Our primary concern is confidentiality. An attacker must be prevented from distinguishing two program executions that differ only in their confidential inputs, because the attacker might otherwise learn something about the inputs.

As usual, we assume that there is a lattice $\mathcal{L}$ of security labels [7, 11]. Lattice elements describe restrictions on the propagation of the information they label; labels higher in the lattice describe data whose use is more restricted. For confidentiality, labels higher in the lattice describe more confidential data whose dissemination should be restricted. In this paper we will leave the lattice abstract; however, it is often useful to think about the simple two-point lattice containing just the elements $\mathtt{L}$ (low) and $\mathtt{H}$ (high), where $\mathtt{L} \sqsubseteq \mathtt{H}$ but $\mathtt{H} \not\sqsubseteq \mathtt{L}$.

The terminology "ℓ_1 is protected by ℓ_2" is used to indicate that $\ell_1 \sqsubseteq \ell_2$—intuitively it is secure to treat data with label ℓ_1 as though it has label ℓ_2 because the latter label imposes more restrictions on how the data is used. The terms "high-security" and "low-security" describe data whose labels are relatively high or low in the lattice, respectively.

2.1 Noninterference and nondeterminism

Suppose that we have a *low-equivalence* relation $\approx_\zeta$ that relates two program expressions e and e' if they differ only in high-security input data. Here ζ ("zeta") is a label that defines "high-security": high-security data is any whose label ℓ does not satisfy $\ell \sqsubseteq \zeta$. The low observer (attacker) can see only data protected by ζ.

For example, consider pairs $\langle h, l\rangle$, where h ranges over some high-security data and l ranges over low-security data. An observer with low-security access (only permitted to see the l component) can see that the pairs $\langle \mathtt{attack\ at\ dawn}, 3\rangle$ and $\langle \mathtt{do\ not\ attack}, 4\rangle$ are different (because $3 \neq 4$), but will be unable to distinguish the pairs $\langle \mathtt{attack\ at\ dawn}, 3\rangle$ and $\langle \mathtt{do\ not\ attack}, 3\rangle$. We have:

$$\langle \mathtt{attack\ at\ dawn}, 3\rangle \approx_\ell \langle \mathtt{do\ not\ attack}, 3\rangle$$
$$\langle \mathtt{attack\ at\ dawn}, 3\rangle \not\approx_\ell \langle \mathtt{do\ not\ attack}, 4\rangle$$

Noninterference says that this equivalence relation also captures the distinctions that can be made by observing the *execution* of the two expressions; an observer able to see only low-security data learns nothing about high-security inputs by watching the program execute. Suppose we have an evaluation relation $e \Downarrow v$. Evaluations of two expressions will be indistinguishable as long as they produce indistinguishable results: If $e \approx_\zeta e'$, then

$$(e \Downarrow v \wedge e' \Downarrow v') \Rightarrow v \approx_\zeta v' \qquad (1)$$

Note that values v and v' need not be strictly equal. The equivalence relation $\approx_\zeta$ on values captures the idea that the low-security observer may be unable to see certain parts of the program output—it relates any two values whose differences are not observable by the low observer.

This definition of security is appealing, although it does permit high-security information to leak through termination behavior. Because termination behavior communicates an average of one bit of information in an information-theoretic sense, we follow common practice by ignoring this covert channel [8, 45, 17, 27].

This definition does not apply straightforwardly to concurrent languages. First, the idea that a program terminates with a single result is less appropriate for a concurrent language, where programs may produce observable effects while continuing to run. This can be addressed by describing the observations about program execution as a finite or infinite event trace. Suppose that a machine configuration m consists of a pair $\langle M, e\rangle$ where M is the state of the memory and e is the executing program. The statement $m \Downarrow T$ means that the configuration m may produce the execution trace T. For the language defined in this paper, the event trace consists of the sequence of memory states $[M_0, M_1, M_2, \ldots]$ that occur as the program executes.

A second, more significant difficulty with concurrent languages is nondeterminism; there may be many traces T such that $m \Downarrow T$. For example, the language presented below has a nondeterministic operational semantics that allows any possible interleaving of the evaluations of two parallel processes. Informally, there are two rules for evaluating an expression $e_1 \mid e_2$ where the symbol $\mid$ represents parallel composition:[1]

$$\frac{\langle M, e_1\rangle \to \langle M', e_1'\rangle}{\langle M, e_1 \mid e_2\rangle \to \langle M', e_1' \mid e_2\rangle} \quad \frac{\langle M, e_2\rangle \to \langle M', e_2'\rangle}{\langle M, e_1 \mid e_2\rangle \to \langle M', e_1 \mid e_2'\rangle}$$

The thread scheduler resolves this nondeterminism by choosing a thread to run at each execution step.

A standard way to handle nondeterminism is a *possibilistic* generalization of noninterference in which the computations of two expressions are considered indistinguishable as long as every possible computation of one expression is possible for the other expression as well [23]; that is, $m_1 \approx_\zeta m_2$ must imply:

$$\begin{array}{lcl} \forall T_1 \,.\, m_1 \Downarrow T_1 & \Rightarrow & \exists T_2 \,.\, m_2 \Downarrow T_2 \wedge T_1 \approx_\zeta T_2 \\ \forall T_2 \,.\, m_2 \Downarrow T_2 & \Rightarrow & \exists T_1 \,.\, m_1 \Downarrow T_1 \wedge T_1 \approx_\zeta T_2 \end{array} \qquad (2)$$

If the evaluation relation is deterministic, this security condition is noninterference. Importantly, it can be proved in essentially the same way as noninterference. One common technique is to show that each step of computation preserves the low-level equivalence between configurations [15, 45].

However, a system that satisfies the possibilistic security condition may have a refinement that does not. As a result, an observer may be able to distinguish two such programs by comparing the probabilities of possible executions. For example, consider this program:

```
l := true | l := false | l := h        (A)
```

Here the boolean variables `l` and `h` are low- and high-security locations respectively. This program is secure if the low observer cannot distinguish between the following two programs:

```
l := true | l := false | l := true     (A1)
l := true | l := false | l := false    (A2)
```

Suppose that the low observer is unable to distinguish two traces if their final memories have the same values in location `l`. In both programs the location `l` may end with the contents `true` or `false`, so the possible low observations of the two programs are identical. However, an attacker may be able to infer some information about the initial value of `h`. For example, the thread scheduler may tend to run the final expression of the three last, so that both programs usually have the effect `l := h`. Even if the scheduler randomly chooses ready threads to execute, `l` will be more likely to contain `h` than ¬`h`. In general, if the distribution of the final values of `l` is different in the two programs, it is undoubtedly because the assignment `l := h` is leaking information about `h` via (perhaps probabilistic) refinement of thread scheduler nondeterminism.

Even the simple program `(l := true | l := false)` might be used to violate confidentiality, despite not mentioning high-security data. For example, the code for the assignment `l := false` might happen to fall on the same cache line as a piece of data used by another apparently secure program running on the same system. If accesses to this other data are conditioned on confidential information, it could make the assignment `l := false` more likely to come last, winning the write–write race. Thus, an attacker might learn about confidential data by introducing such programs into a running system. Attacks of this sort have been demonstrated [42].

[1]The actual operational semantics of $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ is given in Figure 3.

2.2 Internal vs. external timing

Many previous security-typed language approaches for concurrent programs [43, 41, 19, 5, 36] aim to prevent high-security information from affecting the termination behavior and timing behavior of a program. This decision strengthens possibilistic noninterference by considering execution time to be observable by the low-security observer. Confidential information thus cannot be encoded in the execution time of one thread and then transferred to a second thread, as in the following example that exploits thread timing behavior:

```
x := true;                                              (B)
  (if h then delay(100) else skip; x := false)
| (delay(50); l := x;  ...)
```

This program initializes variable `x` to `true` and then spawns two threads. The first thread assigns `x` the value `false` either immediately or after a delay. The second thread waits long enough that the assignment to `x` is likely to have occurred when `h` is false; this thread then assigns `l` the value of `x`. Depending on the thread scheduler, this program can reliably copy the contents of `h` into the variable `l`—an information leak. Its bandwidth can be increased by placing the code inside a loop.

If program execution time is considered low-observable, information flow control requires that the execution time of low-equivalent programs be equal. Because proving statements about execution time is difficult, the type systems proposed to enforce this security condition have tended to be very restrictive. For example, these type systems have usually ruled out programs like the following, because its running time depends on high-security information:

```
x := true;                                      (C)
(if h then delay(100) else skip; x := false)
```

A related way to treat timing channels is to pad code to eliminate them [3, 36], for example by ensuring that the same time is taken by both branches of an `if`. However, this approach does not easily handle loops, recursive functions, or instruction-cache effects.

Here we have taken a different approach, drawing a distinction between the internally and externally observable timing behavior of programs. External observations are those made by an observer outside the system, timing the program with mechanisms external to the computing system. Internal observations are those made by other programs running on the same system. In principle, internal observations are more easily controlled because other programs can be subjected to a validation process before being allowed to run. Typically, internal observations also offer greater potential for high-bandwidth information transfer, so they may also be more important to control. In this work, the focus is on controlling information flows that are internal to the system. Because there are many external flows, most of which are difficult to control (e.g., conversation between users of the system), and other techniques that can be applied to controlling them (e.g., auditing, dynamically padding total execution time), this decision seems reasonable.

The beneficial effect of this decision is that example (C) is considered secure because its timing channels are only external; example (B) is considered insecure because it contains an internal timing channel made observable by a race.

2.3 Low-security observational determinism

These weaknesses of possibilistic security conditions led McLean [24] and Roscoe [32] to propose using low-security observational determinism to generalize noninterference to nondeterministic systems. Low-security observational determinism is essentially noninterference as given above (1), but applied to a nondeterministic system. In our framework the observational determinism can be expressed straightforwardly. Similarly to noninterference, we have two arbitrary initial configurations m and m' that the low observer cannot distinguish ($m \approx_\zeta m'$). To avoid information flows, they must produce indistinguishable traces T and T':

$$(m \Downarrow T \wedge m' \Downarrow T') \Rightarrow T \approx_\zeta T' \qquad (3)$$

Thus, to a low-level observer who is unable to distinguish states that differ only in high-security components, a system satisfying this condition appears deterministic.

2.4 Race freedom

To obtain deterministic results from the low-security viewpoint, it is *not* necessary that evaluation itself be deterministic, which is fortunate because nondeterminism is important. Nondeterministic evaluation allows thread scheduler behavior to depend on aspects of the run-time system not under the programmer's control: for instance, the operating system, the available resources, or the presence of other threads. The problem is how to permit useful nondeterminism without creating security holes.

Our insight is that requiring *race freedom* is a solution to this problem. To be considered secure, a program must enforce an ordering on any two accesses to the same memory location, when at least one of the accesses is a write. This ordering ensures that the sequence of operations performed at a single memory location is deterministic.

Race freedom rules out insecure programs like example (A) because it has a write–write race to the location `l`. However, the program (`l`$_1$ `:= true | l`$_2$ `:= true`) is considered secure because it writes to two different locations, even though the ordering of these writes is nondeterministic. Disallowing races reduces the expressiveness of the language, but because races are difficult to reason about, programmers rarely use them intentionally; races are usually bugs.

Making programs race-free weakens the ability of a thread to observe the behavior of other threads. The observational powers of the low observer are weakened correspondingly. This weakening is expressed formally in the relation $T \approx_\zeta T'$ that captures when the low observer is able to distinguish two traces. Two traces T and T' are related if they are equivalent up to stuttering and prefixing at every memory location, considered *independently* of other locations.

To be more precise, let M, a memory, be a finite map from locations L to values v. Then $M(L)$ is the contents of location L. Let $T(L)$ be the projection of the trace T onto a single memory location L; that is, if $T = [M_0, M_1, M_2, \ldots]$ then $T(L) = [M_0(L), M_1(L), M_2(L), \ldots]$. A sequence of values $[v_0, v_1, v_2, \ldots]$ is related to another sequence of values $[v'_0, v'_1, v'_2, \ldots]$ if $v_i \approx_\zeta v'_i$ for all i up to the length of the shorter sequence. Then $T \approx_\zeta T'$ if for all locations L, $T(L)$ is equivalent up to stuttering to $T'(L)$, or vice versa.

Combining this definition of trace equivalence with observational determinism (3) yields a new security condition that allows high-security information to affect the *external* termination and timing behavior of a program, while preventing any effect on *internal* termination and timing behavior.

Two aspects of this definition merit discussion. First, allowing one sequence to be a prefix of the other permits an *external* nontermination channel that leaks one bit of information, but removes the obligation to prove program termination. However, this decision implies that the $\approx_\zeta$ relation on traces is not an equivalence relation (transitivity fails). To see why, consider the three traces $T_1 = [v_0, v_1]$, $T_2 = [v_0, v_1, v_2]$, $T_3 = [v_0, v_1, v_3]$, we have $T_2 \approx_\zeta T_1$ and $T_1 \approx_\zeta T_3$ but $T_2 \not\approx_\zeta T_3$. Consequently, we prove the security property in terms of a more primitive simulation relation $\lesssim_\zeta$ (defined in Section 5.3) that is transitive.

Second, considering each memory location independently is justified because internal observations of the two locations can only observe the relative ordering of their updates by using code that contains a race—and programs containing races are disallowed. By requiring only per-location ordering of memory operations, this definition avoids the restrictiveness incurred by timing-sensitive definitions of security.

3 Synchronization mechanisms

The previous section presents a new security condition for concurrent imperative programming languages. A concurrent language can be made more expressive by exploiting the added flexibility in the condition. This section and the next present such a language, called $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$.

The race freedom requirement of the security definition implies that threads cannot asynchronously communicate through shared memory. Consequently, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ uses message passing for interthread communication, and the communication is regulated to prevent illegal information flows. Also, because the security definition relies on sequencing writes to memory, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ needs a thread synchronization mechanism. Despite its importance for practical concurrent programming and impact on information-flow properties, synchronization has only recently been studied in the context of information security [19, 30, 18]. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ provides a thread-synchronization mechanism based on the same abstraction it uses for message passing.

3.1 Message passing

To support both thread synchronization and communication, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ uses message passing. The programming model of $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ is based on the join calculus [13], which supports first-class channels; $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ extends it with *linear channels* and *linear handlers*. This linear synchronization mechanism provides additional structure that allows the type system to more accurately describe synchronization information flows.

The $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ notation for sending a value v on a channel `c` is `c`(v). Messages may contain no data, in which case the send is written `c()`, or they may contain multiple values, in which case the send is written `c`$(v_1, \ldots, v_n)$. A send on a channel may cause the activation of a corresponding *message handler* (or simply *handler*), for example: `c`$(x) \triangleright$ `l :=` x

When a message is sent on channel `c`, the handler triggers, creating a new thread to execute `l :=` x with the message contents bound to the variable x. The message is consumed by the handler. Handlers may invoke their own channel, allowing encoding of loops and recursive functions.

Handler definitions are introduced via `let`-syntax. For example, a $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ program described above is:

```
let c(x) ▷ l := x in c(t)
```

Multiple threads might attempt to send messages on a single channel concurrently, creating *send contention* and possibly race conditions. Lexical scoping of channel names makes it impossible to introduce *receive contention*, in which multiple handlers vie for a message.

In $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ channels are first-class and may be passed as values in the messages sent on other channels. For example, the channel `double` sends two (empty) messages on any channel it is given as an argument:

```
let double(c) ▷ c() | c() in
let d() ▷ P in double(d)
```

Ordinary handlers remain permanently in the program environment once declared, and are able to react to any number of messages sent on the channels they define. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ also supports *linear channels*, on which exactly one message must be sent. The symbol $\multimap$ is used in place of $\triangleright$ to indicate that a message handler is linear. For example, the following program declares a linear channel `k` that must be used exactly once along each execution path, as shown in this example:

```
let k(x) ⊸ l := x in
  if h then k(t) else k(f)
```

Linear handlers are guaranteed to be invoked exactly once unless the computation diverges. Consequently, no information is transmitted by the fact that the linear handler is run. Furthermore, linear handlers cannot create nondeterminism because there is never any send contention. These observations enable more precise information-flow checking.

Channel arguments are given a security label that restricts what data may be sent on them. This suggests an alternate formulation of the security condition of Section 2.3: a program could be equipped with a distinguished output channel that accepts only low values, and we could require that the sequence of values sent on the output channel is deterministic. A distinguished input channel would also be required to allow the program context to acknowledge the output. The difference between the two approaches is not large, because memory locations can be viewed as external processes that receive messages at every write.

3.2 Synchronization

So far, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ has not addressed the issue of synchronization between concurrent threads. Rather than directly incorporating mutexes or semaphores as primitives, we adopt *join patterns* as found in the join calculus [13]. With this design, handlers may block waiting for messages on multiple channels, permitting synchronization on several threads of execution. For example, the following handler waits for messages on channels `input`$_1$ and `input`$_2$ before starting the thread "`let z...`".

```
input₁(x) | input₂(y) ▷ let z = x + y in output(z)
```

If some of the channels in the join pattern never receive a message, the handler never triggers.

This approach simplifies reasoning about synchronization. Unlike mutexes or semaphores, which may be used anywhere within a program, join patterns limit synchronization to points at which messages are received.

Despite this abstraction[2], join patterns support a range of useful synchronization idioms [13]. In particular, they provide synchronous message passing between two threads. Suppose thread P_1 wants to send the message `m` synchronously on the channel c to thread Q_1, after which it continues as P_2. Thread Q_1 blocks waiting for the message `m` and then continues as thread Q_2(`m`). In traditional message-passing notation, this situation might be expressed by the following program:

```
cobegin (P₁;send_c(m);P₂)|(Q₁;recv_c(x);Q₂(x)) coend
```

Writing $R_1; R_2$ for sequential composition of (sequential) processes R_1 and R_2, this example can be written using $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$'s join patterns:

```
let send_c(x) | recv_c() ▷ P₂ | Q₂(x)
in P₁;send_c(m) | Q₁;recv_c()
```

[2]Join patterns could be *implemented* using mutexes or semaphores.

```
let k0() | k1() ⊸ T in
let k2() | k3() ⊸ S; k0() in
  P;( Q1;( R1; k1() | R2; k3()) | Q2; k2())
```

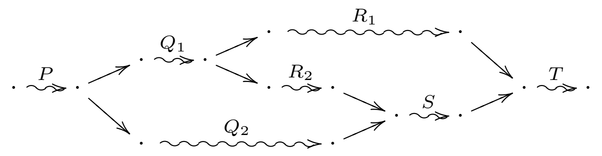

Figure 1. Nonnested synchronization

$\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ also allows join patterns for linear channels. For example, the program below declares a handler for two linear channels $\mathtt{k}_0$ and $\mathtt{k}_1$:

```
let k0() | k1(x) ⊸ P in Q
```

Channels $\mathtt{k}_0$ and $\mathtt{k}_1$ must be used exactly once in each possible execution path of the process Q.

Join patterns in $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ permit complex synchronization behavior. For instance, the program shown in Figure 1 has the synchronization structure pictured there, where wavy arrows represent computation and straight arrows represent process forking or synchronizing message sends. The example is a single-message producer-consumer pair. Note that this program uses channel `k0` inside the body of the handler defining channels `k2` and `k3`.

The typical synchronization mechanisms for shared-memory programming are locks, such as mutexes and semaphores. Like other process calculi [26, 13], $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ does not directly support these mechanisms. Though their addition would be straightforward, locks do not interact with information flow analysis as congenially as message-passing does. Locks are primarily used to obtain atomicity, which is useful for writing correct programs, but it provides little help in reasoning about timing flows. For example, given a critical section controlled by a mutex, the order of arrival of two threads can leak as much information as if there were no mutex. Locks can be used as limited message-passing mechanisms, and it is plausible that a precise information flow analysis could be constructed for those uses.

3.3 Expressiveness

Security-typed languages rule out apparently insecure programs and thus may reduce expressiveness. While $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ has a relatively rich collection of features in comparison with the previous work, it is difficult to directly compare expressiveness because of the differing definitions of security.

Channels in $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ are quite expressive. A channel is an extension to a continuation: both accept a value and cause some computation to occur, possibly using the value. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ is in fact an extension of a secure CPS language introduced in the authors' earlier work [49]. Thus adding concurrency causes no loss of expressiveness, unlike in various previous secure concurrent languages that have ruled out high-security loop guards [38, 36]. However, $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ does rule out programs that other systems consider secure [17, 38, 19, 36], because those programs allow refinement attacks. It is our belief that these different approaches embody a tradeoff in secure systems design. In cases where absolute security is mandatory, the more restrictive languages may be required. However, there are many circumstances in which external timing and termination channels are acceptable risks.

Linearity is an important feature because it improves reasoning about implicit flows: information flows arising from program control structure [8]. In the producer-consumer example of Figure 1, knowing that the code at T is executed gives no more information than knowing that P was executed. Linearity makes it possible to determine this despite the complex control structure. Prior secure concurrent languages are either unable to express this program or reject it as insecure, which is unfortunate because the producer-consumer idiom is useful.

4 $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$: A secure concurrent calculus

This section introduces the formal semantics for $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$, including its syntax, operational semantics, and type system.

4.1 Syntax and operational semantics

Figure 2 shows the syntax for $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ programs. Base values in $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ are channel names c, memory locations L, or booleans `t` and `f`. The metavariable f ranges over channels and variables that have channel or linear channel type. A *process* ("*thread*" and "*process*" are used interchangeably) P consists of a sequence of `let`-declarations and primitive operations followed by either the terminal process `0`, an `if` expression, or P_1 | P_2. Message sends and handlers are as described in the previous section. Nonlinear join patterns may bind linear variables y (although they are not required to), but linear join patterns never bind linear variables. This restriction prevents problems in sequencing linear sends.

It is helpful to define a sequential subset of $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$: processes not containing the | symbol. If $P(y)$ is a sequential process that contains one free linear channel variable y, the process P; Q is sugar for `let` y`()` $\multimap$ Q `in` $P(y)$.

Figure 2 also describes a program's dynamic state. A memory M contains channel handler definitions in addition to ordinary mappings between locations and their values; its domain is generalized to include the join patterns J defines. The function $\mathrm{dom}(M)$ is the set of locations L and join patterns J that appear in M. If $L \in \mathrm{dom}(M)$ then we write $M(L)$ for the value v such that $[L \mapsto v] \in M$. Similarly, if $J \in \mathrm{dom}(M)$, we write $M(J)$ for the (open) process P such that $[J \rhd P] \in M$. A *synchronization environment* S stores the linear handlers that have been declared by the program, using similar notation. Program-counter labels (pc) are used to track implicit information flows [49, 30]. The syntax $[\mathsf{pc} : P]$ associates a top-level process P with a pc label. A collection of such processes running concurrently is

x, f	$\in$	$\mathcal{V}$	Variable names
bv	$::=$	c	Channel value
	$\mid$	L	Reference value
	$\mid$	$\texttt{t} \mid \texttt{f}$	Boolean values
v	$::=$	x	Variables
	$\mid$	bv_ℓ	Secure values
lv	$::=$	$y \mid c$	Linear values
$prim$	$::=$	v	Values
	$\mid$	$v \oplus v$	Boolean operations
	$\mid$	$!v$	Dereference
f	$::=$	$x \mid y \mid c$	Variables or channels
J	$::=$	$f(\vec{x}, y)$	Nonlinear channel
	$\mid$	$f(\vec{x})$	Linear channel
	$\mid$	$J \mid J$	Join pattern
P	$::=$	$\texttt{let}\ x = prim\ \texttt{in}\ P$	Primitive operation
	$\mid$	$\texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ P$	Reference creation
	$\mid$	$\texttt{set}\ v := v\ \texttt{in}\ P$	Assignment
	$\mid$	$\texttt{let}\ J \triangleright P\ \texttt{in}\ P$	Handler definition
	$\mid$	$\texttt{let}\ J \multimap P\ \texttt{in}\ P$	Linear handler definition
	$\mid$	$v\,(\vec{v}, lv^{\mathsf{opt}})$	Message send
	$\mid$	$lv\,(\vec{v})$	Linear message send
	$\mid$	$\texttt{if}\ v\ \texttt{then}\ P\ \texttt{else}\ P$	Conditional
	$\mid$	$(P \mid P)$	Parallel processes
	$\mid$	0	Inert process
M	$::=$	$M[L \mapsto v]$	Memory location L storing v
	$\mid$	$M[\mathsf{pc} : J \triangleright P]$	Message handler
	$\mid$	$\cdot$	
S	$::=$	$S[\mathsf{pc} : J \multimap P]$	Linear handler
	$\mid$	$\cdot$	
N	$::=$	$\cdot \mid N \mid [\mathsf{pc} : P]$	Process pool
m	$=$	$\langle M,\ S,\ N \rangle$	Machine configuration

$$bv_\ell \sqcup \ell' \stackrel{\text{def}}{=} bv_{(\ell \sqcup \ell')}$$

$$\mid_i P_i \stackrel{\text{def}}{=} P_1 \mid \ \ldots\ \mid P_n \quad (i \in \{1, \ldots, n\})$$

Figure 2. Process syntax & notation

$$M,\ \mathsf{pc} \models v \Downarrow v \sqcup \mathsf{pc} \qquad \frac{M(L) = v}{M,\ \mathsf{pc} \models !L \Downarrow v \sqcup \mathsf{pc}}$$

$$M,\ \mathsf{pc} \models n_\ell \oplus n'_{\ell'} \Downarrow (n [\![\oplus]\!] n')_{\ell \sqcup \ell'} \sqcup \mathsf{pc}$$

$$\frac{M,\ \mathsf{pc} \models prim \Downarrow v}{\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{let}\ x = prim\ \texttt{in}\ e])\rangle \\ \rightarrow \langle M,\ S,\ (N \mid [\mathsf{pc} : e\{v/x\}])\rangle\end{array}}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{let}\ x = \texttt{ref}\ v\ \texttt{in}\ P])\rangle \\ \rightarrow\ \langle M[L \mapsto v],\ S,\ (N \mid [\mathsf{pc} : P\{L_{\mathsf{pc}}/x\}])\rangle \\ \text{where } (L \notin \mathrm{dom}(M))\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{set}\ L_\ell := v\ \texttt{in}\ P])\rangle \\ \rightarrow\ \langle M[L \mapsto v \sqcup \ell \sqcup \mathsf{pc}],\ S,\ (N \mid [\mathsf{pc} : P])\rangle \\ \text{where } (L \in \mathrm{dom}(M))\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{if}\ \texttt{t}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2])\rangle \\ \rightarrow\ \langle M,\ S,\ (N \mid [\mathsf{pc} \sqcup \ell : P_1])\rangle\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{if}\ \texttt{f}_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2])\rangle \\ \rightarrow\ \langle M,\ S,\ (N \mid [\mathsf{pc} \sqcup \ell : P_2])\rangle\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ \ldots\ \mid f_n(\vec{x}_n) \triangleright P_1\ \texttt{in}\ P_2])\rangle \\ \rightarrow\ \langle M[\mathsf{pc} : c_1(\vec{x}_1) \mid \ \ldots\ \mid c_n(\vec{x}_n) \triangleright P_1\{(c_i)_{\mathsf{pc}}/f_i\}],\ S, \\ \qquad (N \mid [\mathsf{pc} : P_2\{(c_i)_{\mathsf{pc}}/f_i\}])\rangle \\ \text{where the } c_i \text{ are fresh}\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : \texttt{let}\ f_1(\vec{x}_1) \mid \ldots \mid f_n(\vec{x}_n) \multimap P_1\ \texttt{in}\ P_2])\rangle \\ \rightarrow\ \langle M,\ S[\mathsf{pc} : c_1(\vec{x}_1) \mid \ldots \mid c_n(\vec{x}_n) \multimap P_1], \\ \qquad (N \mid [\mathsf{pc} : P_2\{c_i/f_i\}])\rangle \\ \text{where the } c_i \text{ are fresh}\end{array}$$

$$\begin{array}{l}\langle M[\mathsf{pc} : c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \triangleright P],\ S, \\ \quad (N \mid_i [\mathsf{pc}_i : c_{i\ell_i}(\vec{v}_i, lv_i^{\mathsf{opt}})])\rangle \\ \rightarrow\ \langle M[\mathsf{pc} : c_1(\vec{x}_1, y_1^{\mathsf{opt}}) \mid \ldots \mid c_n(\vec{x}_n, y_n^{\mathsf{opt}}) \triangleright P],\ S, \\ \qquad (N \mid [\ell : P\{\vec{v}_i \sqcup \mathsf{pc}_i/\vec{x}_i\}\{lv_i/y_i\}^{\mathsf{opt}}])\rangle \\ \text{where } \ell = \bigsqcup_i \mathsf{pc}_i \sqcup \ell_i\end{array}$$

$$\begin{array}{l}\langle M,\ S[\mathsf{pc} : c_1(\vec{x}_1) \mid \ldots \mid c_n(\vec{x}_n) \multimap P],\ (N \mid_i [\mathsf{pc}_i : c_i(\vec{v}_i)])\rangle \\ \rightarrow\ \langle M,\ S,\ (N \mid [\mathsf{pc} : P\{\vec{v}_i \sqcup \mathsf{pc}_i/\vec{x}_i\}])\rangle\end{array}$$

$$\begin{array}{l}\langle M,\ S,\ (N \mid [\mathsf{pc} : P \mid Q])\rangle \\ \rightarrow\ \langle M,\ S,\ (N \mid [\mathsf{pc} : P] \mid [\mathsf{pc} : Q])\rangle\end{array}$$

Figure 3. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ operational semantics

called a *process pool*. A *machine configuration* m is a triple $\langle M,\ S,\ N\rangle$, where M is a memory, S is a synchronization environment, and N is a process pool.

Figure 3 contains the operational semantics for $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$, instrumented to track information flow within executing programs—that this instrumentation is correct is the purpose of the security proof. The rules define a transition relation $m_1 \rightarrow m_2$ between machine configurations, though space limitations preclude full explanation of the semantics. Evaluation of primitive operations is described by a large-step evaluation relation $M, \mathsf{pc} \models \mathit{prim} \Downarrow v$.

The evaluation rule for a process about to do a memory update $[\mathsf{pc} : \texttt{set}\ L_\ell := v\ \texttt{in}\ P]$ updates the contents of memory location L to contain a security value v whose label is bounded below by ℓ, the security label of the reference itself (needed to prevent information leaks through aliasing) and pc (needed to prevent implicit information flows).

The two rules for evaluating conditional expressions show how the program counter label approximates the implicit information flows that arise due to program control behavior. The process $[\mathsf{pc} : \texttt{if}\ t_\ell\ \texttt{then}\ P_1\ \texttt{else}\ P_2]$ evolves to the process $[\mathsf{pc} \sqcup \ell : P_1]$, where the new program counter label is the join of the old one and the label of the value that regulates the conditional—any assignment operations that occur in P_1 will propagate the label $\mathsf{pc} \sqcup \ell$ and so track the implicit information introduced by the branch.

Note that the rules for evaluating handler definitions install their handlers in the appropriate environment, with the channels renamed to prevent collisions; linear handlers also record the pc at the point of their definition. Correspondingly, the rules for sending messages look up the channel in the appropriate environment and start a thread to execute the body. The rule for the new pc differs between linear and nonlinear messages; in the linear case the pc of the sending context(s) is discarded and the pc existing at the evaluation of the handler is used instead—possibly lowering pc. Linearity ensures that this is safe [49]: once the handler has been reached, it must also be invoked (unless that thread of control does not terminate). Therefore, the future computation learns nothing from its invocation.

The operational semantics in the figure are too rigid: they require the right-most processes to take part in the computational step. Because thread scheduling should ignore the syntactic ordering of process pools and processes running concurrently, we introduce structural equivalences on processes and process pools, allowing arbitrary reordering of items separated by $\mid$, and discarding of halted processes ($\mathsf{0}$). These structural equivalences ($N_1 \equiv N_2$, $P_1 \equiv P_2$) are the least symmetric, transitive congruences allowing these transformations. Finally, two machine configurations $\langle M_1, S_1, N_1\rangle$ and $\langle M_2, S_2, N_2\rangle$ are structurally equivalent if they are α-equivalent and $N_1 \equiv N_2$.

The syntax-independent operational semantics is given by the transition relation $\Rrightarrow$ defined from the $\rightarrow$ relation by

pc, ℓ	$\in$	$\mathcal{L}$	Security labels
s	$::=$	t_ℓ	Security types
t	$::=$	$\texttt{bool}$	Booleans
	$\mid$	$[\mathsf{pc}](\vec{s}, k^{\mathsf{opt}})$	Channel types
	$\mid$	$s\ \texttt{ref}$	Reference types
k	$::=$	$(\vec{s})$	Linear channel types
Γ	$::=$	$\cdot \mid \Gamma, x\!:\!s$	Type contexts
H	$::=$	$\cdot$	Empty memory type
	$\mid$	$H, [L\!:\!s]$	Location type
	$\mid$	$H, [c\!:\!s]$	Channel definition type
K	$::=$	$\cdot \mid K, y\!:\!k$	Linear type contexts
T	$::=$	$\cdot \mid T, c\!:\!k$	Synchronization state types

$$\mathsf{label}(t_\ell) \overset{\mathrm{def}}{=} \ell$$

Figure 4. Type syntax

composition with structural equivalence:

$$m_1 \Rrightarrow m_2 \iff \exists m_1', m_2'.\ m_1 \equiv m_1' \rightarrow m_2' \equiv m_2$$

4.2 $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ type system

The language $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ has a nonstandard type system that enforces the security condition. Figure 4 shows the types for $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ programs. They are divided into security types and linear types. Base types, t, consist of booleans, channel types, and references; security types t_ℓ are pairs consisting of a base type and a security label annotation.

The channel type $[\mathsf{pc}](\vec{s}, k^{\mathsf{opt}})$ has any number of nonlinear arguments and at most one linear argument. The $[\mathsf{pc}]$ component of a channel type is a lower bound on the security level of memory locations that might be written to if a message is sent on this channel.

The linear types are channels $(\vec{s})$ that accept nonlinear arguments. A linear message does not itself reveal information about the sending context (although its contents might), so linear channel types do not need a $[\mathsf{pc}]$ component.

The security lattice is lifted to a subtyping relation $\leq$ on $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ types, in the usual manner [45, 49]. In particular, nonlinear channel types are contravariant in both their pc-label and their argument types; reference types are invariant.

A type context Γ is a finite map from nonlinear variables to their types. Linear type contexts K are finite maps from linear variables to linear types. A memory type, H (for *heap*), is a mapping from locations and channels to their types. A synchronization state type T similarly maps linear channels to their linear types.

The typing judgments are defined by the rules in Figure 5. These judgments make use of auxiliary judgments that ensure values, linear values, and primitive operations are well-typed (Figure 6). For space reasons we have omitted the straightforward judgments for deciding that memories, synchronization environments, and process pools are well-formed, as well as the subtyping and subsumption rules.

The type system guarantees the following properties:

- Explicit and implicit insecure information flows are ruled out, if the program is also race-free.
- Channel names introduced by a linear handler are used exactly once in each possible future execution path.

Typing judgments have the form $H; \Gamma; T; K\ [\mathsf{pc}] \vdash P$, which asserts that process P is well-typed in the context defined by H, Γ, T, K, and a program-counter label pc. Nonlinear contexts Γ permit weakening and contraction, whereas linear contexts K do not; thus, a process typed assuming a set of linear message handlers must use all of them.

The type system uses the pc label to bound what can be learned by seeing that the program execution has reached P. The $[\mathsf{pc}]$ component of the judgment is thus a lower bound on the label of memory locations that may be written by P. The rule that increases pc is IF, because branching transfers information to the program counter.

Most of the typing rules are similar in spirit to those in recent security-typed languages [17, 49, 30], though there are a few rules of special interest.

Concurrent processes $P_1 \mid P_2$ are checked using the program-counter label of the parent process, as shown in rule PAR of Figure 5. The two processes have access to the same nonlinear resources, but the linear resources must be partitioned between them.

The typing rules LET and LETLIN make use of auxiliary operations that extract variable binding information from handler definitions. A join pattern J yields a collection Γ_f of channels it defines and a set of variables bound in the body of the handler definition Γ_{args}. For nonlinear join patterns, the linear variables form a synchronization context K. The operation $J \leadsto_{\mathsf{pc}} \langle \Gamma_f;\ \Gamma_{args};\ K\rangle$, defined in Figure 7 collects these channel names and variables for nonlinear join patterns and assigns them types. A similar operation $J \leadsto \langle K;\ \Gamma_{args}\rangle$ defined for linear join patterns extracts the synchronization point K and the context for the handler body, Γ_{args}.

Rule LET checks the body of the handler under the assumption that the arguments bound by the join pattern have the appropriate types. Nonlinear handlers cannot capture free linear values or channels, because that would potentially violate their linearity. Consequently, the only linear resources available inside the body P_1 are those explicitly passed to the handler: K_{args}. Note that the channels defined by the nonlinear handler (Γ_f) are available inside the handler body, which allows recursion. The process P_2 has access to the newly defined channels (in Γ_f) and to the previously

$$\text{PRIM}\quad \frac{\begin{array}{c} H; \Gamma\ [\mathsf{pc}] \vdash prim : s \\ H; \Gamma, x{:}s; T; K\ [\mathsf{pc}] \vdash P \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{let}\ x = prim\ \texttt{in}\ P}$$

$$\text{REF}\quad \frac{\begin{array}{c} H; \Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s) \\ H; \Gamma, x{:}s\ \texttt{ref}_{\mathsf{pc}}; T; K\ [\mathsf{pc}] \vdash P \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{let}\ x = \texttt{ref}\, v\ \texttt{in}\ P}$$

$$\text{ASSN}\quad \frac{\begin{array}{c} H; \Gamma \vdash v : s\ \texttt{ref}_\ell \quad H; \Gamma; T; K\ [\mathsf{pc}] \vdash P \\ H; \Gamma \vdash v' : s \quad \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{label}(s) \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{set}\ v := v'\ \texttt{in}\ P}$$

$$\text{IF}\quad \frac{\begin{array}{c} H; \Gamma\ [\mathsf{pc}] \vdash v : \texttt{bool}_\ell \\ H; \Gamma; T; K\ [\mathsf{pc} \sqcup \ell] \vdash P_i \quad (i \in \{1,2\}) \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{if}\ v\ \texttt{then}\ P_1\ \texttt{else}\ P_2}$$

$$\text{ZERO}\quad H; \Gamma; \cdot; \cdot\ [\mathsf{pc}] \vdash 0$$

$$\text{PAR}\quad \frac{H; \Gamma; T_i; K_i\ [\mathsf{pc}] \vdash P_i \qquad (i \in \{1,2\})}{H; \Gamma; T_1, T_2; K_1, K_2\ [\mathsf{pc}] \vdash P_1 \mid P_2}$$

$$\text{LET}\quad \frac{\begin{array}{l} J \leadsto_{\mathsf{pc}} \langle \Gamma_f;\ \Gamma_{args};\ K_{args}\rangle \\ H; \Gamma, \Gamma_f, \Gamma_{args}; \cdot, K_{args}\ [\mathsf{pc}] \vdash P_1 \\ H; \Gamma, \Gamma_f; T, K\ [\mathsf{pc}] \vdash P_2 \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash \texttt{let}\ J \triangleright P_1\ \texttt{in}\ P_2}$$

$$\text{LETLIN}\quad \frac{\begin{array}{l} J \leadsto \langle K_f;\ \Gamma_{args}\rangle \\ H; \Gamma, \Gamma_{args}; T_1; K_1\ [\mathsf{pc}] \vdash P_1 \\ H; \Gamma; T_2; K_2, K_f\ [\mathsf{pc}] \vdash P_2 \end{array}}{H; \Gamma; T_1, T_2; K_1, K_2\ [\mathsf{pc}] \vdash \texttt{let}\ J \multimap P_1\ \texttt{in}\ P_2}$$

$$\text{SEND}\quad \frac{\begin{array}{l} H; \Gamma \vdash v : [\mathsf{pc}'](\vec{s}, k^{\mathsf{opt}})_\ell \\ H; \Gamma\ [\mathsf{pc}] \vdash v_i : s_i \\ T; K \vdash lv^{\mathsf{opt}} : k^{\mathsf{opt}} \\ \mathsf{pc} \sqcup \ell \sqsubseteq \mathsf{pc}' \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash v(\vec{v}, lv^{\mathsf{opt}})}$$

$$\text{LINSEND}\quad \frac{\begin{array}{c} T; K \vdash lv : (\vec{s}) \\ H; \Gamma\ [\mathsf{pc}] \vdash v_i : s_i \end{array}}{H; \Gamma; T; K\ [\mathsf{pc}] \vdash lv(\vec{v})}$$

Figure 5. Process typing

$$H;\Gamma \vdash x : \Gamma(x) \qquad \cdot;\, y{:}k \vdash y : k$$

$$H;\Gamma \vdash \mathtt{t}_\ell : \mathtt{bool}_\ell \qquad c{:}k;\, \cdot \vdash c : k$$

$$H;\Gamma \vdash \mathtt{f}_\ell : \mathtt{bool}_\ell \qquad \frac{\vdash k_1 \leq k_2 \quad T;K \vdash lv : k_1}{T;K \vdash lv : k_2}$$

$$H;\Gamma \vdash L_\ell : H(L) \sqcup \ell$$

$$H;\Gamma \vdash c_\ell : H(c) \sqcup \ell \qquad \frac{\vdash s_1 \leq s_2 \quad H;\Gamma \vdash v : s_1}{H;\Gamma \vdash v : s_2}$$

$$\frac{H;\Gamma \vdash v : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{H;\Gamma\ [\mathsf{pc}] \vdash v : s}$$

$$\frac{H;\Gamma \vdash v : \mathtt{bool}_\ell \quad H;\Gamma \vdash v' : \mathtt{bool}_\ell \quad \mathsf{pc} \sqsubseteq \ell}{H;\Gamma\ [\mathsf{pc}] \vdash v \oplus v' : \mathtt{bool}_\ell}$$

$$\frac{H;\Gamma \vdash v : s\ \mathtt{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell)}{H;\Gamma\ [\mathsf{pc}] \vdash\ !v : s \sqcup \ell}$$

Figure 6. $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ value and operation types

$$f(\vec{x}) \rightsquigarrow_{\mathsf{pc}} \langle f : [\mathsf{pc}](\vec{s});\ \vec{x}:\vec{s};\ \emptyset\rangle$$

$$f(\vec{x}, y) \rightsquigarrow_{\mathsf{pc}} \langle f : [\mathsf{pc}](\vec{s}, k);\ \vec{x}:\vec{s};\ y:k\rangle$$

$$\frac{\begin{array}{c} J_1 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f1};\ \Gamma_{args1};\ K_1\rangle \\ J_2 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f2};\ \Gamma_{args2};\ K_2\rangle \end{array}}{J_1 \mid J_2 \rightsquigarrow_{\mathsf{pc}} \langle \Gamma_{f1}, \Gamma_{f2};\ \Gamma_{args1}, \Gamma_{args2};\ K_1, K_2\rangle}$$

$$f(\vec{x}) \rightsquigarrow \langle f : (\vec{s});\ \vec{x}:\vec{s}\rangle$$

$$\frac{J_1 \rightsquigarrow \langle K_1;\ \Gamma_{args1}\rangle \quad J_2 \rightsquigarrow \langle K_2;\ \Gamma_{args2}\rangle}{J_1 \mid J_2 \rightsquigarrow \langle K_1, K_2;\ \Gamma_{args1}, \Gamma_{args2}\rangle}$$

Figure 7. Join pattern bindings

available resources. Rule LETLIN operates similarly but external linear channels may be used in the handler body.[3]

The rule for type-checking sends on nonlinear channels requires that the channel type and the types of the values passed in the message agree. Also, the program counter at the point of the send must be protected by the label of the message handler, which rules out implicit information flows.

Sending a message on a linear channel does not impose any constraints on the pc label at the point of the send, because there is no information revealed by the act of sending on a linear channel. Note that the contents of the messages are labeled with the pc label, because the linear message might contain information about the program counter.

[3]Unlike the related linear-continuation type system [49], the type system presented in Figure 5 does not explicitly enforce any ordering constraints on the use of linear channels, relying instead on race freedom. Race freedom implies that there is a *causal* relationship between the linear synchronization handlers such that any two handlers that interfere are totally ordered. Interfering linear handlers are used sequentially, so updates to memory locations mentioned in the handlers are deterministic.

4.3 Race prevention and alias analysis

As discussed above, two concurrently running threads might leak confidential information if they have write–write or read–write races. Rather than further complicating the type system with race-condition analysis, we assume that a separate program analysis rejects programs that contain such races. This strategy modularizes the proof of security: Any program analysis that soundly guarantees race freedom can be used in conjunction with this type system.

The remainder of this section formalizes our race freedom condition and sketches some ways that existing technology can be used to determine whether a given program satisfies the condition.

Intuitively, the definition of race freedom requires that steps performed by parallel threads can be interleaved in any order. Formally, a configuration m is **race-free** whenever $m \Rrightarrow^* m'$ and $m' \Rrightarrow m_1$ and $m' \Rrightarrow m_2$ and $m_1 \not\equiv m_2$ imply that there exists an m'' such that $m_1 \Rrightarrow m''$ and $m_2 \Rrightarrow m''$. An open term is race-free whenever its closed instances are race-free.

This is a strong notion of race freedom (it implies confluence), though certainly sufficient to rule out timing leaks that may occur between threads. It is possible to weaken the definition of race freedom to consider harmful only nondeterminism apparent from the memory, but even with a weakened definition of race freedom, nondeterminism on nonlinear channels can cause races.

There are a number of ways to establish race freedom. One approach is to use an alias analysis to soundly approximate the set of locations and channels written to (or sent messages) by a thread P. Call this set $\mathsf{write}(P)$. By determining which locations are potentially read by P (a set $\mathsf{read}(P)$), an analysis can prevent races by requiring the following (and its symmetric counterpart) for any subprograms P_1 and P_2 that might execute concurrently:

$$\mathsf{write}(P_1) \cap (\mathsf{read}(P_2) \cup \mathsf{write}(P_2)) = \emptyset$$

An alias analysis constructs finite models of the dynamic behavior of a program in order to approximate which references are instantiated with which memory locations at run time. The more closely the abstract model agrees with the true behavior of the system, the more accurate aliasing information can be. Formulating such an alias analysis is orthogonal to this work; instead, we have simply stated the race-freedom property the analysis must enforce.

One trivially sound analysis is to assume that any process might read or write any reference. Such a rough approximation to the actual aliasing forces the program to be sequential. A second possibility is to ensure that references and nonlinear channels are used sequentially. Simple syntactic constraints can enforce this property [31]. Sequentiality can also be formulated in the type system [19], but doing so is rather complex.

Another possibility is to track aliasing directly in the type system [40, 46], which would potentially permit very fine-grained control of concurrency. More generally, pointer or shape analysis can be used to approximate $\mathsf{read}(-)$ and $\mathsf{write}(-)$ [21, 9, 10, 33]. Because $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$'s sequential core is essentially an imperative language with `goto`, we expect that these existing alias analyses can be adapted to this setting.

5 Security condition

We now present more formally the security condition that the $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ language enforces and give an outline of the proof techniques used.

Well-typed $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ programs satisfy the determinism-based security condition described informally in Sections 2.3 and 2.4. The proof strategy is to show that regardless of the high-security inputs to a program, its low-security memory access behavior can be simulated by a single, deterministic program that differs only in its high-security parts. The existence of a common simulation implies that low-security behavior reveals nothing about the inputs.

The remainder of this section sketches the proof that the type system in conjunction with a race-freedom analysis implies the security condition. A complete proof is available [48].

5.1 Subject reduction

Subject reduction is a crucial first step of the security proof because it establishes that evaluation preserves well-formedness.

Lemma 5.1 (Subject reduction) *Suppose that*

$$H;T \vdash \langle M,\, S,\, N\rangle \text{ and } \langle M,\, S,\, N\rangle \to \langle M',\, S',\, N'\rangle$$

Then there exist H' and T' such that $H';T' \vdash \langle M',\, S',\, N'\rangle$. Furthermore, H' extends H, and T and T' agree on the channels in their intersection.

Proof: The proof is by cases on the evaluation step used. A number of additional lemmas are needed:

- Substitution: the usual lemmas for type preservation when substituting for linear and nonlinear variables;
- Weakening: preservation of typing judgments when program counter labels are lowered or heap types and synchronization environments are extended. □

5.2 Observational equivalence

The ζ-equivalence relation indicates when two values look the same to a low-level observer.

Definition 5.1 (ζ-equivalence) *Let ζ-equivalence (written $\approx_\zeta$) be the family of symmetric binary relations inductively defined as follows.*

- *For values:*

$$H;\Gamma \models v_1 \approx_\zeta v_2 : t_\ell \quad \Leftrightarrow$$
$$H;\Gamma \vdash v_i : t_\ell \;\wedge\; (\ell \sqsubseteq \zeta \Rightarrow v_1 = v_2)$$

- *For linear values:*

$$T \models c_1 \approx_\zeta c_2 : k \quad \Leftrightarrow \quad T(c_i) = k \;\wedge\; c_1 = c_2$$

5.3 Process simulation

Generalizing ζ-equivalence to processes is more involved because there can be both high-security and low-security computations running simultaneously, making it harder to relate corresponding parts of subprograms. Bisimulation-based proof techniques for noninterference are no longer appropriate. Rather than giving a definition of $\approx_\zeta$ for traces directly, we instead give a simulation relation $\lesssim_\zeta$. Two programs are then ζ-equivalent if they can both be simulated by the same program.

The simulation relation is induced by the typing structure of a source machine configuration. Intuitively, if $m \lesssim_\zeta m'$ then configuration m' can simulate the low-security behavior of m while ignoring both the timing and termination behavior of the high-security computation in m.

Definition 5.2 (ζ-simulation) *Let $\lesssim_\zeta$ be the relation (mutually) inductively defined as shown in Figures 8, 9, 10, 11 and in the rules below.*

For configurations: SIM-CONFIG

$$\frac{\begin{array}{l} H \vdash M_1 \lesssim_\zeta M_2 \\ H; T_1, T_2 \vdash S_1 \lesssim_\zeta S_2, T_2 \\ H; \cdot; T_2; \cdot \vdash N_1 \lesssim_\zeta N_2 \end{array}}{H; T_1, T_2 \vdash \langle M_1,\, S_1,\, N_1\rangle \lesssim_\zeta \langle M_2,\, S_2,\, N_2\rangle}$$

For processes with $\mathsf{pc} \not\sqsubseteq \zeta$: SIM-HIGH-PROC

$$\frac{\mathsf{pc} \not\sqsubseteq \zeta \quad T(c_i) = (\vec{s_i}) \quad H;\Gamma \vdash \vec{v_i} : \vec{s_i}}{H;\Gamma;T;\cdot\,[\mathsf{pc}] \vdash P \lesssim_\zeta \;|_i\; c_i(\vec{v_i})}$$

For processes that are well-typed with $\mathsf{pc} \sqsubseteq \zeta$, the $\lesssim_\zeta$ relationship acts homomorphically on the typing rule of the term, replacing the judgment $H;\Gamma \vdash v : s$ with the equivalence rule $H;\Gamma \vdash v_1 \approx_\zeta v_2 : s$ (and similarly for primitive operations). For example, the simulation for conditionals is derived from the typing rule IF*:*

$$\frac{\begin{array}{l} \mathsf{pc} \sqsubseteq \zeta \\ H;\Gamma[\mathsf{pc}] \vdash v_1 \lesssim_\zeta v_2 : \mathtt{bool}_\ell \\ H;\Gamma;T;\cdot\,[\mathsf{pc} \sqcup \ell] \vdash P_{1i} \lesssim_\zeta P_{2i} \quad i \in \{1,2\} \end{array}}{\begin{array}{l} H;\Gamma;T;\cdot\,[\mathsf{pc}] \vdash \mathtt{if}\; v_1 \;\mathtt{then}\; P_{11} \;\mathtt{else}\; P_{12} \lesssim_\zeta \\ \qquad \mathtt{if}\; v_2 \;\mathtt{then}\; P_{21} \;\mathtt{else}\; P_{22} \end{array}}$$

The most important part of the $\lesssim_\zeta$ relation is SIM-HIGH-PROC. This rule says that any process that is well typed with a pc label not protected by ζ can be simulated by the process that just sends a response on each of the linear channels. Intuitively, this simulation bypasses all of the potential high-security computation performed in P and simply returns via the linear-channel invocations. Importantly, for high-security process $P \lesssim_\zeta P'$ the simulation P' always terminates, even if P does not; the simulation ignores the termination behavior of P.

Observe that any value returned from a high-security context via linear channels must itself be high-security, because its label must protect that context's program-counter label. Therefore, it does not matter what value is returned in the ζ-simulation because that value is not observable anyway.

Also note that if there are no linear channels in the context, the rule SIM-HIGH-PROC says that $P \lesssim_\zeta 0$—the high-security process P has no way of affecting low-security memory locations, so from the low-security view, P may as well not exist.

In this setting the $\lesssim_\zeta$ relation is more fundamental and easier to work with than $\approx_\zeta$, because $\approx_\zeta$is not transitive and hence not an equivalence relation. Two machine configurations are ζ-equivalent if they are both simulated by the same configuration:

Definition 5.3 (ζ-equivalence for configurations) *Configurations m_1 and m_2 are ζ-equivalent, written $H;T \models m_1 \approx_\zeta m_2$, if and only if there exists a configuration m such that $H;T \vdash m_1 \lesssim_\zeta m$ and $H;T \vdash m_2 \lesssim_\zeta m$.*

To prove that $\lambda^{\mathrm{PAR}}_{\mathrm{SEC}}$ satisfies the security condition, we first show that the simulation preserves typing and respects the operational semantics.

Lemma 5.2 (Simulation preserves typing) *If $H;T \vdash m$ and $H;T \vdash m \lesssim_\zeta m'$ then $H;T \vdash m'$.*

Proof: By induction on the derivation of $H;T \vdash m$; the inductive hypothesis must be extended to the other judgment forms. The one interesting case is SIM-HIGH-PROC, which holds because each free linear channel c_i mentioned in P is used exactly once in the simulation $\mid_i c_i(\vec{v_i})$. □

Now we establish that the simulation respects the operational semantics.

Lemma 5.3 ($\rightarrow$-simulation) *If $H;T \models m_1 \lesssim_\zeta m_2$ and $m_1 \rightarrow m_1'$ then either $m_1' \lesssim_\zeta m_2$ or there exists H', T', and m_2' such that $m_2 \rightarrow m_2'$ and $H', T' \models m_1' \lesssim_\zeta m_2'$.*

Proof: This is proved by separately considering low-security and high-security evaluation steps ($\mathsf{pc} \sqsubseteq \zeta$ and $\mathsf{pc} \not\sqsubseteq \zeta$, respectively). In either case it is shown that a step of m_1 can be simulated by zero or one steps performed by its simulation m_2. The proofs are by cases on the evaluation step taken by m_1. A similar $\Rrightarrow$-simulation result follows. □

Next, we show that the race-freedom condition implies deterministic updates. Given a configuration $m = \langle M, S, N\rangle$, we define $m(L)$ as $M(L)$.

SIM-VAL
$$\frac{H;\Gamma \models v_1 \approx_\zeta v_2 : s \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s)}{H;\Gamma[\mathsf{pc}] \vdash v_1 \lesssim_\zeta v_2 : s}$$

SIM-BINOP
$$\frac{\begin{array}{l} H;\Gamma \models v_{11} \approx_\zeta v_{12} : \mathtt{bool}_\ell \\ H;\Gamma \models v_{21} \approx_\zeta v_{22} : \mathtt{bool}_\ell \qquad \mathsf{pc} \sqsubseteq \ell \end{array}}{H;\Gamma[\mathsf{pc}] \vdash v_{11} \oplus v_{12} \lesssim_\zeta v_{21} \oplus v_{22} : \mathtt{bool}_\ell}$$

SIM-DEREF
$$\frac{H;\Gamma \models v_1 \approx_\zeta v_2 : s\ \mathtt{ref}_\ell \quad \mathsf{pc} \sqsubseteq \mathsf{label}(s \sqcup \ell)}{H;\Gamma[\mathsf{pc}] \vdash !v_1 \lesssim_\zeta !v_2 : s \sqcup \ell}$$

Figure 8. Primitive operation simulation

SIM-EMPTY
$$\frac{\begin{array}{l} H \vdash M \\ \forall L \in \mathrm{dom}(M).\mathsf{label}(H(L)) \not\sqsubseteq \zeta \\ \forall c \in \mathrm{dom}(M).\mathsf{label}(H(c)) \not\sqsubseteq \zeta \end{array}}{H \vdash M \lesssim_\zeta \cdot}$$

SIM-LOC
$$\frac{H \vdash M_1 \lesssim_\zeta M_2 \quad H \models v_1 \approx_\zeta v_2 : H(L)}{H \vdash M_1[L \mapsto v_1] \lesssim_\zeta M_2[L \mapsto v_2]}$$

SIM-J
$$\frac{\begin{array}{l} H \vdash M_1 \lesssim_\zeta M_2 \\ J \rightsquigarrow_{\mathsf{pc}} \langle \{c_i : H(c_i)\};\ \{\vec{x_i} : \vec{s_i}\};\ \{y_i^{\mathsf{opt}} : k_i^{\mathsf{opt}}\}\rangle \\ H(c_i) = [\mathsf{pc}](\vec{s_i}, k_i^{\mathsf{opt}}) \\ H; \vec{x_i} : \vec{s_i}; \cdot; \{y_i^{\mathsf{opt}} : k_i^{\mathsf{opt}}\}\ [\mathsf{pc}] \vdash P_1 \lesssim_\zeta P_2 \end{array}}{H \vdash M_1[J \rhd P_1] \lesssim_\zeta M_2[J \rhd P_2]}$$

Figure 9. Memory simulation

SIM-S-EMPTY
$$\frac{\begin{array}{l} H;T \vdash S;T' \\ \forall[\mathsf{pc} : J \multimap P] \in S.\mathsf{pc} \not\sqsubseteq \zeta \end{array}}{H;T \vdash S \lesssim_\zeta \cdot, T'}$$

SIM-S-HANDLER
$$\frac{\begin{array}{l} H;T \vdash S_1 \lesssim_\zeta S_2, T_1 \\ J \rightsquigarrow \langle \{c_i : T(c_i)\};\ \{\vec{x_i} : \vec{s_i}\}\rangle \\ T(c_i) = (\vec{s_i}) \quad T_2 \subseteq T \\ H; \vec{x_i} : \vec{s_i}; T_2; \cdot\ [\mathsf{pc}] \vdash P_1 \lesssim_\zeta P_2 \end{array}}{\begin{array}{l} H;T \vdash S_1[\mathsf{pc} : J \multimap P_1] \lesssim_\zeta \\ \qquad S_2[\mathsf{pc} : J \multimap P_2], T_1, T_2 \end{array}}$$

Figure 10. Synchronization environment simulation

SIM-NONE
$$H;\Gamma;\cdot;\cdot \vdash \cdot \lesssim_\zeta \cdot$$

SIM-PROC
$$\frac{\begin{array}{l} H;\Gamma;T_1;K_1 \vdash N \lesssim_\zeta N' \\ H;\Gamma;T_2;K_2\ [\mathsf{pc}] \vdash P \lesssim_\zeta P' \end{array}}{\begin{array}{l} H;\Gamma;T_1,T_2;K_1,K_2 \vdash N \mid [\mathsf{pc} : P] \lesssim_\zeta \\ \qquad N' \mid [\mathsf{pc} : P'] \end{array}}$$

SIM-EQ
$$\frac{\begin{array}{l} N_1 \equiv N_2 \qquad N_3 \equiv N_4 \\ H;\Gamma;T;K \vdash N_2 \lesssim_\zeta N_3 \end{array}}{H;\Gamma;T;K \vdash N_1 \lesssim_\zeta N_4}$$

Figure 11. Process pool simulation

Lemma 5.4 (Race freedom and determinism) *Suppose that both $m_{(0,0)} \Rrightarrow^* m_{(i,0)}$ and $m_{(0,0)} \Rrightarrow^* m_{(0,j)}$ where the last step of each evaluation sequence makes a low-observable change to location L, but no prior step does. Then $m_{(i,0)}(L) \approx_\zeta m_{(0,j)}(L)$.*

Proof: By induction on (i, j), applying a collection of related lemmas that use race freedom to reason about equivalence of configurations after single syntax-independent evaluation steps. □

To talk about programs that differ only in high-security inputs, we use *substitutions*. A substitution γ is a finite map from variables to values. If $\mathit{\Gamma}$ is a typing environment and γ is a substitution, we write $\gamma \models \mathit{\Gamma}$ to mean that γ assigns each variable a value of the type required by Γ. The notation $\gamma(e)$ is short-hand for simultaneous capture-avoiding substitutions. We write $H \models \gamma_1 \approx_\zeta \gamma_2 : \mathit{\Gamma}$ if the two substitutions satisfy $\mathit{\Gamma}$ and map each variable to equivalent values given heap type H.

The following lemma says that starting from a configuration with an open process pool and closing it under equivalent substitutions yields equivalent configurations.

Lemma 5.5 (Simulations and high-substitution) *Suppose the following hold:*

$$\begin{array}{l} H \vdash M \text{ and } H; T_1, T_2 \vdash S; T_1 \text{ and } H; \mathit{\Gamma}; T_2; \cdot \vdash N \\ \text{and} \quad H \models \gamma_1 \approx_\zeta \gamma_2 : \mathit{\Gamma} \\ \text{and} \quad \forall x \in \mathrm{dom}(\mathit{\Gamma}).\ \mathsf{label}(\mathit{\Gamma}(x)) \not\sqsubseteq \zeta \end{array}$$

Then for any configuration m,
$H; T \models \langle M, S, \gamma_1(N)\rangle \lesssim_\zeta m$ *implies*
$H; T \models \langle M, S, \gamma_2(N)\rangle \lesssim_\zeta m$

Proof: An easy induction on the typing derivation for process pool N. The inductive cases follow from the construction of $\lesssim_\zeta$. □

5.4 Determinism-based security

Given the technical machinery above, we can state the security theorem as follows.

Theorem 5.1 (Determinism-based security) *Let ζ be an arbitrary label in the security lattice. Suppose that*

$$H; x\!:\!s; \cdot; \cdot \vdash N$$

Let an initial memory M be given such that $H \vdash M$ and suppose that whenever $H; \cdot \vdash \langle M, \cdot, N\rangle \lesssim_\zeta m$ the simulation m is race-free. Let location $L \in \mathrm{dom}(H)$ be given such that $\mathsf{label}(H(L)) \sqsubseteq \zeta$. Further suppose that $\mathsf{label}(s) \not\sqsubseteq \zeta$. Then for any two values v_1 and v_2 such that $H; \cdot \vdash v_i : s$ the sequence of values stored in memory location L during the evaluation of $\langle M, \cdot, N\{v_1/x\}\rangle$ is a prefix of the sequence of values stored in L by $\langle M, \cdot, N\{v_2/x\}\rangle$ (or vice-versa).

Proof: By Lemma 5.5 there exists a configuration m' that simulates both configurations. Suppose, for the sake of contradiction, that the evaluations disagree on the $n+1^{st}$ update to the location L. We derive a contradiction by induction on n, using Lemma 5.4. □

Note that in order to establish that a given program is secure, not only must the program be well-typed, but *all* of its ζ-simulations must be race-free. An easy lemma shows that if $m \lesssim_\zeta m'$ and m is race-free then m' is race-free. Consequently, it suffices to perform the race-freedom analysis on the original source program—if the original program is race-free, all of its simulations are too.

6 Related work

There has been a long history of information-flow research based on trace models of computer systems [14, 23, 16, 25, 47] and process algebras [34, 39, 35]. Early programming languages work in this area was initiated by Denning [8] and Reynolds [31].

A few researchers have investigated noninterference-based type systems for concurrent languages and process calculi. Smith and Volpano have assumed a fixed number of threads and a uniform thread scheduler; however, their work accounts for probabilistic thread scheduling and relaxes the constraints due to timing channels [43, 44, 41].

McLean [24] and Roscoe [32] proposed determinism-based definitions of noninterference for trace-based and labeled-transition systems. Their approach has not been used previously in type systems for programming languages.

Focardi and Gorrieri [12] have implemented a flow-checker for a variant of Milner's calculus of concurrent systems (CCS). Honda et al. have proposed a similar system for the π-calculus in which they can faithfully encode Smith and Volpano's language [19]. Their work relies on a sophisticated type system that distinguishes between linear channels, affine channels, and nonlinear channels. Both of these approaches use bisimulation to prove possibilistic noninterference properties. A similar approach is taken by Abadi and Gordon to prove the correctness of cryptographic protocols in the Secure Pi Calculus [2], but they do not enforce information-flow policies.

Hennessy and Riely consider information-flow properties in the asynchronous π-calculus [18]. Like the definition used here, their may-testing version of noninterference is timing- and termination-insensitive, though possibilistic. Their language does not support synchronous communication or refinement of the flow analysis via linearity constraints.

Pottier [29] gives an elementary proof of possibilistic noninterference for a variant of the π-calculus, but its type system is restrictive because it does not make a distinction between linear and nonlinear channel usage. Conchon [6] also gives a similar information analysis for the join calculus, with a strong security condition based on bisimulation.

Sabelfeld and Sands have considered concurrent languages in a probabilistic setting [38], achieving scheduler independence. They use a novel probabilistic bisimulation approach to specify noninterference properties, and have used the techniques to prove correct Agat's program transformation for eliminating timing channels [3]. However, this transformation does not allow high-pc loops or recursive functions, and it allows leaks through instruction-cache effects. Mantel and Sabelfeld [22] have also considered type systems for multithreaded secure message-passing languages, including the use of Agat's transformation to pad timing channels [36] and thus achieve scheduler independence.

The design of $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$ was inspired by concurrent process calculi such as the π-calculus [26] and especially the join calculus [13].

7 Conclusions

This paper makes two contributions: first, it presents and formalizes in a language setting a definition of information security based on observational determinism. This definition has the advantage over possibilistic security properties that it is immune to refinement attacks, eliminating covert channels based on the observation of the probabilities of possible results. Two insights make the observational determinism an effective basis for a security condition: internal and external timing channels are treated differently, and programs are required to be race-free, eliminating their ability to obtain information from the internal timing channels. Based on these insights, the attacker is formally modeled as having less power to observe differences between program executions, with the result that the security definition avoids some of the restrictiveness of alternative noninterference definitions. Thus, this work allows a tradeoff in expressiveness based on an estimation of the power of the attacker.

The second contribution is a demonstration that this new definition of security can be enforced by the type system of a concurrent language with expressive thread communication primitives in conjunction with a program analysis used to guarantee race freedom. Factoring the security analysis into a type system and an alias analysis simplifies the type system and permits more sophisticated analyses to be plugged in to obtain more precise characterizations of information flow. The language and its type system support linear message handlers that can synchronize on messages from multiple threads; this feature enables a less restrictive treatment of implicit flows. The $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$ language is not a full-featured programming language; nevertheless, it includes the key features that research on process calculi have shown to be important for concurrent programming. Therefore, the security analysis of $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$ should extend to other concurrent languages that support message passing. In addition, $\lambda_{\mathrm{SEC}}^{\mathrm{PAR}}$ is likely to be useful in designing secure intermediate languages, just as secure sequential calculi have been [49, 50].

Acknowledgments

The authors would like to thank Dan Grossman, E Lewis, Greg Morrisett, Andrei Sabelfeld, Fred Schneider, and Stephanie Weirich for their suggestions about earlier drafts of this work. The comments of the anonymous reviewers also improved the paper.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 147–160, San Antonio, TX, Jan. 1999.

[2] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, Jan. 1999.

[3] J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, Jan. 2000.

[4] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[5] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. *Merci, Maurice, A mosaic in honour of Maurice Nivat*, 281(1):109–130, June 2002.

[6] S. Conchon. Modular information flow analysis for process calculi. In I. Cervesato, editor, *Proc. Foundations of Computer Security Workshop*, Copenhagen, Denmark, 2002.

[7] D. E. Denning. A lattice model of secure information flow. *Comm. of the ACM*, 19(5):236–243, May 1976.

[8] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[9] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyand k-limiting. In *Proc. of the '94 SIGPLAN Conference on Programming Language Design*, pages 230–241, 1994.

[10] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive points-to analysis in the presence of function pointers. In *Proc. of the '94 SIGPLAN Conference on Programming Language Design*, pages 242–256, June 1994.

[11] R. J. Feiertag. A technique for proving specifications are multilevel secure. Technical Report CSL-109, SRI International Computer Science Lab, Menlo Park, California, Jan. 1980.

[12] R. Focardi and R. Gorrieri. The compositional security checker: A tool for the verification of information flow security properties. *IEEE Transactions on Software Engineering*, 23(9), Sept. 1997.

[13] C. Fournet and G. Gonthier. The Reflexive CHAM and the Join-Calculus. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 372–385, 1996.

[14] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, Apr. 1982.

[15] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society Press, Apr. 1984.

[16] J. Gray III and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–176. IEEE Computer Society Press, 1992.

[17] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, Jan. 1998.

[18] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. *ACM Transactions on Programming Languages and Systems*, 24(5), Sept. 2002.

[19] K. Honda and N. Yoshida. A uniform type structure for secure information flow. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, pages 81–92, Jan. 2002.

[20] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, Oct. 1973.

[21] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proc. of the SIGPLAN '92 Conference on Programming Language Design*, June 1992.

[22] H. Mantel and A. Sabelfeld. A unifying approach to the security of distributed and multi-threaded programs. *Journal of Computer Security*, 2002. To appear.

[23] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186. IEEE Computer Society Press, May 1988.

[24] J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1), 1992.

[25] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93. IEEE Computer Society Press, May 1994.

[26] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100(1):1–77, 1992.

[27] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, Jan. 1999.

[28] J. Palsberg and P. Ørbæk. Trust in the λ-calculus. In *Proc. 2nd International Symposium on Static Analysis*, number 983 in Lecture Notes in Computer Science, pages 314–329. Springer, Sept. 1995.

[29] F. Pottier. A simple view of type-secure information flow in the π-calculus. In *Proc. of the 15th IEEE Computer Security Foundations Workshop*, 2002.

[30] F. Pottier and V. Simonet. Information flow inference for ML. In *Proc. 29th ACM Symp. on Principles of Programming Languages*, Portland, Oregon, Jan. 2002.

[31] J. C. Reynolds. Syntactic control of interference. In *Proc. 5th ACM Symp. on Principles of Programming Languages (POPL)*, pages 39–46, 1978.

[32] A. W. Roscoe. CSP and determinism in security modeling. In *Proc. IEEE Symposium on Security and Privacy*, 1995.

[33] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proc. of the ACM SIGPLAN 1999 Conference on Programming Language Design*, pages 77–90, May 1999.

[34] P. Ryan. A CSP formulation of non-interference and unwinding. *Cipher*, pages 19–30, 1991.

[35] P. Ryan and S. Schneider. Process algebra and non-interference. In *Proc. of the 12th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[36] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. In *Proceedings of the 9th Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[37] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan. 2003.

[38] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. of the 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[39] S. Schneider. Security properties and CSP. In *Proc. IEEE Symposium on Security and Privacy*, 1996.

[40] F. Smith, D. Walker, and G. Morrisett. Alias types. In *Proc. of the 9th European Symposium on Programming*, volume 1782 of *Lecture Notes in Computer Science*, pages 366–381, 2000.

[41] G. Smith. A new type system for secure information flow. In *Proc. of the 14th IEEE Computer Security Foundations Workshop*, pages 115–125. IEEE Computer Society Press, June 2001.

[42] T. V. Vleck. Timing channels. poster session, IEEE TCSP conference, Oakland CA, May 1990. Available at http://www.multicians.org/timing-chn.html.

[43] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Journal of Computer Security*, 7(2,3):231–253, Nov. 1999.

[44] D. Volpano and G. Smith. Verifying secrets and relative secrecy. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 268–276. ACM Press, Jan. 2000.

[45] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[46] D. Walker and G. Morrisett. Alias types for recursive data structures. In *Workshop on Types in Compilation*, Sept. 2000.

[47] A. Zakinthinos and E. S. Lee. A general theory of security properties and secure composition. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, 1997.

[48] S. Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, 2002.

[49] S. Zdancewic and A. C. Myers. Secure information flow via linear continuations. *Higher Order and Symbolic Computation*, 15(2–3):209–234, Sept. 2002.

[50] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.