Securing Interactive Systems

Willard Rafnsson


© 2014 Willard Rafnsson

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 3670
ISSN 0346-718X

Technical Report 109D
Department of Computer Science and Engineering
Research group: Language-based Security

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31-772 1000

Printed at Chalmers Reproservice
Göteborg, Sweden 2014
Abstract

This thesis provides means to achieve end-to-end information-flow security in interactive systems. The elusiveness of this problem stems from the fact that interaction patterns, primitives, synchronous communication and nondeterminism combine in ways where seemingly innocuous systems compromise security in unexpected ways under interaction.

We study what it means for interactive systems to not leak information about confidential behavior into observable behavior in a nondeterministic setting. We focus on two properties: progress-sensitive noninterference (PSNI), requiring that observable behavior is invariant to confidential input, and progress-insensitive noninterference (PINI), permitting confidential input to impede the ability of a system to make progress on its observable output. The latter is a popular target of information-flow security enforcement mechanisms, e.g. JSFlow, Paragon, LIO and Jif. We formalize PINI and PSNI extensionally, based on the view the attacker has on the interaction. To identify the essence of interactive systems security, we explore classes of attacks PSNI and PINI must guarantee protection against, and find previous work ignores classes of attacks powered by varied presence of input – a high-bandwidth channel in the concurrent setting. This is due to limitations in the model used for system environments.

To address this, we devise a new, preservation-based, formalization of noninterference. Since preservation-based noninterference guarantees secure systems interact securely, it is compositional; we prove this for a core of combinators, and derive from it a rich language of security-preserving combinators. While both PSNI and PINI are preserved under arbitrary wirings, the latter is not preserved fairly; it relies fundamentally on lack of scheduling fairness to guarantee security of interactions, and is therefore unfit for autonomous interactive systems security.

To facilitate building secure systems in parts, we advance secure multi-execution (SME): a combinator which repairs insecurities. SME thus makes any interactive system, secure or not, readily pluggable into a secure composed system. We prove soundness for all fair schedulers, and redesign SME to enforce PSNI, obtaining a more semantics-preserving combinator. We give a language-independent model for information release in SME. For scenarios where semantics must be preserved, we present type-based enforcements of PSNI and PINI. The type systems guarantee absence of leaks through challenging constructs e.g. dynamic event handlers and lazy class initialization. Lastly, we give a combinator which places a logarithmic bound on leaks through progress. Together with the type-based enforcement of PINI, we get a permissive hybrid enforcement of a stronger property.

Keywords: semantics-based security, language-based security, information-flow security, information-flow control, program analysis, runtime enforcement, program transformation, concurrency, parallelism, multi-threading, scheduling, fairness, covert channels
ACKNOWLEDGEMENTS

My journey to a PhD has been full of adventure experienced in great company, leaving me with friendships, stories and fond memories which I will treasure for as long as I live. For this, I thank my advisor and friend Andrei Sabelfeld. Thank you for all the laughter we have shared, for the guidance you have provided me, and for catalyzing my transition to Willard 2.0. It truly has been a fortune and pleasure to have you as my mentor. I am looking forward to doing more research with you in the next several months, and to the fun we will have in the future.

Thank you Dave Sands, for your friendship, support and genuine concern for my personal and professional well-being, and for our many whiteboard sessions. Thank you Wolfgang Ahrendt, for expanding my taste in music, for serving on my PhD committee, and for your friendship. Thank you Devdatt Dubhashi for broadening my perspective on the world and the people in it.

My praise, admiration and gratitude go to the wonderful people in my department, for contributing to a relaxed, social atmosphere that is a pleasure to be in. Special thanks go to my fellow true gentlemen on the fifth floor, for all the procrastination and social events. Thanks go to Daniel Hedin, for being my consultant on life, for our amusing discussions, fruitful collaborations, after-hours fun, and for helping me learn Swedish. I thank my office mates: Arnar, my academic twin, for our shared experiences, and Pablo, for fueling my intrigue of brewed beverages, for tolerating my peculiarities, and for all the fun we had so far.

To my thesis opponent, Fred Schneider, I extend my thanks for providing enlightening and valuable feedback to an early draft of this thesis. My gratitude to Cormac Flanagan, Shriram Krishnamurthi, Andrew Myers, and Heiko Mantel for my inspirational visits to your groups. Many thanks to Cédric Fournet for the many enlightening chats we have had through the years. And thank you Kaj, for your patience and understanding.

My friends and family, new and old, deserve thanks. Pelle for the many late-night hangouts and for making me feel like I have an extended family in Sweden. Hans Hüttel for academic support and for recommending a PhD at Chalmers. My old friends Anders and Arild from AAU for our long coffee-addled nights of hacking and fun, and for sparking my computer-science passion. My ancient friends Jan and Ulrik whom I can always count on. My family, near and far, for their support.

At last, my endless gratitude goes to the love of my life, Freja. Mussi, du er min drøm. Thank you for all of your unconditional warmth and support, and for sharing your life with me. I look forward to growing old with you.
Contents

Abstract iii
Acknowledgements v

0 Introduction 1
  0.1 Information-Flow Control 2
    0.1.1 Policies 2
    0.1.2 Enforcement 3
  0.2 Challenges 4
  0.3 Thesis Contributions 7
    0.3.1 Interactive Systems 7
    0.3.2 Compositional Security 8
    0.3.3 Secure Multi-Execution 9
    0.3.4 Event-based Communication 9
    0.3.5 Class Initialization 10
  0.4 Conclusion 11

1 Securing Interactive Programs 17
  1.1 Introduction 17
  1.2 Framework 21
    1.2.1 Interactive Programs 22
    1.2.2 Observables 22
    1.2.3 Strategies 23
    1.2.4 Noninterference 24
  1.3 Generalized strategies for noninterference 25
    1.3.1 Total Strategies 25
    1.3.2 Deterministic Strategies 26
    1.3.3 Public Presence Labels 27
  1.4 Relation to streams for noninterference 29
  1.5 Compositionality 33
  1.6 Enforcement 37
    1.6.1 Syntax 38
chapter 0

Introduction

Our world is increasingly more connected and governed by software. We may realize this while riding a driverless subway, surfing the web to do on-line banking, shopping and social networking on our mobile devices, or while at the doctor having our implanted medical devices calibrated. Consequently, security becomes an ever-growing concern, due to the growing risk of attack, on the confidentiality and integrity of data, on our privacy, and on the reliability of real-time and safety-critical systems.

While security is traditionally enforced as operating-system-level policies on files, most attacks are application-level, typically taking the form of a buffer overrun, code injection, or race condition, bypassing operating system enforcement. Language-based security aims to provide protection from application-level attacks by use of programming-language techniques, e.g. static analysis, model checking, program transformation, reference monitors and property-based testing.

While many tools and techniques for guaranteeing the absence of certain attacks on applications have been proposed, they have not seen widespread use amongst application developers. Static enforcements do not lend themselves well to software already deployed, and are usually too restrictive. Dynamic enforcements incur a run-time overhead, and usually enforce weaker properties. Stakes are often not deemed high enough to mandate investment in the effort required to utilize these solutions. Finally, systems are often large and complex, with components written in different programming languages and distributed across different architectures; existing solutions typically lack the modularity and compositionality results needed to achieve end-to-end system-wide guarantees.

To achieve end-to-end security, we propose a formally-verified core, securely-modified shell approach. Here, the (few) core and safety-critical components, such as the hardware and operating system, are verified during development e.g. through static analysis or model checking, by virtue of being written in a domain-specific language, or by being generated correctly from specification. The (many) non-critical applications are then made secure through semantic modification, until formally-verified alternatives become available. Compositionality results are then leveraged for achieving end-to-end system-wide guarantees.
Chapter 0 : Introduction

This thesis provides means to achieve end-to-end information-flow security in a general interactive systems setting. We study what it means for interactive systems to not leak information about confidential behavior, and formalise this as noninterference properties. We show that the properties are preserved under composition, and provide a rich language for building large systems from parts. We explore combinators which repair insecurities, enabling immediate secure plugging of non-critical components into a secure composition. For critical components, we give static and hybrid enforcements of our properties that track information-flows through hitherto-unstudied language constructs.

0.1 Information-Flow Control

Information enters a program on sources and exits on sinks. As it runs, a program creates dependencies in the data it operates on. If an output on a sink depends on an input on a source, then there is an information flow from said source to said sink; the output, and knowledge of this dependency, can be used to reason about the input. If this flow of information is undesired, then an information leak has occurred.

The need to track the flow of information in programs has spawned the field information-flow control [11]. Since each program is written in a programming language, and these have a rigorous semantics, we can apply language-based techniques to analyze, in an automated manner, a program for information flows occurring in it, to enforce that the program satisfies a security policy of interest [24]. The choice of policy is a balancing act between the attacks the policy guarantees protection against (and therefore, to which extent it is preserved under composition), and the desired permissiveness of the enforcement mechanism.

0.1.1 Policies

A security policy places restrictions on the permitted dependencies between sources and sinks. To specify which dependencies are allowed, we typically label the sources and sinks with a confidentiality level, and (partially) order them. A popular example of such ordered levels are the confidentiality levels \(L\) (low, public) and \(H\) (high, secret), ordered as \(\subseteq = \{(L, L), (L, H), (H, H)\}\). So, if an output on an L sink depend on input from an H source, an information leak occurs.

The baseline for our policies is the notion of noninterference [10, 14], which states that any two runs of a program with the same L inputs will produce the same L outputs. Taken as-is, this condition is strong, as it requires that L outputs be independent of H inputs. While desirable, devising an enforcement mechanism for this condition which is sound and permissive is an ongoing challenge. Furthermore, the full strength of this condition is not always needed; depending on the attacker we are modeling, an instantiation of noninterference which allows some leaks may be acceptable. Finally, a carefully chosen instantiation can be enforced in a sound and permissive way.
A classic instantiation of noninterference in literature is termination-insensitive noninterference \cite{28, 24, 23}. Consider a batch program, which input is an initial memory, and output is the memory at program termination, both of which have $H$ and $L$ parts. The termination of some batch programs, such as \texttt{while \{skip\}}, can depend on $H$ input. Since the presence of output in such programs depends on $H$ inputs, they cannot be noninterfering. However, observing program termination gives at most 1 bit of information, which might be acceptable leak in some scenarios, say, if the program is executed only once, and the space of secret values is large. Thus the noninterference notion can be weakened, made insensitive to termination, by only requiring two runs with the same $L$ inputs to produce the same $L$ outputs when both runs terminate.

0.1.2 Enforcement

In information-flow control, a security policy is accompanied by a permissive enforcement mechanism, proven sound wrt. our security policy. When run on a program, if the enforcement reports a positive result, then the soundness proof implies that the program satisfies the policy.

There are several ways to achieve this effect, all of which rely strongly on the semantics of the movers of information, the language constructs. One is by use of static analysis, an analysis of the program performed before executing it. Typically, these take the form of a security type system \cite{28, 17} which, by tracking the confidentiality level of information contained in variables and program context, (over-)approximates information flows occurring in (an over-approximation of) the control flow paths the program can take. This makes it possible to guarantee the nonexistence of leaky control flow paths. One advantage of this approach is that the policy is enforced before the program is ever run, thus the program has no runtime overhead. Another is the ability of this approach to reason about all control flow paths; the analysis can ensure that an observer of $L$ outputs cannot, by inferring which control flow path was not taken by the program, learn about $H$ inputs. Since analysis is performed before the program is run, the enforcement has no access to runtime information. This means that a static enforcement cannot permissively enforce programs using highly dynamic language constructs, such as \texttt{eval}. The reason is that a large/infinite control flow branching occurs at these control points, so the static enforcement must make coarse approximations. The static enforcement in \cite{28} enforces termination insensitive noninterference for a simple while-language.

The counterpart of static analysis is dynamic analysis, which in our setting is usually a security monitor \cite{27, 3}. Here, inputted data is labeled, on run time, with the confidentiality level of the information the data carries, which then propagates through the channel with the data. When the monitor sees an output of data containing a $H$ label on a $L$ sink, the monitor prevents the leak, typically by blocking the program. While this blocking of the program can leak information, an advantage of dynamic analysis is that, since the analysis has access to the current control flow path, highly dynamic language constructs can be treated in
a permissive manner. However, the enforcement has runtime overhead, and cannot, without losing permissiveness, guarantee the absence of leaks through the observation that the program did not take a particular control flow path. Indeed, for any dynamic information-flow monitor, there is secure program, rejected by said monitor, which a sound static enforcement accepts.

The field of information flow control is diverse, addressing other aspects of security such as information integrity [6] and erasure [8]. Policies for runtime information declassification [26] have been developed. Information flow control can also be used to mitigate leaks through timing [11]. Static-dynamic hybrid enforcements have been developed to achieve a more permissive enforcement [25]. Policies have even been enforced in language-independent ways [12, 4]. We refer the reader to [24] for a detailed survey of the field.

0.2 Challenges

The field of information-flow control has matured greatly during the last two decades. Theoretical frameworks for information flow control, consisting of a policy and enforcement for a particular language, exist for many programming paradigms. Today, developments in the field are aimed at bridging the gap between theory and practice, to achieve end-to-end security of real systems. This thesis focuses on the distributed setting in general, with the web as an example scenario. Here, both the client-side JavaScript and the server-side code, written in i.e. Java, must be secured. Information-flow control of JavaScript [12, 16, 13, 19] and Java-like languages [20, 5, 15] is a popular area of research. We outline here the frontiers of information flow control and present intricate scenarios that are hard to handle or out of reach of current theory and tools for information flow control, to motivate our advancements in the field.

In the setting of interactive programs, an information channel which has received little attention is that of program progress [3], that is, whether more outputs are forthcoming or not. An L-observer capable of observing whether more L-outputs are forthcoming can, together with observing L-outputs while the program runs, and knowing the source code of the program, make precise inferences on which trace the program took, and thus which states the program reached. Contrast this to termination observations, that is, whether or not a program terminates. Here the inference on which trace the program took is coarse; the set of possible traces is partitioned in two, the ones where the program terminates, and the ones where the program diverges. Observing termination can thus convey at most 1 bit of information. In the interactive program setting, we commonly assume that an attacker cannot (directly) observe whether a program has terminated or not, so working with a termination insensitive noninterference is reasonable. However, the termination insensitive noninterference policy for batch programs described above is too weak for interactive programs, as it places no constraints on leaks through progress – a high-bandwidth channel. Consider, for instance, the left-hand side program in Figure 0.1. Each run of this program, with $h \geq 0$
initially, eventually outputs the $H$ input on a $L$ channel, and diverges. The termination behavior of this program does not depend on $H$ input; since no run ever terminates, this program is noninterfering according to the above-mentioned policy, yet leaks the whole secret input on $H$. The secret leaks through program progress; an observer seeing 0 followed by 1 knows that $h > 0$. This policy is therefore progress-insensitive [2]; secret values are permitted to affect whether more outputs are forthcoming or not. An interactive variation of this program is given on the right-hand side of Figure 0.1. Here, the program begins by receiving a $H$ value and storing it in memory. Then, for every $L$ value received, if the $L$ value equals the $H$ value, the system diverges, and otherwise returns to a state ready for $L$ input. The analog here is password guessing scenario; each time the observer inputs a message, he obtains information about the value received from $H$. A more appropriate instantiation is a progress-sensitive [2] noninterference, stating that any two program runs with the same $L$ inputs will produce the same $L$ output messages. However, progress-sensitive noninterference is hard to enforce permissively (explained later). There may be useful policies in the space between these all-or-nothing policies which allow low-bandwidth leaks through progress observations while at the same time being permissively enforceable.

While progress-sensitive noninterference policies have been devised [9, 22], the formulation of these makes a seemingly innocuous simplification, namely that of environment totality. Here, the environment of a program always has input available for the program to process. In practice, however, this is not necessarily the case, as the progress of one program can depend on the presence of an output from another program, in particular if the former blocks on input. The presence of a message can easily depend on data, as for instance in

$$\text{in } a \; x; \; \text{if } x = 0 \; \text{then out } b \; 0 \; \text{endif.}$$

This program only produces an output on $b$ if the input on $a$ is 0. Since data can contain confidential information, we have a possibility of leaking through the presence of messages. Environment totality thus reduces the space of attacks which the security policy guarantees protection against. For instance, in $H \; x; \; \text{out } L \; 0$ is secure under total environments, as the program always outputs 0 on $L$, since the environment always supplies input on $H$. In practice, this program leaks the presence of a message on $H$ to $L$, which is 1 bit of information. This leak can be magnified, such as in the program in Figure 0.2. This program consists of three programs, separated by ||, running concurrently.
The program attempts to read from the environment in a nonblocking manner on channel $H$. The program keeps trying to read until no input is ready for it during the nonblocking read, in which case $h \neq \text{UNDEFINED}$. Then the program produces an output on $L$. Now, if the environment always has input ready for this program,
repeat {
    poll H (h)
} until h != UNDEFINED
out L (0)

Figure 0.3: Extortionist

as is the case when the environment is total and communication between the environment and the program is synchronous, then this program never produces output on L. However, if environments can be nontotal, then for the environment which never has input ready for this program (which has the same observable behavior), this program eventually produces the L output. This program is problematic for compositionality, since it places a demand on the environment to produce H output infinitely to preserve the noneventuality of an output on L. However, this program does satisfy forward correctability [18], a policy in event-based systems formalisms which is touted as the weakest compositional security policy. This policy is formalized using traces of inputs and outputs, and accepts this program since it only considers possible observable behavior for provision of arbitrary (but finite) numbers of H input.

0.3 Thesis Contributions

In this section, we give a high-level overview of the contributions of each paper, along with a statement of contribution of the thesis author.

0.3.1 Interactive Systems

We demonstrate the intricacies of achieving compositional security in interactive programs by giving an example of secure components which, in composition, yield an insecure system, leaking information through the presence of messages. To plug leaks through message presence, we generalize the framework from [9, 22] for securing interactive programs by removing the environment totality assumption and by adding presence levels on channels. This yields a progress-sensitive possibilistic noninterference policy which does not permit leaks through message presence. In studying attacks on interactive systems, we show that it is sufficient to consider deterministic environments which, for each input channel, are total up to a certain (finite) number of inputs, after which the environment provides no more inputs on that channel. In the special case where all message presence is L, our security policy coincides with that of [9, 22]. In another special case where programs are deterministic, we demonstrate a similar result as [9], that it suffices to consider environments which are finite lists of inputs on each channel. We prove an appealing compositionality result: Parallel composition of secure programs results in a secure thread pool. Finally we illustrate how to track presence levels by means of static flow-sensitive analysis, and prove that the
Chapter 0: Introduction

static analysis enforces our security policy. The main lesson learned here is that if blocking on H-presence input can affect which L-observables the program can produce, an information leak occurs. While this can be enforced by raising the \( pc \) to H after a H-presence read, this is not particularly permissive, since the following secure program would be rejected:

\[
\text{(skip } \text{ in } H \ h) \text{; out L } 0.
\]

So tracking presence information in the presence of nondeterminism, synchronous communication and varied interaction pattern is a delicate matter.

Statement of Contribution  This paper was co-authored with Daniel Hedin and Andrei Sabelfeld. Willard is mainly responsible for formalizing and writing the results presented in the paper.

This paper has been published in the proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF 2012).

0.3.2 Compositional Security

We develop a system model which simplifies compositional reasoning of interactive systems, by assuming that systems are input total, that is, always ready to receive anything, and output productive, that is, always able to send something. We present a novel way to define noninterference: preservation-based noninterference. It requires that for any behavior a system can perform, the system can match the public projection of said behavior, regardless of how H information is fed into the system while it does so. This formalization facilitates a technique for proving compositionality: to show that a composition can match a sequence of actions, schedule the source of the next observable output in the sequence of actions. We demonstrate that this style of noninterference definition deals with problematic scenarios like such as scheduling, high interaction loops, and high output starvation, in an appropriate way. For progress-sensitive and progress-insensitive noninterference framed in this definition style, we prove that both are preserved under arbitrary binary composition. However, progress-insensitive noninterference relies fundamentally on lack of fairness to compose for even simple product compositions; components might need to starve one another to prevent information leaks. We therefore deem progress-insensitive noninterference unfit for reasoning about security for autonomous interactive systems. We give a rich language of combinators based on a core of combinators, and derive from their compositionality that the language is a language of secure combinators.

Statement of Contribution  This paper was co-authored with Andrei Sabelfeld. Willard is responsible for formalizing and writing the results presented in the paper.
0.3.3 Secure Multi-Execution

To facilitate building secure systems in parts, we advance a recent [12] runtime enforcement of timing-sensitive noninterference: secure multi-execution (sme). The sme of a program, in a H-L security lattice, runs two copies the program: a H-copy, and a L-copy. The L-copy gets L system input, and L system output comes from the L-copy only. The H-copy gets a copy of L-input after the L-copy has received it. H system output comes from the H-copy only. This approach achieves noninterference by isolation; the part of the system responsible for L output never receives H input from the environment. This approach thus defines a combinator which repairs insecurities in a given program, making any interactive system pluggable into a secure composition by smeing it. Our advancement introduce presence levels, and give more accurate results describing to which extent sme preserves the semantics of the original program by the above transformation: if the program was secure, then its per-channel input-output behavior is unaffected. Further, we prove that sme is secure for all fair schedulers. Next, we modify sme to enable information release between components in sme. Since sme is language-independent, our formalization of declassification is purely semantic, in contrast to many previous syntactic approaches to information release, where application logic in part defines the security policy it should satisfy [26]. Finally, we present a fully transparent sme; by relaxing the guarantee to progress-sensitive noninterference, we get for a redesigned sme that when applied on a program, will, modulo timeouts, fully preserve the input-output behavior of secure programs. Behavior is thus only modified as the result of a detected insecurity, or a timeout (which could also be the result of an insecurity). We furthermore demonstrate how, during runtime, the fully transparent sme can detect concrete attacks, that is, a proof by counterexample that the wrapped program does not satisfy noninterference.

Statement of Contribution This paper was co-authored with Andrei Sabelfeld. Willard is mainly responsible for formalizing and writing the results presented in the paper.

This paper has been published in the proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013).

0.3.4 Event-based Communication

We show that the bandwidth through progress observations can be bounded tightly by, after each input, forcing a progress-insensitive noninterfering program to buffer its outputs until the program is ready to consume input. We present a progress-bounded noninterference policy defined in terms of phases in the input stream; we obtain these phases by partitioning an input stream around observable inputs, and including each observable input in the (possibly empty) part immediately before it. Our policy now requires that any two runs of a program with the same L inputs must produce pairwise the same L outputs, until one run diverges silently on a whole phase; then no constraint is placed on the remainder of the other
run. We show that leaks through progress observations in a program following our policy are bounded by \( \log(n + 1) \), where \( n \) is the number of \( L \) inputs to the program. We then prove that progress-insensitive noninterfering program which buffers its output in the manner described above follows our policy. This gives rise to a hybrid enforcement mechanism; for an extension of the toy-subset of JavaScript from [7] with event hierarchies and handler switching, we give a combination of a flow-sensitive static dependency tracker (which tracks flows through these added constructs), and a program transformation which makes a program buffer its outputs. The main lesson here is that progress observations are a matter of great concern, and that buffering outputs mitigates leaks through progress observations. Further, this mitigation can be achieved in complete separation from any other security analysis performed on the program, say, by giving the mitigator as a combinator.

**Statement of Contribution** This paper was co-authored with Andrei Sabelfeld. Willard is mainly responsible for formalizing and writing the results presented in the paper.

This paper has been published in the proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS 2011).

### 0.3.5 Class Initialization

To avoid leaks through class initialization statuses, Jif disallows nonconstant non-primitive field initializers. Since these are useful, we opt instead to track flows through class initializations to guarantee the absence of leaks through initialization failure. We formalize the semantics of class initialization for a subset of Java with exception handling. We then demonstrate how to track information flows through class initialization statuses by a static type-and-effect system. The type system tracks flows through dependencies in the class hierarchy and field initializers, and tracks every context in which each class can be initialized in, to prevent leaks though class initialization such as the one presented previously. The effects of the type-and-effect system are class initialization statuses of classes, which we track to know which classes must be initialized at each point during the program run. At last we give a non-standard proof that the type-and-effect system enforces termination-insensitive noninterference. The main challenge here was to establish invariants which enable *compositional reasoning at the type level*.

**Statement of Contribution** This paper was co-authored with Keiko Nakata and Andrei Sabelfeld. This paper is based on an earlier conference version [21], and Willard is mainly responsible for formalizing and writing the results presented in this extended version. This extended version presents a more permissive enforcement which tracks flows through dependencies in field initializers and class hierarchies, and gives a soundness proof of this enforcement.

This paper has been published in the IEEE Transactions on Dependable and Secure Computing (TDSC) journal, Volume 10, Issue 1, Jan.-Feb. 2013.
0.4 Conclusion

The contributions of the papers enable repairing insecurities in programs by runtime enforcement or program transformation, and proving programs secure through program analysis. Compositionality results then give that programs, secured in either of these two approaches, composed in arbitrary ways, yield a secure composite system. This facilitate end-to-end information-flow security reasoning with little effort, and therefore has great potential for practical application.

References


Securing Interactive Programs

ABSTRACT  This paper studies the foundations of information-flow security for interactive programs. Previous research assumes that the environment is total, that is, it must always be ready to feed new inputs into programs. However, programs secure under this assumption can leak the presence of input. Such leaks can be magnified to whole-secret leaks in the concurrent setting. We propose a framework that generalizes previous research along two dimensions: first, the framework breaks away from the totality of the environment and, second, the framework features fine-grained security types for communication channels, where we distinguish between the security level of message presence and message content. We show that the generalized framework features appealing compositionality properties: parallel composition of secure program results in a secure thread pool. We also show that modeling environments as strategies leads to strong compositionality: various types of composition (with and without scoping) follow from our general compositionality result. Further, we propose a type system that supports enforcement of security via fine-grained security types.

1.1 Introduction

Motivation  Is program $\text{in}_H(x); \text{out}_L(1)$ secure? This program receives an input on a secret (high-confidentiality) channel $H$, stores it in variable $x$, and outputs constant 1 on a public (low-confidentiality) channel $L$. Upon observing the low output, the attacker can deduce that a high input has been received. Hence, the presence of high input is revealed (but not its value). This kind of leak is often undesirable. For example, whether or not any communication with a medical web site takes place in a given browser tab should not be revealed to any web sites that are opened in the other tabs of the browser. Further, this leak can be magnified in the presence of concurrency. Consider the following two programs:

$$\text{in}_{H_0}(x); \text{out}_L(0)$$

and

$$\text{in}_{H_1}(x); \text{out}_L(1)$$
where both $H_0$ and $H_1$ are high channels. Say these programs are run in parallel with the following thread:

$$\text{in}_{H}(x); \text{if } x \text{ then out}_{H_1}(1) \text{ else out}_{H_0}(1)$$

First, a secret value is received and stored in the variable $x$. Depending on its value, a message is sent on either $H_1$ or $H_0$. The parallel composition (where the scope of $H_0$ and $H_1$ is made internal to the thread pool) is obviously insecure. Indeed, the parallel composition leaks whether or not the secrets entered on the high channel $H$ is zero. The result of the leak is sent out on the low channel $L$. It is straightforward to turn this example into a whole-secret leak by wrapping the above threads into loops. The last thread can then walk through the bits of a secret and the first two threads output the bits on the public channel. (A similar example can be constructed with a single high channel.)

$$\text{while } 1 \text{ do (in}_{H_0}(x); \text{out}_{L}(0))$$
$$|| \text{while } 1 \text{ do (in}_{H_1}(x); \text{out}_{L}(1))$$
$$|| \text{in}_{H}(h);$$
$$\text{for } b \text{ in bits}(h) \text{ do }$$
$$\text{if } b \text{ then out}_{H_1}(1) \text{ else out}_{H_0}(1)$$

As simple as this, the example points out a gap in the research on security for interactive programs. Clearly, when the presence of messages is secret, the program $\text{in}_{H}(x); \text{out}_{L}(1)$ is leaky. Surprisingly, the state-of-the-art in security for interactive programs [24, 9] imposes the assumption of totality of the environment, prescribing that the environment must always be ready to feed new inputs into programs. (Note that totality of environments is not to be confused with the notion of input totality [21], which requires that a system in any state can accept an input.)

On the other side of the spectrum is work that distinguishes between the security of message presence and content [29, 2, 26]. Such a distinction is important for modeling encryption, where the observation of a ciphertext reveals message presence but not its original content. However, an unaddressed limitation of this work is that it does not model full interaction with users or programs, but simply assumes all input is precomputed and provided by streams of values. In addition, there is also work that neither models strategies nor protects the presence of secret input [2, 4, 6].

We propose a framework that bridges this gap and generalizes previous research along two dimensions: breaking away from the totality of the environment and featuring fine-grained security types for communication channels. The rest of this section provides background on the security of interactive programs, positions our contributions, and overviews the results contained in the paper.
Background We study reactive programs that are capable of consuming input from the environment, perform internal computation, and produce output to the environment.

Despite a body of work on interaction in process calculi [12, 28, 14, 27, 15, 25, 17] and event-based systems [18, 19, 29], relatively little has been done on tracking the flow of information through language constructs in interactive languages. The state of the art in the area of security for interactive programs is primarily the work by O’Neill et al. [24] and Clark and Hunt [9]. O’Neill et al. argue for the need of direct reasoning on the security of interactive programs, to complement the body of work on the security of interactive systems in general. Inspired by Wittbold and Johnson’s nondeducibility on strategies [33], O’Neill et al. investigate the security of interactive programs in the presence of user strategies. A key example that shows intricacies of reasoning about interactive programs originates from Wittbold and Johnson:

\[
\text{while } l \text{ do (}
\begin{align*}
x &:= 0 \parallel x := 1; \\
in_H(x); \\
\text{out}_H(x); \\
in_H(y); \\
\text{out}_L(x \oplus y)
\end{align*}
\]

where \(\parallel\) is nondeterministic choice. Assume the program operates on binary values. Given an observation of low output, any high input is consistent on the high channel. However, the environment can make the program propagate a secret value \(z\) to the low channel. All the environment needs to do is to take the output (value of \(x\)) and xor it with \(z\) and provide \(x \oplus z\) as input. This example motivates the need to reason about security of programs in the presence of strategies [33]. Clark and Hunt [9] focus on reducing the security for interactive programs to the security of programs that operate on streams of inputs (without feedback). They prove that it makes no difference in a deterministic setting whether the environment is represented by strategies or streams.

Let us draw closer attention to the interactive setting of the previous work [24, 9]. Programs interact with strategies using communication channels. The strategies are functions that, given a trace that is observed on a certain channel (or generally, a set of channels at a given observation level), produce a value that serves as the next input for the program. A critical assumption in this work is the totality of the environment, which demands that strategies must always be able to produce new inputs: there is no way for the environment to block the program by not supplying an input, as demonstrated earlier.

We argue that the assumption of totality limits the space of possible attacks in an undesirable way. Recall the program \(\text{in}_H(x); \text{out}_L(1)\). This program is considered secure in [24, 9]. However, when the environment has the possibility of providing or not providing an input on the secret channel, the program is clearly insecure. The output on the low channel leaks the (one-bit) information
about whether or not an input is provided on the high channel.

Further, totality hinders the compositionality of security definitions. Recall the example with the three threads that opens this section. Similarly to the initial program, these programs are considered secure. Yet their parallel composition leaks high information on the low channel.

Consider a possible modification of the previous models to mimic nontotal environments by providing a special “no further input available” value. With such a modification, program $\text{in}_H(x); \text{out}_L(1)$ can be ruled out as insecure, provided that the input construct adequately treats the “no further input available” value by blocking or crashing. We choose to explicitly model the possibility for the environment to block the program, so that we do not need to modify the semantics for input.

**Contributions** This paper presents a generalized framework for security via strategies, where the totality assumption is dropped: the framework includes both total and nontotal strategies. Further, we parametrize our policies in the level of message presence. In other words, we distinguish between the security level of message presence (existence) and message content.

We illustrate that our generalized framework does not break the relation to deterministic strategies and streams from the work by Clark and Hunt for total strategies. Indeed, we are able to “replay” the results by Clark and Hunt (summarized below).

We show that the generalized framework features appealing compositionality properties: parallel composition of secure program results in a secure thread pool. The power of strategies gives us strong compositionality: we illustrate that various types of composition (with and without scoping) follow from our general compositionality result.

Finally, we provide a type system to illustrate how security can be enforced for our framework. The fine-grained types distinguish between the levels of message presence and content and provide elegant rules for typing the parallel composition.

**Overview** Our results, combined with the results from previous work, form the following big picture, displayed by the diagram in Figure 1.1. The leftmost column in the diagram comes from Clark and Hunt’s work [9]. The diagram relates sets of programs that correspond to the security conditions and the type system. As we walk through the diagram, we will informally introduce the notation for the sets. This notation is formalized later in the paper.

Our main security condition is strategy-based noninterference. The set of secure programs according to this condition is $\text{Strat-NI}$. The first series of results (presented in Section 1.3) positions our condition with respect to other strategy-based definitions. We show that it makes no difference for the security of a program whether the definition only considers deterministic strategies (which corresponds to the set $\text{DS-NI}$). This generalizes the result for total strategies [9]. We
1.2 Framework

As outlined above, our aim is to secure information flows in interactive programs. We address this issue in an adaption of a standard framework for interactive programs [24, 9, 6, 26]. Here, information can only enter and exit programs through channel-based message passing. Each channel comes with a label expressing the confidentiality level of the information it carries. We then ensure that confidential information in inputs does not influence which public inputs and outputs the program can perform.
1.2.1 Interactive Programs

Inputs $i$, outputs $o$, and messages $a$, last of which we also refer to as (inter)actions, are given by

$$i ::= ?a \cdot v \quad o ::= !a \cdot v \quad \tau$$

where $?a \cdot v$ (resp. $!a \cdot v$) denotes a message received (resp. sent) on channel $a$ carrying value $v$, and $\tau$ denotes a computation step other than an interaction with an environment. Here, $a$ and $v$ respectively range over the (unspecified nonempty) sets $A$ and $V$. Channels are the only external interface to our systems, and are therefore the only medium by which information can enter and exit our systems.

Our model of computation is a labeled transition system (LTS). An LTS is a triple $(S, A, \{ \triangleleft | a \in A \})$, where $S$ is a set of states, $A$ a set of actions, and for all $a \in A$, $\triangleleft \subseteq S \times S$. We write $s \triangleleft s'$ for some $s'$. The behavior of an interactive program can be given as an LTS as follows.

(1.1) **Definition**  An input-output LTS (IOLTS) is a LTS, with $A$ ranged by $a$, which is input-neutral, that is for all $s \in S$, if $\exists v \cdot s \xrightarrow{?a \cdot v}$, then $\forall v \cdot s \xrightarrow{?a \cdot v}$.

Intuitively, if $s$ is an IOLTS state which can, as its next computation step, input $i$ and enter state $s'$, then $s \triangleleft s'$. Likewise, $s \triangleleft s'$ if $s$ can output $o$ and enter $s'$. Practical computation models native to this paradigm include event loops and programs written in Erlang and JavaScript. Here, states correspond to program configurations, that is, code (and possibly a code pointer) paired with its environment, and actions express an interaction of a running program with its context. Section 1.6 demonstrates how to give the semantics of programs written in a simple nondeterministic imperative programming language as an IOLTS.

(1.2) **Definition**  An IOLTS is deterministic iff

1) If $s \xrightarrow{a \cdot v_1}, s_1, s \xrightarrow{a \cdot v_2}, s_2$ and $a_1 \neq a_2$, then $a_1 = ?a \cdot v_1$ and $a_2 = ?a \cdot v_2$ for some $\alpha, v_1$ and $v_2$.

2) If $s \xrightarrow{a}, s_1, s \xrightarrow{a}, s_2$, then $s_1 = s_2$.

Pt. 1) says that if $s \xrightarrow{?a \cdot v}$, then $s \triangleleft \alpha \cdot v$ iff $\alpha \in \{ ?a \cdot v | v \in V \}$, and implicitly, if $s \xrightarrow{?a}$, then $s \triangleleft \alpha$ iff $a = 0$. Pt. 2) says that $s$ has no internal nondeterminism.

Let $Tr$ denote the set $a^*$ of traces, ranged by $t$. We write $s \xrightarrow{t} s'$ when we have $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} s_n$ for some $t = a_1 \cdots a_n$ and $s_1, \ldots, s_n$ with $s_n = s'$. We let $t \downarrow, t \uparrow$ and $t \uparrow_\alpha$ denote the inputs, outputs and $\alpha$-messages, in $t$, respectively. That is, if $t = ?o \cdot 0.\alpha' \cdot ?a \cdot 2.\alpha 3$, then $t \downarrow = ?o \cdot 0.\alpha' \cdot 2, t \uparrow = !a \cdot 1.\alpha 3$ and $t \uparrow_\alpha = ?o \cdot 0.\alpha 3$. We write $t \leq t''$ when, for some $t', t'' = t.t'$.

1.2.2 Observables

The observables of an interactive program are its inputs and outputs. Whether a message on a channel is observable or not is indicated by the security levels
associated with the channel. We assume a lattice of security levels \((L, \sqsubseteq)\), with \(L\) ranged by \(l\), expressing levels of confidentiality. In our examples, \(L = \{L, H\}\) and \(\sqsubseteq = \{(L, L), (L, H), (H, H)\}\), \(H\) for “high” and \(L\) for “low” confidentiality. The channel-to-levels labeling is denoted \(\gamma : \mathbb{C} \to \sqsubseteq\). Here, if \(\gamma(\alpha) = l_1^{l_2}\), with \(l_2\) abbreviating \((l_2, l_1)\), then \(l_1\) is the confidentiality level of values (content) passed on \(\alpha\), and \(l_2\) the confidentiality level of the presence of a message on \(\alpha\). In examples, we will frequently represent a channel by its security label. We abbreviate \(L^L, H^L\) and \(H^H\) by \(L, M\) and \(H\), respectively. No observer can see \(\tau\) actions.

The security labels express who can observe what. An observer is associated a security level \(l\), indicating that the observer is capable of observing values on \(\alpha\) if \(l_1 \sqsubseteq l\), and the presence of messages on \(\alpha\) if \(l_2 \sqsubseteq l\), where \(\gamma(\alpha) = l_1^{l_2}\). We denote the \(l\)-observables in \(t\) by \(t \downarrow l\). For example, for \(t = ?L0::H1::M2::L3\), \(t \downarrow L1 = ?L0::M \Box ::L3\). Here, \(\Box \notin \forall\) is a “blank”, representing an unobservable value.

(1.3) **Definition**  
\(t_1\) and \(t_2\) are \(l\)-equivalent, written \(t_1 \equiv_l t_2\), iff \(t_1 \downarrow l = t_2 \downarrow l\).

So, \(l\)-equivalent traces are observably equivalent to an \(l\)-observer. We write \(t \leq_l t''\) if \(t =_l t'\) for some \(t' \leq t''\).

1.2.3 Strategies

The inputs to our systems come from the environment in which the system runs. The environment might vary the input on a channel \(\alpha\) depending on which interaction trace \(t\) the environment has observed. Further, the environment might pick an input value nondeterministically, or not input any value at all. We model an environment as a mapping from interaction traces \(t\) and channels \(\alpha\) to the (possibly empty) set of values \(V\) from which the environment draws a value to input on request on \(\alpha\) after observing \(t\). To ensure that an observer can attribute observably different interaction traces to a leak in the interactive program, we place two restrictions on the environments we take into consideration in our framework. First, if the observer can observe values passed on \(\alpha\), then the environment must be defined the same way on \(\alpha\) for all observably equivalent traces. Second, if the observer can observe the presence of messages passed on \(\alpha\), then the environment must, for each set of observably equivalent traces, map all of them to (possibly different) values, or map none of them to a value\(^1\). We refer to these environments as strategies.

(1.4) **Definition**  
A strategy is a function \(\omega : \mathbb{C} \to Tr \to \mathcal{P}(V)\) such that, for all \(\alpha\); \(\gamma(\alpha) = l_1^{l_2}\), with \(\omega_\alpha\) denoting \(\omega(\alpha)\),

\[
\begin{align*}
t_1 \equiv_{l_1} t_2 & \implies \omega_\alpha(t_1) = \omega_\alpha(t_2) \\
t_1 \equiv_{l_2} t_2 & \implies \omega_\alpha(t_1) \neq \omega_\alpha(t_2).
\end{align*}
\]

\(^1\)Not considering strategies which, given an interaction trace \(t\) and an input request on channel \(\alpha\), either return a value, or no value, does not affect our security results (as our policy is possibilistic).
Here, \( \mathcal{P}(A) \) is the powerset of \( A \), and \( A \sim B \) is defined as \( A = \emptyset \iff B = \emptyset \), and satisfies the following property.

\[
A = B \implies A \sim B.
\] (1.3)

Let \( \text{Strat} \) denote the set of strategies. A strategy which always inputs on \( \alpha \) the number of past \( \alpha \)-outputs can be defined as \( \omega_\alpha(t) = |t \upharpoonright \alpha| \).

(1.5) **DEFINITION** \( \omega \) and \( \omega' \) are \( l \)-equivalent, written \( \omega =_l \omega' \), iff, for all \( \alpha \); \( \gamma(\alpha) = l_1^{l_2} \) and \( t \),

\[
l_1 \subseteq l \implies \omega_\alpha(t) = \omega'_\alpha(t)
\]
\[
l_2 \subseteq l \implies \omega_\alpha(t) \equiv \omega'_\alpha(t)
\]

It is instructive to look at strategies as restrictions on which traces are possible; \( t \) is consistent with \( \omega \), written \( \omega \models t \), iff for all \( ?\alpha v, t' \) and \( t'' \) for which \( t = t'.?\alpha v.t'' \), \( v \in \omega_\alpha(t') \). Running a system under a strategy thus constrains the traces which the system can perform, as system inputs must come from the strategy; \( s \) produces \( t \) under \( \omega \), written \( \omega \models t \), iff \( s \xrightarrow{\omega} t \).

### 1.2.4 Noninterference

Our security policy of interest is that of *possibilistic noninterference* from [9], which is a generalization of Definition 1 from [24]. The policy states that under observably equivalent strategies, drawn from \( W \subseteq \text{Strat} \), the respective sets of traces \( s \) produces under either of them are observably equivalent.

(1.6) **DEFINITION** \( s \) is \( W \)-noninterfering iff

\[
\forall l . \forall \omega_1, \omega_2 \in W . \omega_1 =_l \omega_2 \implies \\
\forall t_1 . \omega_1 \models s \xrightarrow{\omega_1} \implies \\
\exists t_2 . \omega_2 \models s \xrightarrow{\omega_2} \land t_1 =_l t_2.
\] (NI)

The larger \( W \) is, the larger the space of attacks a \( W \)-noninterfering \( s \) is protected from.

(1.7) **DEFINITION** A \( W \)-attack is a 4-tuple \( (l, \omega_1, \omega_2, t_1) \) where \( \omega_1, \omega_2 \in W, \omega_1 =_l \omega_2 \) and \( \omega_1 \models t_1 \). It is an attack on \( s \) iff

1) \( \omega_1 \models s \xrightarrow{\omega_1} \), and
2) \( \forall t_2 . \omega_2 \models s \xrightarrow{\omega_2} \implies t_2 \not=_l t_1. \)

It is easy to see that \( s \) is \( W \)-noninterfering iff there is no \( W \)-attack on \( s \). Let \( W\text{-NI} \) denote the set of \( W \)-noninterfering programs. We say \( s \) is noninterfering iff \( s \in \text{Strat-NI} \).

We get the following lemma from Definition 1.6, since a \( W_1 \)-attack on \( s \) is a \( W_2 \)-attack on \( s \), for any \( W_1 \subseteq W_2 \).
1.3 Generalized strategies for noninterference

We now study the interplay between interactive programs, channel labelings and strategies.

1.3.1 Total Strategies

First, we contrast Strat with the total strategies considered in [9], a subset of which is considered in [24].

(1.9) definition \( \omega \) is total iff \( \forall \alpha, t . \omega_\alpha(t) \neq \emptyset \).

Let \( W_T \) denote the set of total strategies in \( W \). The set of all total strategies is thus \( Strat_T \). As outlined in the introduction, programs which are protected against \( Strat_T \)-attacks may still have \( Strat \)-attacks.

(1.10) theorem \( Strat_NI \subsetneq Strat_T-NI \).

Proof. Lemma 1.8 gives \( Strat_NI \subsetneq Strat_T-NI \). We now show the existence of a program in \( Strat_T-NI \) which is not in \( Strat_NI \). Program \( \text{in}_H x ; \text{out}_L \emptyset \) is such a program. It is in \( Strat_T-NI \) since totality of strategies in \( Strat_T \) gives

\[
\forall \omega \in Strat_T . \exists v . \omega | = s \Downarrow \text{H}^v,
\]

and thus

\[
\forall \omega \in Strat_T . \exists v . \omega | = s \Downarrow \text{H}^v . \text{L}^0.
\]

So (NI) holds with \( l \) instantiated to \( L \). (NI) with \( l \) instantiated to \( H \) follows from \( \omega_1 =_H \omega_2 \implies \omega_1 = \omega_2 \). However, this program is not \( Strat_NI \). In particular, consider

\[
\omega_1, \alpha(t) = \begin{cases}
\{42\}, & \text{if } \alpha = H \\
\emptyset, & \text{otherwise}
\end{cases}
\omega_2, \alpha(t) = \emptyset
\]

Clearly, \( \omega_1 =_L \omega_2 \). However,

\[
\omega_1 | = s \Downarrow \text{H}^{42} . \text{L}^0,
\]

and

\[
\forall t . \omega_2 | = s \Downarrow \implies t = \epsilon \neq L \Downarrow \text{H}^{42} . \text{L}^0.
\]

Thus \( (L, \omega_1, \omega_2, \text{H}^{42} . \text{L}^0) \) is a Strat-attack on this program. This program is thus not in \( Strat_NI \). \( \square \)
It is worth noting at this point that if we consider the class of channel labelings where the presence and content of messages are labeled with the same security level,

\[ \text{img}(\gamma) = \{ l^I | l \in \mathcal{L} \}, \]

then by (1.3), the definition of strategies and \( l \)-equivalence becomes the same as the one given in [9] and [24]. Furthermore, \( \text{Strat}_T \)-noninterference becomes the same policy as the one given in Definition 8 in [9] and, for a subset \( \text{Strat}_N \) of \( \text{Strat}_T \) (called “narrow” strategies in [9]) and for deterministic programs, \( \text{Strat}_N \)-noninterference becomes the same policy as the one given in Definition 1 in [24]. Since \( \text{inh} \ x \); \( \text{out}_L \ 0 \) has no \( \text{Strat}_T \)-attacks and is deterministic, it is secure according to both these policies.

1.3.2 Deterministic Strategies

We have just witnessed the existence of a program which has no \( \text{Strat}_T \)-attacks, but which has \( \text{Strat} \)-attacks, meaning that there are interesting attacks in the space between \( \text{Strat}_T \)- and \( \text{Strat} \)-attacks. We now take a closer look at this space, characterizing a small subset of it as being of interest. We first consider deterministic strategies.

\[ (1.11) \textbf{DEFINITION } \omega \text{ is deterministic iff } \forall \alpha, t, |\omega_\alpha(t)| \leq 1. \]

Let \( \text{DS} \) denote the set of deterministic strategies. We sometimes write \( \omega_\alpha(t) = \perp \) instead of \( \omega_\alpha(t) = \emptyset \) when \( \omega \) is deterministic. [9] shows that when considering whether a \( s \) is protected against \( \text{Strat}_T \)-attacks or not, it is sufficient to consider \( \text{DS}_T \)-attacks; this is Theorem 1 therein.

\[ (1.12) \textbf{PROPOSITION ([9]) } \text{Strat}_T \text{-NI} = \text{DS}_T \text{-NI}. \]

It turns out that the same holds for strategies in general.

\[ (1.13) \textbf{THEOREM } \text{Strat-NI} = \text{DS-NI}. \]

\[ \text{Proof. } \text{Same as proof of Theorem 1 in [9], as totality is never invoked in the proof.} \]

This theorem rules out the need to take into consideration attacks which utilize nondeterminism to cause a leak. However, the theorem does not make clear which abilities the attacker \textbf{must} have in order to create a leak in interactive programs which are not in \( \text{Strat-NI} \). For instance, do some programs \textbf{need} an infinite supply of input, perhaps only on some channels and not others, to leak? Do we need to take into consideration strategies which “discriminate” against some interaction traces by, say, providing input only if an even number of \( L \)-outputs has occurred prior?

The answer to both of these questions is no. We only need to consider those DS-attacks \((l, \omega_1, \omega_2, t_1)\) where \( \omega_1_\alpha \) and \( \omega_2_\alpha \) always feed \( \alpha \)-input on request, as long as the interaction trace has fewer \( \alpha \)-inputs than \( t_1 \) has. Furthermore, \( \omega_2 \) does
not need to feed input on channels with \( \not \in l \) presence at all. The proof of this can be found in the appendix.

(1.14) \textbf{LEMMA} If \((l, \omega_1, \omega_2, t_1)\) is a Strat-attack on \(s\), then for some \(\omega'_1\) and \(\omega'_2\), where for all \(t\) and \(\alpha\); \(\gamma(\alpha) = l_1^{l_2}\),

\[
(|t|_a| \prec |t_1|_a| \iff \omega'_1(\alpha) \neq \emptyset),
\]

\[
l_2 \subseteq l \implies (|t|_a| \prec |t_1|_a| \iff \omega'_2(\alpha) \neq \emptyset), \text{ and}
\]

\[
l_2 \not\subseteq l \implies \omega'_2(\alpha) = \emptyset,
\]

\((l, \omega'_1, \omega'_2, t_1)\) is a DS-attack on \(s\).

1.3.3 Public Presence Labels

The presence of an input on a channel with a secret presence label has a significant impact on the space of attacks to be considered to determine whether an interactive program is noninterfering or not. Consider for instance the class of channel labelings where message presence is always public. We refer to these \(\gamma\) as \(lp\) labelings.

\[
\text{img}(\gamma) = \{l^l | l \in L\}.
\]

It turns out that when a program is labeled with such a labeling, then it is sufficient to consider only Strat\(\gamma\)-attacks to determine whether an interactive program is in Strat-NI or not. Before proving this claim, we establish two simple, but useful, lemmas. The first lemma states that when \(\omega_1 = l \omega_2\), then for any \(\alpha\) with \(l\)-observable presence, \(\omega_{1\alpha}\) and \(\omega_{2\alpha}\) will be (un)defined on exactly the same traces.

(1.15) \textbf{LEMMA} If \(\omega_1 = l \omega_2\), then for all \(t\) and \(\alpha\); \(\gamma(\alpha) = l_1^{l_2}\), if \(l_2 \subseteq l\), then

\[
\omega_{1\alpha}(t) = \emptyset \iff \omega_{2\alpha}(t) = \emptyset.
\]

\textbf{Proof.} Follows from (1.3) and Definition 1.5. \(\square\)

It is never the case that an attack works as a consequence of one strategy supplying input on a channel with observable presence, and the other strategy not doing so (on an observably equivalent trace).

(1.16) \textbf{LEMMA} For all \(s\), Strat-attacks \((l, \omega_1, \omega_2, t_1)\) on \(s\), \(\alpha\); \(\gamma(\alpha) = l_1^{l_2}\) and \(t\), if \(l_2 \subseteq l\), then

\[
\forall v. \omega_2 \models s \overset{t}{\rightarrow} s' \wedge \omega_2(\alpha)_v(g(t) \models \emptyset) \implies t.\alpha v \not\in_l t_1
\]

\textbf{Proof.} Let \(s\), and Strat-attack \((l, \omega_1, \omega_2, t_1)\) on \(s\), be given. By Definition 1.7 Pt. 1), we have \(\forall t, \alpha, v,\)

\[
t.\alpha v \leq t_1 \implies v \in \omega_{1\alpha}(t) \neq \emptyset.
\]
By Definition 1.4 we get \( \forall t, t', \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \sqsubseteq l \),
\[
t =_{l_2} t' \implies (\omega_{j_a}(t) = \emptyset \iff \omega_{j_a}(t') = \emptyset).
\]

Since \( \omega_1 =_l \omega_2 \), Lemma 1.15 yields \( \forall t, \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \sqsubseteq l, v \),
\[
\omega_1(t) = \emptyset \iff \omega_2(t) = \emptyset.
\]

Together, this gives \( \forall t, \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \sqsubseteq l, v \),
\[
t ? \alpha \lor \leq_l t_1 \implies \omega_2(t) \neq \emptyset.
\]

By contraposition, we get \( \forall t, \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \sqsubseteq l, v \),
\[
\omega_2(t) = \emptyset \implies t ? \alpha \lor \leq_l t_1.
\]

(1.4) follows by specializing the premise of the implication. \( \square \)

We are now ready to prove the above-stated claim.

(1.17) THEOREM Strat-NI = Strat_T-NI for lp \( \gamma \).

Proof. We prove DS-NI = DS_T-NI; the result will then follow from Theorem 1.13 and Proposition 1.12. DS_T \( \subseteq \) DS by definition of DS_T. By Lemma 1.8, DS-NI \( \subseteq \) DS_T-NI. We now show DS-NI \( \supseteq \) DS_T-NI, i.e.,
\[
\forall s . s \in DS_T-NI \implies s \in DS-NI.
\]

We show instead the contrapositive. That is,
\[
\forall s . s \notin DS-NI \implies s \notin DS_T-NI. \tag{1.5}
\]

Let \( s \notin DS-NI \) be given. Then \( s \) has some DS-attack \((l, \omega_1, \omega_2, t_1)\), that is, for some \( l \), deterministic \( \omega_1 \) and \( \omega_2 \), and \( t_1 \),

1. \( \omega_1 =_l \omega_2 \),
2. \( \omega_1 \models s \overset{t_1}{\rightarrow} \),
3. \( \forall t_2 . \omega_2 \models s \overset{t_2}{\rightarrow} \implies t_2 \neq_{l} t_1 \).

By Lemmas 1.15 and 1.16, and by lp, we get \( \forall t, \alpha \),
\[
\omega_{1_\alpha}(t) = \bot \iff \omega_{2_\alpha}(t) = \bot \tag{1.6}
\]
and \( \forall t, \alpha, v \),
\[
\omega_2 \models s \overset{\cdot}{\rightarrow} s' \land s' \overset{\cdot \alpha \lor v}{\rightarrow} \land \omega_{2_\alpha}(t) = \bot \implies t ? \alpha \lor \lor \leq_l t_1, \forall v. \tag{1.7}
\]
In particular, this holds if we fix $v$ to a constant $k$. Let

$$\omega'_{ja}(t) = \begin{cases} k, & \text{if } \omega_{ja}(t) = \bot \\ \omega_{ja}(t), & \text{otherwise.} \end{cases}$$

We show that $(l, \omega'_1, \omega'_2, t_1)$ is a $\text{DS}_{\text{T}}$-attack on $s$, that is,

i) $\omega'_1 = l \omega'_2$

ii) $\omega'_j$ is a deterministic total strategy,

iii) $\omega'_1 \models s \xrightarrow{t_1}$

iv) $\forall t_2. \omega'_2 \models s \xrightarrow{t_2} t_2 \neq l t_1.$

It is easy to see that $\forall t, \alpha,$

$$\omega_{ja}(t) = \omega_{na}(t) = \bot \implies \omega'_{ja}(t) = \omega'_{na}(t)$$

This, 1) and (1.6) gives i). For $\alpha$ for which $\gamma(\alpha) = l t_1^2$, since

$$t = l t' \implies \omega_{ja}(t) = \bot \iff \omega_{ja}(t') = \bot$$

and

$$t = l t' \implies t = l t',$$

then either $\omega'_{ja}(t) = \omega'_{ja}(t') = k$ or $\omega'_{ja}(t) = \omega_{ja}(t)$ and $\omega'_{ja}(t') = \omega_{ja}(t')$. Thus, since $\omega_j$ is deterministic, then by definition of $\omega'_j$, $\omega'_{ja}$ is total. Since $\omega_j$ is deterministic, then by definition of $\omega'_j$, so is $\omega'_j$. So ii) holds. It is easy to see that $\forall t, \alpha,$

$$\omega_{ja}(t) \neq \bot \implies \omega_{ja}(t) = \omega'_{ja}(t)$$

This, and 1), gives iii). By (1.7), iv) holds.

Thus $s \notin \text{DS}_{\text{T}}$-NI. Since $s$ was arbitrary, (1.5) holds.

As a consequence, programs such as $\text{in}_M x ; \text{out}_L 0$, which we already know is $\text{Strat}_{\text{T}}$-noninterfering, are thus Strat-noninterfering, since all interaction occurs on public presence channels, and now flow of message content occurs.

### 1.4 Relation to streams for noninterference

As we saw in (1.2), in the presence of nondeterminism, to succeed, an attack $(l, \omega_1, \omega_2, t_1)$ sometimes needs to adapt to observed nondeterministic choices, to then force the program down different control flow paths, to then make a trace $t_1$ producible under $\omega_1$ not matchable under $\omega_2$. However, as demonstrated in [9] Theorem 2, in the Strat setting, if the program in question is deterministic (as many programs are), there are no nondeterministic choices made by the program.
for an attack to cleverly adapt to. The space of attacks we need to consider is thus much simpler; it suffices to consider strategies which can be expressed as a (possibly infinite) list of messages on each input channel, independent on interaction traces. These are stream strategies.

\[ \Omega \text{ is a stream strategy iff it is deterministic and for all } \alpha, |t_1 \uparrow \alpha| = |t_2 \uparrow \alpha| \Rightarrow \omega_\alpha(t_1) = \omega_\alpha(t_2). \]

Let SS denote the set of stream strategies in W.

**Proposition ([9])** Strat\(_T\)-NI = SS\(_T\)-NI for deterministic s.

We prove this result in the Strat setting. First we establish an insightful lemma; it says that, when a deterministic program is run under a deterministic strategy, then it produces a unique (possibly infinite) sequence of interactions, of which all interaction traces are prefixes. This is Lemma 4 in [9].

**Lemma ([9])** If s and \( \omega \) are deterministic, then if \( \omega \models s \downarrow \), and \( \omega \models s \downarrow \alpha \), then \( t_1 \leq t_2 \) or \( t_2 \leq t_1 \).

Proof. Same as proof of Lemma 4 in [9], as totality is never invoked in the proof. \( \square \)

We now prove the above-stated result in the Strat setting. The proof borrows the idea from [9] of “streamifying” deterministic strategies on the set of traces on which they are defined. Lemma 1.20 then gives us that this change does not affect the set of traces a deterministic program produces under the modified strategies.

**Theorem** Strat-NI = SS-NI for deterministic s.

Proof. We prove DS-NI = SS-NI; the result will then follow from Theorem 1.13. SS \( \subseteq \) DS by definition of SS. By Lemma 1.8, DS-NI \( \subseteq \) SS-NI. We show DS-NI \( \supseteq \) SS-NI. That is,

\[ \forall s. s \in \text{SS-NI} \Rightarrow s \in \text{DS-NI}. \]

We show instead the contrapositive. That is,

\[ \forall s. s \notin \text{DS-NI} \Rightarrow s \notin \text{SS-NI}. \]

Let \( s \notin \text{DS-NI} \) be given. Then there is a DS-attack \((l, \omega_1, \omega_2, t_1)\) on s, that is,

1. \( \omega_1 =_l \omega_2 \),
2. \( \omega_1 \models s \downarrow \),
3. \( \forall t_2. \omega_2 \models s \downarrow \alpha \Rightarrow t_2 \neq t_1 \).
and, by Lemma 1.14,

$$|t \upharpoonright \alpha|? < |t_1 \upharpoonright \alpha|? \iff \omega_{t_a}(t) \neq \emptyset,$$

$$l_2 \subseteq l \implies (|t \upharpoonright \alpha|? < |t_1 \upharpoonright \alpha|? \iff \omega_{t_a}(t) \neq \emptyset) \tag{1.8}$$

Let

$$\omega'_{j_a}(t) = \{ v \mid \omega_j \models s \xrightarrow{t',\alpha v} \text{ for some } t' \text{ with } |t \upharpoonright \alpha|? = |t' \upharpoonright \alpha|? \}$$

We must show that

i) \( \omega'_1 = \omega'_2 \),

ii) \( \omega' \) is a strategy,

iii) \( \omega'_1 \models s \downarrow\alpha_l \),

iv) \( \omega'_2 \models s \downarrow\alpha_l \implies t_1 \neq t_2, \forall t_2 \).

For \( \alpha = l_1^2 \), since

$$t = l_2 t' \implies |t \upharpoonright \alpha|? = |t' \upharpoonright \alpha|?$$

and

$$t = l_1 t' \implies t = l_2 t',$$

then either

$$\omega_{j_a}(t) = \omega_{j_a}(t') = \omega'_{j_a}(t) = \omega'_{j_a}(t') = \bot$$

or

$$\omega'_{t_a}(t) = \omega'_{t_a}(t) = V$$

for some set \( V \neq \emptyset \) of values. Thus, since \( \omega_j \) are strategies, ii) holds.

Before proceeding with proving i), we show that \( \omega'_j \) is deterministic, and at the same time a stream strategy. We first show \( \omega'_j \) is deterministic. Assume the contrary. Then there are some \( t'_1, t'_2, v_1, v_2 \) such that

a) \( \omega'_j \models s \xrightarrow{t'_1,\alpha v_1}, \omega'_j \models s \xrightarrow{t'_2,\alpha v_2} \),

b) \( |t'_1 \upharpoonright \alpha|? = |t'_2 \upharpoonright \alpha|? \), and

c) \( v_1 \neq v_2 \).

Lemma 1.20 gives \( t'_1,\alpha v_1 \leq t'_2,\alpha v_2 \) or \( t'_2,\alpha v_2 \leq t'_1,\alpha v_1 \). Assume wlg. that \( t'_1,\alpha v_1 \leq t'_2,\alpha v_2 \). Two cases to consider.

\( t'_1,\alpha v_1 < t'_2,\alpha v_2 \): Then \( t'_1,\alpha v_1 \leq t'_2 \). But

$$|t'_1,\alpha v_1 \upharpoonright \alpha|? = |t'_1 \upharpoonright \alpha|? + 1 > |t'_1 \upharpoonright \alpha|? = |t'_2 \upharpoonright \alpha|?,$$
contradicting b).

\[ t_1' \Box \alpha v_1 = t_2' \Box \alpha v_2: \quad \text{Then } t_1' = t_2'. \quad \text{But now Definition 1.2 Pt. 1) gives } v_1 = v_2, \]

contradicting c).

So \( \omega_j' \) is deterministic. By definition, \( \omega_j' \) is also a stream strategy. By (1.8),

\[
|t \downarrow_{\alpha}| = |t_1 \downarrow_{\alpha}| \quad \iff \quad \omega_{\alpha_1}'(t) \neq \emptyset
\]

\[
\omega_{\alpha_2}'(t) \neq \emptyset \quad (1.9)
\]

\[
l_2 \subseteq l \quad \implies \quad \omega_{\alpha_1}'(t) \neq \emptyset
\]

\[
l_2 \not\subseteq l \quad \implies \quad \omega_{\alpha_2}'(t) = \emptyset.
\]

We return to proving i). Let \( t \) be arbitrary. To show i), we must show that for all \( \alpha; \gamma(\alpha) = l_1' \) for which \( l_2 \subseteq l \),

\[
a') \; \omega_{\alpha}(t) = \omega_{\alpha}'(t),
\]

\[
b') \; l_1 \subseteq l \quad \implies \quad \omega_{\alpha}(t) = \omega_{\alpha}'(t).
\]

There are three cases to consider.

1') There exists no \( t' \) for which we have \( |t' \downarrow_{\alpha}| = |t \downarrow_{\alpha}| \), \( \omega_1 \models s \xrightarrow{t'.\alpha v_1}, \)

\( \omega_2 \models s \xrightarrow{t'.\alpha v_2} \), for any \( v_1, v_2 \). Then \( \omega_{\alpha_1}'(t) = \omega_{\alpha_2}'(t) = \bot \), satisfying a'),

and satisfying b') regardless of whether \( l_1 \subseteq l \) holds or not.

2') There exists a \( t' \) for which we have \( |t' \downarrow_{\alpha}| = |t \downarrow_{\alpha}| \), \( \omega_1 \models s \xrightarrow{t'.\alpha v_1}, \)

\( \omega_2 \models s \xrightarrow{t'.\alpha v_2} \), for some \( v_1 \) and all \( v_2 \). Then \( \omega_{\alpha_1}'(t) = \omega_{\alpha_2}'(t) = \bot \), contradicting 1), regardless of whether \( l_1 \subseteq l \) holds or not. So a') and b') hold. (the proof of the symmetric case is analogous, so we omit it wlg.)

3') There exists a \( t' \) for which we have \( |t' \downarrow_{\alpha}| = |t \downarrow_{\alpha}| \), \( \omega_1 \models s \xrightarrow{t'.\alpha v_1}, \)

\( \omega_2 \models s \xrightarrow{t'.\alpha v_2} \), for some \( v_1, v_2 \). Then \( \omega_{\alpha_1}'(t) = v_1 \neq \bot \) and \( \omega_{\alpha_2}'(t) = v_2 \neq \bot \), satisfying a'). Assume \( l_1 \subseteq l \). Then by 1), \( \omega_{\alpha_1}(t') = \omega_{\alpha_2}(t') = v \)

for some \( v \). Then \( v_1 = v \) and \( v_2 = v \), so \( \omega_{\alpha_1}'(t) = \omega_{\alpha_2}'(t) \).

So i) holds.

It remains to show iii) and iv). To do this, we show

\[
\omega_j \models s \xrightarrow{t} \iff \omega_j' \models s \xrightarrow{t} \quad (1.10)
\]

We proceed by induction in \( n = |t \downarrow_{\alpha}| \).

\( n = 0 \): Assume \( \omega_j \models s \xrightarrow{t} \). Then \( s \xrightarrow{t} \). Since \( \omega_j' \models t \) holds vacuously, \( \omega_j' \models s \xrightarrow{t} \)

holds. This proves the forward implication of (1.10). The proof for the reverse implication is analogous. Thus (1.10) holds.

\( n + 1 \), assuming \( n \): Assume (1.10) holds \( \forall t \) with \( |t \downarrow_{\alpha}| = n \). This is our induction hypothesis (IH). We prove (1.10) for \( t \) with \( |t \downarrow_{\alpha}| = n + 1 \). Let
1.5 Compositionality

A common scenario for interactive programs is when they interact with other interactive programs. It is therefore of key importance to ensure that the interaction of secure interactive programs does not create an information leak. As we saw in the introduction, a seemingly innocuous leak through message presence in one program can be magnified when that program is run in parallel with other interactive programs. We show that Strat-NI is compositional. That is, the parallel composition of noninterfering programs yields a noninterfering program. We thus guarantee the absence of high-bandwidth leaks through message presence in the concurrent setting.

In theory and practice, there are scenarios and primitives for almost any wirings of output channels to input channels, fixed or runtime-changing, scoped or unscoped. To stay as general as possible, we leave it to the environment to decide how to route messages. This yields the very simple semantics displayed in
Figure 1.2: Labeled Reduction Relation for Interactive Programs in Parallel

Figure 1.2. Here, any message outputted by resp. inputted to the parallel composition $s_1 \parallel s_2$ of $s_1$ and $s_2$ is produced resp. consumed by exactly one of the parallel components $s_1$ and $s_2$. A parallel composition of interactive programs is then itself an interactive program, and thus, all the results from Sections 1.2 through 1.4 apply to them. This "strategies as glue" approach has much appeal for our security purposes; since all plugging and routing is performed by strategies, and since strategies are general, we can, by picking the right strategy, model different types of composition. For instance, $(1.1)$, with $H_0$, $H_1$, $H$ and $L$ aliased $H_0$, $H_1$, $M$ and $L$, would in practice ideally scope channels $H_0$ and $H_1$ to be internal to the parallel composition, and wire output-$H_j$ to input-$H_j$, leaving input channel $M$ and output channel $L$ as the external interface. A strategy can achieve this wiring by implementing a buffer on $H_0$ and $H_1$. A strategy for $\alpha$ which implements a message buffer$^2$ for channel $\alpha$ is given below, defined using pattern-matching syntax similar to that of Haskell and ML.

$$\omega_{\alpha} (t ! \alpha v . t' ? \alpha v . t'') \mid (t \mid_\alpha = t' \mid_\alpha = \epsilon) = \omega_{\alpha} t'.t''$$

$$\omega_{\alpha} (t ! \alpha v . t') \mid (t \mid_\alpha = t' \mid_\alpha = \epsilon) = \nu$$

$$\omega_{\alpha} t = \bot$$

Using the above definition for $H_0$ and $H_1$, we obtain the desired wiring, where no externals influence the communication on $H_0$ and $H_1$ (the reason for scoping them).

Note that our compositionality result can only be used to reason about the security of programs which are running in parallel. Consider $s_A[s_B]$: a program $s_A$ which after some computation steps forks $s_B$ as a new thread. The behavior of $s_A[s_B]$ can be given as an IOLTS in terms of parallel composition. When the forking occurs, $q$ becomes a parallel component. Thus, a secure $s_A[s_B]$, in parallel with any IOLTS, yields a secure composition. However, $s_B$ is not a parallel component until $s_A[s_B]$ has forked $s_B$. Indeed, if $s_A[s_B] \not\to s_A' \parallel s_B$, then even if $s_A'$ and $s_B$, and thus $s_A' \parallel s_B$, are secure, then this does not imply the security of $s_A[s_B]$ as the occurrence of $t$ can leak information. We discuss how to track information flows in the presence of a forking construct in Section 1.6.

We now prove our compositionality result. The idea here is that given an attack on $s_A \parallel s_B$, we obtain an attack on either $s_A$ or $s_B$ by incorporating $s_A$ into the

$^2$Inputting from an empty buffer is impossible, and messages in the buffer are inputted in FIFO order.
attack environment to produce an attack on \(s_B\), and vice versa.

(1.23) **Theorem** For all \(s_A\) and \(s_B\),

\[
s_A, s_B \in \text{Strat-NI} \implies s_A \parallel s_B \in \text{Strat-NI}.
\]

**Proof.** We show the contrapositive. That is,

\[
s_A \parallel s_B \notin \text{Strat-NI} \implies s_A \notin \text{Strat-NI} \lor s_B \notin \text{Strat-NI}.
\]

Assume \(s_A \parallel s_B \notin \text{Strat-NI}\). \(s_A \parallel s_B \notin \text{DS-NI}\) by Theorem 1.13. Then there is a DS-attack \((l, \omega_1, \omega_2, t_1)\) on \(s_A \parallel s_B\). Particularly,

\[
\forall t_2, \omega_2 \models s_A \parallel s_B \xrightarrow{t_2} t_2 \neq t_1. \tag{1.11}
\]

By Lemma 1.14,

\[
\forall \alpha; \gamma(\alpha) = l_1^2 \cdot l_2 \not\subseteq l \implies \omega_2 = \emptyset.
\]

Assume (towards a contradiction) that \(s_A, s_B \in \text{Strat-NI}\). Then, by Definition 1.6, we have for \(k \in \{A, B\},\)

\[
\forall \omega_{1k}, \omega_{2k} \in \text{Strat}, \omega_{1k} =_l \omega_{2k} \\
\forall t_{1k} \cdot \omega_{1k} \models s_k \xrightarrow{t_{1k}} \\
\exists t_{2k} \cdot \omega_{2k} \models s_k \xrightarrow{t_{2k}} \land t_{1k} =_l t_{2k}. \tag{1.12}
\]

Pick \(t_{1A}\) and \(t_{1B}\) such that \(s_A \xrightarrow{t_{1A}} s_B \xrightarrow{t_{1B}}\) and \(t_{1A} \parallel t_{1B}\). Here, for any \(\hat{\ell}, \tilde{\ell}_1\) and \(\tilde{\ell}_2\) \(\hat{\ell} \parallel \tilde{\ell}_1 \parallel \tilde{\ell}_2\) iff \(\hat{\ell}\) is an interleaving of \(\tilde{\ell}_1\) and \(\tilde{\ell}_2\). Recall the assumption that \(\omega_1 \models s_A \parallel s_B\). We construct \(\omega_{1A}, \omega_{2A}, \omega_{1B}\) and \(\omega_{2B}\) s.t. \(\omega_{1A} =_l \omega_{2A}\) and \(\omega_{1B} =_l \omega_{2B}\) which, by (1.12), contradict (1.11). Let \(j \in \{1, 2\}, k, k' \in \{A, B\}, \)

\[
\omega_{jk_{\alpha}}(t) = \{v \mid \exists t'_{k'}, t'_{k}, t' = l_1, t'_{k} \land t'_{k} \parallel t'_{\alpha} \parallel t'_{\alpha} \parallel t'_{\alpha} ;
\]

\[
\land t'_{\alpha} \leq t, t'_{\alpha} \leq s_k \xrightarrow{t_{\alpha}}
\]

\[
\land t'_{\alpha} \leq l, t_{\alpha} \land s_k \xrightarrow{t_{\alpha}}
\]

We show that \(\omega_{jk}\) are strategies. Let \(t = l_1, t'\). Assume \(\omega_{jk_{\alpha}}(t) \neq \emptyset\). Then for some \(v, v \in \omega_{jk_{\alpha}}(t)\). Let \(t'_{k}, t'_{k}\) and \(t'_{k}\) be the evidence that \(v \in \omega_{jk_{\alpha}}(t)\) The only condition on \(t\) for \(v \in \omega_{jk_{\alpha}}(t)\) to hold is \(t = l_1, t'_{k}\). Since \(t = l_2, t'\), \(t = l_{2}, t'_{k}\) by transitivity. Thus \(t'_{k}, t'_{k}\) and \(t'_{k}\) are evidence of \(v \in \omega_{jk_{\alpha}}(t')\). So \(v \in \omega_{jk_{\alpha}}(t') \neq \emptyset\). \(\omega_{jk_{\alpha}}(t) = \omega_{jk_{\alpha}}(t')\). Let \(t = l_1, t'\). Since \(l_2 \leq l_1, t = l_2, t'\). Thus \(\omega_{jk_{\alpha}}(t) = \omega_{jk_{\alpha}}(t')\).

We show that \(\omega_{1k} =_l \omega_{2k}\). Let \(t\) and \(\alpha; \gamma(\alpha) = l_1^2\) be arbitrary. Case on \(l\).

\(l_1 \subseteq l\): Assume w.l.g. that \(v \in \omega_{1k}(t)\). \(v \in \omega_{2k}(t)\) must be shown. Define \(p_j\)
such that

\[
(\exists t'_1 . p_j(t, t'_1, \hat{v}) \land \omega_j \models s_k \parallel s_B^L_{2,3} v \\
\iff \hat{v} \in \omega_{jk_a}(t).
\]

Observe that \( p_1 = p_2 \). Since \( v \in \omega_{jk_a}(t) \), we get for some \( t'_1 \),

\[
p_1(t, t'_1, v) \land \omega_1 \models s_k \parallel s_B^L_{2,3} v.
\]

Since \( \omega_1 = \omega_2 \), we get \( \omega_2 \models s_k \parallel s_B^L_{2,3} v \). Since \( p_2 = p_1 \), we get \( p_2(t, t'_1, v) \). Thus \( v \in \omega_{2k_a}(t) \). Thus we have

\[
\forall v . v \in \omega_{1k_a}(t) \iff v \in \omega_{2k_a}(t).
\]

\( l_1 \not\subseteq l \) and \( l_2 \subseteq l \): Assume w.l.o.g. that \( v \in \omega_{1k_a}(t) \). We must show that \( v' \in \omega_{2k_a}(t) \). Define \( p_j \) such that

\[
(\exists t'_1, \alpha \in \omega_{jk_a}(t), (t', t'_1, t'_1) \land t'_1.2\alpha \hat{v} \leq_l t'_1k \land s_k \parallel s_B^L_{2,3} v \\
\iff \hat{v} \in \omega_{jk_a}(t).
\]

Observe that \( p_1 = p_2 \). Since \( v \in \omega_{jk_a}(t) \), we get for some \( t'_1 \) and \( t'_1k \),

\[
(p_1(t, t'_1, t'_1k) \land t'_1k.2\alpha v \leq_l t'_1k \land s_k \parallel s_B^L_{2,3} v \\
\iff \hat{v} \in \omega_{jk_a}(t).
\]

Since \( \omega_1 = \omega_2 \), we get \( \omega_2 \models s_k \parallel s_B^L_{2,3} v \). Since \( p_2 = p_1 \), we get \( p_2(t, t'_1, t'_1k) \). Thus \( v' \in \omega_{2k_a}(t) \). Thus we have

\[
(\exists v . v \in \omega_{1k_a}(t)) \iff (\exists v' . v' \in \omega_{2k_a}(t)).
\]

\( l_2 \not\subseteq l \): The condition on \( t \) and \( \alpha \) in Definition 1.5 is vacuously true in this case.

Since \( t \) and \( \alpha \) were arbitrary, we get \( \omega_{1k} = \omega_{2k} \).

We show that \( \omega_{1A} \models s_A^L_{1k} \). We already have \( s_A^L_{1k}, s_B^L_{1k}, t_{1A} || t_{1B} \) and \( \omega_1 \models s_A \parallel s_B \). We proceed by induction in \( n = |t_{1A}| \).

\( n = 0 \): Since \( s_A^L_{1k} \) and \( t_{1A} \) has no inputs, \( \omega_{1A} \models s_A^L_{1k} \) holds vacuously.

\( n + 1 \), given \( n \): Assume \( \omega_{1A} \models s_A^L_{1k} \) for \( t_{1A} \) with \( |t_{1A}| = n \); this is our induction hypothesis (IH). For some \( t''_{1A} \) with \( |t''_{1A}| = n + 1 \), we have \( t'_{1A} = t''_{1A} \land v \land t''_{1A} \). By (IH) we have \( \omega_{1A} \models s_A^L_{1k} \). By definition of \( t'_{1A} \), we
have for some $t'_1B$ and $t'_1$ for which $t'_1B \leq t_1B$, $t'_1 \leq t_1$, and $t'_1A \parallel t'_1B$ that $\omega_{1A} \vdash s_A \parallel s_B^{t'_1A/\omega_{1A}}$. By $s_A^{t_1A}$ and $s_B^{t_1B}$ we get $s_A^{t_1A}$ and $s_B^{t_1B}$. Since $s \leq s_i$, we get by definition of $\omega_{1A}$ that $v \in \omega_{1A}(t'_1A)$. Thus $\omega_{1A} \vdash s_A^{t'_1A/\omega_{1A}}$. Since $|t''_1|_\parallel = 0$ and $s_A^{t_1A}$, we get $\omega_{1A} \vdash s_A^{t''_1}$.

Likewise (swap As and Bs), $\omega_{1B} \vdash s_B^{t''_1}$.

Assume that $\omega_{2A} \vdash s_A^{t''_1}$ and $\omega_{2B} \vdash s_B^{t''_1}$, such that $t_{1A} = t_{2A}$ and $t_{1B} = t_{2B}$. We show that there then is a $t_2$ for which we have $t_2 = t_1$ and $\omega_2 \vdash s_A \parallel s_B^{t''_1}$, contradicting (1.11). We consider the interesting case where $|t_{2A}|_{\parallel > 0}$ and $|t_{2B}|_{\parallel > 0}$ (uninteresting case: if $|t_{2A}|_\parallel = 0$, then $s_A \parallel s_B \not\in DS$ (DS-NI)). Let $t_{2A} = t_{2A}^{i_A} \cdot t_{2B}$ and $t_{2B} = t_{2B}^{i_B} \cdot t_{2B}''$ such that $|t_{2A}|_{\parallel} = 0$ and $|t_{2B}'_1|_{\parallel} = 0$. Let $i_A = \alpha_A \cdot v_A$ and $i_B = \alpha_B \cdot v_B$. By definition of $(\omega_{2A})_{\alpha_A}$ and $(\omega_{2B})_{\alpha_B}$, we have that $v_A \in (\omega_{2A})_{\alpha_A}(t_{2A}^{i_A})$ and that $v_B \in (\omega_{2B})_{\alpha_A}(t_{2B}^{i_B})$. Thus for some $t_A^{i_A}$ and $t_B^{i_B}$ for which $t_A^{i_A} \parallel i_A \leq t_{1A}$ and $t_B^{i_B} \parallel i_B \leq t_{1B}$, we have $\omega_{2A} \vdash s_A \parallel s_B^{t_A^{i_A}}$ and $\omega_{2B} \vdash s_B^{t_B^{i_B}}$. Since $\omega_{2A}^{t_A^{i_A}} = 0$ for all $\alpha'_A = 0^{t_A^{i_A}}$ with $t_A^{i_A} \not\in t_{1A}$, $t_A^{i_A} \parallel i_A = t_{1A}^{i_A}$ and $t_B^{i_B} \parallel i_B = t_{1B}^{i_B}$, and then either $t_A^{i_A} \parallel i_A \leq t_{1A}$, then $t_B^{i_B} \parallel i_B \leq t_{1B}$, or $t_B^{i_B} \parallel i_B \leq t_{1A}$, then either $t_A^{i_A} \parallel i_A \leq t_{1A}$, then $t_B^{i_B} \parallel i_B \leq t_{1B}$, or $t_B^{i_B} \parallel i_B \leq t_{1A}$, then $t_A^{i_A} \parallel i_A \leq t_{1B}$, then either $t_B^{i_B} \parallel i_B \leq t_{1A}$, or $t_B^{i_B} \parallel i_B \leq t_{1B}$. Assume $t_A^{i_A} \parallel i_A \leq t_{1A}$, then $t_B^{i_B} \parallel i_B \leq t_{1B}$ wlg.

Then by definition of $t_A^{i_A}$ and $t_B^{i_B}$, $t_A^{i_A} \parallel i_A \leq t_{1A}$ (since $\parallel i_A$ is $l$-equivalent with the last observable input in $t_1$). Thus $t_1 = t_A^{i_A}$ for some $t_A^{i_A}$ and $t_B^{i_B}$ for which $t_A^{i_A} \parallel i_A \leq t_{1A}$ and $|t_A^{i_A} |_{\parallel i_A} = 0$. Now, there is some $t''_1$ for which $t''_1 \parallel i''_1 = t''_1A$. For any such $t''_1$, since $|t''_1|_{\parallel} = 0$ and, as established before, $\omega_2 \vdash s_A \parallel s_B^{t''_1}$, $\omega_2 \vdash s_A \parallel s_B^{t''_1}$. But $t_A^{i_A} \parallel i''_1 \parallel i''_1A = t_1A$, contradicting (1.11).

Thus, either $s_A \not\in Strat-NI$ or $s_B \not\in Strat-NI$.

It is worth noting that the compositionality result does not condition on whether the interactive programs are deterministic or not. This means that programmers can write deterministic programs, establish that the programs are secure using an enforcement mechanism for deterministic programs, freely compose the programs, and obtain a guarantee that the composition is secure, even though || introduces nondeterministic behavior.

1.6 Enforcement

This section presents a small nondeterministic imperative programming language with input and output primitives, and develops a type system that differentiates between the security of the presence of messages and the security of their content. We show that the type system enforces DS-NI. In addition, we show how the language (and type system) can be safely extended with top-level parallelism, and establish soundness of the extension. Finally, we illustrate how to track flows in the presence of a fork command.
1.6.1 Syntax

Assume a standard expression language ranged over by $e$. The commands form a while language, extended with nondeterministic choice ($c_1 \parallel c_2$), input ($\text{in}_\alpha(x)$), and output ($\text{out}_\alpha(e)$).

$$c ::= x := e | \text{in}_\alpha(x) | \text{out}_\alpha(e) | c_1; c_2 | c_1 \parallel c_2 | \text{if } e \text{ then } c_1 \text{ else } c_2 | \text{while } e \text{ do } c$$

1.6.2 Semantics

Let $\mathbb{V}$ be any set containing 0, and let $\sigma$ range over variable environments, i.e., maps from variables to values. Following [24] let $\sigma(e)$ denote the evaluation of $e$ in $\sigma$. The configurations are pairs of variable environments and commands. The semantics of commands is of the form $\langle \sigma_1, c_1 \rangle \xrightarrow{a} \langle \sigma_2, c_2 \rangle$, read the configuration $\langle \sigma_1, c_1 \rangle$ evaluates in one step to the configuration $\langle \sigma_2, c_2 \rangle$ with action $a$.

The rules of the semantics can be found in Table 1.1. Most rules are entirely standard. Nondeterminism and input is modeled by underspecification. In the former case, either of the rules $\text{nd1}$ and $\text{nd2}$ can be chosen to evaluate nondeterministic choice. In the latter case, the strategy the program is run against selects which input transitions are possible, see Section 1.2. Terminal configurations contain special-purpose command $\text{skip}$.

The relation in Table 1.1 forms an IOLTS in the sense of Section 1.2 with $\langle \sigma, c \rangle$ as states and $\xrightarrow{}$ denoted $\rightarrow$. The behavior of $c$ is modeled as an IOLTS by $\langle \langle \rangle, c \rangle$, where $\langle \rangle = \lambda x. \ 0$ (the initial configuration of $c$).

1.6.3 Type system

Let the type environments $\Gamma$ be maps from variables to security levels, $l$. For clarity let $pc$ denote the security context, used to track implicit flows, and let $\tau$ denote security levels of values.

The expression typing judgments are given as $\Gamma \vdash e : \tau$, where $\Gamma(x) \subseteq \tau$ for each $x$ occurring in $e$, and the typing judgments for commands have the form $pc,l,\Gamma \vdash_\gamma c : l'$, where the lexical context $pc$ is a lower bound of the effects (assignments, inputs and outputs) in $c$, the blocking context $l$ is a lower bound of the actions (inputs and outputs) of $c$, and the blocking level $l'$ an upper bound of the security level of the blocking behavior of $c$. In addition, the typing judgments are parametrized on a channel labeling $\gamma$ that ranges over $L$, $M$, and $H$.

The type rules for commands are found in Table 1.2. They are standard apart from the addition of the blocking context and the blocking level. Rule $\text{asn}$ prohibits implicit flows by taking the security context into account, where the security context is raised to the security level of the expressions guarding the control flow in rules $\text{if}$, and $\text{wh}$.

The blocking level expresses whether the blocking behavior of a command depends on secrets or not. Hence, a command with secret blocking level, $pc,l,\Gamma \vdash_\gamma$
\begin{center}
\begin{tabular}{|l|l|}
\hline
\textbf{asn} & $\langle \sigma, x := e \rangle \rightarrow \langle \sigma[x \mapsto \sigma(e)], \text{skip} \rangle$ \\
\hline
\textbf{nd1} & $\langle \sigma, c_1 \parallel c_2 \rangle \rightarrow \langle \sigma, c_1 \rangle$ \\
\hline
\textbf{nd2} & $\langle \sigma, c_1 \parallel c_2 \rangle \rightarrow \langle \sigma, c_2 \rangle$ \\
\hline
\textbf{if1} & $\sigma(e) \neq 0$ \\
\hline
\textbf{if2} & $\langle \sigma, \text{if } e \text{ then } c_1 \text{ else } c_2 \rangle \rightarrow \langle \sigma, c_1 \rangle$ \\
\hline
\textbf{if} & $\sigma(e) = 0$ \\
\hline
\textbf{scq1} & $\langle \sigma_1, c_1 \rangle \xrightarrow{a} \langle \sigma_2, c'_1 \rangle \quad c'_1 \neq \text{skip}$ \\
\hline
\textbf{scq2} & $\langle \sigma_1, c_1 \rangle \xrightarrow{a} \langle \sigma_2, \text{skip} \rangle$ \\
\hline
\textbf{wh1} & $\langle \sigma, \text{while } e \text{ do } c \rangle \rightarrow \langle \sigma, \text{while } e \text{ do } c \rangle$ \\
\hline
\textbf{wh2} & $\langle \sigma, \text{while } e \text{ do } c \rangle \rightarrow \langle \sigma, \text{skip} \rangle$ \\
\hline
\textbf{in} & $\langle \sigma, \text{in}_\alpha(x) \rangle \xrightarrow{\alpha_v} \langle \sigma[x \mapsto v], \text{skip} \rangle$ \\
\hline
\textbf{out} & $\langle \sigma, \text{out}_\alpha(e) \rangle \xrightarrow{\alpha_{\sigma(e)}} \langle \sigma, \text{skip} \rangle$ \\
\hline
\end{tabular}
\end{center}

Table 1.1: Semantics of Commands
\[
\begin{align*}
\text{nd} & \quad pc, l_1, \Gamma \vdash_{\gamma} c_1 : l_2 \quad pc, l_1, \Gamma \vdash_{\gamma} c_2 : l_2 \\
& \quad pc, l_1, \Gamma \vdash_{\gamma} c_1 ; c_2 : l_2 \\
\text{asn} & \quad \Gamma \vdash e : \tau \quad pc \sqcup \tau \subseteq \Gamma(x) \\
& \quad pc, l, \Gamma \vdash_{\gamma} x := e : L \\
\text{seq} & \quad pc, l_1, \Gamma \vdash_{\gamma} c_1 : l_2 \quad pc, l_1 \sqcup l_2, \Gamma \vdash_{\gamma} c_2 : l_3 \\
& \quad pc, l_1, \Gamma \vdash_{\gamma} c_1 ; c_2 : l_2 \sqcup l_3 \\
\text{inL} & \quad \gamma(\alpha) = L \\
& \quad L, L, \Gamma \vdash_{\gamma} \text{in}_\alpha(x) : L \\
\text{inM} & \quad \gamma(\alpha) = M \quad H \subseteq \Gamma(x) \\
& \quad L, L, \Gamma \vdash_{\gamma} \text{in}_\alpha(x) : L \\
\text{inH} & \quad \gamma(\alpha) = H \quad H \subseteq \Gamma(x) \\
& \quad pc, l_1, \Gamma \vdash_{\gamma} \text{in}_\alpha(x) : H \\
\text{outL} & \quad \gamma(\alpha) = L \quad \Gamma \vdash e : L \\
& \quad L, L, \Gamma \vdash_{\gamma} \text{out}_\alpha(e) : L \\
\text{outM} & \quad \gamma(\alpha) = M \\
& \quad L, L, \Gamma \vdash_{\gamma} \text{out}_\alpha(e) : L \\
\text{outH} & \quad \gamma(\alpha) = H \\
& \quad pc, l, \Gamma \vdash_{\gamma} \text{out}_\alpha(e) : L \\
\text{if} & \quad \Gamma \vdash e : \tau \quad pc \sqcup \tau, l_1, \Gamma \vdash_{\gamma} c_1 : l_2 \\
& \quad pc \sqcup \tau, l_1, \Gamma \vdash_{\gamma} c_2 : l_2 \\
\text{wh} & \quad \Gamma \vdash e : \tau \quad pc \sqcup \tau, l_1, \Gamma \vdash_{\gamma} c : l_2 \quad l_2 \sqsubseteq l_1 \\
& \quad pc, l_1, \Gamma \vdash_{\gamma} \text{while} e \text{ do } c : pc \sqcup \tau \sqcup l_2 \\
& \quad pc_2, l_2, \Gamma \vdash_{\gamma} c : l_3 \\
\text{sb} & \quad pc_1 \sqsubseteq pc_2 \quad l_1 \sqsubseteq l_2 \quad l_3 \sqsubseteq l_4 \\
& \quad pc_1, l_1, \Gamma \vdash_{\gamma} c : l_4
\end{align*}
\]

Table 1.2: Type rules of commands
1.6 Enforcement

c : H, may diverge and/or input-block, depending on secrets. This implies that any succeeding commands cannot be allowed to have public actions, i.e., the succeeding command must be typable in a secret blocking context, as seen in rule seq. Otherwise a leak occurs, as demonstrated in the introduction. There the example program in_H(x); out_L(1) was used, but any source of secret blocking must lead to the same restrictions, as illustrated by rules inH and wb.

In contrast to previous work [24], our type system allows secret blocking levels. A program well-typed with a secret blocking level can block depending on secret information, but that blocking will not influence public actions performed by the program. This discipline is similar to the handling of while loops by Boudol and Castellani [7] and Smith [31], where assignments to public variables are prevented from happening after any possibility of entering loops with high guards. A key difference in our approach is the blocking context, which distinguishes internal side effects (due to assignment) from external side effects (due to output and input, which are not treated by Boudol, Castellani, or Smith). Assuming h is secret and l is public, it allows us to rightfully accept secure programs like in_H(x); l := 1 and (while h do h := h); l := 1, which are problematic in the shared-memory concurrency setting considered by Boudol, Castellani, and Smith.

All commands with public actions — rules inL, inM, outL, and outM — are constrained to run in public security and blocking level. This means that the program in_H(x); out_L(1) is not accepted by the type system, whereas in_M(x); out_L(1) is. Also, the order of instructions matters as indicated by the threading in rule seq. Hence, out_L(1); in_H(x) is accepted by the type system since the secret-presence input occurs after the public-presence output.

(1.24) **Theorem (Soundness of the type system)**

\[
p_c, l_1, \Gamma \vdash c : l_2 \quad \iff \quad \langle \langle \rangle, c \rangle \in DS-NI
\]

*Proof.* The proof can be found in the appendix. □

1.6.4 Parallel Composition

We add top-level parallel composition to the language. A program p is a command or two programs in parallel.

\[
p ::= c \mid p_1 \parallel p_2
\]

We obtain a semantics for parallel composition by lifting the parallel composition operator to the level of IOLTSs; with

\[
[c] = \langle \langle \rangle, c \rangle
\quad \quad \quad
[p_1 \parallel p_2] = [p_1] \parallel [p_2],
\]

the semantics for parallel composition is as presented in Section 1.5. Typing parallel composition is done by typing all participating commands using the same
channel labeling.

\[
\text{parc} \quad \frac{\Gamma \vdash \gamma \text{pc} : \text{l}\_1}{\Gamma \vdash \gamma c : \text{l}\_2}
\]

Since the type system guarantees DS-NI, the soundness of parallel composition follows from the soundness of the type system and the compositionality result of Section 1.5.

(1.25) **Theorem**  Soundness of parallel composition.

\[
\Gamma \vdash p = \text{[p]} \in \text{DS-NI}
\]

**Proof.** Immediate from Theorems 1.24 and 1.23. □

1.6.5 Fork command

On a last note, say our language contains a \texttt{fork}(c) primitive, which, when executed, will cause \langle\sigma, c\rangle to run in parallel with the executing IOLTS, where \sigma is either \langle\rangle or (a copy of) the variable environment of the executing IOLTS. If \emph{c} produces \emph{L} effects, and \texttt{fork}(\emph{c}) is executed in a \emph{H} context, then an information leak can occur, as in \emph{in}_\text{M}(h); \text{if} h \text{ then } \texttt{fork}(\texttt{out}_\text{L}(0)) \text{ else } h := h. To track flows in the presence of \texttt{fork}(\emph{c}), we suggest typing \emph{c} under the context of creation of \langle\sigma, c\rangle, as in the following rule.

\[
\text{fork} \quad \frac{\Gamma \vdash \gamma \text{pc} : \text{l}\_1}{\Gamma \vdash \gamma \text{fork}(\emph{c}) : \text{l}}
\]

Note the l’ because \emph{c} cannot, by blocking, constrain the behavior of the IOLTS executing \texttt{fork}(\emph{c}).

1.7 Related work

Security of interactive systems has been investigated in the context of process calculi [12, 28, 14, 27, 15, 25, 17] and event-based abstractions [18, 19, 29]. Connections with security models for more concrete programming languages have been made [20, 13]. However, relatively little has been done on tracking the flow of information through language constructs in interactive languages.

**Strategy-based models** Wittbold and Johnson [33] are the first to define strategy-based information-flow security. In a language-based setting, O’Neill et al. [24] investigate the security of interactive programs in the presence of user strategies. They present a strategy-based security condition and a type system that guarantees security. Our framework generalizes this work by distinguishing the security level for message presence and removing the assumption of the totality for
strategies. Compared to the type system by O’Neill et al., our type system (i) tracks the level of message presence, (ii) handles parallel composition, and (iii) has more permissive rules for loops.

Clark and Hunt [9] prove that it makes no difference in a deterministic setting whether the environment is represented by strategies or streams. Our results can be seen as a generalization along two dimensions. The first dimension allows both total and nontotal strategies. The second dimension parametrizes in the presence level for channels.

Stream-based models Streams are commonly used for representing the interaction environment of programs. Our generalization of Clark and Hunt’s results ensures that using streams does not sacrifice generality, as long as programs are deterministic.

Sabelfeld and Mantel [29] investigate the impact of different types of channels (secret, encrypted, public) and different types of communication (synchronous and asynchronous) on information-flow security. The encrypted channel is similar to our low-presence channel, where only the presence (not the content) of messages is visible to attackers. The origins of presence and content levels are in security labels for datatypes. For example, Jif [22, 23] allows arrays, where the length of the array is public but the individual elements are secret.


Askarov et al. [3] clarify the impact of leaking information via intermediate output. They investigate a condition that is insensitive to computation progress and show that the attacker cannot learn secret information in polynomial time in the size of the secret. This implies that restrictions on language constructs that might result in abnormal termination or divergence, originating in classical security analysis [10, 32] and supported in modern information-flow tools Jif [23], FlowCaml [30], and the SPARK Examiner [5, 8], are not strong enough to prevent brute-force attacks.

Bohannon et al. [6] propose security definitions for reactive systems that correspond to four indistinguishability relations on streams. They emphasize CP-security (sensitive to computation progress) and ID-security (insensitive to computation progress and thus similar to the one by Askarov et al. [3]).

In a stream-based setting, Rafnsson and Sabelfeld [26] distinguish the security level of message presence and content, accommodate new handler creation, and deploy output buffering to reduce leaks through intermediate output to at most one bit per consumed public input.

Devriese and Piessens [11] suggest splitting the execution of a program onto threads operating at different security levels. Only the thread at a given level is allowed to produce output on a channel labeled with the level. An input at a given security level is processed by the thread at that level and forwarded to threads that
are above in the security hierarchy. With some care taken when scheduling the threads (as spelled out by Kashyap et al. [16]), it is possible to achieve both timing- and termination-sensitive noninterference.

**Local interaction**  Almeida Matos et al. [1] consider local synchronous composition of threads under cooperative scheduling. They study a reactive setting, where threads can broadcast and react to local signals. They propose a formalization of noninterference for this setting and a type system that enforces it. The focus is primarily on suspension features and leaks associated with them.

### 1.8 Conclusion

We have presented a generalized framework for securing interactive programs. The framework drops the assumption from previous work that strategies must be always able to feed new input into the system. Further, the framework enables fine-grained security types for channels, distinguishing between the security level of message presence and content.

We have established compositionality of the security condition: assorted compositions of secure threads result in a secure thread pool. We have showed an enforcement of the condition via a type system. The type system capitalizes on the distinction between the security level of message presence and content, as well as on the compositionality properties.

Future work is focused on exploring the impact of nondeterminism on the security condition. We are interested in tight stream-based approximations of strategy-based security as well as in a type system that tracks the interplay between nondeterminism and interaction.

### Acknowledgments

This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR.

### References


Chapter 1: Securing Interactive Programs


Appendix

Notation

In the following, $ ::= ? | !$. The inputs in $t$, written $t ?$, is given by

$$
\alpha v. t' ? = \begin{cases} 
\alpha v. (t' ?) & \text{if } \alpha = ? \\
t ? & \text{otherwise.}
\end{cases}
$$

The outputs in $t$, written $t !$, is defined analogously. The $\alpha$-messages in $t$, written $t \alpha$, is given by

$$
\alpha v. t' \alpha = \begin{cases} 
\alpha v. (t' \alpha) & \text{if } \alpha' = \alpha \\
t \alpha & \text{otherwise.}
\end{cases}
$$
The $l$-observables in $t$, written $t ↾ l$, is given by

$$αv.t ↾ l = \begin{cases} 
  αv.(t ↾ l), & \text{if } γ(α) = l \text{ and } l_1 ⊑ l \\
  α □ .(t ↾ l), & \text{if } γ(α) = l \text{ and } l_2 ⊑ l \\
  t ↾ l, & \text{otherwise.}
\end{cases}$$

For all of these $|$-operators, $ε | = ε$ and $(τ.t) | = t$. $P(A)$ is the powerset of $A$. $|A|$ is the number of elements of $A$. $|t|$ is the number of actions in $t$. $t' ≤ t$ if there is some $t''$ for which $t = t't''$.

**Bounded Strategies**

Consider strategies which, for each $α$, are total on the set of traces traces which contains a number of $α$-inputs less than some bound $n$. We refer to these as bounded strategies.

(1.26) **Definition** $ω$ is bounded if for all $α$, there is an $n$ for which

$$|t ↾ α|_2 < n ⇐⇒ ω_α(t) ≠ \emptyset.$$  

Let $W_B$ denote the set of bounded strategies in $W$.

(1.27) **Definition** $ω$ refines $ω'$, written $ω ≤ ω'$, iff for all $α$ and $t$,

$$ω_α(t) ⊆ ω'_α(t).$$

(1.28) **Lemma** ([9]) If $ω ≤ ω'$ and $ω | = s ⇔$, then $ω' | = s ⇔$.

It turns out we only need to consider attacks containing deterministic, bounded strategies when checking for insecure flows in an interactive program.

(1.29) **Proposition** $\text{Strat-NI} = \text{DS}_B\text{-NI}$.

**Proof.** We prove $\text{DS-NI} = \text{DS}_B\text{-NI}$; the result will then follow from Theorem 1.13. $\text{DS}_B ⊆ \text{DS}$ by definition of $\text{Strat}_B$. $\text{DS-NI} ⊆ \text{DS}_B\text{-NI}$ by Lemma 1.8. We now show $\text{DS-NI} ≥ \text{DS}_B\text{-NI}$. That is,

$$∀s . s \in \text{DS}_B\text{-NI} ⇒ s \in \text{DS-NI}.$$  

We show instead the contrapositive. That is,

$$∀s . s \notin \text{DS-NI} ⇒ s \notin \text{DS}_B\text{-NI}.$$  

(1.13)

Let $s \notin \text{DS-NI}$ be given. Then there is a $\text{DS}$-attack $(l, ω_1, ω_2, t_1)$ on $s$. Particularly,

1. $ω_1 =_l ω_2$,
2. $ω_1 | = s \downarrow l$, 
3. $ω_2 | = s \downarrow l$. 

Let $s \notin \text{DS-NI}$ be given. Then there is a $\text{DS}$-attack $(l, ω_1, ω_2, t_1)$ on $s$. Particularly,

1. $ω_1 =_l ω_2$,
2. $ω_1 | = s \downarrow l$, 
3. $ω_2 | = s \downarrow l$. 


3. \( \forall t_2 \cdot \omega_2 \models s \xrightarrow{t_2} \implies t_2 \neq t_1. \)

Let
\[
\omega_j'(t) = \begin{cases} 
\bot, & \text{if } |t|-\alpha| ? \geq |t_1|-\alpha| ? \\
\omega_j(t), & \text{otherwise}
\end{cases}
\]

We must show that
i) \( \omega_1' = t \omega_2' \),

ii) \( \omega_j' \) is a strategy,

iii) \( \omega_1' \models s \xrightarrow{t_1} \),

iv) \( \forall t_2 \cdot \omega_2' \models s \xrightarrow{t_2} \implies t_1 \neq t_2. \)

We have i) from 1) since, for all \( t \) and \( \alpha \),
\[
\omega_{1_\alpha}(t) \neq \omega_{1_\alpha}'(t) = \bot \implies \omega_{2_\alpha}(t) = \bot
\]
\[
\omega_{2_\alpha}(t) \neq \omega_{2_\alpha}'(t) = \bot \implies \omega_{1_\alpha}'(t) = \bot.
\]

For \( \alpha \) for which \( \gamma(\alpha) = l^{l_1} \), since
\[
t = l_2 \ x \ t' \implies |t|-\alpha| ? = |t'-\alpha| ?
\]
and
\[
t = l_1 \ x \ t' \implies t = l_2 \ x \ t',
\]
then either
\[
\omega_{j_\alpha}(t) = \omega_{j_\alpha}(t') = \omega_{j_\alpha}'(t) = \omega_{j_\alpha}'(t') = \bot
\]
or
\[
\omega_{j_\alpha}'(t) = \omega_{j_\alpha}(t)
\]
and
\[
\omega_{j_\alpha}'(t') = \omega_{j_\alpha}(t').
\]

Thus, since \( \omega_j \) are strategies, ii) holds. Since for all \( \alpha \),
\[
t \ ? \alpha \ v \leq t_1 \implies |t|-\alpha| ? < |t_1|-\alpha| ?,
\]
we get \( \omega_1'(t) = \omega_{1_\alpha}(t) = \bot \). Thus \( \omega_1 \models t_1 \). Since \( s \xrightarrow{l_1} \), we get iii) by definition of \( s \xrightarrow{l_1} \). We have iv) by Lemma 1.28 since \( \omega_2' \leq \omega_2 \).

By Lemmas 1.15 and 1.16, \( \forall t, \alpha; \gamma(\alpha) = l^{l_1}; \ l_2 \subseteq l \),
\[
\omega_{1_\alpha}(t) = \bot \iff \omega_{2_\alpha}'(t) = \bot \quad (1.14)
\]
and \( \forall t, \alpha; \gamma(\alpha) = l^{l_1}; \ l_2 \subseteq l, \ v,
\[
\omega_2' \models s \xrightarrow{l_2} \ s' \ \wedge \ s' \xrightarrow{\ ? \alpha \ v} \wedge \omega_{2_\alpha}'(t) = \bot \implies t \ ? \alpha \ v \notin l \ t_1, \forall v.
\quad (1.15)
\]
In particular, this holds if we fix \( v \) to a constant \( k \). Let

\[
\hat{\omega}'_{j\alpha}(t) = \begin{cases} 
  k, & \text{if } |t|_{\alpha}\gamma < |t_1|_{\alpha}\gamma \land \omega'_j(t) = \bot, \\
  \omega'_j(t), & \text{otherwise.}
\end{cases}
\]

\[
\omega''_{1\alpha}(t) = \hat{\omega}'_{1\alpha}(t)
\]

\[
\omega''_{2\alpha}(t) = \begin{cases} 
  \bot, & \text{if } \alpha = l_1^{l_2} \land l_2 \not\subseteq l, \\
  \hat{\omega}'_{2\alpha}(t), & \text{otherwise.}
\end{cases}
\]

We must show that

a) \( \omega''_1 \neq \omega''_2 \),

b) \( \omega''_j \) is a strategy,

c) \( \omega''_1 \models s \models_{\xi_1} \),

d) \( \omega''_2 \models s \models_{\xi_2} \Rightarrow t_1 \neq t_2, \forall t_2. \)

We have a) from i) since, for all \( t \) and \( \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \subseteq l, \)

\[
\omega'_{1\alpha}(t) \neq \omega''_{1\alpha}(t) = \bot \Rightarrow \omega''_{2\alpha}(t) = \bot \\
\omega'_{2\alpha}(t) \neq \omega''_{2\alpha}(t) = \bot \Rightarrow \omega''_{1\alpha}(t) = \bot.
\]

It is easy to see that \( \forall t, \alpha; \gamma(\alpha) = l_1^{l_2}; l_2 \subseteq l, \)

\[
\omega'_{1\alpha}(t) = \omega'_{2\alpha}(t) = \bot \Rightarrow \omega''_{1\alpha}(t) = \omega''_{2\alpha}(t)
\]

This, i) and (1.14) gives a). It is easy to see that \( \forall t, \alpha, \)

\[
\omega'_{1\alpha}(t) \neq \bot \Rightarrow \omega'_{1\alpha}(t) = \omega'_{j\alpha}(t).
\]

This, and a), gives c). By (1.15) and by definition of \( \omega''_{2\alpha} \), d) holds.

By definition of \( \omega'_j \) and \( \omega''_j \), \( \omega'_1 \) and \( \omega''_2 \) are deterministic, and bounded, strategies. Thus \( s \not\in DS_{B-NI} \). Since \( s \) was arbitrary, (1.13) holds. \( \square \)

Lemma 1.14 follows from the proof of Proposition 1.29.

**Soundness of the type system**

We perform the proof in the instrumented semantics of O’Neill et al. [24] \((c, \sigma, \psi, t, \omega)\) extended with our richer model of channels. This extension is straightforward and does not significantly change the proofs.

In essence our type system is the type system of O’Neill et al. extended with the blocking context and blocking level. For programs typed \( pc, L, \Gamma \vdash \gamma c : L \) their soundness proof applies with minor modifications, since a public blocking context guarantees that the program is free from secret blocking. We begin by
establishing a few lemmas relating our type system and semantics to the type system of O’Neill et al.

(1.30) **Lemma** If channels \( \alpha \) s.t. \( \gamma(\alpha) = M \) on the left hand side are interpreted as \( H \) on the right hand side we have the following result.

\[
p, L, \Gamma \vdash_\gamma c : L \Rightarrow \Gamma \vdash c : pc \ cmd
\]

*Proof.* By structural induction on \( c \). \( \square \)

Here, \( \Gamma \vdash c : pc \ cmd \) is the typing judgment of [24]. Semantically, we have the following correspondence between our semantics and the semantics of [24].

(1.31) **Lemma** For all \( \omega, \sigma, c \) and \( t \),

\[
\omega \models \langle \sigma, c \rangle \Downarrow \Rightarrow \exists \psi, \psi', c', \sigma'. (c, \sigma, \psi, \langle \rangle, \omega) \rightarrow (c', \sigma', \psi', t, \omega)
\]

*Proof.* The existence of \( \psi \) and \( \psi' \) corresponding to the nondeterministic choices is immediate. The result follows. \( \square \)

For programs free from blocking the correspondence goes the other direction.

(1.32) **Lemma** For commands \( c \) s.t. \( p, L, \Gamma \vdash_\gamma c : L \) it holds that

\[
(c, \sigma, \psi, \langle \rangle, \omega) \rightarrow (c', \sigma', \psi', t, \omega) \Rightarrow \omega \models \langle \sigma, c \rangle \Downarrow
\]

*Proof.* The result follows from the fact that \( c \) is free from secret blocking, which is given by \( p, L, \Gamma \vdash_\gamma c : L \). \( \square \)

(1.24) **Theorem** (*Soundness of the type system*)

\[
p, l_1, \Gamma \vdash_\gamma c : l_2 \Rightarrow \langle \langle \rangle, c \rangle \in DS-NI
\]

*Proof.* Let \( \sim_L \) be defined as in [24] with the exception that \( \omega_1 \sim_L \omega_2 \) is taken to be \( \omega_1 =_L \omega_2 \). The proof proceeds by case analysis on \( p, l_1, \Gamma \vdash_\gamma c : l_2 \).

\[
p, L, \Gamma \vdash_\gamma c : L \] This case is equivalent to the proof of [24]. Given \( \omega_1 =_L \omega_2 \) and \( \sigma_1 \sim_L \sigma_2 \), for \( \omega_1 \models \langle \sigma_1, c \rangle \Downarrow \), show that there exists a \( t_2 \) such that \( \omega_2 \models \langle \sigma_2, c \rangle \Downarrow \) and \( t_1 \sim_L t_2 \). Lemma 1.31 gives us that there exists \( \psi_1, \psi'_1, c'_1, \sigma'_1 \), s.t. \( (c, \sigma_1, \psi_1, \langle \rangle, \omega_1) \rightarrow (c'_1, \sigma'_1, \psi'_1, t_1, \omega_1) \), and Lemma 1.30 gives us that \( \Gamma \vdash c : pc \ cmd \). Theorem 2 of [24] gives us that \( (c, \sigma_2, \psi_2, \langle \rangle, \omega_2) \rightarrow (c'_2, \sigma'_2, \psi'_2, t_2, \omega_2) \) and \( t_1 \sim_L t_2 \) for some \( \psi_2, \psi'_2, c'_2, \sigma'_2 \). Now, Lemma 1.32 allows us to establish \( \omega_2 \models \langle \sigma_2, c \rangle \Downarrow \) and the result follows.

\[
p, H, \Gamma \vdash_\gamma c : l \] Immediate, since \( c \) is low-silent.
Assume $\omega_1 =_L \omega_2$, and $\sigma_1 \sim_L \sigma_2$, and two executions $\omega_1 \models \langle \sigma_1, c \rangle \xrightarrow{\text{H}}$, and $\omega_2 \models \langle \sigma_2, c \rangle \xrightarrow{\text{L}}$. It is easy to show that there exists a prefix $c'$, and two suffixes $c_1, c_2$ such that $\omega_1 \models \langle \sigma_1, c' ; c_1 \rangle \xrightarrow{\text{L}}$, and $\omega_2 \models \langle \sigma_2, c' ; c_2 \rangle \xrightarrow{\text{L}}$, where $pc, L, \Gamma \vdash \gamma c' : H$, and $pc, L, \Gamma \vdash \gamma c_2 : H$, and where $c_1, c_2$ is prefixed by either $\text{out}_H(x)$ a secret conditional or a secret while. In either case, we have that $c_1$ and $c_2$ are low-silent (the secret conditional, or secret while are low-silent in the bodies, since they constitute secret contexts; any suffixes typed $pc, H, \Gamma \vdash \gamma c : H$ and are, hence, low-silent. Now, the result for $pc, L, \Gamma \vdash \gamma c' : L$ from above allows us to establish low-equivalence on the parts of the traces leading up to $c_1$ and $c_2$; from this $t_1 \sim_L t_2$ follows.

\[\Box\]
Compositional Information-flow Security for Interactive Systems

ABSTRACT  To achieve end-to-end security in a system built from parts, it is important to ensure that the composition of secure components is itself secure. This work investigates the compositionality of two popular conditions of possibilistic noninterference. The first condition, progress-insensitive noninterference (PINI), is the security condition enforced by practical tools like JSFlow, Paragon, LIO, Jif, FlowCaml, and SPARK Examiner. We show that this condition is not preserved under fair parallel composition: composing a PINI system fairly with another PINI system can yield an insecure system. We explore constraints that allow recovering compositionality for PINI. Further, we develop a theory of compositional reasoning for progress-sensitive noninterference (PSNI). In contrast to PINI, we show what PSNI behaves well under composition, with and without fairness assumptions. Our work is performed within a general framework for nondeterministic interactive systems.

2.1 Introduction

Modularity and compositionality are essential for the design and construction of modern computing systems. A major challenge is secure composition: to achieve end-to-end security in a system built from parts, it is important to ensure that the composition of secure components is itself secure. Secure composition is particularly intricate because security conditions are often fragile under system behavior modifications. Adding, removing, or modifying a single trace or event can break the security of a system [31].

This paper studies the foundations of secure composition. Our focus is on specifying confidentiality (or dual flavors of integrity [7,8]) by defining what constitutes secure information flow through computing systems.

Modeling security for interactive systems  Given the importance of the subject, it is not surprising that the literature has explored security of communic-
ating systems and their composition in assorted settings, discussed in more detail in Section 2.6. For example, security has been addressed in reactive systems (e.g., [2, 10, 42]). Reactive systems exercise a restrictive pattern of communication, when the system waits for an input, and once an input has been received it proceed with executing an appropriate handler until completion, possibly producing some output on the way. Event systems have been a focus of several previous approaches (e.g., [27, 28, 53, 56]). Events have different levels of sensitivity with the goal of protecting both presence and content of events but not distinguishing between the two. Interaction patterns in some of this work are fixed. For example, Wittbold and Johnson [53] assume the pattern of receiving a secret and a public event, followed by sending a secret and public event. With process calculi as the underlying models, a line of work (e.g., [17, 21, 22, 39, 45, 46]) studies secure interaction, inheriting the concrete features from the process calculus and not distinguishing between the sensitivity of presence and content of events. Closer to our work are formalizations that operate on labeled transition systems [14, 36, 41]. However, the above work models environments as strategies, which are separate from the computational model. These strategies can always receive and are always willing to send messages.

Thus, key questions that have not been addressed by previous work are: *) what is an appropriate general model of security of interactive systems, *) how to distinguish sensitivity of message presence and content in such a model, *) how do we model environments as part of the system, and *) how do we provide flexible ways for composing.

Progress-sensitive and progress-insensitive security  The focus of our work is two popular security conditions. The first condition, progress-insensitive noninterference (pini) [3, 4, 10], prevents information leaks from secret sources to public sinks, but allows secrets to affect progress of public computation. Thanks to the liberty it provides for handling loop constructs, this condition has been a popular target for such practical security tools as JSFlow [20], Paragon [12], LIO [50], Jif [33], FlowCaml [49], and SPARK Examiner [5]. The second condition is progress-sensitive noninterference (psni) [4, 10, 36, 42], which does not allow leaks via progress. The advantage of psni is that it provides stronger security guarantees and that it is not susceptible to laundering secrets by brute-force attacks [3] or re-running programs [9]. An important question that has not been previously answered by previous work is how do pini and psni behave under composition?

Contributions  The paper delivers the following contributions. The first contribution is a general security framework to model security of interactive systems. We obtain full generality by adopting labeled transition systems as the underlying model. In contrast to previous work, environments are tightly integrated into the computation model allowing flexibility in modeling the nature of what systems may be interacting with. The expressiveness of the environment model is key for compositionality results. Unifying the assumptions on environments
and systems provides us with a generic system model. More importantly, it also paves the way for secure composition thanks to the possibility that systems and environments can be manipulated interchangeably. The second contribution is *combinators* for composing systems. The combiners allow flexibility in how exactly composed components can interact with each other. This is in contrast to previous work where composition is typically restricted to a single way. The third contribution allows us further generality in the modeling of interaction: we distinguish between the sensitivity of message *presence* and message *content* without restricting communication paradigms. The fourth contribution is the study of *compositionality for PINI and PSNI*. We find that PINI is not preserved under fair parallel composition: composing a PINI system fairly with another PINI system can yield an insecure system, and thus, compositionality of PINI relies fundamentally on unfairness. We explore constraints that allow recovering compositionality for PINI. Further, we develop a theory of compositional reasoning for progress-sensitive noninterference (PSNI). In contrast to PINI, we show what PSNI behaves well under composition, with and without fairness assumptions.

**Organization**  The rest of the paper proceeds as follows. Section 2.2 presents the general setting of interactive systems, as specified by labeled transition systems. Section 2.3 presents the security definitions. Section 2.4 establishes compositionality properties for a core of combiners, with and without fairness assumptions. Section 2.5 demonstrates the generality of our results by providing a language of secure combiners. Section 2.6 reports on related work. Section 2.7 offers conclusions and points to worthwhile future work.

### 2.2 Interactive Systems

We present a language-independent framework for reasoning about the behavior of autonomous interactive programs. The framework functions as the foundation for our technical contribution, and unifies several previous frameworks for interactive program security [10, 14, 36, 41, 42].

#### 2.2.1 Computation Model

Our model of computation is a *labeled transition system* (LTS). An LTS is a triple \((\mathcal{P}, \mathcal{A}, \xrightarrow{\cdot})\), where \(\mathcal{P}\) is a set (of processes), \(\mathcal{A}\) is a set (of action labels), and \(\xrightarrow{\cdot} \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}\) (a labeled transition relation). Let \(p\) and \(a\) range over \(\mathcal{P}\) and \(\mathcal{A}\) respectively. Computation occurs in discrete steps (transitions). The label on a step is the *effect* of said step. These effects are the *only* external interface to our processes; they are “black boxes” in every other respect. \(p \xrightarrow{a} p'\) iff \((p, a, p') \in \xrightarrow{\cdot}\), and \(p \xrightarrow{a} p'\) iff \(p \xrightarrow{a} p'\) for some \(p'\).

The processes we consider interact with their environment through channel-based message-passing. They have two kinds of effects: (message-)input, denoted
Chapter 2: Compositional Information-flow Security for Interactive Systems

i, and output, denoted o. A message, m, is a value v on a channel c.

\[ a ::= i | o | i ::= ?m \quad o ::= !m \quad m ::= cv \] (act)

Here, \( cv \) (resp. \( ?cv \)) denotes a message sent (resp. received) on channel c carrying value v. We let i, o, m, c and v range over I, O, M, C and V, respectively.

(2.1) **Definition** An input-output LTS (LTS\_IO) is an LTS with action labels as defined in (act).

This notion of computation applies to a wide range of programs and systems. For instance, Bohannon et al. give the semantics of a JavaScript-like language as an LTS\_IO in [10, 11], and Rafnsson et al. give the semantics of an imperative language with I/O as an LTS\_IO in [41], demonstrating that LTS\_IO is rich enough to model anything from batch-style programs which receive one i, produce one o and terminate, to reactive systems which continuously interact. Since all of our results apply to LTS\_IOs, our contributions are general.

To illustrate, the following program reads binary sensor values until receiving 1, at which time the program outputs an alert message. Next to it is a graph of an LTS\_IO describing its input-output behavior.

```
repeat
  in sensor b
until b = 1
out alert 1
```

Here, \( p_{s0} \xrightarrow{\text{sensor}} p_{s0} \xrightarrow{\text{sensor}} p_{s1} \xrightarrow{\text{alert}} p_{sa} \) represents the program receiving value 0 on channel sensor twice, returning to its initial state each time, then receiving a 1, and subsequently outputting value 1 on channel alert. No more actions follow, so \( p_{sa} \not\rightarrow \), i.e., \( p_{sa} \) is a terminal state.

The LTS\_IO above abstracts from the internal actions (e.g. evaluating the loop condition). To model such noninteraction, we distinguish a channel for internal action, and denote any message on it by \( \diamond \). A more accurate LTS\_IO is then

While the above graphs have a finite number of nodes, the LTS\_IO semantics of many programs have infinitely many nodes (nodes are the only way to represent state).

### 2.2.2 Behaviors

To study the behavior of an LTS\_IO, we consider sequences of actions performed by it, that is, solutions to the equation

\[ Z = A \times Z. \] (seq)
The smallest solution containing $\epsilon \notin A$ is the set $Z = \mathcal{T}$ of traces, that is, (finite) lists of actions, ranged over by $t$. For instance, with "e" denoting the empty trace and "", the constructor ("cons") operator, $t_s = \text{sensor1.!alert1.e}$ is a trace. Alternatively, $\mathcal{T}$ could be defined as $\mathcal{T} = A^\omega$, i.e., the Kleene closure of $A$, or inductively, by asserting that $\epsilon \in \mathcal{T}$, and specifying, given an action $a$ and a trace $t \in \mathcal{T}$, how to obtain a new trace in $\mathcal{T}$ ($a.t \in \mathcal{T}$). For simplicity, we usually omit the $\epsilon$ trailing a nonempty trace, use "." to also denote trace concatenation ($e.t = t$ and $(a.t').t = a.(t'.t)$), and write $t^n$ for the concatenation of $t$ $n$ times ($t^0 = \epsilon$ and $t^{n+1} = t.t^n$). We write $t \leq t'$ when $t' = t.t''$ for some $t''$.

The largest nonempty solution\(^1\) to (seq) is the set $Z = \mathcal{S}$ of streams, that is, infinite lists of actions, ranged over by $s$. For instance, with "." denoting the deconstructor operator, $s_s$ for which $(.) s_s = (\text{sensor0}, s_s)$ is a stream, i.e., the infinite sequence of $\text{sensor0}$ actions. Alternatively, $\mathcal{S}$ could be defined as $\mathcal{S} = A^\omega$, i.e. the language of all infinite (not finite\(^2\)) words over $A$, or coinductively, by specifying, given a stream $s \in \mathcal{S}$, how to obtain an action and a new stream in $\mathcal{S}$ (apply "." on $s$). For simplicity, we let $a.s$ denote any stream $s'$ for which $(.) s' = (a, s)$, and let $t^\omega = t.(t^\omega)$. We define $t \leq s$ and $t.s$ the same way we did for traces. For details on coinduction, see [6, 23].

The behaviors of processes are as follows. For each $p$, $p \triangleleft p$, and $p \triangleleft$ if $p \triangleleft p'$ $\bowtie$. Likewise, $p \triangleleft$ only if $p \triangleleft p'$ $\triangleleft$. Let $\mathcal{T}(p) = \{ t \mid p \triangleleft t \}$ and $\mathcal{S}(p) = \{ s \mid p \triangleleft s \}$ be the set of traces and streams of $p$, i.e. the trace- and stream-semantics of $p$. For instance, $p_{s_0} \triangleleft$, and $p_{s_0} \triangleleft$, so $t_s \in \mathcal{T}(p_{s_0})$ and $s_s \in \mathcal{S}(p_{s_0})$. Traces and streams can themselves be regarded as $\text{LTS}_{\text{IO}}$, simply by defining $a.t \triangleleft t$ and $a.s \triangleleft s$. This will turn out to be useful later.

At first, it appears that stream semantics are less expressive than trace semantics, since e.g., $\mathcal{S}(p_{s_0}^\infty) = \{ \text{sensor0} \}$ and $t_s \notin \text{sensor0}^\infty$. However, this is only the case for $\text{LTS}_{\text{IO}}$ that have terminal states. We say $p$ is productive if it has no terminal states (this is defined formally in Definition 2.38 pt. 4)). For productive $p$, we have that
\[
\forall s. s \in \mathcal{S}(p) \implies (\forall t \leq s. t \in \mathcal{T}(p)). \quad (\mathcal{S} \Rightarrow \mathcal{T})
\]

Unless we state otherwise, we assume throughout our development that $p$ is productive. This assumption is made without loss of generality, since any $p$ which is not productive can be modeled as a productive $\text{LTS}_{\text{IO}}$, as follows. First, we model termination as an action by distinguishing a termination channel, and denoting any message on said channel by $\ast$. The obituary wrapper $O$ announces termination of the wrapped process, leaving its behavior in other ways intact.

\[
\frac{p \rightarrow_{\text{O}}} {O(p) \triangleleft p} \quad \frac{p \triangleleft p'} {O(p) \triangleleft O(p')}\]

The zombie wrapper $Z$ keeps terminated processes productive by enabling the !\diamondsuit

---

\(^1\)It exists and is unique [6]

\(^2\) $B^n$ and $B^\omega$ are disjoint; $B^{\infty} = B^n \cup B^\omega$. 
action in terminal states.
\[ p \rightarrow Z(p) \stackrel{\circ}{\rightarrow} Z(p) \]
\[ p \triangleleft p' \rightarrow Z(p) \triangleleft Z(p') \]

Thus, \( Z \circ O \) attaches this to terminal states in a process:

\[ \text{\#} \]

Now any property \( \phi \) on \( T(p) \) can be stated as a property \( \phi' \) on \( S(Z(O(p))) \) such that \( \phi(T(p)) \iff \phi'(S(Z(O(p)))) \), since \( T(p) = \{ t \mid ( \exists s \in S(Z(O(p))) \cdot t \leq s ) \wedge ( \forall t' \cdot t' \not\preceq t ) \} \).

However, trace semantics and stream semantics are not equally expressive; the reverse implication of \( (S \Rightarrow T) \) does not hold in general. To see this, consider these \( \text{LTS}_{\text{IOs}} \).

Let \( p_L \) (resp. \( p_R \)) denote \( O \) in the left (resp. right) \( \text{LTS}_{\text{IO}} \) above. Then \( T(p_L) = T(p_R) \). However, \( S(p_L) \neq S(p_R) \); in particular, for \( s_0 = !\text{zero}0\omega \), \( p_L \triangleleft s_0 \), but \( p_R \ntriangleleft s_0 \), since \( p_R \) always performs an arbitrary, but finite, number of \( !\text{zero}0 \) actions, and then starts performing only \( !\text{eof}0 \) actions. This illustrates that there are scenarios where trace semantics is too weak; in the above, trace semantics, depending on perspective, hides the eventuality of an \( !\text{eof}0 \) action in \( p_L \), or hides the possible non-eventuality of an \( !\text{eof}0 \) action in \( p_R \). We suspect this difference in expressiveness only occurs in productive \( \text{LTS}_{\text{IO}} \) in the presence of unbounded nondeterminism [44], present in \( p_R \) as it nondeterministically picks a number \( n \) from the full range of natural numbers.

Finally, if we instead expressed semantics as a function mapping input behaviors to output behaviors, we would not have a complete rule for composition, as the composition would suffer from the Brock-Ackerman anomaly [13], an issue addressed by [52].

2.2.3 Comparing Behaviors

To reason about the behavior of processes, we need ways to compare behaviors. Comparing traces for equality can easily be done inductively component-wise, and a proof of trace equality will be inductive (thus finite). But what about infinite streams? One way is to define stream difference as the eventual failure of component-wise equality, and define stream equality as non-difference (i.e. nonexistence of inductive proof of difference). Instead we will use the following coinductive definition which captures the same idea as component-wise trace equality.
\subsection*{2.2 Interactive Systems}

**Definition** \( \mathcal{R} \subseteq S \times S \) is a *strong stream relation* iff
\[
\forall s_1, s_2 \cdot s_1 R s_2 \implies \\
\exists a, s'_1, s'_2 \cdot s_1 \xrightarrow{a} s'_1 \land s_2 \xrightarrow{a} s'_2 \land s'_1 R s'_2,
\]
s\_1 and s\_2 are *strongly stream related*, \( s_1 = s_2 \), iff there exists a strong stream relation \( \mathcal{R} \) such that \( s_1 R s_2 \).

This is a strong bisimulation on streams. Since strong stream relations are equivalence relations, we refer to them as strong stream equivalences and say \( s_1 \) and \( s_2 \) are (strongly stream) equivalent iff \( s_1 = s_2 \). To prove that \( s_{0s} \) and \( s_{0e} = !zero0.\text{eof}^\omega \) are not equivalent, assume \( s_{0s} R s_{0e} \) for some strong stream equivalence \( \mathcal{R} \). Then \( s_{0s} R !\text{eof}^\omega \), since \( s_{0s} \xrightarrow{\text{zero}0} s_{0e} \) and \( !\text{zero}0.\text{eof}^\omega \xrightarrow{!\text{zero}0} \text{eof}^\omega \). But \( s_{0s} \xrightarrow{\text{zero}0} !\text{zero}0 \text{eof}^\omega \) and \( !\text{zero}0 \equiv \text{eof}^\omega \), contradicting that \( \mathcal{R} \) is a strong stream equivalence. Thus \( s_{0s} \neq s_{0e} \). An inductive proof of difference would use the fact that \( !\text{zero}0.\text{!zero}0 \leq s_{0e} \) and \( !\text{zero}0 \text{!zero}0 \not\leq s_{0s} \); this leads us to the following useful observation, proven by Park in [38],

\[
s_1 = s_2 \iff \forall t \cdot t \leq s_1 \iff t \leq s_2.
\]

Ultimately, we need to compare *observable* behavior to reason about information-flow security of programs. Which behavior is observable and which is not is a topic better left for Section 2.3; for now, we let \( \bullet \) range over unobservable actions. Furthermore, we let \( \bullet \) function as a wildcard, so multiple occurrences of \( \bullet \) in the same context can represent different mathematical objects. We will then define *observational equality* as component-wise equality of observables. To obtain the next observable action, we use the following “weak” labeled transition relation: \( p \xleftarrow{\cdot} p' \text{ iff } a \neq \bullet \land (p \xrightarrow{\cdot} p' \lor (p \xleftarrow{\cdot} p'' \land p \xrightarrow{\cdot} p')) \).

**Definition** \( \mathcal{R} \subseteq S \times S \) is a *weak stream relation* iff
\[
\forall s_1, s_2 \cdot s_1 R s_2 \implies (s_1 \xrightarrow{\cdot} \land s_2 \xrightarrow{\cdot}) \lor \\
\exists a, s'_1, s'_2 \cdot s_1 \xrightarrow{a} s'_1 \land s_2 \xrightarrow{\cdot} s'_2 \land s'_1 R s'_2.
\]
s\_1 and s\_2 are *weakly stream related*, \( s_1 \equiv s_2 \), iff there exists a weak stream relation \( \mathcal{R} \) such that \( s_1 R s_2 \).

This is a coinductive definition of a weak bisimulation on streams. For instance, \( s_{0s} \equiv (\bullet.\text{sensor}0)^\omega \). Again, since weak stream relations are equivalence relations, we refer to them as weak stream equivalences and say \( s_1 \) and \( s_2 \) are weakly (stream) equivalent iff \( s_1 \equiv s_2 \). Roughly, the inductive definition of \( t_1 \equiv t_2 \) is Definition 2.3 with the implication reversed (\( t_1 \equiv t_2 \) if \( t_1 \xrightarrow{\cdot} \land t_2 \xrightarrow{\cdot} \) or \( t_1 \xrightarrow{\cdot} t_1' \land t_2 \xrightarrow{\cdot} t_2' \land t_1' \equiv t_2' \)). We write \( t \leq t' \) when \( t \equiv t'' \) for some \( t'' \leq t' \), and define \( t_1 \leq s_2 \) similarly. Note that \( s_1 \neq s_2 \) does not necessarily have an inductive proof; this is the case when the observables of \( s_1 \) strictly prefix the observables in \( s_2 \) for some \( t \), \( s_1 \xrightarrow{t} s_2', s_2 \xrightarrow{t} s_2' \) and \( s_1 \equiv s_2' \), but \( \bullet \xrightarrow{\cdot} \) does not have an inductive proof, since this is an assertion about all elements in an infinite sequence. A proof of \( t \leq s \) is inductive, since \( t \) is finite. The following analog of \((=\text{-and-ts})\) is therefore of interest.

\[
s_1 \equiv s_2 \iff \forall t \cdot t \leq s_1 \iff t \leq s_2.
\]
The other means we consider of comparing observables stipulates that there is never a disagreement in which observable comes next in two streams. That is, component-wise comparison of observables does not eventually yield two observables which differ. This holds e.g. when \( s_1 \approx s_2 \) or when the observables in \( s_1 \) prefix the observables in \( s_2 \).

\[ (2.4) \textbf{DEFINITION} \ R \subseteq S \times S \text{ is a feeble stream relation iff} \]
\[
\forall s_1, s_2 : s_1 R s_2 \implies s_1 \trianglerighteq \lor s_2 \trianglerighteq \lor
\]
\[
\exists a, s_1', s_2' : s_1 \xrightarrow{a} s_1' \land s_2 \xrightarrow{a} s_2' \land s_1 R s_1'
\]
\[
s_1 \text{ and } s_2 \text{ are feebly stream related, } s_1 \approx s_2, \text{ iff there exists a feeble stream relation } R
\]
\[
such that s_1 R s_2. \quad \diamondsuit
\]

This is a coinductive definition. This relation is reflexive and symmetric, but not transitive, and therefore not an equivalence relation; \( \text{sensor}\_\_0 \cdot \_\_0^\omega \approx \_\_0^\omega \) and \( !\text{zero}\_\_0 \cdot \_\_0^\omega \approx \_\_0^\omega \), but \( \text{sensor}\_\_0 \cdot \_\_0^\omega \neq !\text{zero}\_\_0 \cdot \_\_0^\omega \) (indeed, for all \( s, s \approx \_\_0^\omega \)). To contrast, \( !\text{zero}\_\_0 \cdot \_\_0^\omega \neq \_\_0^\omega \). Again, the inductive definition of \( t_1 \approx t_2 \) is Definition 2.4 with reversed implication (\( t_1 \approx t_2 \) if \( t_1 \rightarrow a \), \( t_2 \rightarrow a \), or \( t_1 \xrightarrow{a} t_1' \land t_2 \xrightarrow{a} t_2' \land t_1' \approx t_2' \)). On traces, \( t_1 \approx t_2 \) iff \( t_1 \leq t_2 \) or \( t_2 \leq t_1 \). Note that \( s_1 \neq s_2 \) has an inductive proof; this is because the eventuality of a component-wise difference can be represented as a trace \( t \) for which \( t \leq s_1 \) but \( t \not\approx s_2 \) (or vice versa). With \( t \approx s \) defined in the obvious way (i.e. \( t \approx s \) if \( t \cdot \_\_0^\omega \approx s \)), \( t \not\approx s \) therefore also has an inductive proof. This yields

\[ s_1 \approx s_2 \quad \text{iff} \quad \forall t. t \not\approx s_1 \iff t \not\approx s_2. \quad (\approx\text{-and-ts}) \]

\[ (2.5) \textbf{LEMMA} \ (\approx) \subseteq (\approx) \subseteq (\approx). \]

We close with a justification for focusing on stream semantics as a basis for comparing the behavior or processes. As noted earlier, actions are the only external interface to our processes. Therefore, when reasoning about security, we will model attackers’ knowledge obtained via extensional observations. Here, not allowing the attacker to observe the state of the program is natural. While branching-time equivalence (bisimulation on processes) is stricter than trace equivalence, it is too strict for our purposes, since it makes distinctions on the internal branching structure of processes [34]. We therefore opt for stream equivalence: a linear-time behavioral equivalence which subsumes trace equivalence.

2.2.4 Interaction

The input-output behavior in the LTS\_IO semantics of a program implies a particular model of interaction used by said program. Typically, secure information flow frameworks in the reactive, process algebra, and interactive program setting, adopt synchronous communication on channels as the model of communication [10, 14, 17, 21, 36, 39, 41, 42, 46]. Here, if we take the processes in these computation models and put them in interaction, then processes can block on both input and output. We argue that synchronous communication is not a good fit for general system composition, as the exhibited blocking behavior makes (compositional) reasoning about process behavior nontrivial. Progress of synchronously
interactive programs is highly context sensitive; nonwillingness of one component to receive can halt the progress of a sender component. For instance, consider the following.

Let \( p''_L \) (resp. \( p''_M, p''_R \)) denote the left (resp. middle, right) LTS above. Consider \( p''_M \) in interaction with \( p''_R \). Then it is possible for these components to synchronize on \( \texttt{b0} \). Now add \( p''_L \) to the composition, let \((p''_L, p''_M)\) synchronize first, and then let \((p''_M, p''_R)\) synchronize if they can; otherwise synchronize \((p''_L, p''_M)\) twice. Under this scheduling, it is impossible for \( p''_R \) to perform action \( \texttt{tb0} \), since whenever \( p''_R \) is willing to synchronize on \( \texttt{b} \), \( p''_M \) is not ready to do so, and vice versa. Just by adding a component \( p''_L \) which sends no messages to the parallel composition, we have enabled a cooperative starvation of \( \texttt{b0} \) by \( p''_L \) and \( p''_M \). Furthermore, under this scheduling, \( p''_R \) performs no actions, although it is given the opportunity to do so at regular intervals. However, looking at the stream of synchronizations, it is not evident that \( p''_R \) was given the opportunity to act. The same situation arises if we reverse the direction of the \( \texttt{b} \) channel. 

Fairness is thus nontrivial under synchronous communication.

In practice, output blocking is typically avoided by buffering channels, making communication asynchronous. Then, \( p''_R \) would put \( \texttt{tb0} \) into a buffer without delay and move on with its computation, while \( p''_M \) would subsequently read from the buffer whenever \( p''_M \) is ready to do so. This effect is also achieved by requiring that there is always a receiver ready for each send. This holds if each component in a composition is input total [26,29], i.e., always ready to receive input. As argued in [56], input totality simplifies reasoning about composed systems considerably, since an input-total process cannot control its environment by not desiring certain input. Input totality abstracts from how asynchrony is achieved, and thus our framework generalizes more concrete approaches e.g. input queues [51] and buffered channels [47]. To make fairness visible in a stream of an interaction between two components, we ensure each component is always able to perform an action, regardless of which environment it is in. We get this by assuming that processes are output productive, i.e., always capable of producing output. This makes our processes similar to Input/Output Automata [26], in that processes are autonomous; a process can always make progress on its productions. Like in [26], we say a stream is fair if it contains an infinite number of outputs, i.e., is in the set 

\[ S_F = \{ s \in S \mid \forall t \leq s . \exists t', o', t.t'.o' \leq s \} \]

Let \( S_F(p) = S(p) \cap S_F \). Since \( \texttt{!⋄} \) models noninteraction, receivers of \( \texttt{!⋄} \) should not react to it. Such receivers are atemporal. For further justification and merits of
this model of concurrency, see Section 2.6.

(2.6) **Definition (Interactive LTS$_{IO}$)** $p$ is

1. **input total** \( \forall i \cdot p \Downarrow \) iff \( \forall t, p' \cdot t \Downarrow p' \implies \)
2. **output productive** \( \exists o \cdot p \Downarrow \) \( o \cdot p' \Downarrow \)
3. **atemporal** \( \forall i, p'' \cdot p' \Downarrow p'' \implies p' = p'' \).

$p$ is interactive iff $p$ satisfies 1), 2) and 3).

Putting two interactive LTS$_{IO}$ in nonblocking interaction is now a simple matter of making all output of one process become the only input to the other (and vice versa). We write $p$ under $p'$ as $p' \models p$. $p$ produces $s$ under $p'$, $p' \models p \Downarrow$, iff, $p \Downarrow$ and $p' \Downarrow$. Similarly for traces. Function $\cdot^{-1}$ reverses direction of actions, that is, $\epsilon^{-1} = \epsilon$, $(tm.t)^{-1} = ?m.t^{-1}$, and $(?m.t)^{-1} = !m.t^{-1}$, and similarly for streams. We refer to $p'$ here as the environment of $p$, and we will study the behavior of $p$ under different environments when reasoning about security of $p$. This is in stark contrast with previous work on security of LTS$_{IO}$ [14, 36, 41] which considers (classes of) strategies as environments, i.e. functions of type $\mathbb{T} \rightarrow \mathcal{C} \rightarrow \mathcal{P}(\mathbb{V})$. One noteworthy feature which secure $p$-environments have which secure strategy-environments do not have is $p$ can force a secret input to occur before a public input, as input streams can in the reactive systems setting [10]; indeed, if $p'$ and $p$ are deterministic, and the interaction pattern is fixed, $p'$ will behave like an input stream. Our framework thus unifies several previous frameworks for interactive program security.

We assume $p$ is interactive throughout our development, unless stated otherwise. While these are strong restrictions to impose on an LTS$_{IO}$, any LTS$_{IO}$ can be modeled as an interactive LTS$_{IO}$ without disabling behaviors. For instance, for $p$ which can discriminate on which channel to receive on next, like the LTS$_{IO}$ in [14, 36, 41], the buffer wrapper $B^i$ associates an input queue with each channel, which $p$ can then receive on at its leisure. $B^i(p) = B^i(\epsilon, p)$ is input total.

\[
\begin{array}{c}
\begin{array}{c}
p \Downarrow p' \\
B^i(t, p) \Downarrow B^i(t, p')
\end{array} &
\begin{array}{c}
p \Downarrow \text{cv} \quad p' \Downarrow \text{cv'} \leq t \\
B^i(t, ? \text{cv}, t', p) \Downarrow B^i(t, t', p')
\end{array} &
\begin{array}{c}
p \Downarrow \text{cv} \quad p' \Downarrow \text{cv'} \leq t \\
B^i(t, p) \Downarrow B^i(t, i, p)
\end{array}
\end{array}
\]

For programs which never discriminate on which channel to receive from, like the LTS$_{IO}$ in [10, 42, 57], the FIFO wrapper $F^i$ buffers input and delivers it to $p$ on demand, in FIFO order. For such $p$, $F^i(p) = F^i(\epsilon, p)$ is input total.

\[
\begin{array}{c}
\begin{array}{c}
p \Downarrow p' \\
F^i(t, p) \Downarrow F^i(t, p')
\end{array} &
\begin{array}{c}
p \Downarrow p' \\
F^j(i, t, p) \Downarrow F^j(t, p')
\end{array} &
\begin{array}{c}
p \Downarrow p' \\
F^j(t, i, p) \Downarrow F^j(t, i, p)
\end{array}
\end{array}
\]

Any $p$ which is not output productive has the potential to block on input. The wait wrapper $W$ empowers any such $p$ with the ability to, instead of block, wait
as an internal action. For any \( p \), \( W(p) \) is output productive.

\[
\exists i, \cdot p \Downarrow \exists \circ, \cdot p \Downarrow \quad W(p) \Downarrow W(p)
\]

At last, the *atemporal* wrapper \( T \) ignores any \( \circ \) actions. \( T(p) \) is atemporal, for any \( p \).

\[
\begin{align*}
T(p) & \Downarrow T(p) & p \Downarrow p' & a \neq \circ \\
& \Downarrow & & \Downarrow 
\end{align*}
\]

### 2.3 Security of Interactive Systems

Equipped with the tools from the previous section, we develop notions for reasoning about information-flow security in our setting. We present two popular conditions of *possibilistic noninterference*: PSNI and PINI. While PSNI is well studied in our setting [14, 36, 41], we give the first formalization of PINI in a general interactive setting; PINI has thus far only been presented in the interactive setting with restricted forms of interaction [3, 10].

#### 2.3.1 Observables

The observables of a process are its effects. We assume a lattice \( (\mathcal{L}, \sqsubseteq) \), with \( \mathcal{L} \) ranged over by \( \ell \), of security levels expressing levels of *confidentiality*. Each channel \( c \) is labeled with two security levels; \( \pi(c) \) is the level of the *presence* of a message on \( c \), and \( \kappa(c) \) is the level of the *content*, or value, of a message on \( c \). In examples, we frequently represent a channel by its security levels, writing \( \kappa(c) \pi(c) \) in place of \( c \). A classic example is the lattice \( \mathcal{L} = \{L, H\} \) ("low" (public) and "high" (secret)) and \( \sqsubseteq = \{(L, L), (L, H), (H, H)\} \). We let \( L, M \) and \( H \) denote \( L^L \), \( H^L \) and \( H^H \), respectively. We let \( T \) resp. \( \bot \) denote the top resp. bottom element in the security lattice. Let \$ := ? | ! \$ and define \( \pi(\text{cv}) = \pi(c) \) and \( \kappa(\text{cv}) = \kappa(c) \). Then \( \pi(\circ) = \kappa(\circ) = T \). For termination-sensitive reasoning in this framework, set \( \pi(\star) = \kappa(\star) = \bot \) and impose a restriction similar to atemporal for termination actions.

The security labels express *who* can observe *what*. An observer is associated a security level \( \ell \). An \( \ell \)-observer is capable of observing the presence (resp. content) of messages on \( c \) if \( \kappa(c) \subseteq \ell \) (resp. \( \pi(c) \subseteq \ell \)). Let \( s \upharpoonright \ell \) be the stream where, component-wise, each action has been replaced with what an \( \ell \)-observer observes in the action.

\[
(s_{cv}, s) \upharpoonright \ell = \begin{cases} 

\cdot, s \upharpoonright \ell, & \text{if } \pi(c) \nsubseteq \ell \\

s_{cv}, s \upharpoonright \ell, & \text{if } \pi(c) \subseteq \ell \wedge \kappa(c) \nsubseteq \ell \\

s_{cv} \cdot, s \upharpoonright \ell, & \text{otherwise.}
\end{cases}
\]

For instance, \( (\text{?H}0, \text{?M}1, \text{!L}2, \text{!o}^\omega) \upharpoonright \ell = \text{?Md} \cdot \text{!L}2, \text{!o}^\omega \). Here, \( d \) is a constant. Let \( s_1 \approx_{\ell} s_2 \) iff \( s_1 \upharpoonright \ell = s_2 \upharpoonright \ell \), and \( s_1 \approx_{\ell} s_2 \) iff \( s_1 \upharpoonright \ell \approx s_2 \upharpoonright \ell \). Similarly for traces.
2.3.2 Noninterference

The idea behind noninterference is as follows. Assume an \( \ell \)-observer observes all he is privileged to observe. A process is noninterfering if, based on \( \ell \)-observables, the \( \ell \)-observer learns nothing he is not privileged to learn, i.e., unobservable input does not interfere with observable behavior\(^3\). Noninterfering processes are thereby not responsible for leaks.

To attribute a detected insecurity to the process under scrutiny, we study its behavior under secure environments. Typically, definitions of noninterference state that a process exhibits observably equivalent behavior, under any pair of noninterfering observably equivalent environments. In our setting, this leads to a circularity, since environments are processes. Previous work avoids this circularity by

a) using simpler environments for which noninterference and observational equivalence are trivial \([3, 17, 36]\),

b) defining observational equivalence on processes, and noninterference as self-equivalence \([10,48]\), or

c) defining noninterference as invariance of observable behavior to insertion / deletion of unobservable input \([24]\).

We find that none of these approaches can be applied directly to our setting. Since compositionality is a main concern in this paper, we need environments to be part of the computation model, ruling out a). Since self-equivalences are bisimulation relations, they are branching-time equivalences, rejecting e.g. the following program \( \text{plinear} \) since it can enter the “else” branch, where it can leak information, even through it can also always take the “then” branch on \( x \), where no information leaks\(^4\). We therefore find b) too strict.

\[
x = 0 | 1 \\
\text{out} \ H \ x \\
\text{poll} \ H \ h \\
\text{if} \ h = \text{UNDEFINED} \ \text{then} \ h = 0 \ \text{end if} \\
\text{if} \ x \mod 2 = 1 \\
\text{then out} \ L \ (0 | 1) \\
\text{else out} \ L \ (h \mod 2) \\
\text{end if}
\]

Approaches based on c) are defined on traces, and have similar problems as b) in that they reject the above program; inserting \( ?!H1 \) immediately after \( !\.H0 \) in \( !.H0.1^*!.L0 \) makes the subsequent \( !L0 \) impossible.

\(^3\)In our possibilistic setting, what this means is any difference in observable behavior should be attributable to nondeterministic choices.

\(^4\)Here, \text{poll} c x \) is a nonblocking input interacting with a buffering context, similar to \text{if-receive} in \([47]\). If a c-input is waiting in the buffer, consume it. Otherwise, write \text{UNDEFINED} to \( x \).
What we desire is a property which stipulates that a process can (by making “the right” nondeterministic choices) preserve the possibility of a sequence of observables, under insertion of unobservable input during execution. To ensure that a stream is not only possible due to the presence of unobservable input, we require the above for streams which contain no unobservable input, i.e., streams over 
\[ A_\ell = \emptyset \cup \{ ?cv \mid \pi(c) \subseteq \ell \wedge (\kappa(c) \not\subseteq \ell \implies v = d) \}. \]
Since a process cannot leak information if denied the opportunity to produce output, we focus on fair runs of processes. We define the above as a coinductive predicate as follows.

\[ (2.7) \text{ Definition} \quad p_0 R_\ell \text{-preserves the possibility of } s_0 \text{ after } t \text{ through } s, \quad \text{preserve}_{p_0,s_0}^{R_\ell}(t,s), \]
is the largest predicate satisfying each of the following.

1) \[ \forall o \leq s \cdot \exists s' \in A_\ell^o. \quad s_0 R_\ell t.o.s' \wedge p_0 \xrightarrow{t.o.s'} \wedge \text{preserve}_{p_0,s_0}^{R_\ell}(t.o,s') \]
2) \[ \forall i \leq \ell s \cdot \exists s' \in A_\ell^o. \quad s_0 R_\ell t.i.s' \wedge p_0 \xrightarrow{t.i.s'} \wedge \text{preserve}_{p_0,s_0}^{R_\ell}(t.i,s') \]
3) \[ \forall i \simeq \ell s \cdot \exists s' \in A_\ell^o, i \leq i.o. \quad s_0 R_\ell t.i.o.s' \wedge p_0 \xrightarrow{t.i.o.s'} \wedge \text{preserve}_{p_0,s_0}^{R_\ell}(t.i.o,s') \]

The intuition behind 1), 2) and 3) is as follows. Consider a \( p \) and a fair \( s \) for which \( p \nleftrightarrow s \), and assume \( s \simeq \ell s' \) for some fair \( s' \in A_\ell^o \). Consider \( \text{preserve}_{p,s}^{\leq}(\epsilon,s') \). Then 1) corresponds to the scenario where the environment feeds no input into \( p \), so to match \( s, p \) merely emits \( s' \), since we already have that \( s \simeq \ell s' \). However, \( p \) may eventually need low-presence input to match \( s \). Further, if \( p \) expects \( ?M \) instead for any \( v \in \mathbb{V} \). Finally, at any point, the environment can introduce a \( ?Hv \) into \( p \). To match the rest of \( s \) under these circumstances, \( p \) is allowed to choose a different \( s' \) in \( A_\ell^o \) matching the rest of \( s \). However, the choice of \( s' \) for \( p \) to follow affects crucially the ability of \( p \) to eventually perform a desired future action. Our property must ensure that this eventuality not only holds for the “right” \( H \) input (or lack thereof) or values in \( H \) messages; the possible eventuality must be independent on \( H \) input. When devising our property, we encountered three notable scenarios which our property needed to deal with to guarantee eventuality of certain actions of interest: i) scheduling, ii) high interaction loops, and iii) high output starvation. In i), since both \( p \) and its environment can be the producer behind the \( k \)th action in an interaction, \( p \) might only be able to choose a matching \( s' \) if \( p \) produces the next message. However, the demand \( p \) places on its possibilistic scheduling may conflict with
the demands the environment places on the scheduling to preserve security, i.e. when both \( p \) and its environment only preserve confidentiality if they each are the producer behind the first message in the interaction. Having one yield fully to the demands of the other will make the security property stipulate security for all schedulers, which, while interesting\(^5\), we find to be too conservative; we wish for our scheduler to be possibilistic (and therefore a source of nondeterminism which can conceal information leaks). In ii), future observables can be starved as a result of either unfortunate scheduling, or \( H \) input. To illustrate, consider e.g. the following process \( p_{\text{loop}} \).

![Diagram of \( p_{\text{loop}} \)](image)

After receiving \( ?H0 \), this process insists on outputting \( H0 \) before outputting \( L0 \). If the environment insists (to not leak) on outputting \( H0 \) before receiving \( L0 \), action \( !L0 \) is deferred, possibly indefinitely, as is possible when \( p_{\text{loop}} \) interacts with a variant of itself which first outputs \( H0 \). The process and its environment here are engaged in a livelock, exhibiting behavior reminiscent of a “hallway dance”, or more accurately, an Alphonse-Gaston routine\(^6\) [37]. In the presence of high interaction loops, security must ensure that \( p \) can eventually produce its next observable, regardless of what environment \( p \) is run under. Dually, the environment must tolerate receiving the next observable at any point in time, as different processes under it can demand high interaction loops terminated at different times. Thus 2), along with ensuring that \( s \) remains possible for arbitrary insertion of high input\(^7\), also guarantees possible eventual termination of high interaction loops between a secure \( p \) interacting with a secure environment (at the whim of the producer of the next observable). Finally, 3) is designed to addressiii), e.g., to ensure that after \( p \) has produced all its observables, it can still be scheduled fairly. Consider this example of high output starvation, \( p_{\text{starve}} \).

![Diagram of \( p_{\text{starve}} \)](image)

Consider environments \( p_1 = F^3(\text{Md}^\omega) \) and \( p_2 = F^3(\text{M}0^\omega) \). Then \( p_1 \models p_{\text{starve}} \) can match \( ?M0^\omega \) fairly, while \( p_2 \models p_{\text{starve}} \) cannot; while \( p_{\text{starve}}(\text{Md}!H0)^\omega \), the first \( i \leq_f (\text{Md}!H0)^\omega \) fed to \( p_{\text{starve}} \) by \( p_2 \) is \( i = ?M0 \), making the matching stream with \( i \) as first action \( ?M0.(\text{Md}!H0)^\omega \); compared to \( (\text{Md}!H0)^\omega \), the first output has been deferred. Indeed, the only way for \( p_2 \models p_{\text{starve}} \) to match \( ?M0^\omega \) is to not produce output at all, thus starving \( p_{\text{starve}} \). To address this, 3) requires that when \( p \) has produced all observable output and is being fed a sequence of input, \( p \) can

---

\(^5\) Since it implies security under scheduler refinement, studied e.g. in [58].

\(^6\) “After you”, followed by a back-and-forth “No, you first.” ad infinitum.

\(^7\) For all \( s, v \) and \( c \) for which \( \pi(c) \not\subseteq v \). If \( \leq_f \) holds for all \( s, v \) and \( c \), then the next observable in \( s \) is an input.
at some point cut into the (otherwise infinite) stream of input \( \tilde{i} \) after a (finite) trace of inputs \( \tilde{i} \leq \tilde{i} \) and produce an unobservable output \( \epsilon \) while still preserving possibility of \( s \).

Our “preservation-based noninterference” approach can be viewed as a hybrid of b) and c). In contrast to b), we do not require security to hold in all reachable states. In contrast to c), we do not require that each \( s \in \mathcal{S}(p) \) satisfies insert/delete conditions; only that some \( \tilde{s} \in \mathcal{R}_{\ell} s \) does, that is, that \( p \) can make the right nondeterministic choices to preserve the possibility of \( \mathcal{R}_{\ell} s \).

\[
(2.8) \text{DEFINITION } \quad p \text{ is } \mathcal{R}\text{-noninterfering iff } \forall \ell. \forall s \in \mathcal{S}_{\ell}(p). \exists s' \in \mathcal{S}(p) \cap \mathcal{A}_{\ell} s \mathcal{R}_{\ell} s' \land \text{preserve}_{\ell}(s, s').
\]

\[
(2.9) \text{DEFINITION } \quad p \in \text{PNI if } p \text{ is } \approx\text{-noninterfering.}
\]

\[
(2.10) \text{DEFINITION } \quad p \in \text{PINI if } p \text{ is } \approx\text{-noninterfering.}
\]

Definition 2.10 is the first definition of \text{PINI} in a general nondeterministic interactive setting. It differs from previous \text{PINI} formalizations [3, 4, 10] in that input to the process is not fixed before the process is run; rather, the environment is permitted to adapt its input based on prior process output. Our definition can be improved, however; consider \( p_{\text{echo}} \), a process which outputs anything it receives in FIFO order (outputs \( \triangledown \) when its FIFO is empty), except when \( \triangledown \mathcal{H}_0 \) is received; then \( p_{\text{echo}} \) immediately becomes \( \mathcal{F}((\triangledown \mathcal{H}_0)) \). Consider environments \( p_1 = \mathcal{H}_0 \triangledown \mathcal{H}_0 \) and \( p_2 = \mathcal{H}_0 \triangledown \mathcal{H}_0 \). Then \( p_1 \models p_{\text{echo}} \mathcal{F}((\mathcal{H}_0 \triangledown \mathcal{H}_0)) \), which is \( \approx \)-matched by \( p_2 \models p_{\text{echo}} \mathcal{H}_0 \triangledown \mathcal{H}_0 \mathcal{H}_0 \). However, the moment \( p_2 \models p_{\text{echo}} \) inputs twice after \( \triangledown \mathcal{H}_0 \), \( \approx \)-equivalence with \( (\mathcal{H}_0 \triangledown \mathcal{H}_0) \) is lost. Thus, in general, a \( p \) satisfying Definition 2.10 might have to starve the environment to \( \approx \)-match a stream. This is not unlike \text{id}-security in [10], which allows a reactive system to ignore an observable input ready in the environment by diverging silently while reacting to a previous input. While we could adjust our definition of \( \approx \) to cover the above scenario, the main reason we introduce \text{PINI} is to study its compositionality properties, and the adjusted \text{PINI} would fail to be compositional in the ways our \text{PINI} does.

\( \text{PSNI} \) is strictly stronger than \text{PINI}. This follows from Definition 2.8, Lemma 2.5 and this program which, wrapped in \( W \circ B' \circ Z \), is in \text{PINI} \setminus \text{PSNI}.

\[
in H h
\]

\[
\text{if } (h \text{ mod } 2) = 0 \text{ then out } L 0 \text{ end if}
\]

\[
(2.11) \text{LEMMA } \quad p \in \text{PSNI} \implies p \in \text{PINI}
\]

Pt. 2) in Definition 2.7 enables a simple proof technique for guaranteeing eventualty of observable actions in an interaction: schedule the producer of the next observable action. To see this technique in action, we refer the reader to the proofs of Theorems 2.23 and 2.16 in the appendix and Section 2.4.4 respectively.

### 2.3.3 Noninterference Under Environments

To facilitate evaluation of the relative merits of our preservation-based formalization of progress-(in)sensitive noninterference, and to demonstrate the generality
of our framework, we give more conventional definitions of psni and pini under environments à la [14, 36, 41].

(2.12) **Definition** \( p_2 \mathcal{R}_E\text{-simulates} p_1 \) iff
\[
\forall s_1 \in S_F(p_1) \cdot \exists s_2 \in S_F(p_2) \cdot s_1 \mathcal{R}_E s_2.
\]
p_1, p_2 are \( \mathcal{R}_E\)-equivalent, \( p_1 \equiv_{\mathcal{R}_E} p_2 \), iff, \( p_1 \mathcal{R}_E\text{-simulates} p_2 \) and vice versa. 

(2.13) **Definition** \( p_2 \mathcal{R}_E\cdot E\text{-simulates} p_1 \) iff
\[
\forall p'_1, p'_2 \in \text{psni} \cdot p'_1 \equiv_{\mathcal{R}_E} p'_2 \Rightarrow
\forall s_1 \in S_F \cdot p'_1 \models p_1 
\Downarrow
\exists s_2 \in S_F \cdot p'_2 \models p_2 \Rightarrow \wedge s_1 \mathcal{R}_E s_2.
\]

(2.14) **Definition** \( p \in \text{psni}_E \) iff \( \forall E \cdot p \equiv_{\mathcal{R}_E} \text{-}E\text{-simulates} p \).

(2.15) **Definition** \( p \in \text{pini}_E \) iff \( \forall E \cdot p \equiv_{\mathcal{R}_E} \text{-}E\text{-simulates} p \).

We use psni environments in the definition of piniE because otherwise, piniE becomes too conservative. To see this, say we defined piniE by relaxing \( \equiv_{\mathcal{R}_E} \) to \( \equiv_{\mathcal{R}_E} \) and environment assumption psni to pini in the definition of psniE. Now consider the following process \( p_{\text{piniE}} \in \text{psni} \).

Consider environments \( p_1 = F^i(\text{!L0.}!\omega) \) and \( p_2 = F^i(\text{to}^{\omega}) \), for which we have \( p_1, p_2 \in \text{pini} \) and \( p_1 \equiv_i p_2 \). However, while \( p_1 \models p_{\text{piniE}} \Rightarrow F^i(\text{!L0.}!\omega) \), \( p_2 \models p_{\text{piniE}} \) cannot fairly either match these observables or remain silent; eventually, an \text{!L1} occurs, and so, for any stream \( s \in S_F \) for which \( p_2 \models p_{\text{piniE}} \Rightarrow \text{!L1} \), \( p \neq \mathcal{R}_E \text{!L0.}!\omega \). So, while replacing \( \equiv_{\mathcal{R}_E} \) in psniE with \( \equiv_{\mathcal{R}_E} \) weakens the security definition, relaxing the assumptions on the environments to pini strengthens it immensely; the behavior of a process would need to be invariant to L input as well as H input to satisfy piniE.

For our preservation-based definitions to be useful on its own, they need to be stronger than the standard environment-based definitions; then we will know that processes satisfying our preservation-based definitions are safe against all attacks which processes satisfying the environment-based definition are safe against. This turns out to be the case. The proof is in the appendix.

(2.16) **Theorem** \( p \in \text{psni} \Rightarrow p \in \text{psni}_E \)

(2.17) **Theorem** \( p \in \text{pini} \Rightarrow p \in \text{pini}_E \)

We suspect the reverse implication of these theorems to be false. To show that \( p \) satisfies the constraint imposed by Definition 2.7 pt. 2) using assumption \( p \in \text{pini}_E \), it seems we need to propose an environment which outputs a different observable if it receives an unobservable first. However, such an environment is
not $\text{psni}$ (resp. $\text{pini}$). These theorems give us a sense of assurance, however; if a property is too weak, say, $\mathcal{P} = \{ p \in \text{LTS}_{\text{IO}} \mid p \text{ is interactive} \}$, then $p \in \mathcal{P} \iff p \in \text{PE}$, since $\text{PE}$ places demands on $p$ beyond $p \in \mathcal{P}$.

For similar reasons as for Lemma 2.18, $\text{psni} \text{E}$ is strictly stronger than $\text{pini} \text{E}$.

(2.18) **Lemma**  $p \in \text{psni} \text{E} \implies p \in \text{pini} \text{E}$

Finally, we consider to which extent $\text{psni} \text{E}$ (resp. $\text{pini} \text{E}$) permits processes to starve the environment to preserve confidentiality. A process starving the environment is exerting control over the possibilistic scheduling of processes, which violates our desire for processes to be autonomous. Therefore, ideally, $\text{psni} \text{E}$ and $\text{pini} \text{E}$ should reject processes which might need to starve the environment to preserve confidentiality. We say $p$ produces $s$ fairly under $p'$, $p' \models_{\text{F}} p \Rightarrow s$, iff $p' \models p \Rightarrow s$ and $s^{-1} \in \mathcal{S}_{\text{F}}$. Now let $\text{psni} \text{EF}$ (resp. $\text{pini} \text{EF}$) be defined as $\text{psni} \text{E}$ (resp. $\text{pini} \text{E}$) with "$\models$" replaced by "$\models_{\text{F}}$". It turns out that $\text{psni} \text{EF}$ and $\text{psni} \text{E}$ are equivalent, and thus, that $p \in \text{psni} \text{E}$ preserves interaction fairness when matching behaviors. However, as hinted at earlier, due to the way we formalized $\text{pini} \text{E}$, $p \in \text{pini} \text{E}$ may need to starve $p_2 \approx t p_1$ to match a behavior that $p_1 \models p$ can perform.

(2.19) **Theorem**  $p \in \text{psni} \text{EF} \iff p \in \text{psni} \text{E}$

(2.20) **Theorem**  $p \in \text{pini} \text{EF} \iff p \in \text{pini} \text{E}$

2.3.4 **Contrast to Trace-based Properties**

Finally, to emphasize the novelty of our security properties, we demonstrate that they rule out classes of attacks which trace-based security properties, classically considered in work on compositionality in event systems [27,31,56], do not guarantee protection from. Consider the following program, which we refer to as the extortionist.

```
repeat
  poll H h
until h ≠ undefined
out L 0
```

This program, turned into the process $p_{\text{extort}}$ by buffering input, will repeatedly attempt a read on $H$ in a nonblocking manner from its buffering context; if no $H$ input is available in the context, then the program outputs $L 0$.

We have $F^2(\langle H \omega \rangle) \models p_{\text{extort}} \xrightarrow{(0, \langle H \rangle)\omega}$. However, we have that $F^2(\langle 0 \rangle) \models p_{\text{extort}}$ cannot match this behavior, and thus, $p_{\text{extort}} \notin \text{psni} \text{E}$. Indeed, since there is no fair stream in $\langle H \omega \rangle$ matching this behavior, we also have $p_{\text{extort}} \notin \text{psni}$.

However, the above program does, for instance, satisfy forward correctability [24]. This is due to the fact that it is defined in terms of a trace semantics, and therefore cannot properly deal with the definite (non)eventuality of $\langle 0 \rangle$0. Pick a trace. If it has the $L$ output, and you insert the $H$ input anywhere, with no $H$ input after it, then eventually, the $L$ output will emerge, since no more $H$ input follows. Remove a $H$ input, and the same happens. If the trace does not have the
Chapter 2 : Compositional Information-flow Security for Interactive Systems

2.4 Compositional Security

To study how our security properties behave under composition, we present a minimal combinator language for building systems from parts. The core of this language is complete in the sense that arbitrary wirings between components can be constructed, yet structured in the sense that the possible routes that data can take in the composed system are clearly defined by the combinators used (as opposed to being partially defined by the (un)willingness of components to synchronize on certain channels at different times).

2.4.1 First Attempt

A minimal approach to enable two processes in a composed system to interact is to introduce a loop combinator \( \lceil \cdot \rceil \), illustrated in Figure 2.1. This is the approach taken in functional reactive programming [35], where loops and simple products together enable modeling of arbitrary wirings.

\[
p := \ldots | \lceil p \rceil
\]

In \( \lceil p \rceil \), an output from \( p \) is sent to the environment, and at the same time, a copy of said output is sent back into \( p \) as input. Any input to \( \lceil p \rceil \) is handed to \( p \). The semantics of \( \lceil \cdot \rceil \) is as follows.

\[
\begin{align*}
p \xrightarrow{\omega} p' & \quad p' \xrightarrow{\omega^{-1}} p'' \quad \lceil \cdot \rceil \\
[\lceil p \rceil] \xrightarrow{\omega} [p''] & \quad [p'] \xrightarrow{\omega} [p'']
\end{align*}
\]

However, convenient as this combinator is, it enables a process to engage in a high interaction loop with itself. Consider again \( p_{\text{loop}} \), a process satisfying \( \text{PINI} \). While it is possible to schedule \( p_{\text{loop}} \) under an arbitrary environment such that \( !L0 \) eventually occurs, this is not the case for \( \lceil p_{\text{loop}} \rceil \); while \( F^3(\lceil !L0.(!H0) \rceil) \models [p_{\text{loop}}] !L0.(!H0) \), \( F^3(\lceil !L0.(!H0) \rceil) \models [p_{\text{loop}}] \) cannot match this behavior; to get to the \( ?L0 \) without producing \( !L0 \), \( p_{\text{loop}} \) must consume \( ?H0 \). This sends \( p_{\text{loop}} \) to its leftmost state, where it returns directly after performing \( !H0 \), as the next action of \( p_{\text{loop}} \) is invariably \( !H0 \). While this poses no problems for \( \text{PINI} \) (since a
2.4 Compositional Security

The issue with \([\_]\) is that it can prevent the source of an output from making progress on its observable productions, by immediately following each output it makes with an input sent directly to the source, in one step of the whole system. Therefore, if our combinators are to compose under PSNI, they need to at most enable output to reach any part of the system except its source, in one step of the whole system. We provide two combinators which we deem to be core combinators, sufficient to construct arbitrary such wirings: “and”, and “route”.

\[ p ::= \ldots | p \odot p | r(p) \]

These each take (possibly compound) interactive LTS\(_{\text{IO}}\) as parameter and yield a compound interactive LTS\(_{\text{IO}}\). The structure that they impose is illustrated in Figure 2.2.

2.4.2 Core Combinators

The and combinator produces a composite system from parts. An input to \(p_1 \odot p_2\) is sent to both \(p_1\) and \(p_2\), while output from \(p_1 \odot p_2\) comes from exactly one of \(p_1\) and \(p_2\), and is copied into the other as input, thus exhibiting feedback. This combinator is the enabler of communication, functioning as the “glue” with which we wire together larger systems from parts. Our \(\odot\) combinator most closely resembles the (full, arbitrary, hook-up) binary composition typically used in event-based formalisms [27], or alternatively, a broadcasting variant [40] of parallel composition in process algebra [17]. The semantics of \(\odot\) is as follows. It is clear that \(\odot\) is associative and commutative.

\[
\begin{align*}
  p_L &\xrightarrow{\odot} p'_L, & p_R \xrightarrow{\odot^{-1}} p'_R, & p_L \xrightarrow{\odot} p'_L, & p_R \xrightarrow{\odot^{-1}} p'_R, \\
  p_L \odot p_R &\xrightarrow{\odot} p'_L \odot p'_R, & p_L \odot p_R &\xrightarrow{\odot^{-1}} p'_L \odot p'_R, & p_L \odot p_R &\xrightarrow{\odot} p'_L \odot p'_R.
\end{align*}
\]
The router combinator \( \langle \rangle \) wraps an LTS in a context which routes messages. The routing is defined by a router function \( r : A \rightarrow A \) satisfying \( r(\mathbb{I}) \subseteq \mathbb{I} \), \( r(\mathbb{O}) \subseteq \mathbb{O} \) and \( r(\emptyset) = \emptyset \). Then in \( r(p) \), an output \( o \) leaving \( p \) is replaced with \( r(o) \), and when an input \( i \) arrives at \( r(p) \), \( r(i) \) is received by \( p \). Whereas \( \oplus \) is an enabler of flows, where a composed system wires each output to each component (save the output source), \( \langle \rangle \) can be used to control which underlying component receive which input, and to hide certain output from certain components. One example of the use of \( r \) is to map input actions on a particular channel (i.e. carrying high data) to \(?\emptyset\) to put the channel out of scope of the wrapped process. The semantics of \( \langle \rangle \) are the following:

\[
\frac{p \xrightarrow{r(i)} p'}{r(p) \xrightarrow{r(i)} r(p')}
\]

\[
\frac{p \xrightarrow{r(o)} p'}{r(p) \xrightarrow{r(o)} r(p')}
\]

### 2.4.3 Point-to-Point Variant

Instead of our broadcasting combinator semantics, one could alternatively opt for one which, instead of sending a message along both branches of a split arrow, sends it along exactly one of them. This yields a point-to-point message-passing semantics (which is still asynchronous), represented by the “xor” combinator.

\[
p ::= \ldots | p \boxplus p
\]

### Xor

The xor combinator \( \boxplus \) is similar to parallel composition in point-to-point message-passing formalisms [17]. Any action performed by \( p_L \boxplus p_R \) is performed by exactly one of \( p_L \) and \( p_R \). The semantics of \( \boxplus \) is as follows. It is clear that \( \boxplus \) is associative and commutative.

Using \( \boxplus \) and \( \langle \rangle \) as the combinator core would be viable in a synchronous concurrency model; there, input is only delivered to an intended receiver. However,
in a setting where each component always waits for input on every channel, \(\oplus\) nondeterministically pick a component to receive the input. The input can therefore be sent along a branch in the composition which is not intended to receive the input (a constraint modeled using a router which maps it to \(?\circ\)) and therefore never reach the intended target. We therefore find that \(\oplus\) is not a good fit in our framework – at least not as a replacement for \(\otimes\). There are some merits to including \(\oplus\) in a language based on our combinator core, e.g. if, at the combinator level, one wishes to model nondeterministic dispatching of input to servers.

### 2.4.4 Compositionality

We now explore the compositionality of our security properties, to then give a language of secure combinators for building secure systems from secure parts. While security properties are known to be fragile under composition [31], the proof technique arising from the design of our security properties yields positive results.

And

We begin with the most important combinator, \(\otimes\). It composes under \(\text{PSNI}\).

(2.23) **Theorem** \(p_L, p_R \in \text{PSNI} \implies p_L \otimes p_R \in \text{PSNI} \).

The proof of this is as follows. For any \(p_L \otimes p_R \not\rightarrow\), we must show existence of a \(p_L \otimes p_R \not\rightarrow s\) for which \(s' \in A_L^{\omega,s}, s \equiv_{\ell} s'\) and \(\text{preserve}_{p_L \otimes p_R}(\epsilon, s')\). Since \(p_L \otimes p_R \not\rightarrow\), there are some \(s_L, s_R\) for which \(p_L \not\rightarrow s_L, p_R \not\rightarrow s_R\) and \(s_L \not\rightarrow s_R\). Indeed, for any \(n\)th action \(a_L, a_R\) and \(a\) in \(s_L, s_R\) and \(s\) respectively, if \(a = i\) (environment input) for some \(i\), then \(a_L = i = a_R\), and if \(a = o\) (component output), then either \(a_L = o\) and \(a_R = o^{-1}\), or \(a_L = o^{-1}\) and \(a_R = o\).

We obtain \(s'\) by “zipping” \(s'_L\) and \(s'_R\) in a manner guided by the observables in \(s_L\) and \(s_R\) (observables in both are the same as observables in \(s\)) modulo direction) as follows:

Assume for \(s'_L, s'_R \in A_L^{\omega}, t_L, t_R, t\) that \(s_L \equiv_{\ell} t_L, s'_L, s_R \equiv_{\ell} t_R, s'_R, p_L \not\rightarrow s_L, p_R \not\rightarrow s_R\), where \(p_L \not\rightarrow p_{L,s_L}(t_L, s'_L), p_R \not\rightarrow p_{R,s_R}(t_R, s'_R)\). We show existence of \(s''_L, s''_R \in A_L^{\omega}, t'_L, t'_R, t'\) for which \(s_L \equiv t'_L, s'_L, s_R \equiv_{\ell} t'_R, s'_R, p_L \not\rightarrow s''_L, p_R \not\rightarrow s''_R\), where \(p_L \not\rightarrow p_{L,s_L}(t'_L, s''_L), p_R \not\rightarrow p_{R,s_R}(t'_R, s''_R)\), and \(\text{preserve}_{p_{L,s_L}}(t'_L, s''_L, t' \in A_L^{\omega}, t' \equiv_{\ell} s, t'_L \equiv_{\ell} t'_R, t'_L < t'_R, t' < t, \text{ and } s''_L \not\equiv_{\ell} s''_R \implies t \not\equiv_{\ell} t'\). (\(\ast\)).

Assume \(o_L \equiv_{\ell} s\) for some \(o_L \not\equiv_{\ell} s\) (proof for \(o_R \equiv_{\ell} s\) case obtained by swapping L and R). Then \(s''_L = o_L.o_L.s''\) for some \(o_L \equiv_{\ell} s\) and \(s''_L\). Through repeated application of Def. 2.7 pt. 1), we get \(s_L \equiv t_L.o_L.o_L.s''_L, p_L \not\rightarrow t_L.o_L.o_L.s''_L, p_L \not\rightarrow p_{L,s_L}(t_L, o_L.o_L.s''_L)\). Through repeated application of Def. 2.7 pt. 2), we have some \(s''_R\) for which we have \(s_R \equiv t_R.o_R.o_R.s''_R, p_R \not\rightarrow t_R.o_R.o_R.s''_R, p_R \not\rightarrow p_{R,s_R}(t_R, o_R.o_R.s''_R)\). Further, we have that \(t_L.o_L.o_L.o_L.s''_L, t_R.o_R.o_R.o_R.s''_R\), and \(p_L \not\rightarrow p_{L,R}(o_L.o_L, o_R.o_R)\). Set \(s''_L = s''_L, s''_R = s''_R, t'_L = t_L.o_L.o_L, t'_R = t_R.o_R.o_R, t' = t.o_L.o_L, and we get (\(\ast\)).
Assume $i \leq \ell \hat{s}_L$ and $i \leq \ell \hat{s}_R$ for some $i \notin \bar{s}$ and $i \in \mathbb{A}_\ell$. Through single application of Def. 2.7 pt. 2), we have for some $\hat{s}_L' \in \mathbb{A}_\ell^\omega$ that $s_L \cong \ell t_L.,i.,\hat{s}_L', p_L \leftarrow t_L.,i.,\hat{s}_L'$. Through single application of Def. 2.7 pt. 2), we have for some $\hat{s}_R'$ that $s_R \cong \ell t_R.,i.,\hat{s}_R', p_R \leftarrow t_R.,i.,\hat{s}_R'$, and preserve $p_{L,s_L}(t_L,\hat{s}_L')$ (the same argument with $L$ and $R$ swapped also holds). Further, we have that $t_L.,i.,\hat{s}_L' \leftarrow t_L.,i.,\hat{s}_L'$ and $p_L \oplus p_R \leftarrow t_L.,i.,\hat{s}_L'$. Set $\hat{s}_L' = \hat{s}_L', \hat{s}_R' = \hat{s}_R'$, $t_L' = t_L.,i.,\hat{s}_L'$, $t_R' = t_R.,i.,\hat{s}_R'$, and $t' = t,i,$ and we get ($\ell$).

Assume $\hat{s}_L \cong \ell \hat{s}_R \cong \ell \bar{s}$ (equally valid proof obtained by swapping $L$ and $R$ in the following). Then $\hat{s}_L = o_L, \hat{s}_L''$ for some $o_L \cong \ell \bar{s}$ and $\hat{s}_L''$. Through single application of Def. 2.7 pt. 1), we have that $s_L \cong \ell t_R.,o_L.,\hat{s}_L''$, $p_L \leftarrow t_R.,o_L.,\hat{s}_L''$, and preserve $p_{L,s_L}(t_L,\hat{s}_L'')$. Through single application of Def. 2.7 pt. 2), we have for some $\hat{s}_R''$ that $s_R \cong \ell t_R.,o_L.,\hat{s}_R''$, $p_R \leftarrow t_R.,o_L.,\hat{s}_R''$, and preserve $p_{R,s_R}(t_R,\hat{s}_R'')$. Further, we have that $t_L.,o_L.,\hat{s}_L'' \leftarrow t_R.,o_L.,\hat{s}_R''$, and $p_L \oplus p_R \leftarrow t_R.,o_L.,\hat{s}_R''$. Set $\hat{s}_L'' = \hat{s}_L''$, $\hat{s}_R'' = \hat{s}_R''$, $t_L' = t_R.,o_L.,\hat{s}_L''$, $t_R' = t_R.,o_L.,\hat{s}_R''$, and $t' = t.o_L.$, and we get ($\ell$).

Since $p_{L,R} \in \text{PSNI}$, there exist $p_L \leftarrow t_L.$, $p_R \leftarrow t_R.$ for which $s_L', s_R', \in \mathbb{A}_L^\omega$, $s_L \cong \ell s_L', s_L \cong \ell s_L'$, preserve $p_{L,s_L} (s_L', s_L)$ and preserve $p_{R,s_R} (s_R', s_R)$. Then ($\ell$) gives us a list of traces $t_0 < t_1 < \ldots$ for which $p_L \oplus p_R \leftarrow t_j$, and $t_j \in \mathbb{A}_L^\omega$ for all $j \geq 0$. Let $s'$ be the fixed point of these traces. Then $p_L \oplus p_R \leftarrow t_j$, and $s' \in \mathbb{A}_L^\omega$, $s \cong \ell s'$. To establish preserve $p_{L,R}(s', s')$, we proceed as follows. Assume for $\hat{s}_L, \hat{s}_R \in \mathbb{A}_L^\omega$, $t_L, t_R, t$ that $s_L \cong \ell t_L, \hat{s}_R \cong \ell t_R, \hat{s}_R, p_L \leftarrow t_L, s_L \leftarrow t_L, t, s_L \leftarrow t_L, t, s_L$, and preserve $p_{L,R} (t_L, \hat{s}_L, \hat{s}_R, \hat{s}_R)$, preserve $p_{R,s_R} (t_R, \hat{s}_R, \hat{s}_R, \hat{s}_R)$, preserve $p_{R,s_R} (t_R, \hat{s}_R, \hat{s}_R, \hat{s}_R)$. To prove preserve $p_{L,R}(s', s')$, we must show for each of Def. 2.7 pt. 1)-3) that there exist $s' \in \mathbb{A}_L^\omega$, $t' \in \mathbb{A}_L^\omega$, $s \cong \ell s'$, and $p_L \oplus p_R \leftarrow t'.s'$. As prescribed, we let $s_L \cong \ell t_L, \hat{s}_L \cong \ell t_R, \hat{s}_R, p_L \leftarrow t_L, s_L \leftarrow t_L, s_L$, and $t' \in \mathbb{A}_L^\omega$, $s \cong \ell s'$, and $p_L \oplus p_R \leftarrow t'.s'$. Further, we have that $t_L.,o_L. \leftarrow t_R.,o_L. \leftarrow t_L.\leftarrow t_R.\leftarrow t_O.$ and $p_L \oplus p_R \leftarrow t_O.$ Set $\hat{s}_L' = \hat{s}_L', \hat{s}_R' = \hat{s}_R'$, $t_L' = t_L.,o_L., \hat{s}_L'$, $t_R' = t_R.,o_L., \hat{s}_R'$, and use the above-described approach to obtain $s'$ from these, and we get (+).

For pt. 2), assume $i \leq \ell \hat{s}_L$ for some $i$. Then either $o \leq \hat{s}_L$ or $o \leq \hat{s}_R$. As prescribed, we let $o \leq \hat{s}_L$ (and that $o \leq \hat{s}_L$ came from $p_{L,R}$ during construction of $s'$). Since preserve $p_{L,s_L}(t_L, \hat{s}_L)$, we have through single application of Def. 2.7 pt. 1) that for some $\hat{s}_L' \in \mathbb{A}_L^\omega$, $s_L \cong \ell t_L.,i.,\hat{s}_L'$, $p_L \leftarrow t_L.,i.,\hat{s}_L'$, and preserve $p_{L,s_L}(t_L, \hat{s}_L')$. Since preserve $p_{R,s_R}(t_R, \hat{s}_R)$, we get through single application of Def. 2.7 pt. 2) some $\hat{s}_R'$ for which $s_R \cong \ell t_R.,i.,\hat{s}_R'$, $p_R \leftarrow t_R.,i.,\hat{s}_R'$, and preserve $p_{R,s_R}(t_R, \hat{s}_R')$. Further, we have that $t_L.,i., \leftarrow t_R.,i., \leftarrow t_L.,i., \leftarrow t_R.,i., \leftarrow t_o.$ and $p_L \oplus p_R \leftarrow t_o.$ Set $\hat{s}_L' = \hat{s}_L', \hat{s}_R' = \hat{s}_R'$, $t_L' = t_L.,i., \hat{s}_L'$, $t_R' = t_R.,i., \hat{s}_R'$, and use the above-described approach to obtain $s'$ from these, and we get (+).

For pt. 3), assume $\ell \cong \hat{s}_L$ for some $\ell$ By preserve $p_{L,s_L}(t_L, \hat{s}_L)$ (the same argument with $L$ and $R$ swapped also holds), we get through single application of
Def. 2.7 pt. 3) some $\tilde{i}$, $o$ and $s'_L$ for which $\tilde{i} \leq \tilde{i}$, $s_L \cong t_{L,i,o,s'_L}$, $p_L \overset{t_{L,i,o,s'_L}}{\rightsquigarrow}$, and preserve$_{p_L,t_{L,i,o,s'_L}}(t_{L,i,o,s'_L})$. By repeated application of pt. 2) for each $i$ in $i$ and for $o^{-1}$, we have for some $s''_R$ that $s''_R \cong t_{R,i,o^{-1},s''_R}$, $p_R \overset{t_{R,i,o^{-1},s''_R}}{\rightsquigarrow}$, and preserve$_{p_R,t_{R,i,o^{-1},s''_R}}(t_{R,i,o^{-1},s''_R})$. Set $s''_L = s''_R$, $s'''_R = s''''_R$, $t''_L = t_{L,i,o}$, $t''_R = t_{R,i,o^{-1}}$, $t' = t_i.o$, and use the above-described approach to obtain $s'$ from these, and we get (+).

Thus preserve$_{p_L,t_{L,i,o,s'_L},(\epsilon,s')}$, which completes this proof.

Furthermore, it turns out that PNI composes under $\otimes$, if we e.g. change the way $s'_L$ and $s_R$ are zipped to form $s'$ such that if one runs out of observables, say, $s'_L$, then the rest of $s'$ is set to the rest of $s'_R$.

(2.24) COROLLARY $p_L, p_R \in$ PNI $\implies p_L \otimes p_R \in$ PNI.

Xor

The proof that PSNI and PNI compose under $\oplus$ is similar to the above “zip-and-preserve” proof of compositionality of $\otimes$. Since environment input does not enter both components, and since output from a component does not both go to the environment and the other component, the zipping procedure consults $s$ in addition to $s_L$ and $s_R$ since $s$ tells us where observables in $s_L$ and $s_R$ came from and went to.

(2.25) COROLLARY $p_L, p_R \in$ PSNI $\implies p_L \oplus p_R \in$ PSNI.

(2.26) COROLLARY $p_L, p_R \in$ PNI $\implies p_L \oplus p_R \in$ PNI.

Router

Since a router can route any message to any channel, and change values in messages, a router has the capacity to reveal the presence of H-presences message or values in H-content messages. However, as long as a router function never moves information down in the security lattice, wrapping a secure process in it yields a secure process.

(2.27) DEFINITION $r$ is $\ell$-secure iff $\forall c, c_1, v, v_1$.

\[ r(\$cv) = \$c_1v_1 \land \kappa(c) \not\subseteq \ell \implies \]

- $\pi(c) \subseteq \ell \implies \forall \ell', c_2, v_2 \cdot r(\$cv') = \$c_2v_2 \implies c_1 = c_2$
- $\pi(c) \not\subseteq \ell \implies \pi(c) \not\subseteq \ell$

We say $r$ is secure if it is $\ell$-secure for all $\ell$.

Compositionality of $\langle \rangle$ is immediate from Definition 2.7.

(2.28) COROLLARY $p \in$ PSNI $\implies r\langle p \rangle \in$ PSNI for secure $r$.

(2.29) COROLLARY $p \in$ PNI $\implies r\langle p \rangle \in$ PNI for secure $r$. 

\[ \]
2.4.5 Fairness

As we have discussed, autonomy, and therefore fairness, is a key feature of interactive LTS\(_{\text{IO}}\). However, the proof we just saw does not rule out the possibility of starvation. For instance, in the zipper given in the proof of \(\oplus\), when \(\hat{s}_L\) has an infinite number of observables and \(\hat{s}_R\) has no observable output, the zipper can ignore the remainder of \(\hat{s}_R\). Also, the \textsc{pini} zipper ignores the remainder of \(\hat{s}_R\) when \(\hat{s}_L\) has no more observables. The central question here is whether our security properties require starvation of components to remain compositional. This prompts us to study \emph{fair} combinator. What we find is that whereas \textsc{psni} composes freely assuming fairness, \textsc{pini} relies \emph{fundamentally on lack of fairness} to be compositional for even simple product compositions.

\textbf{Fair Composition}

A fair combinator permits only fair behaviors, i.e. ones which do not starve components, always allowing each to eventually make progress on its outputs.

\[(2.30) \text{DEFINITION} \quad \text{For } p_L \oplus p_R \downarrow, s \text{ is } \oplus\text{-fair}, \text{iff, } \forall t.\]

\[\exists t_{L,R}, o_{L,R}, s_L, s_R, p_{L,R}, p_{L,R}' : p_L \oplus p_R \Rightarrow p_{L,R} \Rightarrow p_{L,R}' \text{ and } s = t_{L,R}, o_{L,R}, s_L, s_R \wedge\]

\[p_L \oplus p_R \triangleleft t_{L,R} \Rightarrow p_{L,L} \oplus p_R \Rightarrow p_{L,L}' \Rightarrow p_{R,R} \Rightarrow p_{L,R}' \wedge\]

\[p_L \oplus p_R \triangleleft t_{L,R} \Rightarrow p_{L,L} \Rightarrow p_R \Rightarrow p_{R,R} \Rightarrow p_{L,R}' \wedge\]

\[p_{L,L} \Rightarrow p_{L,L}' \wedge p_R \Rightarrow p_{R,R} \Rightarrow p_{L,R}' \wedge\]

\[p_{R,R} \Rightarrow p_{R,R}' \wedge p_{L,R} \Rightarrow p_{L,R}' \wedge\]

\[p_L \oplus_F p_R \downarrow \text{iff } p_L \oplus p_R \downarrow \text{ and } s \text{ is } \oplus\text{-fair}. \]

We let \(\bfi\) be defined in a similar manner.

\textbf{\textsc{psni} Composes Fairly}

Modifying the zip-and-preserve proof above for the compositionality of \textsc{psni} under \(\oplus\) and \(\bfi\) is easy; when \(\hat{s}_R\) runs out of observable output, we zip in such a way that \(\hat{s}'\) takes turns in pulling an output from \(\hat{s}_R\) and \(\hat{s}_L\) into \(t\), irrespective of when and how many input appears before the outputs. We therefore have the following.

\[(2.31) \text{COROLLARY } p_L, p_R \in \textsc{psni} \implies p_L \bfi p_R \in \textsc{psni}.\]

\[(2.32) \text{COROLLARY } p_L, p_R \in \textsc{psni} \implies p_L \oplus_F p_R \in \textsc{psni}.\]

\textbf{\textsc{pini} Composes Unfairly}

However, the same cannot be said for \textsc{pini}; it fails to be preserved under even simple fair products. The way in which \textsc{pini} fails to compose is \emph{not} due to the simplification we made of the definition of \textsc{pini} discussed in Section 2.3.2. We therefore maintain that, in the interactive setting, \textsc{pini} relies \emph{fundamentally} on
unfairness to be preserved under composition, making it a poor target property for reasoning about security of autonomous processes.

(2.33) **THEOREM**  \( p_L, p_R \in \text{PINI} \not\Rightarrow p_L \boxast_F p_R \in \text{PINI} \).

To prove this, we give two programs which satisfy PINI which fair composition fails to be PINI. It illustrates how a progress-difference in one component can translate into an explicit flow under composition. Consider this program \( p_A \),

\[
\text{in } M \ h \\
\text{if } h \text{ mod } 2 = 0 \text{ then} \\
\text{out } L \ 0 \\
\text{end if}
\]

and the following program \( p_B \).

\[
\text{out } L \ 1
\]

Wrapped in \( W \circ B \circ Z \), both programs satisfy \( \text{PINI} \). In fact, \( p_B \) satisfies PSNI. Now consider two environments \( p_1 = F(\text{?M0.}!L_0.!) \) and \( p_2 = F(\text{?M1.}!L_1.!) \). Then we have that \( p_1 \models p_A \boxast_F p_B \) can match \( \text{?M0.}!L_0.!) \). However, \( p_2 \not\models p_A \boxast_F p_B \) cannot match this behavior; \( p_A \) produces no observables, and the composition cannot postpone the observable \( p_B \) wishes to perform indefinitely, so at best, \( p_1 \models p_A \boxast_F p_B \) can match \( \text{?M1.}!L_1. ! \), which is not \( \approx \)-equivalent to \( \text{?M0.}!L_0.!) \). Thus \( p_A \boxast_F p_B \not\in \text{PINI} \).

This same counterexample shows PINI also fails to compose under \( \boxast_F \).

(2.34) **COROLLARY**  \( p_L, p_R \in \text{PINI} \not\Rightarrow p_L \boxast_F p_R \in \text{PINI} \).

While PINI may be justifiable if it composes under simpler combinators, the above counterexample also applies to the \( \oplus_F \) and \( \boxplus_F \) combinators (see Section 2.5), and we have pointed out that PINI does not behave well under cascade (see our rationale for considering PSNI environments when defining PINIE in Section 2.3.3).

(2.35) **COROLLARY**  \( p_L, p_R \in \text{PINI} \not\Rightarrow p_L \oplus_F p_R \in \text{PINI} \).

(2.36) **COROLLARY**  \( p_L, p_R \in \text{PINI} \not\Rightarrow p_L \ominus_F p_R \in \text{PINI} \).

It is also worthy of note that \( p_B \in \text{PSNI} \). This means that even if a process is the only PINI process in a composition with PSNI processes, it cannot be guaranteed that the composition even satisfies PINI. The only way we can be sure of this in general is if PINI operates in a disjoint part of the security lattice, as then, the varied presence of the PINI component will not interfere with the behavior of the other components.

For these reasons, we deem PINI unfit for reasoning about security of autonomous processes.

### 2.5 Language of Secure Combinators

To showcase the generality of our combinator core and compositionality results, we give an rich language of combinators with which to build secure systems from
secure parts. Since the combinators are implemented in terms of core combinators, they all compose under $\text{PSNI}$ and $\text{PNI}$, and compose fairly under $\text{PSNI}$.

2.5.1 Derived Combinators

Figure 2.3 contains the full set of binary combinators which, for each component, has a path from input, through it, to output. Some of them appear regularly in literature on compositionality of security properties, e.g. product, (relaxed) cascade and feedback [27, 31, 55]. We show how these, and other, combinators can be implemented in terms of our core combinators. For comparison, we have placed their operational semantics in the appendix.

**Relaxed Cascade Feedback**

This combinator, denoted $\otimes$, behaves like $\otimes$, except that input to the composition is only delivered to the left component in the composition. The combinator can therefore be seen as a relaxed relaxed cascade. Using our combinators, we can implement $\otimes$ as

$$p_L \otimes p_R = r_E(r_L(p_L) \oplus r_R(p_R)).$$
where
\[
\begin{align*}
  r_E(i) &= i_E \\
  r_L(o) &= o_L \\
  r_R(o) &= o_R \\
  r_E(o_L) &= o \\
  r_L(i_E) &= i \\
  r_R(i_E) &= ?\diamond \\
  r_E(o_R) &= o \\
  r_L(i_R) &= ?\diamond \\
  r_R(i_L) &= i.
\end{align*}
\]

Here, \(a_X\) is \(a\) which message is labeled\(^8\) \(X\). The label expresses who is the producer behind \(a\); \(E\) is the environment, \(L\) is \(p_L\) and \(R\) is \(p_R\). Thus, for instance, \(r_R(i_E) = ?\diamond\) expresses that \(p_R\) should not receive a message that originated from the environment.

## Relaxed Cascade

This combinator, denoted \(\otimes\), is a relaxation of cascade (i.e. sequential) composition often considered in work in compositionality. Like in a cascade, input to a relaxed cascade enters only the first component, and output from the second component is sent only to the environment. However, output from the first component is replicated and sent both to the environment and the second component. Using our core, we can implement \(\otimes\) as
\[
  p_L \otimes p_R = r_E(r_L(p_L) \otimes r_R(p_R)),
\]
where
\[
\begin{align*}
  r_E(i) &= i_E \\
  r_L(o) &= o_L \\
  r_R(o) &= o_R \\
  r_E(o_L) &= o \\
  r_L(i_E) &= i \\
  r_R(i_E) &= ?\diamond \\
  r_E(o_R) &= o \\
  r_L(i_R) &= ?\diamond \\
  r_R(i_L) &= i.
\end{align*}
\]

## Cascade

Also known as sequential composition, combinator \(\ominus\) is a basic combinator typically seen as a primitive in various combinator formalisms (e.g. [35]). Input entering a cascade enters the first component only, output from the first component enters the second component only, and output from the second component becomes the output of the cascade. Using our core, we can implement \(\ominus\) as
\[
  p_L \ominus p_R = r_E(r_L(p_L) \ominus r_R(p_R)),
\]
where
\[
\begin{align*}
  r_E(i) &= i_E \\
  r_L(o) &= o_L \\
  r_R(o) &= o_R \\
  r_E(o_L) &= i \circ \diamond \\
  r_L(i_E) &= i \\
  r_R(i_E) &= ?\diamond \\
  r_E(o_R) &= o \\
  r_L(i_R) &= ?\diamond \\
  r_R(i_L) &= i.
\end{align*}
\]

\(^8\) By partitioning \(C\) or \(\forall\).
Feedback

A specialization of $\otimes$, this combinator, denoted $\ominus$, isolates the right-component, making it interact only with the left component. This is useful for modeling interaction with a closed system. We implement $\ominus$ as

$$p_L \ominus p_R = r_E\langle r_L\langle p_L \rangle \otimes r_R\langle p_R \rangle \rangle,$$

where

$$r_E(i) = i_E \quad r_L(o) = o_L \quad r_R(o) = o_R$$
$$r_E(o_L) = o \quad r_L(i_E) = i \quad r_R(i_E) = ?$$
$$r_E(o_R) = ! \quad r_L(i_R) = i \quad r_R(i_L) = i.$$

Buffered Loop

Placing a FIFO as the right component of $\ominus$ yields a buffered loop combinator, denoted $\lceil \cdot \rceil$. To see this, let $p_F = W(F(\epsilon))$, where $F$ is defined by $F(\delta) \Downarrow F(\delta.i^{-1})$ and $F(o.\delta) \Downarrow F(\delta)$. Using this process, we can implement the buffered loop combinator as follows.

$$[p] = p \ominus p_F.$$

This loop combinator avoids the compositionality issues $[\cdot]$ has by storing loop messages in the FIFO $p_F$ (which $p$ can consume from at its leisure), instead of jamming them directly into $p$. This ensures that $p$ can make progress on its outputs.

Generator

Any interactive LTS_{IO} can be used as a source of information by never delivering input to it. We define the generator combinator $[\cdot]$ as

$$[p] = r_{drop}\langle p \rangle,$$

where $r_{drop}(i) = ?$ and $r_{drop}(o) = o$. When used in conjunction with the binary combinators in Figure 2.3, we obtain three new combinators for introducing information into a system. These are $p \oplus [p']$ ($p'$ streams to the environment), $p \odot [p']$ ($p'$ streams to $p$) and $p \otimes [p']$ ($p'$ streams to both), the last of which can be illustrated as follows.

While one might consider including $r_{mute}(\cdot)$ where $r_{mute}(o) = !$ and $r_{mute}(i) = i$ as a combinator, we find that for any binary combinator $\odot$, $p \odot r_{drop}\langle p' \rangle$ behaves either as $p$ or as $r_{drop}\langle p' \rangle$ (latter semantically equivalent to $F^i(\omega))$. 

2.5 Language of Secure Combinators

2.5.2 Compositionality

Since all of the combinators presented in Figure 2.3 are specializations of \( \odot \) and \( \langle \rangle \), we have for all of them, and their counterparts based on \( \boxdot \) instead of \( \odot \), the following compositionality properties.

(2.37) **COROLLARY** For each binary combinator \( \odot \) in Figure 2.3,

\[
\begin{align*}
& p_L, p_R \in \text{PINI} \implies p_L \odot p_R \in \text{PINI}, \\
& p_L, p_R \in \text{PSNI} \implies p_L \odot p_R \in \text{PSNI}, \\
& p_L, p_R \in \text{PSNI} \implies p_L \odot_F p_R \in \text{PSNI}.
\end{align*}
\]

For each corresponding operator \( \boxdot \) based on \( \boxdot \),

\[
\begin{align*}
& p_L, p_R \in \text{PINI} \implies p_L \boxdot p_R \in \text{PINI}, \\
& p_L, p_R \in \text{PSNI} \implies p_L \boxdot p_R \in \text{PSNI}, \\
& p_L, p_R \in \text{PSNI} \implies p_L \boxdot_F p_R \in \text{PSNI}.
\end{align*}
\]

For \( \lceil \cdot \rceil \), and its counterpart based on \( \boxdot \),

\[
\begin{align*}
& p \in \text{PINI} \implies \lceil p \rceil \in \text{PINI}, \\
& p \in \text{PSNI} \implies \lceil p \rceil \in \text{PSNI}, \\
& p \in \text{PSNI} \implies \lceil p \rceil_F \in \text{PSNI}.
\end{align*}
\]

For \( \langle \cdot \rangle \),

\[
\begin{align*}
& p \in \text{PINI} \implies \langle p \rangle \in \text{PINI}, \\
& p \in \text{PSNI} \implies \langle p \rangle \in \text{PSNI}.
\end{align*}
\]

2.5.3 Building Secure Systems From Parts

By the above result, we now have a rich toolset for building secure wholes from secure parts. Large systems are often developed in a modular manner, in different programming languages, and once deployed, run distributed over a network. Our combinators facilitate a divide-and-conquer approach to building large secure systems. Parts can be proven secure by use of language-based or language-independent enforcement mechanisms that target our security properties. Once the parts are proven secure, we have that the whole, assembled using our combinators, is secure. Combinators can be used to model the network topology (how the parts are “hard-wired” together), while routers can express data-dependent traffic routing in the network.

The combinators and our system model can also be used to formalize the concurrency semantics in a programming language, like Erlang. Furthermore, by proposing suitable primitive interactive LTSIO, our combinators can be a programming language for writing asynchronous message-passing systems. One could, say, replace \( \langle \rangle \) with \( \odot \) as a primitive, if the one-step routing delay this introduces at the semantics level is not problematic. However, for such a language to be expressive, combinators for programmatically changing the wiring of components (like e.g. switches in functional reactive programming [35] name-passing in process algebra [21]) should be devised.
2.6 Related work

To aid in understanding the relative merits of the various models of interaction we are about to discuss, we classify $\text{LTS}_{\text{IO}}$ based on the interaction behavior they exhibit.

(2.38) **Definition** ($\text{LTS}_{\text{IO}}$ classification) $p$ is

1) input value neutral
   
   \[ \forall v \cdot p' \overset{1}{\Rightarrow} p' \] \iff \forall t, t' \cdot p \overset{1}{\Rightarrow} p' \]

2) input neutral
   
   \[ \forall i \cdot p' \overset{1}{\Rightarrow} i' \] \iff \forall t, t' \cdot p \overset{1}{\Rightarrow} p' \]

3) reactive
   
   \[ (\exists i \cdot p' \overset{1}{\Rightarrow}) \iff \forall t, t' \cdot p \overset{1}{\Rightarrow} p' \]

4) productive
   
   \[ \exists a \cdot p' \overset{1}{\Rightarrow} \] \iff \forall t, t' \cdot p \overset{1}{\Rightarrow} p' \]

5) internally deterministic
   
   \[ \forall a \cdot p_1, p_2 \overset{1}{\Rightarrow} p_1' \land p_2 \overset{1}{\Rightarrow} p_2' \iff p_1' = p_2' \]

6) input deterministic
   
   \[ \forall i_1, i_2 \cdot p \overset{1}{\Rightarrow} i_1' \land p' \overset{1}{\Rightarrow} i_2' \iff \exists c \cdot i_1, i_2 \in \{\text{?cv} \mid v \in \forall\} \]

7) output deterministic
   
   \[ \forall o_1, o_2 \cdot p \overset{1}{\Rightarrow} o_1' \land p' \overset{1}{\Rightarrow} o_2' \iff o_1 = o_2 \]

Event systems [19, 27, 28, 30, 53, 56] are essentially $\text{LTS}_{\text{IO}}$ with no restrictions applied. Trace semantics is used as the underlying notion of behavioral equivalence. Compositionality of information flow properties, under a binary operator which implicitly wires matching communication channels internally, has been thoroughly studied in this setting in theories developed for reasoning about compositionality [27, 31, 56]. McLean [31], Zakinthinos and Lee [55] showed that noninference, separability and perfect security are all compositional, and McLean further showed that generalized noninference and generalized noninterference compose under product. Johnson and Thayer showed that forward correctability is fully compositional [24]. McCullough first demonstrated that generalized noninterference is not fully compositional [28]. However, Zakinthinos and Lee have shown that generalized noninterference composes under certain conditions: under a relaxed cascade [54], if every feedback loop involves at least three components [55], or if a delay component is inserted into the feedback of high events [54]. Mantel [27] derived all the above results save the last two using his modular assembly kit for security properties (MAKS). He also derived several new conditional compositionality results, and showed that a weakened forward correctability is compositional. Our psni composes under routing, product, and under cascade and feedback provided a FIFO is placed between components. Our combinators offer a structured way of composing secure systems from parts; no wiring is implicit, and the possible routes that data can take are clearly defined by structure.
2.6 Related work

and routers. Our properties use stream semantics as the underlying behavioral equivalence, which we have argued and demonstrated, makes more, desirable distinctions, enabling us to reject the “extortionist” program given in Section 2.3.4.

Process calculi for security [17, 21, 22, 39, 45, 46] have $\text{LTS}_{\text{IO}}$ as their underlying semantics. They study the use of algebraic properties of concrete concurrency constructs. We are more abstract, providing results for $\text{LTS}_{\text{IO}}$ directly. We assume input totality in our framework, which induces a concurrency semantics free of output blocking, similar to mailboxes in the Actor model [1] (implemented in Erlang), message queues in JavaScript, and buffered I/O in most programming languages. Assuming input totality simplifies system composition considerably [56]. The parallel composition operator also implicitly wires channels. Bisimulation on processes is typically provided as the primary tool for behavioral reasoning. Since bisimulation is a branching-time, it makes undesired distinctions, which our behavioral equivalence avoids.

Reactive systems [10, 42, 57] are, in the sense of Definition 2.38, reactive, input neutral and productive $\text{LTS}_{\text{IO}}$. Bohannon et al. [10] present and contrast four stream-based possibilistic noninterference definitions, emphasizing $\text{cp}$-security and $\text{id}$-security, and give a type-based enforcement of $\text{id}$-security. While $\text{id}$-security and $\text{cp}$-security don’t exclude nondeterministic programs, $\text{id}$- and $\text{cp}$-security are very restrictive for nondeterministic programs, rejecting programs which conceal information using nondeterministic choice, and therefore essentially becoming as strict as low observational determinism [58] or security under refinement [36]. Our definition of $\text{PINI}$ can therefore be perceived as a more faithful generalization of $\text{PINI}$ from [3] to nondeterministic systems, or as a generalization of $\text{id}$-security to nondeterministic systems with intermediate input. We have shown that $\text{PINI}$ does not compose under fair schedulers. The counterexamples can easily be expressed in the language of [10], and thus $\text{id}$-security does not compose fairly. While the transducer impression that reactive systems have suggests easy composition, reactive systems are not input total; non-willingness of one component to receive can halt progress of another component that wishes to send.

Our security framework has the most in common with the $\text{LTS}_{\text{IO}}$- and strategy-based frameworks for possibilistic noninterference [14, 36, 41, 43]. O’Neill et al. [36] present a single-threaded programming language which $\text{LTS}_{\text{IO}}$ is input neutral, internally- and output-deterministic. The target property is strategy-based $\text{PSNI}$, originally inspired by nondeducibility on strategies by Wittbold and Johnson [53]. Extending the language with nondeterministic choice (making their $\text{LTS}_{\text{IO}}$ no longer deterministic), they modify $\text{PSNI}$ to require noninterference under refinement (arbitrary determinization of all nondeterministic choices prior to execution). Clark and Hunt [14] instead give a possibilistic version of $\text{PSNI}$, show that it is sufficient to guarantee security under deterministic strategies to prove that a program is $\text{PSNI}$, and that stream strategies are sufficient if the program is internally-, input- and output-deterministic, a result used by [10]. In both of these settings, strategies are total; strategies are always willing to receive, and regardless of when and on which channel a program blocks on, the strategy
has input available on said channel. While this may be a good fit when strategies model local memory, as demonstrated by Rafnsson et al. [41], this total strategies assumption ignores classes of realizable attacks which encode secrets in the varied presence of messages in a concurrent setting. This motivates distinguishing between sensitivity of message presence and content, as we do in the present paper, which none of the work discussed so far does, save for [41–43]. This idea can be traced back to Sabelfeld and Mantel [47], who study public (L), encrypted (M) and secret (H) communication channels in a concurrent setting, and Myers [32], who distinguishes between sensitivity of data structure length and content. Rafnsson et al. [41] show that $\text{PSNI}$ composes under product. All three of [14, 36, 41] use trace semantics as a basis for behavioral equivalence. We show $\text{PSNI}$ composes under all of our combinators, with stream semantics as the basis for behavioral equivalence, and with environments which are part of the computation model.

Asynchronous testing faces the same difficulties with blocking behavior as we face when putting two $\text{LTS}_{\text{IO}}$ in interaction. Whereas Verhaard et al. [51] solve the issues by equipping a tester and the implementation under test with an input queue, our assumption of input totality effectively means our interactive systems have input queues baked in. The Input/Output Automata model [26] is similar to our $\text{LTS}_{\text{IO}}$ model. It has input totality as a fundamental assumption, and is designed to reason about system composition and fairness. We have not seen this model applied in research on information-flow security. The concurrency semantics induced by our combinators is similar to that employed by signal processing formalisms, like Kahn networks [18, 25] and dataflow programming languages, e.g. Functional Reactive Programming languages [16, 35]. Indeed, our combinators are reminiscent of the signal function constructors in [35].

Finally, end-to-end security is easier to achieve if we, alongside combinators which compose secure components securely, have combinators which repair insecurities. Rafnsson and Sabelfeld [42] give such a combinator which puts a logarithmic bound on leaks through progress in a $\text{id}$-secure program. Secure Multi-Execution [15, 43] is a promising new technique which, through program transformation or dynamic monitoring, modifies (modestly) the semantics of any program to become that of a secure program. Devriese and Piessens [15] prove that the approach enforces timing sensitive noninterference, while Rafnsson and Sabelfeld [43] show that, by relaxing the guarantee to $\text{PSNI}$, the semantics of secure programs is modified less.

### 2.7 Conclusion

We have presented a framework for secure composition. We have achieved generality along several dimensions: (i) our underlying systems are general labeled transition systems; (ii) our composition is facilitated by a number of combinators; (iii) we distinguish between the security level of message presence and content; and (iv) we study both progress-sensitive and progress-insensitive security definitions. While the latter is a popular policy for practical tools, our findings point to
the importance of the former in the context of secure system composition. Our findings also provide new insights on the impact of fairness for the security of system composition.

Future work includes investigation of composition in the presence of insecure components. We plan to extend our combinator set with enhanced combinators that are able to “repair” insecure components and make them readily pluggable into a secure (composed) system.

Acknowledgments

This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References


Appendix

Proof

(2.16)  \textbf{THEOREM} \quad p \in \text{PSNI} \implies p \in \text{PSNI}_{\text{E}}

\text{Proof.} \quad \text{Assume } p \in \text{PSNI. Let } p_1, p_2 \text{ and } s_1 \text{ such that } p_1, p_2 \in \text{PSNI}, s_1 \in S_F, p_1 \cong_\ell p_2 \text{ and } p_1 \models p \models_\ell \text{ be arbitrary. Since } p_1 \models p \models_\ell, \text{ we have } p \models_\ell. \text{ Since } s_1 \in S_F \text{ and } p \in \text{PSNI}, \text{ there exists a } s_{1P} \in S(p) \cap A^w_\ell \text{ for which } s_1 \cong_\ell s_{1P} \text{ and } \text{preserve}_{psni}(\epsilon, s_{1P}).

\text{Since } p_1 \models p \models_\ell, \text{ we have } p_1 \models_\ell^{-1}. \text{ Assume } s_1^{-1} \in S_F \text{ for now (at the end of this proof, we explain how to adapt it to the scenario } s_1^{-1} \notin S_F). \text{ Since } p_1 \cong_\ell p_2, \text{ there exists a } s_2 \in S_F \text{ for which } p_2 \cong_\ell s_2 \text{ and } s_1^{-1} \cong_\ell s_2. \text{ Since } p_2 \in \text{PSNI}, \text{ there exists a } s_{2E} \in S(p_2) \cap A^w_\ell \text{ for which } s_2 \cong_\ell s_{2E} \text{ and } \text{preserve}_{p_2:s_2}(\epsilon, s_{2E}). \text{ To summarize, } s_{1P} \cong_\ell s_1 \cong_\ell s_2 \cong_\ell s_2 \cong_\ell s_{2E}^{-1} \text{.}
We must show that there exists a \( s_2 \in \mathbb{S}_F \) for which \( p_2 \models p \leq \ell_\mathbb{S} \) and \( s_1 \approx_\ell s_2 \). We obtain \( s_2 \) using

\[
\begin{align*}
\operatorname{zip}(s_{E}, t \cdot \bar{\alpha}.o.p \cdot s.p) \mid \bar{\alpha} p \approx_\ell \epsilon & \wedge \pi(o.p) \subseteq \ell \\
= \bar{\alpha} p \cdot.o.p. \operatorname{zip}(s'_{E}, t \cdot \bar{\alpha}.o.p \cdot s.p), \\
\text{where } \hat{s}_2 & \approx_\ell t \cdot \bar{\alpha}.o.p^{-1} \cdot s'_{E} \\
\land & p_2 \cdot \bar{\alpha}.o.p^{-1} \cdot s'_{E} \\
& \land \operatorname{preserve}_{p_2}((t \cdot \bar{\alpha}.o.p)^{-1}, s'_{E}) \\
\operatorname{zip}(\bar{\alpha} E.o.E \cdot s_{E}, t, s.p) \mid \bar{\alpha} E & \approx_\ell \epsilon \wedge \pi(o.E) \subseteq \ell \\
= (\bar{\alpha} E.o.E)^{-1}. \operatorname{zip}(s_{E}, t \cdot \bar{\alpha} E.o.E^{-1}, s'_{p}), \\
\text{where } s_1 & \approx_\ell (t \cdot \bar{\alpha} E.o.E)^{-1} \cdot s'_{p} \\
\land & p \cdot (t \cdot \bar{\alpha} E.o.E)^{-1} \cdot s'_{p}, \\
& \land \operatorname{preserve}_{p}((t \cdot \bar{\alpha} E.o.E)^{-1}, s'_{p}) \\
\operatorname{zip}(\bar{\alpha} E, t, \bar{\alpha} p) \mid \bar{\alpha} E & \approx_\ell \epsilon \wedge \bar{\alpha} p \approx_\ell \epsilon \\
= \bar{\alpha} E^{-1} \cdot \bar{\alpha} p \cdot.o.p \cdot \operatorname{zip}(s'_{E}, t \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p^{-1}), \\
\text{where } \hat{s}_2 & \approx_\ell t^{-1} \cdot \bar{\alpha} E.o.E^{-1} \cdot \bar{\alpha} p^{-1} \\
\land & p_2 \cdot t^{-1} \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p^{-1} \cdot \bar{\alpha} E \\
& \land \operatorname{preserve}_{p_2}((t^{-1} \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p)^{-1}, \bar{\alpha} E) \\
& \land s_1 \approx_\ell t \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p^{-1} \\
& \land p \cdot t^{-1} \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p^{-1}, \\
& \land \operatorname{preserve}_{p}((t^{-1} \cdot \bar{\alpha} E^{-1} \cdot \bar{\alpha} p)^{-1}, \bar{\alpha} E).
\end{align*}
\]

By setting \( s_2 = \operatorname{zip}(s_{2E}, \epsilon, s_{1p}) \), we get \( p_2 \models p \leq \ell_\mathbb{S} \) and \( s_1 \approx_\ell s_2 \). This can be seen by observing that the middle parameter \( t \) during each corecursive call grows to include one more observable (until it contains all observables, after which it grows with unobservables), and that \( t \leq \ell_\mathbb{S} s_1 \) and \( t^{-1} \leq \ell_\mathbb{S} \hat{s}_2 \).

If \( s_1^{-1} \notin \mathbb{S}_F \), then for the shortest prefix \( t_1 \leq s_1^{-1} \) containing all output in \( s_1^{-1} \), there is a \( \hat{\alpha}_1 \) for which \( p_1 \models_\ell \hat{\alpha}_1 \). We then make sure to not zip beyond the last observable output in \( t_1 \) and still get the desired \( s_2 \) by replacing the last case in the definition of \( \operatorname{zip} \) with one which evaluates to \( \bar{\alpha} p \). \( \square \)

**Combinator Semantics**

**Relaxed Cascade Feedback**

\[
\begin{align*}
\frac{p_1 \Downarrow p'_1 \quad p_2 \Downarrow p'_2 \quad p_2 \Downarrow p'_2 \quad p_1 \Downarrow p'_1 \quad p_2 \Downarrow p'_2}{p_1 \land p_2 \Downarrow p'_1 \land p'_2} & \quad \Downarrow_l \quad \Downarrow_l \\
\frac{p_1 \Downarrow p'_1 \quad p_2 \Downarrow p'_2 \quad p_1 \Downarrow p'_1 \quad p_2 \Downarrow p'_2}{p_1 \land p_2 \Downarrow p'_1 \land p'_2} & \quad \Downarrow_r \quad \Downarrow_r
\end{align*}
\]

**Relaxed Cascade**

\[
\begin{align*}
\frac{p_R \Downarrow p'_R \quad p_L \Downarrow p'_L \quad p_L \Downarrow p'_L \quad p_R \Downarrow p'_R}{p_L \land p_R \Downarrow p'_L \land p'_R} & \quad \Downarrow \quad \Downarrow
\end{align*}
\]
Feedback

\[
\begin{align*}
&\frac{\rho_L \circ \rho'_L \quad \rho_R \circ^{-1} \rho'_R}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R} \\
&\frac{\rho_L \circ \rho_R \circ \rho'_L \parallel \rho'_R \overset{\circ}{\circlearrowleft}}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R} \\
&\frac{\rho_L \circ \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R}
\end{align*}
\]

Cascade

\[
\begin{align*}
&\frac{\rho_R \circ \rho'_R}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R} \\
&\frac{\rho_L \circ \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R} \\
&\frac{\rho_L \circ \rho'_R \parallel \rho_R \overset{\circ}{\circlearrowleft}}{\rho_L \parallel \rho_R \overset{\circ}{\circlearrowleft} \rho'_L \parallel \rho'_R}
\end{align*}
\]

Buffered Loop

Let \([p] = [p]_e\).

\[
\begin{align*}
&\frac{\rho \circ \rho'}{[p]_l \overset{\circ}{\circlearrowleft} [p]_l'_{o.t}} \\
&\frac{\rho \circ^{-1} \rho'}{[p]_{o.t} \overset{\circ}{\circlearrowleft} [p]_{o.t}'} \\
&\frac{\rho \overset{\circ}{\circlearrowleft} \rho'}{[p]_l \overset{\circ}{\circlearrowleft} [p]_l'}
\end{align*}
\]
Secure Multi-Execution: Fine-grained, Declassification-aware, and Transparent

ABSTRACT Recently, much progress has been made on achieving information-flow security via secure multi-execution. Secure multi-execution (SME) is an elegant way to enforce security by executing a given program multiple times, once for each security level, while carefully dispatching inputs and ensuring that an execution at a given level is responsible for producing outputs for information sinks at that level. Secure multi-execution guarantees noninterference, in the sense of no dependencies from secret inputs to public outputs, and transparency, in the sense that if a program is secure then its secure multi-execution does not destroy its original behavior.

This paper pushes the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on the cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows expressing what information can be released. Fourth, we establish a full transparency result showing how secure multi-execution can preserve the original order of messages in secure programs. We demonstrate that full transparency is a key enabler for discovering attacks with secure multi-execution.

3.1 Introduction

As modern attacks are becoming more sophisticated, there is an increasing demand for more advanced protection measures than those offered by standard security practice. We exemplify an instance of the problem with a motivating scenario from web application security, but note that the problem is of rather general nature.
Motivation  In the context of the web, third-party script inclusion is pervasive. It drives the integration of advertisement and statistics services. As an indicative example, barackobama.com at the time of the 2012 US presidential campaign contained 76 different third-party tracking scripts [41]. The tracking was used for target political advertisement. Script inclusions extend the trusted computing base to the Internet domains of included scripts. This creates dangerous scenarios of trust-abuse. This can be done either by direct attacks from the included scripts or, perhaps more dangerously, by indirect attacks when a popular service is compromised and its scripts are replaced by the attacker. A recent empirical study [28] of script inclusion reports high reliance on third-party scripts. It outlines new attack vectors showing how easy it is to get code running in thousands of browsers simply by acquiring some stale or misspelled domains. Access control mechanisms are of limited use because third-party scripts require access to sensitive information for their proper functionality. This is particularly important for statistics and context-aware advertisement services on the web. Similar scenarios arise in the setting of cloud computing where sharing the resources is desirable but without compromising confidentiality and integrity. This motivates the need for fine-grained information-flow control.

From static to dynamic information-flow control  Tracking information flow in programs is a popular area of research. Static analysis techniques have been extensively explored, leading to tools like Jif [27], FlowCaml [40], and SparkAda Examiner [7] that enhance compilers for Java, Caml, and Ada, respectively. Recently, dynamic monitoring techniques have received increased attention (cf. [24, 23, 39, 37, 4, 5, 20]), driven by the demand to analyze dynamic programming languages like JavaScript. While static analysis either accepts or rejects a given program before it is run, dynamic monitors perform checks at run time. There are known fundamental tensions [35] between static and dynamic analyses, implying that none is superior to the other. Although dynamic analysis might seem intuitively more permissive, it has to conservatively treat the paths that are not taken by the current execution.

Secure multi-execution (SME)  Recently, there has been much progress on SME [16, 9, 22, 21, 6, 8, 18], a runtime enforcement for information flow. In contrast to the monitoring techniques, the goal is not to prevent insecurities but to “repair” them on the fly. This approach is secure by design: security is achieved by separation of computations at different security levels. The original program is run as many times as there are security levels, where outputs at a given security levels are only allowed if the security level of the program is matched with the security level of the output channel. The handling of inputs is slightly more involved because inputs from less restrictive security levels are allowed to be used in computations at more restrictive levels. Secure multi-execution propagates inputs, once they are received, to the runs of the program that are responsible for the computation of outputs at more restrictive levels.
Typically, security levels are drawn from a lattice with the intuition that information from an input source at level \( \ell \) may flow to an output sink at level \( \ell' \) only if \( \ell \subseteq \ell' \) [15]. For simplicity, we will often use the two-level lattice with a secret (high) level and a public (low) level of confidentiality. Figure 3.1 shows program \( P \) with a pair of input sources, labeled high \( H \) and low \( L \), and a similarly-labeled pair of sinks. The baseline policy of noninterference [17] demands that low outputs do not depend on high inputs.

Figure 3.2 shows how secure multi-execution achieves noninterference. Program \( P \) is run twice, as \( P_H \) at high and as \( P_L \) at low levels. The high input is fed into the high run. The low input is fed into both the low and high run. Dummy default values are used whenever the low run asks for high input. High output is produced by the high run, and the low output is produced by the low run, while low output of the high run and high output of the low run are ignored. It is clear from the diagram that noninterference is enforced because the low run, the only producer of low output, never gets access to high input.

In contrast to the traditional dynamic analysis, there is no concern about executions not taken because the control flow of the low run cannot possibly be affected by high input. Further, secure multi-execution provides transparency, in the sense that if a program is secure then its secure multi-execution does not destroy its original behavior.

**Contributions** While secure multi-execution gains increased popularity, there are open challenges that need to be addressed before it can be applied widely. We overview the pros and cons of secure multi-executions compared to traditional information-flow control and, among other findings, point out that secure multi-execution (i) lacks support for fine-grained security levels for communication channels, (ii) relies on restrictive scheduling, (iii) lacks support for declassification, (iv) may reorder messages wrt. the original execution, and (v) lacks support for detecting attacks.

We push the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows ex-
pressing what information can be released. Fourth, we establish full transparency showing how secure multi-execution can preserve the original order of messages in secure programs by barrier synchronization. This enables the use of secure multi-execution to discover attacks on runtime.

3.2 Pros and cons of secure multi-execution

We overview the pros and cons of secure multi-execution with respect to direct information-flow enforcement. The overview has two goals: provide a general basis for deciding on which enforcement mechanism to pick in a particular case and identify the most pressing shortcomings, subject to improvements by this paper. We start by listing of what we view as the pros of secure multi-execution.

Noninterference by design  A significant advantage of secure multi-execution is that it straightforwardly enforces noninterference by a simple access-control discipline: computation responsible for output at a given level never gets access to information at more restrictive or incomparable levels. This provides noninterference guarantees.

Language-independence  A major benefit is that secure multi-execution can be enforced in a blackbox, language-independent, fashion. The enforcement only concerns input and output operations allowing the rest of the language to be arbitrarily complex. This is particularly useful for dynamic languages like JavaScript that are hard to analyze.

Transparency for secure programs  If the original program is secure, there are transparency guarantees that limit ways in which semantics can be modified. The original work on secure multi-execution shows per-channel transparency (or precision in the terminology of Devriese and Piessens [16]). This means that if the original program is secure then, from the viewpoint of each channel, the sequence of I/O events in a given run of a program is the same in the original run and in the multi-executed run.

Transparency at top level  In addition, we note another transparency property, which applies, for example, to programs with no intermediate input: the externally-observable program behavior at the top security level is the same for the original and securely multi-executed runs. This property can be seen from Figures 3.1 and 3.2. Clearly, the original run of the program in Figure 3.1 and the high run of the multi-executed program in Figure 3.2 get the same inputs. Hence, the high output behaviors are the same no matter whether the original program is secure or not.

We now turn to the cons of secure multi-execution.
Coarse-grained labels In work on secure multi-execution so far, communication channels are provided with a single security label. This is often too coarse-grained: for example, the presence of a message might be public but the content is secret. This granularity might be useful for statistics services that might be counting different types of events without revealing their content. For example, Google Analytics is routinely used for various types of counting: how many clicks on the page, how many times a video is played, and how many visitors have viewed a page.

Devriese and Piessens [16] assume total input environments: that the input is always present. This does not allow modeling scenarios where the presence of secret input is secret (for example, whether or not the user visits a health web site). Bielova et al. [9] allow non-total environments but at the price of ignoring information leaks through termination behavior (targeting termination-insensitive noninterference [43]). This implies that the leaks as in the example with the health site are still ignored.

This motivates the need for fine-grained secure multi-execution. We lift the assumption on total input environments and introduce fine-grained labels for communication channels, where the levels of presence and content of messages are distinguished.

Restrictive scheduling With the exception of work by Kashyap [22], secure multi-execution heavily relies on the low-priority scheduler that lets low-computation run until completion before the high run gets a chance to run. The low-priority scheduler is both at the heart of the soundness results by Devriese and Piessens [16] and at the heart of FlowFox [18], an extension of FireFox to enforce secure information flow in JavaScript. The security theorem in the abstract setting of secure multi-execution [16] takes advantage of the low-priority scheduler and establishes timing-sensitive security. This is intuitive because the last access of low data occurs before any high data is accessed. Whenever the timing behavior is affected by secrets, there is no possibility for the attacker to inspect the difference.

However, the situation is different in the presence of handlers. The low-priority scheduler does not scale because it is not possible to extend the low-priority discipline over multiple events—simply because it is not possible to run the low handlers that have not yet been triggered. As a compromise, FlowFox [18] multi-executes JavaScript with the low-priority scheduler on a per-event basis. However, as illustrated by a leak in Appendix A, this strategy is at the cost of timing-sensitive security. All we need to do is to set a low handler to execute after the high run of the main code has finished. Then the low handler can leak via the computation time taken by the high run.

This motivates the need for flexible scheduling strategies and the need for (fair) interleaving of the runs at different levels, as pursued in this paper.

Declassification Declassification is challenging because secure multi-execution is based on separating information at different security levels. Feeding secret in-
formation to a public run might introduce unintended leaks. Coming back to the example of tracking and statistics, we might want to track the popularity of items in a shopping cart or track various average values for transactions.

This motivates the need for declassification in secure multi-execution. The event of declassification should not leak information about the context (branching on a secret and declassifying in the body would leak the boolean value of the secret). It turns out that the support for fine-grained communication channels provides us with a natural treatment of declassification. Indeed, declassification is about communicating a secret value from the high run to the low run, but without leaking through the presence of the communication event. Exactly this is provided by channels with high content and low presence! Hence, a declassification event corresponds to output on a high-content low-presence channel (in view of the high run), and to input on high-content low-presence channel (in the view of the low run).

**Order of events modified** The transparency guarantees of secure multi-execution are per channel, allowing the order of events to be modified across different channels. This leads to unexpected results in an interactive setting.

This motivates the need for stronger transparency, where the behavior of secure programs is unmodified across the different levels. We show how to achieve this by careful scheduling of the runs at the different levels.

**Silent failure** The behavior of secure and insecure programs is silently modified. As mentioned above, there are cases when the run at the top security level is immune to such modifications as it never gets dummy values. However, the behavior at less restrictive levels might be modified, leading to loss of important functionality. This directly connects to undiscovered attacks, addressed below.

**Undiscovered attacks** Related to the silent failure point above, secure multi-execution “repairs” problematic executions on the fly, with no means to identify if there were any attempted attacks and what caused such attacks.

This motivates an enhancement of secure multi-execution that allows for detecting attacks. Intuitively, we introduce barrier synchronization of the runs at the different security levels and track the consistency of the values they produce. In the two-level lattice, we check if the low output produced by the low run matches the value produced by the high run (which is the same as the low output of the original program). If they are inconsistent, we have found an attack. Full transparency is the key for this result because it guarantees that secure programs must have exactly the same I/O behavior as their securely multi-executed versions.

**Nondeterminism** Nondeterminism needs to be reproduced for the executions at different levels. Although this has not been explicitly handled in previous work,
a natural possibility is to assign security level to the source of nondeterminism and propagate it to the relevant executions in a fashion similar to propagating inputs.

**Dummy values**  Dummy values are fed into executions that are not authorized to have access to sensitive input. An unfortunate choice of values might lead to the program crashing. Defensive programming is then needed to ensure that programs are stable under variation of allowed input.

**Performance**  Executing the program several times implies obvious performance overhead. At the same time, secure multi-execution benefits from multicore architectures, in particular when the number of executions is less than the number of cores [16]. Also, as we discuss in Section 3.7, optimizations are possible for simulating multiple executions by computing on enriched values [6].

### 3.3 Framework

We lay the foundation for our technical contributions outlined in the introduction by presenting a framework for information-flow security of interactive programs [29, 14, 12, 31, 30].

#### 3.3.1 Interactive programs

Our model of computation is a *labeled transition system* (LTS). An LTS is a triple \((S, L, \rightarrow)\), where \(S\) is a set (of states), \(L\) is a set (of labels), and \(\rightarrow \subseteq S \times L \times S\) (a labeled transition relation). Computation occurs in discrete steps (transitions), each taking a (unspecified) unit of time. \(s \xrightarrow{l} s'\) iff \((s, l, s') \in \rightarrow\), and \(s \xrightarrow{a} s'\) for some \(s'\).

The systems we consider in this paper interact with their environment through channel-based message-passing. Such systems have three kinds of effects: (message-)input, output, and silence. The two latter effects are “productions”, referred to as output \(o\), and the first effect is a “consumption”, referred to as input \(i\). Collectively, these are actions \(a\).

\[
a ::= i \mid o \quad i ::= ?_c v \quad o ::= !_c v \mid \cdot
\]

Here, \(?_c v\) (resp. \(_c v\)) denotes a message received (resp. sent) on channel \(c\) carrying value \(v\), and \(\cdot\) denotes a non-interaction. \(c\) and \(v\) range over the (nonempty) sets \(C\) and \(V\) resp.. These effects are the only external interface to our systems; systems are “black boxes” in every other respect.

(3.1) **DEFINITION**  An input-output LTS (LTS\(_{IO}\)) is an LTS \((S, L, \rightarrow)\) where \(a\) ranges over \(L\).

Practical languages native to this paradigm include Erlang and JavaScript. Bohannon et al. give the semantics of a JavaScript-like language as an LTS\(_{IO}\) in [12]...
and Rafnsson et al. give the semantics of an imperative language with I/O (used in our examples) as an LTS\(_{\text{IO}}\) in [30].

One element \(\star \in \mathcal{V}\) (blank) is distinguished. When \(s \xrightarrow{c \star} s',\) then \(s\) has waited one time unit for an input on \(c\) without receiving one. \(!c\star\) has no specific meaning.

(3.2) **Definition** Let \(\lambda = (S, L, \rightarrow)\) be an LTS\(_{\text{IO}}\).

1. \(\lambda\) is input-neutral iff
   \[
   \forall s \in S, c \cdot (\exists v \cdot s \xrightarrow{c v}) \implies (\forall v \cdot s \xrightarrow{c v}).
   \]
2. \(\lambda\) is input-blocking iff
   \[
   \forall \{s, s'\} \subseteq S, c \cdot s \xrightarrow{c \star} s' \implies s' = s.
   \]
3. \(\lambda\) is deterministic iff
   a) \(\forall s \in S, a_1, a_2 \cdot s \xrightarrow{a_1} \land s \xrightarrow{a_2} \land a_1 \neq a_2 \implies \exists c, v_1, v_2 \cdot a_1 = ?c v_1 \land a_2 = ?c v_2,\) and
   \[
   b) \forall \{s, s_1, s_2\} \subseteq S, a \cdot s \xrightarrow{a} s_1 \land s \xrightarrow{a} s_2 \implies s_1 = s_2.
   \]

Pt. 1 states that if \(s\) is ready to perform input, \(s\) is receptive to any \(v\). Pt. 2 states that input is a blocking operation. Pt. 3a says if \(s \xrightarrow{c v}\), then \(s \xrightarrow{a}\) iff \(a \in \{?a v \mid v \in \mathcal{V}\}\), and implicitly, if \(s \xrightarrow{a}\), then \(s \xrightarrow{a}\) iff \(a = o\). Pt. 3b says \(s\) has no internal nondeterminism. Unless stated otherwise, any \(s\) we consider in this paper is from a \(\lambda\) satisfying pt. 1, 2 and 3. We discuss the assumption of pt. 2 further in Section 3.4.

### 3.3.2 Traces

A trace is a (finite) list of actions, denoted \(\bar{a}\). We write \(s \xrightarrow{\bar{a}} s_n\) if \(s \xrightarrow{a_1} \ldots s_{n-1} \xrightarrow{a_n}\) for some \(s_1, \ldots, s_n\) and \(\bar{a} = a_1 \ldots a_n\). Let \(\bar{a} \mid_\gamma\), \(\bar{a} \mid_\gamma\) and \(\bar{a} \mid_c\) denote the projection of \(\bar{a}\) to its input-, output- and \(c\)-messages, respectively. E.g., if \(\bar{a} = ?c 1.2c'2.2c lc 4\), then \(\bar{a} \mid_\gamma = ?c 1.2c' \star, \bar{a} \mid_\gamma = 1c'2.2lc 4\), and \(\bar{a} \mid_c = ?c 1.2c 4\). \(\bar{a} \mid_c\) extends to \(\bar{a} \mid_C\) for \(C \subseteq \mathcal{C}\) in the obvious way. We write \(\bar{a} \mid_{x_1, \ldots, x_n}\) as short for \(\bar{a} \mid_{x_1} \ldots \mid_{x_n}\); we refer to each \(x_j\) as a projection predicate. With \(\bar{a}\) defined as above, \(\bar{a} \mid_{c \gamma} = \bar{a} \mid_{c} \gamma = ?c 1\). We write \(\bar{x} \leq \bar{x}'\) when, for some \(\bar{x}'\), \(\bar{x}'' = \bar{x} \cdot \bar{x}'\). For a relation \(R, \bar{a} R x_1, \ldots, x_n \bar{a}'\) is short for \((\bar{a} \mid_{x_1, \ldots, x_n}) R (\bar{a}' \mid_{x_1, \ldots, x_n})\). Note that \((R x_1, \ldots, x_n) x_0 = R x_0, \ldots, x_n\). With \(\bar{a}\) defined as above, \(\bar{a} \leq_{lc} \bar{a}' = ?c 3.4.4c 5\).

### 3.3.3 Observables

The observables of our programs are its effects. The observability of a message is given by the security level associated with the channel carrying the message. As foreshadowed earlier, we assume a lattice \((\mathcal{L}, \sqsubseteq)\), with \(\mathcal{L}\) ranged by \(\ell\), of security levels which express levels of confidentiality. Each channel is labeled with two security levels; \(\pi(c)\) is the level of the presence of a message on \(c\), and \(\kappa(c)\) is
the level of the content or value of a message on \( c \). In examples, we frequently represent a channel by its security labels; we then write \( \kappa(c)\pi(c) \) in place of \( c \) (in code, \( \kappa(c)\pi(c) \)). A classic example is the two-level lattice \( \mathcal{L}_{\text{LH}} = \{L, H\} \) with \( \sqsubseteq = \{L, L\}, (L, H), (H, H)\} \), \( L \) for “low” confidentiality, \( H \) for “high”. We let \( H, M, L \) denote \( H^H, H^L \) and \( L^L \), resp.

Figure 3.3 illustrates the flow of information in the case of the two-level lattice. Input on \( M \) is depicted in-between the high and low input. The presence of such an input is low (this dependency on low is illustrated by the dashed line) while the content is high (this dependency on high is illustrated by the solid line). Similarly, output on \( M \) is depicted in-between the high and low output. Its presence is observable at low level (cf. the dashed arrow), and its value is observable at high level (cf. solid arrow).

The security labels express who can observe what. An observer is associated a security level \( \ell \). An \( \ell \)-observer is capable of observing the presence (resp. content) of a message on \( c \) iff \( \pi(c) \sqsubseteq \ell \) (resp. \( \kappa(c) \sqsubseteq \ell \)). \( \hat{a} \upharpoonright \ell \) removes \( \ell \)-unobservable parts of actions in \( \hat{a} \). For \( \hat{a} = \vdots; \mathcal{L}_0; !\mathcal{H}_1; ?\mathcal{M}_2; ?\mathcal{M}_1 \), \( \hat{a} \upharpoonright \ell = \vdots; \mathcal{L}_0; \vdots; ?\mathcal{M}_2; ?\mathcal{M}_1 \). \( !\mathcal{H}_1 \) got replaced with \( \vdots \) since communication on \( \mathcal{H} \) is unobservable to a \( \mathcal{L} \)-observer (thus looks like a \( \vdots \)). \( ?\mathcal{M}_2 \) got replaced with \( ?\mathcal{M}_2 \) (for a fixed \( \mathcal{d} \in \mathcal{V} \)), since a \( \mathcal{L} \)-observer only observes presence of messages on \( M \) (all \( a \in \{!\mathcal{M}_2 \mid \mathcal{v} \in \mathcal{V}\} \) look the same).

**Timing and progress** Eventually we enforce a property stating that variations in unobservable inputs to a system do not cause an \( \ell \)-observable difference in the traces the system can perform. How we define trace equivalence defines the class of attackers such a property guarantees security against. We consider two classes of attackers: timing- and progress-sensitive ones respectively. To both, a blank input is unobservable since no message is passed. \( \hat{a} \mid * \) replaces all \( ?c \mathcal{L}_* \) with \( \vdots \).

With \( \hat{a} \) defined as above, \( \hat{a} \mid * = \vdots; \mathcal{L}_0; !\mathcal{H}_1; \vdots; ?\mathcal{M}_2; \vdots; ?\mathcal{H}_2 \).

A timing-sensitive (e.g. [1, 16]) attacker measures time between observables in a trace. \( \hat{a} \mid * \) removes trailing \( \vdots \) from \( \hat{a} \). E.g. \( \hat{a} \mid * = \vdots; \mathcal{L}_0; !\mathcal{H}_1; ?\mathcal{M}_2; ?\mathcal{H}_2 \).

Define timing-sensitive \( \ell \)-equivalence \( \approx_{\ell} \) as \( =_{*, \ell, * \ell} \). Consider

\[
\text{in } \mathcal{H} \mathcal{h} \text{ ; out } \mathcal{L} 0
\]

Since this program can perform traces \( \hat{a}_1 = ?\mathcal{H} \mathcal{H}; !\mathcal{L}_0 \) and \( \hat{a}_2 = ?\mathcal{H}_1; !\mathcal{L}_0 \), this program is not secure against a timing-sensitive \( \ell \)-observer since \( \hat{a}_1 \upharpoonright *_{\ell, \ell} = \vdots; \vdots; !\mathcal{L}_0 \neq \vdots; !\mathcal{L}_0 = \hat{a}_2 \upharpoonright *_{\ell, \ell} \) (in both traces, \( !\mathcal{L}_0 \) is produced faster in the latter trace).

A progress-sensitive (e.g. [29, 2, 31]) attacker observes whether there are more observables forthcoming in a trace. \( \hat{a} \mid * \) removes all \( \vdots \) from \( \hat{a} \). With \( \hat{a} \) as above, \( \hat{a} \mid * = \mathcal{L}0; !\mathcal{H}_1; ?\mathcal{M}_2; ?\mathcal{H}_2 \). We define progress-sensitive \( \ell \)-equivalence \( \approx_{\ell} \) as \( =_{*, \ell, * \ell} \). \( \hat{a}_1 \) and \( \hat{a}_2 \) are not evidence that the above program is insecure against progress-sensitive \( \ell \)-observers since \( \hat{a}_1 \upharpoonright *_{\ell, \ell} = !\mathcal{L}_0 = \hat{a}_2 \upharpoonright *_{\ell, \ell} \) (in both traces, \( !\mathcal{L}_0 \)
eventually appears). Note that \((\simeq_\ell) \subsetneq (\simeq_\ell)\). Thus, a progress-sensitive (timing-insensitive) attacker is strictly weaker than a timing-sensitive one.

### 3.3.4 Environments

The inputs to our systems come from the environment in which our systems run. Clark and Hunt [14] have demonstrated that when performing security analysis of programs, an environment does not need to be adaptive in any way to provoke a particular (leaking) behavior from a deterministic program. It is therefore sufficient for our purposes to consider environments represented as a stream (infinite list) of inputs for each input channel. So, an environment \(I\) is a mapping from input channels to the stream of inputs the environment provides on that channel. Since streams can contain blanks, our framework considers attacks powered by delayed input.

Input streams restrict which traces are possible; \(\bar{a}\) is consistent with \(I\), written \(I \models \bar{a}\), iff for all \(c\), with \(I_c = I(c)\),

\[
\forall \bar{a}' \leq \bar{a} \land \exists \bar{i}' \leq I_c \\
( |\bar{a}'| = |\bar{i}'| \land \bar{a}' \leq \bar{i}' \\
\land (\bar{a}' = _\gamma ?c \star \implies \bar{i}' = _\gamma ?c))
\]

Read this as “all \(i\) in \(\bar{a}\) came from \(I\), and \(*\) is read only when \(I\) had no value ready to be read (\(\_\) is a wildcard). Running \(s\) under \(I\) constrains the traces which \(s\) can perform; \(s\) performs \(\bar{a}\) under \(I\), written \(I \models s \bar{a}\), iff \(s \bar{a}\) and \(I \models \bar{a}\).

Consistency implies that \(I\) queues arriving input. Consider

\[
\text{Let } s \text{ be the LTS}_I \text{ state of this program and let } I_1, \ldots, I_5 \text{ such that } I_{1c} = (?c\star)^n, I_{2c} = ?c0.?c0.(?c\star)^n, I_{3c} = ?c2.?c0.(?c\star)^n, I_{4c} = ?c0.?c\star?c\star.?c\star.?c0.(?c\star)^n, \text{ and } I_{5c} = ?c2.?c\star?c\star.?c\star.?c0.(?c\star)^n \text{ be given. Then } I_1 \models s \frac{(\sigma \circ \star)}{n} \text{ for all } n \in \mathbb{N}, I_2 \models s \frac{\sigma \circ \star}{2 \cdot \sigma \circ \star}, I_3 \models s \frac{\sigma \circ \star}{2 \cdot \sigma \circ \star}, I_4 \models s \frac{\sigma \circ \star}{\sigma \circ \star \cdot \sigma \circ \star}, I_5 \models s \frac{\sigma \circ \star}{\sigma \circ \star \cdot \sigma \circ \star}. \text{ Stream equivalence becomes: } I_1 \approx_\ell I_2 \text{ (resp. } I_1 \approx_\ell I_2) \text{ iff } \forall c \cdot \pi(c) \subseteq \ell \implies I_1(c) \approx_\ell I_2(c) \text{ (resp. } I_1(c) \approx_\ell I_2(c)\).
\]

**Totality** An environment is total if it always provides a system with input whenever the system needs it. In our framework, \(I\) is total if \(*\) does not occur in any \(I_c\). Previous work on security for interactive programs [29, 14] assume that environments are total. However, as we have demonstrated previously [30], this assumption limits (undesirably) the space of possible attacks on input-blocking interactive programs, since the presence of a message can vary depending on high data. Consider the program in Section 3.3.3. Let \(I_{1h} = ?H0.(?c\star)^n\) and \(I_{2h} = I_{1c} = I_{2c} = (?c\star)^n\) for all \(c \neq H\). While this program can perform \(?H0.!L0\) under \(I_1\), the program cannot perform an \(\approx_\ell\)-equivalent trace under \(I_2\). Since a program can encode a bit in the presence of a message, these attacks becomes crucial in an interactive setting. To emphasise the gravity of the matter, here are three interactive programs [30], each secure under total environments,
which, when run in parallel, leak the input on H, bit by bit, on L, by encoding
the received value in the presence of H₀ and H₁ messages.

```plaintext
while 1 { in H₁ x ; out L 1 ; out H₁' 42 }
```

```plaintext
while 1 { in H₀ x ; out L 0 ; out H₀' 42 }
```

```plaintext
in H h;
for b in bits(h) {
    if b { out H₁ 42 ; in H₁' x }
    else { out H₀ 42 ; in H₀' x }
}
```

In short, the lack (resp. delay) of input impedes the progress (resp. timing)
behavior of input-blocking interactive systems. To guarantee protection against
attacks powered by varied input presence, nontotal environments (e.g. our I)
must be considered. This in part motivates our fine-grained security types; since
no low observables are allowed to occur after a high input, the only way for an
input-blocking system to input a high value before performing low observables is
if the presence level of the input is low [30].

### 3.4 Fine-grained secure multi-execution

The opening series of our contributions develops a generalization of SME [16]
with respect to several dimensions. We lift the assumption on the totality of
the input environment (that there is always assumed to be input) and on the
cooperative scheduling. Furthermore, we distinguish between security levels of
presence and content of messages. In addition, we generalize SME to arbitrary
deterministic LTS \( IO \) and strengthen the guarantees SME provides.

By design, our formalization of SME ensures
that the \( \ell \)-observable part of the interaction on
channels with \( \ell \) presence depends only on \( \ell \)-
observable parts of input on channels with \( \sqsubseteq \ell \)
presence, thus enforcing a noninterference policy.
Figure 3.4 illustrates the intuition in our handling
of the channels with fine-grained security levels
for the two-level lattice. In addition to propagat-
ing low input to the high run (as in Figure 3.2),
we propagate to the high run the fact that an M
message has arrived to the low run. This allows
consistent processing of the message. At the out-
put, the presence of an M message is observable
at the low level (cf. dashed output arrow). On the other hand, the value of an M
output is produced by the high run (cf. solid output arrow).

Our SME of \( s \) runs, concurrently, a copy of \( s \) for each level in the security
lattice. The SME of \( s \) can input on \( c \) iff the \( \pi(c) \)-run can input on \( c \). An \( \ell \)-run
which can consume a \( c \)-input with \( \pi(c) \not\subseteq \ell \) is fed a (constant, pre-determined,
input-independent) default value, denoted \( \mathfrak{d} \), by SME. An \( \ell \)-run which can consume its \( n \)th \( c \)-input with \( \pi(c) \supseteq \ell \) gets a copy of the \( n \)th input consumed by the \( \pi(c) \)-run, unless \( \pi(c) \) is yet to consume \( n \) \( c \)-inputs, in which case the \( \ell \)-run blocks until the \( \pi(c) \)-run has done so. However, if \( \kappa(c) \not\supseteq \ell \), then the \( \ell \)-run is fed \( \mathfrak{d} \) instead of the value in the \( n \)th \( c \)-input. The SME of \( s \) can output on \( c \) iff the \( \pi(c) \)-run can output on \( c \). A \( c \)-output produced by a \( \ell \)-run for which \( \pi(c) \neq \ell \) is discarded by SME (as opposed to being sent to the environment). When an \( \ell \)-run produces its \( n \)th \( c \)-output with \( \ell = \pi(c) \), this output is sent straight to the environment, except when \( \pi(c) \neq \kappa(c) \); in that case SME first checks whether the \( \kappa(c) \)-run has already produced its \( n \)th \( c \)-output. If so, then the value of the \( n \)th \( c \)-output produced by the SME of \( s \) becomes the value of the \( n \)th \( c \)-output produced by the \( \kappa(c) \)-run. Otherwise, the value becomes \( \mathfrak{d} \).

We now formalize SME for arbitrary \( s \) satisfying the assumptions in Definition 3.2. Concurrent executions of \( s \) are scheduled by a scheduler.

(3.3) **Def**. A scheduler \( \sigma \) is a LTS with labels ranged by \( \ell \). \( \sigma \) is deterministic iff
\[
\forall \ell, \ell', \sigma \subseteq \ell \land \sigma \subseteq \ell' \implies \ell = \ell'.
\]
\( \sigma \) is fair iff \( \forall \ell, \sigma \not\subseteq \ell \implies \forall \ell'. \exists \ell' \cdot \sigma \Rightarrow \ell'. \)

Unless stated otherwise, \( \sigma \) is deterministic and fair. An example of deterministic and fair schedulers is the round-robin schedulers. For instance, for \( L_{\text{LLH}} \), a scheduler which infinitely repeats H.L or L.H is a deterministic fair scheduler.

The semantics of SME is given in Figure 3.5. A SME state is a triple \((\tilde{a}, \sigma, S)\), where \( \tilde{a} \) is the list of actions which the SME has performed so far, \( \sigma \) the state of the scheduler, and \( S \) contains the state of the \( \ell \)-runs. \( S \) maps each security level \( \ell \) to a pair \((\tilde{a}_\ell, s_\ell)\), where \( \tilde{a}_\ell \) is the list of actions which the \( \ell \)-run has performed so far, and \( s_\ell \) is the current state of the \( \ell \)-run. For a given \( \sigma \), the SME of \( s \), SME\((\sigma, s)\), is defined as SME\((\sigma, s) = (\epsilon, \sigma, L_{\text{H}} \Rightarrow (\epsilon, s))\)). The derivation of any SME state transition begins with the rule

\[
\frac{\tilde{a} \models (\sigma, S) \xrightarrow{a} (\sigma', S')} {\tilde{a}, \sigma, S \xrightarrow{a} (\tilde{a}.a, \sigma', S')} \quad \text{[log]}
\]

The purpose of (log) is to keep track of the interaction \( \tilde{a} \) which SME\((\sigma, s) \) has had with the environment. Note that

\[
\text{SME}(\sigma, s) \xrightarrow{a} (\tilde{a}', \sigma', s') \iff \tilde{a} = \tilde{a}',
\]

so we sometimes omit the trace label on a SME transition. We put \( \tilde{a} \) on the left side of “\( \Rightarrow \)” in the next layer of the semantics, Figure 3.5b, to show that this layer only reads \( \tilde{a} \). The rules at this layer are mainly responsible for, by use of \( \sigma \), deciding which \( \ell \)-run takes a step next, using the rules in the third layer, Figure 3.5a. This layer is responsible for hiding from the previous layer all the different ways which an \( \ell \)-run can take a step without interacting with the environment ((dead), (silence), (old-o), (old-i)), and signaling to the previous layer when the \( \ell \)-run requires I/O with the environment to proceed ((new-o), (new-i)). Each rule at this level appends to the \( \ell \)-run trace the action the \( \ell \)-run...
3.4 Fine-grained secure multi-execution

\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \xrightarrow{\bullet} (\tilde{a}_\ell, s) \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s) \]
\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \] if \( \kappa(c) \not\subseteq \ell \) then \( v = d \)
\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \] if \( \kappa(c) = \ell \) then \( v = v_\ell \) else \( v = d \)
\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \]
\[ \pi(c) \not\subseteq \ell \]
\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \] if \( v = \star \lor \kappa(c) \subseteq \ell \) then \( v_\ell = v \) else \( v_\ell = d \)
\[ s \xrightarrow{\mathcal{CV}} s' \]
\[ (\tilde{a}, \ell) \models (\tilde{a}_\ell, s) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]

(a) SME $\ell$-stepper

\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\bullet} (\tilde{a}_\ell, s) \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s) \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s) \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s) \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]
\[ \sigma \xrightarrow{\ell} \sigma' \]
\[ (\tilde{a}, \ell) \models S(\ell) \xrightarrow{\mathcal{CV}} (\tilde{a}_\ell, s') \]

(b) SME $\ell$-chooser

Figure 3.5: Semantics of SME
performed during the step (not necessarily the same action as the one performed by the SME state). We equate a terminated \( \ell \)-run with an infinitely silent one, as indicated by (dead) (not making terminated runs unschedulable excludes several timing attacks described by Kashyap et al. [22]). SME stores output on channels with presence \( \not\in \ell \) without forwarding it to the environment, as per (old-o). (old-i) covers multiple scenarios for not inputting from the environment on \( c \). (new-o) notifies the previous layer that the \( \pi \)-run performed the corresp. input action. Otherwise, input \( d \) instead. (new-i) indicates that the \( \ell \)-run requires input to proceed, and for the value \( v \) received from the previous layer, indicated on the transition label, instead feeds \( d \) to the \( \ell \)-run iff \( v \neq \star \wedge \kappa(c) \not\subseteq \ell \). The previous layer has two rules for this scenario. When \( \pi(c) \not\subseteq \ell \), then the \( \ell \)-run blocks until the \( \pi(c) \)-run reads on \( c \), by (block-i). When \( \pi(c) \) reads on \( c \), the input is fed to every \( \supseteq \pi(c) \)-run blocking on \( c \), by (i). (new-o) notifies the previous layer that the \( \ell \)-run has a fresh output for the environment. Rule (o) in the previous layer handles this scenario, checking if the \( \kappa(c) \)-run has already provided content for this output, and if so, replaces the value in the output with the value in the corresp. \( \kappa(c) \)-run \( c \)-output. When \( \kappa(c) \neq \pi(c) \), the output value is replaced by \( d \) iff \( \kappa(c) \) has not yet produced the corresponding value. Having the \( \pi(c) \)-run instead wait for the \( \kappa(c) \)-run to reach the corresp. \( c \)-output, or giving responsibility of producing the \( c \)-output to the \( \kappa(c) \)-run, can introduce a leak:

\[
\begin{array}{l}
in \text{ M h;} \; l := 0; \; \text{while} \; l != h \{l := l +1\}; \; \text{out} \; \text{ M h}
\end{array}
\]

Here the time it takes for the \( H \)-run to produce the \( M \)-output, and whether \( H \) produces the output at all, depends on \( h \).

While Figure 3.5b says that SME controls each step of each \( \ell \)-run, in practice the responsibility of SME can be distributed to the \( \ell \)-runs as follows. Each \( \ell \)-run is made responsible for environment I/O on all \( c \in \pi^{-1}(\ell) \), since SME(s) performs I/O iff the \( \ell \)-run of \( s \) performs it. Each \( \ell \)-run makes input on each \( c \in \pi^{-1}(\ell) \) and output on each \( c \in \kappa^{-1}(\ell) \) available in a shared resource (e.g. memory) s.t. \( \subseteq \ell \)-runs can obtain a copy of the input when they need it, and an \( \pi(c) \)-run can obtain the actual value to output on \( c \). Each \( \ell \)-run processes (after sharing, when \( \ell' = \pi(c) \)) \( d \) in place of the inputted value when \( \pi(c) \not\subseteq \ell \subseteq \kappa(c) \), and outputs \( d \) when the \( \kappa(c) \)-run is yet to share the value to put into the output when \( \pi(c) \not\subseteq \kappa(c) = \ell \). This approach is taken in a SME benchmark by Devriese and Piessens [16]. Forcing \( \ell \)-runs to diverge and recording full traces can be avoided [22, 16]. This approach is sound as long as the \( \ell \)-run threads cannot influence the scheduler.

While \( s \) is input-blocking, SME(s) is not; varied presence of input on \( c \in \kappa^{-1}(\ell) \) cannot impede progress or timing of \( \ell \)-runs where \( \ell \not\in \ell' \). This effect is achieved by \( \prec \star \) actions; if SME(s) is in a state where an \( \ell \)-run wants input on \( c \), and \( I \) does not have one ready (yet), the \( \ell \)-run can do a \( \prec \star \) action, allowing SME(s) to pass control to another \( \ell' \)-run. In contrast, the formalization (as op-
posed to the benchmark implementation) of SME by Devriese and Piessens [16] is input blocking; if an $\ell$-run is scheduled before a $\ell'$-run with $\ell \not\subseteq \ell'$, the nonpresence of input on $c \in \pi^{-1}(\ell)$ can interfere with the $\ell'$-run. This hinders sound scheduling of runs for arbitrary nonlinear lattices.

### 3.4.1 Soundness

SME enforces the following property: Under observably equivalent environments, the respective sets of traces produced under any of them are observably equivalent.

**Definition 3.4** $s$ is timing-sensitive, progress sensitive noninterfering ($s \in \text{TSNI}$) iff \[ \forall \ell, I_1, I_2 \cdot I_1 \approx_\ell I_2 \implies \forall \vec{a}_1 \cdot I_1 \models s \vec{a}_1 \implies \exists \vec{a}_2 \cdot I_2 \models s \vec{a}_2 \land \vec{a}_1 \approx_\ell \vec{a}_2. \]

**Theorem 3.5** \[ \forall \sigma, s \cdot \text{SME}(\sigma, s) \in \text{TSNI}. \]

In contrast to Devriese and Piessens [16], who prove soundness for a cooperative scheduler for linear lattices, and to Kashyap et al. [22], who prove soundness for two round-robin schedulers (the “Multiplex-2” and “Lattice-based” approaches), we prove a more general result: soundness for arbitrary deterministic and fair schedulers. While Devriese and Piessens claim their scheduler, called select\text{lowprio}, which executes the $\ell$-runs to completion in increasing order by $\sqsubseteq$, works for any linearized lattice, Kashyap et al. have shown that select\text{lowprio} introduces a timing dependency between $\ell$-runs at incomparable levels in nonlinear lattices. For instance, with $L \defeq \{H, A, B, L\}$ and $\sqsubseteq$ being the reflexive transitive closure of $\{(L, A), (L, B), (A, H), (B, H)\}$, with linearization $L \sqsubseteq A \sqsubseteq B \sqsubseteq H$ and $d = 0$, the time it takes for $!B^1$ to occur is, under select\text{lowprio}, a function of the input on $A^A$.

```plaintext
in A^A a; while a != 0 { a := |a| - 1 }; out B^B 1
```

In the presence of nontotal environments, the situation is even worse; here the presence of input on $A^A$ leaks to $B^B$.

```plaintext
in A^A a; out B^B 1
```

While swapping $A$ and $B$ in the linearization resolves the issue in this program, the following program has no linearization of $L$ for which select\text{lowprio} schedules soundly.

```plaintext
in A^A a; out B^B 1; in B^B b; out A^A 1
```

We show in Appendix 3.8 that the assumption of termination is problematic when program input arrives arbitrarily in time.

The proof of Theorem 3.5 is a corollary of the following lemma, which can be seen by removing the last two elements in the conjunction in the conclusion of the lemma, and comparing the result with Definition 3.4. The details of this, and all other proofs, are in the appendix. We write $S_1 =_\ell S_2$ iff $\forall \ell'' \subseteq \ell \cdot S_1(\ell'') = S_2(\ell'').$
Lemma \((3.6)\) \(\forall s, \sigma, \ell, I_1, I_2 \models I_1 = \ell I_2 \implies\)

\(\forall \vec{a}_1, \sigma_1, S_1 \cdot I_1 \models \text{SME}(\sigma, s) \to (\vec{a}_1, \sigma_1, S_1) \implies\)

\(\exists \vec{a}_2, \sigma_2, S_2 \cdot I_2 \models \text{SME}(\sigma, s) \to (\vec{a}_2, \sigma_2, S_2) \land\)

\(\vec{a}_1 = \ell \vec{a}_2 \land S_1 = \ell S_2 \land \sigma_1 = \sigma_2\)

Such a strong correspondence is achievable since, for each \(\ell' \subseteq \ell\), the \(\ell'\)-run of \(s\), in \(I_1 \models \text{SME}(\sigma, s)\) and \(I_2 \models \text{SME}(\sigma, s)\), behaves as \(I \models s\), where

\[(I \models s}(c)\) = \begin{cases} \text{false} & \text{if } \pi(c) \not\subseteq \ell' \\ I(c) & \text{otherwise.} \end{cases}\]

While for \(\pi(c) \subseteq \ell'\), the number of \(?c\) preceding a \(?c\) of an \(\ell'\)-run in \(I_j \models \text{SME}(\sigma, s)\) compared to \(I \models s\) can differ due to \(\sigma\), this number is the same in \(I_1 \models \text{SME}(\sigma, s)\) compared to \(I_2 \models \text{SME}(\sigma, s)\). Since \(s\) is input-blocking, all three are in the same state at the time of the non-blank read, and consume the same input, by \(I \models s\). Thus, since \(I_1 \models \text{SME}(\sigma, s)\) and \(I_2 \models \text{SME}(\sigma, s)\) are both run under the same \(\sigma\), after any number of transitions, the \(\ell'\)-runs will in both runs have performed the same number of actions, consumed the same inputs, produced the same outputs, and be in the same state.

### 3.4.2 Transparency

We show that SME does not adversely modify the I/O behavior of a program for which changes in \(\ell\)-unobservable input does not affect the \(\ell\)-observable parts of the I/O behavior of the program. We obtain this class of programs by weakening TSN to a timing-insensitive variant by replacing, in Definition 3.7, \(\approx_{\ell}\) with \(\approx_{\ell'\ell}\).

Definition \((3.7)\) \(s\) is timing-insensitive, progress sensitive noninterfering (\(s \in \text{PSNI})\) iff \(\forall \ell, I_1, I_2 \cdot I_1 \models s \approx_{\ell} I_2 \implies\)

\(\forall \vec{a}_1 . I_1 \models s \vec{a} \implies\)

\(\exists \vec{a}_2 . I_2 \models s \vec{a} \land \vec{a}_1 \approx_{\ell} \vec{a}_2\).

Let \(s \in \text{PSNI}, \ell, I\) and \(\sigma\) be arbitrary. In \(s\) and \(\text{SME}(\sigma, s)\), the interaction on \(\ell\)-presence channels is \(\ell\)-equivalent.

Theorem \((3.8)\) \(\forall s \in \text{PSNI}, I, \vec{a}.\)

a) \(I \models s \vec{a} \implies\)

\(\exists \vec{a}' . I \models \text{SME}(\sigma, s) \vec{a} \implies \forall \ell . \vec{a} \leq_{s, \ell, \pi^{-1}(\ell)} \vec{a}'\)

b) \(\forall \sigma . I \models \text{SME}(\sigma, s) \vec{a} \implies\)

\(\exists \vec{a}' . I \models s \vec{a} \implies \forall \ell . \vec{a} \leq_{s, \ell, \pi^{-1}(\ell)} \vec{a}'\)

When all \(\ell\)-presence outputs also have \(\ell\)-content, the \(\ell\)-presence interaction in \(s\) and \(\text{SME}(s)\) is the same. This is an improvement on Theorem 2 in [16] which establishes the \(a\)-part of Theorem 3.8 for the interaction on each channel (as opposed to, for each \(\ell\), the interaction on all channels with presence level \(\ell\)), and
only for terminating runs of termination-sensitive $s$. Furthermore, under $L_{\text{AB}}$,
\texttt{select}_{\text{lowprior}} yields a nontransparent run for the following program, as no $B^{01}$ occurs if no input on $A^h$ arrives.

out $B^0$ 1; in $A^h$ a

However, when $s$ outputs on $c$ with $\kappa(c) \sqsupseteq \pi(c)$, SME$(\sigma, s)$ might replace the value in its corresponding output with $d$. Thus, the timing behavior of $s \in \text{PSNI}$ can impede the ability of a $\sigma$ to soundly schedule runs in SME$(\sigma, s)$ such that the $\kappa(c)$-run reaches the output before the $\pi(c)$-run does irrespective of previously inputted values. In TSNI programs, however, all $\ell$-runs for which $\pi(c) \sqsubseteq \ell$ will reach an output on $c$ after the same number of reduction steps. This includes the $\kappa(c)$-run, since $\pi(c) \sqsubseteq \kappa(c)$. Thus, if we ensure that any $\ell$-run never “outruns” its parent-runs, we can ensure that the content-provider of an output reaches the output before its presence-provider does. It is, however, not sufficient to require, for instance, that at any given point, $H$ has been scheduled more often than $L$, as the $H$-run can waste its turns blocking on channels with presence level $\sqsubseteq H$. The following predicate, when invoked as $\phi(\ell_H, \ell_L, 0, \ell)$, yields $1$ when $\ell_H$, a parent of $\ell_L$, has been scheduled in this manner in $\ell$, and $0$ otherwise.

$$
\begin{align*}
\phi(\ldots, \varepsilon) &= 1 \\
\phi(\ell_H, \ell_L, b_{\text{seen}}, \ell, \ell) | \ell = \ell_H &\quad = \phi(\ell_H, \ell_L, 1, \ell) \\
| \ell = \ell_L \land b_{\text{seen}} &\quad = \phi(\ell_H, \ell_L, 0, \ell) \\
| \ell = \ell_L \land \neg b_{\text{seen}} &\quad = 0 \\
| \text{otherwise} &\quad = \phi(\ell_H, \ell_L, b_{\text{seen}}, \ell)
\end{align*}
$$

(3.9) \textbf{Definition} \quad \sigma \text{ is a high-lead scheduler (} \sigma \in \text{highlead} \text{) if } \forall \ell \cdot \sigma \overset{\ell}{\to} \implies \forall \ell_L, \ell_H \cdot \ell_L \sqsubseteq \ell_H \implies \phi(\ell_H, \ell_L, 0, \ell).

An example of a high-lead scheduler is the round-robin scheduler which schedules levels in (increasing) order of maximal descendancy from the top element in the security lattice (ties broken arbitrarily). For instance, for $L = \{H, A, B, C, L\}$ and $\sqsubseteq$ being the reflexive transitive closure of $\{(L, C), (L, A), (C, B), (A, H), (B, H)\}$, $\sigma$ which infinitely repeats H.B.A.C.L or H.A.B.C.L is a high-lead scheduler. It is for these schedulers that, in $s \in \text{TSNI}$ and SME$(\sigma, s)$, the interaction on $\ell$-presence channels is the same, for all $\ell$.

(3.10) \textbf{Theorem} \quad \forall s \in \text{TSNI}, \sigma \in \text{highlead}, I, \hat{a}.

a) $I \models s \overset{\hat{a}}{\to} \implies \exists \hat{a}' \cdot I \models \text{SME}(\sigma, s) \overset{\hat{a}'}{\to} \land \forall \ell \cdot \hat{a} \leq_{* \pi^{-1}(\ell), \bullet} \hat{a}'$

b) $I \models \text{SME}(\sigma, s) \overset{\hat{a}}{\to} \implies \exists \hat{a}' \cdot I \models s \overset{\hat{a}'}{\to} \land \forall \ell \cdot \hat{a} \leq_{* \pi^{-1}(\ell), \bullet} \hat{a}'$

Thus, if SME puts a $d$ in an $M$ output when run using $\sigma \in \text{highlead}$, then this must have been done to prevent a (timing or progress) leak – a desired effect.
3.5 Declassification

The challenge for declassification in SME is limited communication of information from the high to the low run. This is non-trivial as SME is originally designed to prevent any such leaks. This section demonstrates how information can be intentionally released in SME without violating the guarantees SME was originally designed to provide.

It turns out that the communication model from Section 3.4 is an excellent fit for secure communication between the runs at different levels. A key desired property is to prevent the occurrence of declassification events from leaking information about the context while allowing intended release of the value to be declassified. This is a convenient match with our model that distinguishes the security levels of presence and content. The core idea is depicted in Figure 3.6. Declassification essentially corresponds to routing an M output from the high run into the low run.

A release policy $\rho$ is a subset of $\sqsubseteq$. It indicates which information releases are allowed. When $\rho = \emptyset$, then $\rho$ indicates a classical no-downward-flows policy, that is, one with no information release. When $(\ell, \ell') \in \rho$, then $\rho$ permits the downward flow from $\ell$ to $\ell'$. We write $\ell_0 \rho \ell$ iff

$$\exists \ell_0, \ldots, \ell_n \cdot \ell_n \sqsubseteq \ell \land \forall 0 \leq i < n \cdot (\ell_i, \ell_{i+1}) \in (\sqsubseteq \cup \rho).$$

(3.11) **DEFINITION** \( I_1 \) and \( I_2 \) are $\rho$-$\ell$-equivalent ($I_1 \equiv_\ell^{\rho} I_2$) iff

$$I_1 \equiv_\ell I_2 \land \forall \ell' \cdot \ell' \rho \ell \implies I_1 \equiv_\ell I_2.$$

(3.12) **DEFINITION** \( s \) is $\rho$-releasing TSNI ($s \in \text{TSNI}_\rho$) iff $\forall \ell, I_1, I_2 \cdot I_1 \equiv_\ell^{\rho} I_2 \implies$

$$\forall \tilde{a}_1 \cdot I_1 \models s \triangleleft \tilde{a}_1 \implies \exists a_2 \cdot I_2 \models s \triangleleft \tilde{a}_2 \land \tilde{a}_1 \equiv_\ell \tilde{a}_2.$$

This noninterference policy only states what can be released, without constraining where during control flow information release is permitted. A common construct for aiding programmers in specifying information release, is the construct $\text{declassify}(e, \ell)$, which declassifies the value of expression $e$ (in the state in which this command is executed) to level $\ell$. For instance, only $b$ is intended to leak to L in the below program $p_{d1}$ under lattice $\mathcal{L}_{AB}$ with $\rho = \{(B, L)\}$, but it turns out $a$ leaks as well; a TSNI$_\rho$-insecurity.

```plaintext
in A \ a ; in B \ b ; // p_{d1}
if a { l := declassify(b, L) }
out L \ l
```
The only interface SME has to its $\ell$-runs are action labels. To transfer the right value from the $R$-run to be declassified by the $L$-run, we need to enable the environment (in our case SME) to obtain the value which an $s$ wishes to declassify from an action label, and (optionally) subsequently replace it. To achieve this, we introduce release channels. Let $R \subseteq C$ be the release channels, ranged by $r$. Let $C = C \setminus R$ be the communication channels, ranged by $c$. Let $\varrho : L^2 \rightarrow R$ be bijective. $\varrho$ associates a release channel $\varrho(\ell, \ell')$ with each kind of information release $(\ell, \ell')$ (not all of which are permitted by $\rho$). The following inference rule illustrates how the semantics of declassification, in a simple imperative language with I/O, can be given such that the environment can (optionally) have a program declassify a completely different value (through "in $r$") than the value the program announced it would declassify (through "out $r v$"). This declassify construct is more fine-grained than the standard one as it specifies both the from level $\ell$ and the to level $\ell'$ of the declassification operation.

$$m \models e = v \quad \varrho(\ell, \ell') = r \quad (m, \text{out } r v) \xrightarrow{\varrho} (m', \text{skip})$$

$$(m, x := \text{declassify}(e, \ell \rightarrow \ell')) \xrightarrow{\varrho} (m', x := \text{in } r)$$

To restore the typical semantics of declassification (which does not make declassification an effect) in an $s$, we simply place $s$ in a wrapper which makes communication on release channels a feedback, as per $F(s)$, given below. $F(s)$ only interacts with its environment on communication channels.

$$s \xrightarrow{r v} s' \quad s \xrightarrow{r v} s' \quad s \xrightarrow{\overline{r} v} s' \quad s \xrightarrow{r} \overline{v} a = ! r v$$

$$F(s) \xrightarrow{? r v} F(\overline{r} v, s') \quad F(\overline{r} v, s) \xrightarrow{?} F(s') \quad F(s) \xrightarrow{\overline{r} v} F(s')$$

This wrapper only has the desired effect if $s$ communicates on release channels in the expected manner, by always first performing an output on $r$, and subsequently performing an input on $r$. We define the class of such $s$ now.

(3.13) **Definition** An LTS$_{IO} \lambda = (S, L, \rightarrow)$ is an LTS$_{IO}$ with release (LTS$_{IO}^R$) iff

$$\forall s \in S, r, v, \overline{a}. \quad (s \xrightarrow{\overline{a} r v} \overline{a}', s' \xrightarrow{\overline{a} r v, v r v'} \overline{a}')$$

$\lambda$ is an LTS$_{IO}$ without release (LTS$_{IO}^C$) iff

$$\forall s \in S, \overline{r} v, \overline{a}, v. \quad s \xrightarrow{\overline{a} r v} \overline{a}' \forall s \xrightarrow{\overline{a} r v} \overline{a}'$$

We write $s \in$ LTS$_{IO}^R$ (resp. $s \in$ LTS$_{IO}^C$) iff $s$ is a state in some LTS$_{IO}^R$ (resp. LTS$_{IO}^C$).

We consider only LTS$_{IO}^R$ and LTS$_{IO}^C$ $s$ for the remainder of this section; an $s$ which is neither interacts on some $r$ in an undesired way. We require that $\forall r. \pi(r) = \kappa(r) \land \exists \ell'. \varrho(\kappa(r), \ell') = r$. Read this as "$r$ releases information from $\kappa(r)$ to $\ell'$." The information release which $r$ is responsible for is permitted by the release policy, iff, $(\kappa(r), \ell') \in \rho$.

The semantics of SME extended with release channels (SME$_R$) is given in Figure 3.7. It is parameterized by the release policy $\rho$. By $(r \text{-not})$, any release action which is not permitted by $\rho$ is a feedback. By $(r \text{-declassify})$, when a target $\ell$-run of a re-
(a) SME\(_R\) \(\ell\)-stepper; add Figure 3.5a rules w/ occurrences of \(c\) replaced with \(e\).

\[
\begin{align*}
    s \xrightarrow{r_{-\rightarrow}} s' \\
    (\ddot{a}, \ell) \models (\ddot{a}_f, s) \xrightarrow{r_{\rightarrow}} (\ddot{a}_f \rfloor_{rv} s) \\
    s \xrightarrow{r_{\rightarrow}} s' \\
    (\ddot{a}, \ell) \models (\ddot{a}_f, s) \xrightarrow{r_{\rightarrow}} (\ddot{a}_f \rfloor_{rv} s)
\end{align*}
\]

\(r-o\)

\[
\begin{align*}
    \text{releaseok}(r, \ell) &= (g^{-1}(r) = (\ell', \ldots) \in \rho \land \ell' \rho \ell \land \ell' \not\in \ell)). \\
    \sigma \xrightarrow{\ell} \sigma' \\
    (\ddot{a}, \ell) \models S(\ell) \xrightarrow{r_{\rightarrow}} (\ddot{a}_f \rfloor_{rv} \rfloor_{rv} s) \\
    \neg \text{releaseok}(r, \ell)
\end{align*}
\]

\(r-not\)

\[
\begin{align*}
    S(k(r)) &= (\ddot{a}_k, \ldots) \\
    |\ddot{a}_k|_{\rfloor_{rv}} &< |\ddot{a}_f|_{\rfloor_{rv}}
\end{align*}
\]

\(r-d\)

\[
\begin{align*}
    \ddot{a} \models (\sigma, S) \xrightarrow{\ell} (\sigma', S[\ell \mapsto (\ddot{a}_f \rfloor_{rd} s)]) \\
    \sigma \xrightarrow{\ell} \sigma' \\
    (\ddot{a}, \ell) \models S(\ell) \xrightarrow{r_{rd}} (\ddot{a}_f \rfloor_{rd} s) \\
    \text{releaseok}(r, \ell)
\end{align*}
\]

if \( (\exists \ell_d, \ddot{d}_d . S(\ell_d) = (\ddot{d}_d \rfloor_{rd} \ldots \rfloor_{rd} \ldots) \land |\ddot{a}_k|_{\rfloor_{rv}} = |\ddot{d}_d|_{\rfloor_{rv}} \land \text{releaseok}(r, \ell_d) ) \)
then \( v = d \) else \( v = \nu_k \)

\[
\begin{align*}
    \ddot{a} \models (\sigma, S) \xrightarrow{\ell} (\sigma', S[\ell \mapsto (\ddot{a}_f \rfloor_{rv} \rfloor_{rv} s)])
\end{align*}
\]

(b) SME\(_R\) \(\ell\)-chooser; add Figure 3.5b rules w/ occurrences of \(c\) replaced with \(e\).

Figure 3.7: Semantics of SME\(_R\)
lease reaches a release before the source \( \kappa(r) \)-run does, \( d \) is released instead (same justification as for the treatment of \( M \)-output in SME). By \( (r) \), if a valid release receiver has already received \( d \) for this release, so does the \( \ell \)-run. Otherwise \( \kappa(r) \) has already done the release, so the \( \ell \)-run receives the right value.

### 3.5.1 Soundness

SME\(_{R} \) is sound with regards to any release policy.

\[(3.14) \text{THEOREM } \forall s \in \text{LTS}_{\text{IO}}^{R}, \sigma, \rho \cdot \text{SME}_{R}(\sigma, s, \rho) \in \text{TSNI}_{\rho}.\]

The proof of this theorem follows a similar pattern as the proof of Theorem 3.5, utilizing a lemma which is near-identical to Lemma 3.6.

SME\(_{R} \) does not introduce an information release into a program which does not already release information. The following statements correspond to the conservativeness principle of declassification [38] that stipulates that the security condition for systems with no information release is equivalent to baseline noninterference.

\[(3.15) \text{THEOREM } \forall s \in \text{LTS}_{\text{IO}}^{R}, \sigma, \rho \cdot \text{SME}_{R}(\sigma, s, \rho) \in \text{TSNI.} \]

When no information release is permitted, Definitions 3.12 and 3.4 coincide (\( I_{1} \equiv_{\ell} I_{2} \)), since no step of a trace from \( s \) is derived using a rule from Figure 3.7.

\[(3.16) \text{COROLLARY } \text{TSNI}_{\emptyset} = \text{TSNI}. \]

SME\(_{R} \) prevents all downward flows in all \( s \) which do not announce information release through release actions. This is a corollary of Theorem 3.5, since no step of a trace from \( s \) is derived using a rule from Figure 3.7.

\[(3.17) \text{COROLLARY } \forall s \in \text{LTS}_{\text{IO}}^{C}, \sigma, \rho \cdot \text{SME}_{R}(\rho, s) \in \text{TSNI}. \]

### 3.5.2 Transparency

Information release can impede transparency, even in secure programs. By Corollary 3.17, when \( s \) releases information w/o announcing the release on a release channel, the corresponding control in SME\(_{R}(\rho, s) \) never receives the declassified value. With \( \rho = \{ (H, L) \} \), consider this \( s \).

```
in M \ h \ ; \ out \ L \ h
```

We have \( s \in \text{TSNI}_{\rho} \). However, the L-run in SME\(_{R}(\rho, s) \) never gets the H-value in the H\(^{-}\)-input, and thus, SME\(_{R}(\rho, s) \) cannot be transparent. A similar problem arises when the L-run reaches an information release from H before the H-run does; then the L-run instead receives \( d \). This occurs in the following program \( p_{d2} \) if L is scheduled too often before H.

```
in M \ h ; // p_{d2}
\[ l := \text{declassify}(h, H->L) ; \]
out L \ l
```
It turns out that these are the only inhibitors for transparency. We define the class of programs which only release information through release channels. The definition makes use of a wrapper which binds release channels internally.

\[
\begin{align*}
\frac{s \xrightarrow{rv} s'}{B(\tilde{a}, I, s) \xrightarrow{\bullet} B(\tilde{a} : \tilde{r}_v, I, s')} & \quad \frac{s \xrightarrow{a} s'}{\tilde{a} \xrightarrow{\tilde{r}_v, v \cdot a = \tilde{r}_v} B(\tilde{a}, I, s)} & \quad \frac{s \xrightarrow{\tilde{r}_v} I \models \tilde{a} : \tilde{r}_v}{B(\tilde{a}, I, s) \xrightarrow{\bullet} B(\tilde{a} : a, I, s')} \\
\end{align*}
\]

(3.18) DEFINITION \( s \) is TSNI modulo release (\( s \in \text{TSNI}_{\text{mod}} \)) iff \( \forall I \).

\( (\forall \tilde{r}_v, \tilde{a}_i : \tilde{r}_v \preceq L) \supseteq B(I, s) \in \text{TSNI} \).

Let \( \rho(s) = \{ q^{-1}(r) \mid \exists \tilde{a}, v \cdot s \xrightarrow{\tilde{r}_v, \tilde{a}} \forall s \xrightarrow{\tilde{r}_v, \tilde{a}} \} \) be the releases of \( s \). If \( \rho(s) \subseteq \rho \), then SME\( \subseteq \) will not prevent any declassifications; this enables transparency.

(3.19) THEOREM \( \forall s \in (\text{LTS}_{\text{IO}} \cap \text{TSNI}_{\text{mod}}), \sigma \in \text{highlead}, l, \tilde{a} \).

\( a) \quad I \models F(s) \xrightarrow{\tilde{a}} \supseteq \exists \tilde{a}'. I \models \text{SME}_R(\rho(s), \sigma, s) \xrightarrow{\tilde{a}'} \forall \ell \cdot \tilde{a} \leq_{s, \pi^{-1}(\ell)} \cdot \tilde{a}' \)

\( b) \quad I \models \text{SME}_R(\rho(s), \sigma, s) \xrightarrow{\tilde{a}} \supseteq \exists \tilde{a}'. I \models F(s) \xrightarrow{\tilde{a}'} \forall \ell \cdot \tilde{a} \leq_{s, \pi^{-1}(\ell)} \cdot \tilde{a}' \)

Program \( p_{d1} \) satisfies TSNI\( \subseteq \)mod\( \subseteq \). It is easy to see that if \( \sigma \in \text{highlead} \) and \( \rho = \{(H, L)\} \), then SME\( \subseteq \) routes the value announced by the H-run (which is the value received on M) to the L-run in the desired way, thus yielding a transparent run. When \( \rho = \emptyset \), SME\( \subseteq \) stops the forbidden declassification, retaining soundness. In program \( p_{d1} \), the SME\( \subseteq \) allows the declassification but prevents the implicit flow of \( a \) at the same time! This is a fruitful byproduct of the separation of computation into \( \ell \)-runs; the B-run never obtains H-information, and thus cannot leak it (not even implicitly). At last, SME\( \subseteq \) with \( \rho = \{(H, L)\} \), allows the announced declassification, but stops the explicit flow, in the following program. This indicates that SME\( \subseteq \) not only enforces what is released, but also where in the program release takes place. We discuss this further in Section 3.8.

```
in M h1 ; in M h2 ;
11 := declassify(h1, H->L) ;
12 := h2 ;
out L 11 ; out L 12
```

3.6 Full transparency

This section shows how to achieve fully transparency for secure multi-execution by barrier synchronization. Full transparency, in contrast to per-channel or per-level transparency, guarantees that our SME enforcement preserves the I/O behavior of secure programs, including the ordering of I/O messages. Thanks to such a strong property, we are able to deploy SME to detect attacks.
The core idea is pictorially summarized in Figure 3.8. In contrast to Figure 3.2, we are not ignoring the low output produced by the high run. Instead, we match it with the low output produced by the low run. If the program is secure, this approach guarantees that there may not be any deviation in this matching. Thus, if there is a deviation, it must be due to the insecurity of the original program. From this deviation, we can construct a counterexample against noninterference.

We formalize our approach for a two-level lattice in Figure 3.9. While nothing inhibits the H-run from performing non-L-presence actions (by (H-a) and (●)), a barrier forms when the H-run reaches a L-presence action (by (H-block)). The H-run then only proceeds once the L-run reaches a L-presence action. When the L-run reaches a L-presence action, and the H-run is yet to perform the corresponding action, a barrier forms. The L-run then only proceeds once the H-run has done one of two things. 1) reached an L-observable action before advancing t steps beyond the L-run (by (L-a)), or 2) not (by (L-timeout)). In 1), if the H-run reached a L-observable different action, we note an attack in α1. In 2), we note a timeout in α2 (which might be an attack). Input streams are constructed from traces using ι, defined as

\[(ι(α))_c = (α |_{\ast, \ast, \ast, \ast} \cdot (?c \ast)) ^ \infty.\]  

(3.20) **Definition**  An attack α is a 4-tuple (ℓ, I1, I2, α1) where I1 = ℓ I2 and I1 \models α1. α is an attack on s iff

1) I1 \models s \overset{\alpha_1}{\rightarrow}, and  
2) \forall α_2 \cdot I_2 \models s \overset{\alpha_2}{\rightarrow} \Rightarrow \alpha_2 \neq_{\ell} \alpha_1.

### 3.6.1 Soundness

SME\_T enforces a timing-insensitive noninterference notion. This is easily seen by observing that the traces produced by I \models SME\_T(σ, s) and I |_L \models s are \approx_L-equivalent.

(3.21) **Theorem** \forall σ, σ \cdot SME\_T(σ, s) \in PSNI.

By allowing the L-run to wait up to t steps for the H-run to match an L-observable, SME\_T(s) introduces a timing leak into s, and thus does not enforce TSNI. We note however that SME\_T(σ, s) can be wrapped in a black-box timing leak mitigator to alleviate this weakening of the soundness guarantee [3].

At the point where the H-run deviates from the L-run, the H-run is “frozen” (to avoid leaks), becoming semantically equivalent to a program producing • infinitely, by (conflict). A more practical approach would be to instead have the
\( s \xrightarrow{a} s' \)

if \( a = ?c \nu \) then if \( \kappa(c) \not\subseteq \ell \) then \( \nu = d \) else \( \nu \neq \star \)

\( (\hat{a}, \ell) \models (\hat{a}_H, s) \xrightarrow{a} (\hat{a}_H, a, s') \)

\( \hat{a}.a' \leq \hat{a} \quad \pi(a') = L \quad \hat{a}' \approx_L \hat{a}_H \quad \forall a. s \xrightarrow{a} a \neq_L a' \)

\( (\hat{a}, H) \models (\hat{a}_H, s) \xrightarrow{a} (\hat{a}_H, \ast, s') \)

\( s \xrightarrow{\mathcal{C}V} s' \quad \hat{a}_\ell = \ast, \ell. \quad \pi(c) \subseteq \ell \)

if \( \kappa(c) \subseteq \ell \) then \( \nu = \nu_\ell \) else \( \nu = d \)

\( (\hat{a}, \ell) \models (\hat{a}_\ell, s) \xrightarrow{\mathcal{C}V} (\hat{a}_\ell, \ast c \mathcal{V}_\ell, s') \)

(a) SME\( _T \) \( \ell \)-stepper; add (dead) and (silence) from Figure 3.5a

(\( \ast \)) as in Figure 3.5b (with \( \hat{a}_1, \hat{a}_2 \) equal in states before \& after \( \ast \)).

\( \sigma \xrightarrow{H} \sigma' \quad (\hat{a}, H) \models S(H) \xrightarrow{s} (\hat{a}_H, s_H) \quad \pi(a) = H \)

\( \sigma \xrightarrow{H} \sigma' \quad (\hat{a}, L) \models S(L) \xrightarrow{s} (\hat{a}_L, s_L) \quad \pi(a) = L \)

\( \sigma \xrightarrow{H-a} \sigma' \quad (\hat{a}, \ell) \models S(H) \xrightarrow{s} (\hat{a}_H, s_H) \quad \pi(a) = H \)

\( \sigma \xrightarrow{H-block} \sigma' \quad (\hat{a}, \ell) \models S(L) \xrightarrow{s} (\hat{a}_L, s_L) \quad \pi(a) = L \)

\( \sigma \xrightarrow{L-wait} \sigma' \quad (\hat{a}, L) \models S(L) \xrightarrow{s} (\hat{a}_L, s_L) \quad \pi(a) = L \)

\( \sigma \xrightarrow{L-a} \sigma' \quad (\hat{a}, H) \models S(H) \xrightarrow{s} (\hat{a}_H, s_H) \quad \pi(a) = L \)

(b) SME\( _T \) \( \ell \)-chooser

Figure 3.9: Semantics of SME\( _T \)
H-run behave like it would under SME(s) henceforth. We hypothesize (but do not prove) that this modification of SME\(_T\) yields a sound enforcement.

### 3.6.2 Transparency

Modulo •, SME\(_T\)(s) and s produce the same sequence of I/O (up to an attack or timeout in SME\(_T\)(s)). In contrast to e.g. Devriese and Piessens [16], who swap the order of outputs in the following two programs (linearization B \(\subseteq\) A),

\[
\begin{align*}
&\text{out } H \ 1 \ ; \ \text{out } L \ 1 \\
&\text{out } A^1 \ 1 \ ; \ \text{out } B^0 \ 1
\end{align*}
\]

the I/O correspondence is full. Furthermore, this result guarantees transparency even when s is insecure.

(3.22) **Theorem** \(\forall s, \sigma, I, \tilde{a}^*\).

a) \(I \models s \tilde{a} \implies \exists \tilde{a}' \cdot I \models \text{SME}\(_T\)(\(\sigma, s\)) \xrightarrow{\tilde{a}'} (\ldots, \epsilon, \epsilon) \land |\tilde{a}'|_{\ast, \ast} | \leq |\tilde{a}|_{\ast, \ast} | \land \tilde{a}' \not\leq_{\ast, \ast} \tilde{a}\)

b) \(I \models \text{SME}\(_T\)(\(\sigma, s\)) \xrightarrow{\tilde{a}} (\ldots, \epsilon, \epsilon) \implies \exists \tilde{a}' \cdot I \models s \xrightarrow{\tilde{a}'} \land \tilde{a} \approx \tilde{a}'\)

It is easy to see that if \(s \in \text{TSNI}\), then SME\(_T\) never generates attacks, and thus, s and SME\(_T\)(\(\sigma, s\)) have the same (that is, \(\approx_L\)-equivalent) I/O behavior.

### 3.6.3 Attacks

Any match deviation found by SME\(_T\)(s) (before a timeout), forms the basis of a concrete proof that \(s \notin \text{PSNI}\).

(3.23) **Theorem** \(\forall s, \sigma, I \cdot I \models \text{SME}\(_T\)(\(\sigma, s\)) \rightarrow (\ldots, \alpha, \epsilon) \implies \alpha\) is an L-attack on \(s\).

If a timeout (\(\hat{a}_2\)) is discovered before the discrepancy (\(\hat{a}_1\)), then the mismatch might be consequence of timeout, which is not necessarily the basis of a leak in \(s\).

We end this section with two PSNI-insecure programs, and explain attacks which SME\(_T\) finds on them. Consider

\[
\begin{align*}
&\text{in } M \ h \ ; \ \text{out } L \ h \\
&\text{in } M \ h \ ; \ \text{while } h \neq 0 \ \{ \ h := h - 1 \ \} \ ; \ \text{out } L \ 0 \\
&\text{in } M \ h \ ; \ \text{while } h \neq 0 \ \{ \ h := h - 1 \ \} ; \ \text{out } L \ 0
\end{align*}
\]

With \(d = 0\), \(t = 100\) and initial input 1, SME\(_T\)(s) generates an attack on \(s\) in \(\hat{a}_1\), \((L, I_1, I_2, ?M_d.1!L0)\), where \(I_1(M) = ?M_d.(?M\star)^\infty, I_2(M) = ?M1.(?M\star)^\infty\). Now consider

With \(d = 0\), \(t = 100\) and initial input \(-1\), SME\(_T\)(s) generates an attack on \(s\) in \(\hat{a}_2\), \((L, I_1, I_2, ?M_d.\cdot1!L0)\), where \(I_1(M) = ?M_d.(?M\star)^\infty, I_2(M) = ?M-1.(?M\star)^\infty\). With initial input 1, no attack is generated. With initial input 500, however, an attack is generated which is not an attack on \(s\) (it just took “too long” for the H-run to match \(!L0\).
3.7 Related work

Referring the reader for general overviews on language-based information-flow security [36], on dynamic information-flow control [19], and on declassification [38], we focus on related work on multi-execution.

Li and Zdancewic [25] observe that “a noninterfering program \( f(h, l) \) can usually be factored to a ‘high security’ part \( f_H(h, l) \) and a ‘low security part’ \( f_L(l) \) that does not use any of the high-level inputs \( h \). As a result, noninterference can be proved by transforming the program into a special form that does not depend on the high-level input.” They propose relaxed noninterference that allows information release through a set of prescribed syntactic expressions. This focus is on enforcing relaxed noninterference statically, by a security type system.

Russo et al. [33] sketch the idea of running multiple runs of a program, where each run corresponds to the computation of information at a security level. They discuss that by running the public computation ahead of the secret run, certain classes of timing attacks can be prevented.

Capizzi et al. [13] consider enforcement of secure information flow in the setting of an operating system. The enforcement is based on shadow executions as operating system processes for different security levels. They report on an implementation and an experimental study with benchmarks.

As discussed earlier, Devriese and Piessens [16] develop a general treatment of secure multi-execution at the application level and establish soundness and precision under the assumption of total environments (there is always new input), linear lattices and low priority scheduling.

Bielova et al. [9] investigate multi-execution in a reactive setting. Their model multi-executes Featherweight Firefox [11], a formalization of a web browser as a reactive system. The environments are not necessarily total, but the security guarantee is weaker (than Devriese and Piessens’ [16]): termination-insensitive noninterference. The I/O model targets the browser setting, with handlers under cooperative scheduling. The full version [10] contains an informal discussion of what the authors call sub-input-event security policies, which corresponds to more flexible policies on input events (flexible policies on output events are not considered). These policies are defined by projections that describe how much is visible at each level. This mechanism is however not formalized. A formalization would require reasoning about policy consistency: for example, projections for less restrictive levels should not reveal more than projections for more restrictive levels.

Kashyap et al. [22] show that the low-priority scheduling might exhibit timing leaks for non-linear security lattices, and present several sound schedulers. We show (Appendix 3.8) that in the presence of handlers, it is not necessary for the lattice to be non-linear to produce attacks on the low-priority scheduler. Timing leaks can freely occur in linear lattices, including the simple low-high lattice.

Jaskelioff and Russo [21] implement a monadic library for SME in Haskell. Austin and Flanagan [6] introduce faceted values to simulate secure multi-execution by execution on enriched values. Faceted values can be projected to the different
security levels. The projection theorem assures that a computation over faceted values faithfully simulates non-faceted computations. They show that faceted values guarantee termination-insensitive noninterference. Faceted values provide a viable alternative for an efficient implementation of our technique. Austin and Flanagan also show how to relax noninterference by facet declassification, based on robust declassification [45, 26]. Robust declassification operates on both confidentiality and integrity labels, requiring both data to be declassified and code that does the declassification to be trusted. This leads to the introduction of integrity labels to model trust and integrity checks that the declassification operation is not influenced by untrusted data. This corresponds to the who dimension of declassification [38]. Compared to this approach, our declassification focuses on the what dimension, specifying what source and sinks are affected. We are able to capitalize on what secure multi-execution is best at: built-in security against implicit flows. No matter where in the code declassification occurs, it will not leak information about the context. There is no need to track the integrity of the code in our model.

Barthe et al. [8] present a “whitebox” approach to secure multi-execution. They devise a transformation that guarantees noninterference via secure multi-execution for programs in a language with communication and dynamic code evaluation primitives.

De Groef et al. [18] implement secure multi-execution as an extension of the Firefox browser and report on experiments with browsing the web. Compared to the work by Bielova et al. [9], they multi-execute the actual scripts in web pages rather than the entire browser. The main focus of the experiments is to confirm that the enforcement does not modify the behavior of secure pages in the presence of simple policies.

Compared to the work above, this paper enriches secure multi-execution with the following features: (i) channels with distinct presence and content security levels, (ii) the what dimension of declassification for secure multi-execution, (iii) full transparency results that preserve the order of messages, and (iv) show how secure multi-execution can be used to detect attacks. To the best of our knowledge, none of these features have been previously explored in the context of secure multi-execution.

This paper stands on the ground laid by our previous work [30] on the foundations of security for interactive programs. This earlier work presents a general framework for environments as strategies, lifts the assumptions of the total environment, and distinguishes between the security level of message presence and content in the general setting. While the previous work provides an excellent starting point for the present paper, it does not treat secure multi-execution.

Unno et al. [42] focus on the problem of finding counterexamples against noninterference: pairs of input states that agree on the public parts and lead to paths that disagree on public outputs. Their technique combines type-based analysis and model checking to explore execution paths for programs that may cause insecure information flow. They show that this method is more efficient than model...
checking alone. In comparison, our attack-detection technique does not require program analysis and allows reasoning about the security of individual runs.

In an independent effort, Zanarini et al. [44] apply secure multi-execution for program monitoring in a reactive setting modeled by interaction trees. This work relates to our attack detection results, although it focuses on a more relaxed, progress-insensitive, security condition. Given a program, the goal is to construct a scheduler for secure multi-execution that mimics the execution of the original program. Whenever a deviation is detected, the execution is blocked to avoid leakage. This approach enforces progress-insensitive noninterference.

3.8 Conclusion

Secure multi-execution emerges as a promising technique for enforcing secure information flow. We have overviewed the pros and cons of secure multi-executions and identified most pressing challenges with it. This paper pushes the boundary of what can be achieved with secure multi-execution. First, we lift the assumption from the original secure multi-execution work on the totality of the input environment (that there is always assumed to be input) and on cooperative scheduling. Second, we generalize secure multi-execution to distinguish between security levels of presence and content of messages. Third, we introduce a declassification model for secure multi-execution that allows expressing what information can be released. Fourth, we establish a full transparency result by barrier synchronization of the runs at different security levels. Full transparency guarantees that secure multi-execution preserves the original order of messages in secure programs. We demonstrate that full transparency is a key enabler for discovering attacks with secure multi-execution.

Representing reactive systems in our setting is an interesting topic of future work. In a reactive setting, an incoming input event determines which handler may be triggered. We can model this by tagging input values with a channel id. The program can then pattern-match on the tag to dispatch the value to the handler associated with the channel.

Although our formal results on declassification focus on what information is released, the mechanism also supports where information is released. Indeed, our declassification mechanism restricts release to program points with declassification annotations. Therefore, we expect the mechanism to enforce a variant of intransitive noninterference [32]. An investigation of formal guarantees for the where dimension of declassification is a worthwhile topic for future work.

Future work also includes implementation and case studies. We plan to experiment with modifying the Firefox browser to accommodate our fine-grained, declassification-aware, and transparent secure multi-execution. The modification will allow us to multi-execute JavaScript code in an environment with preemptive scheduling of the runs at different levels.
Acknowledgments  Thanks are due to Frank Piessens for generous feedback, and to Daniel Hedin, and David Sands for the useful discussions. This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References


Appendix

FlowFox leak

The leak exploits the fact that FlowFox [18] multi-executes JavaScript with the low-priority scheduler on a per-event basis. Low priority implies that the low
run is executed first and without preemption by the high run. The low-priority scheduler first applies to the main code. If the main code sets event handlers, they are processed after the multi-execution of the main code. Low handlers are multi-executed. High handlers are only run once, at the high level.

Note that the problem with low-priority scheduling is fundamental because it is not possible to extend the low-priority discipline over multiple events—simply because it is not possible to run the low handlers that have not yet been triggered.

The security theorem in the abstract setting of secure multi-execution [16] takes advantage of the low-priority scheduler and establishes timing-sensitive security. This is intuitive because the last access of the low data occurs before any high data is accessed. This implies that whenever the timing behavior is affected by secrets, there is no possibility for the attacker to inspect the difference.

We show that the situation is different in the presence of handlers. All we need to do is to set a low handler to execute after the high run for the main code has finished. Then the low handler can inspect the computation time taken by the high run. For a simple experiment, we consider the default example policy from the FlowFox distribution\(^1\) in Listing 3.1.

```
//** Example policy file for FlowFox.

Detailed project information and contact address can be found on:

HOWTO: modify the policy rules at the end this file
**/

... non-customizable part of the policy skipped...

/**************************** E D I T *******/
/**************************** ********/
/**************************** ********/

/* Example label conditional function */
var cross_origin = function ([url]) { return (url.indexOf("same-origin") == -1); };

/* Examples */
SME.Label("nsIDOMHTMLDocument_GetCookie").as(SME.Labels.HIGH).default("eat=this");
SME.Label("nsIDOMHTMLImageElement_SetSrc").if(cross_origin).as(SME.Labels.LOW).else(SME.Labels.HIGH);
SME.Label("nsIDOMHTMLScriptElement_SetSrc").as(SME.Labels.LOW).if(cross_origin).else(SME.Labels.HIGH);
```

The listing omits the non-customizable part of the policy, focusing on the sources and sinks. This policy defines same-origin domain as **HIGH** and cross-origin domains as **LOW** (line 16). In order to protect cookies, secret source are defined by

\(^1\)https://distrinet.cs.kuleuven.be/software/FlowFox/
labeling `document.cookie` as `HIGH` (line 19). Lines 20 and 21 define the sinks that correspond to setting the source attributes of image and script HTML elements. These are labeled as `HIGH` for the same origin and `LOW` for the other origins. The intention is to prevent attacks that leak information about the cookie to third-party web sites (any sites other than the site of the web page origin).

Nevertheless, the code in the web page in Listing 3.2 leaks one bit of information about the cookie to the third-party web site `attacker.com`.

Listing 3.2: One-bit timing leak

```html
<html>
  <script>
    var c = new Date();
    var m = c.getTime();
    setTimeout(function() { leak(); }, 1);
    document.cookie = "1";
    //document.cookie = "0";
    var h = parseInt(document.cookie, 10);
    if (h > 0) {
      var t = 0;
      while (t < 1000000) {t++;}
    }
    function leak() {
      var d = new Date();
      var n = d.getTime();
      var x = n - m;
      var s = new Image();
      s.src = "http://attacker.com?v=" + encodeURIComponent(x);
    }
  </script>
</html>
```

Function `getTime()` of the `Date` object returns the number of milliseconds since the midnight of January 1, 1970. First, information about the cookie flows via `document.cookie` into variable `h` (line 8). Depending on the value of `h`, the program might take longer time to execute (line 10). As foreshadowed below, all we need to do is to get a time stamp at the beginning of execution (line 4) and after the high run has finished. The difference in time reveals whether `h` was zero. In order to bypass FlowFox’s multi-execution, we simply create a low handler (line 5) to perform the final time measurement (line 15). Running this page in FlowFox results in a request for an image with URL `http://attacker.com?v=496` (repetitive runs show slight fluctuation around the value of 496). Running the code with line 7 uncommented and line 6 commented out, results in a request for an image with URL `http://attacker.com?v=6` (repetitive runs fluctuate insiginificantly around the value of 6). Hence, we can reliably leak one bit of secret information about the cookies. Clearly, the leak can be easily magnified to leak the entire cookie by walking through it bit-by-bit in a simple loop and sending the results for each bit to the attacker.

Note that changing the policy for `getTime()` to return `HIGH` result does not
close the timing leak. The leak can be still achieved exploiting the difference in the internal timing behavior by a combination of low handlers [34].

While the leak outlined above is achieved by issuing a timeout event, other events (such as user-generated events and XMLHttpRequest) can be used to achieve the same effect.

The low-priority scheduler is both at the heart of the soundness results by Devries and Piessens [16] and at the heart of FlowFox [18]. The experiment points to a fundamental problem with low-priority scheduling. The leak demonstrates that the low-priority scheduler breaks timing-sensitive security and motivates the need for (fair) interleaving of the runs at different levels, as pursued in this paper.

**Projections**

A projection applies as far left as possible (unless indicated otherwise with parentheses). For instance, \( \bar{a} \bar{a} \uparrow \) means \( (\bar{a}, \bar{a}') \uparrow \), not \( \bar{a}.(\bar{a}') \uparrow \). For all projections, \( \epsilon \vdash = \epsilon \). Let $ range over \? and \!. The projections used in the paper are given in Figure 3.10.

**Corrections**

In Figure 3.5a, rule (old-i) is missing a premise \( v \neq \star \). The rule becomes

\[
\frac{s \xrightarrow{cc} s', \bar{a} \ell \cdot cv \leq_{\star, \ell, \cdot} \bar{a} \rightarrow (\bar{a}, \ell, s)}{(\bar{a}, \ell) \models (\bar{a}, \ell, s) \models (\bar{a}, \ell \cdot cv, s')}
\]

The lemma becomes

\[
(3.24) \text{Lemma } \forall s, \sigma, \ell, I_1, I_2 \models I_1 =_\ell I_2 \implies \forall a_1, \sigma_1, S_1 \cdot I_1 \models \text{SME}(\sigma, s) \rightarrow (a_1, \sigma_1, S_1) \implies \exists a_2, \sigma_2, S_2 \cdot I_2 \models \text{SME}(\sigma, s) \rightarrow (a_2, \sigma_2, S_2) \land a_1 =_\ell a_2 \land S_1 =_\ell S_2 \land \sigma_1 = \sigma_2.
\]

In Theorem 3.8, \( \sigma \) should be universally quantified. The theorem becomes

**Theorem 3.8.** \( \forall s \in \text{PSNI}, \sigma, I, \bar{a} \star.

\begin{align*}
a) & I \models s \models a \implies \\
& \exists a' \cdot I \models \text{SME}(\sigma, s) \models a' \land a' \star \models (\sigma, \pi^{-1}(\ell), \bar{a})' \\
b) & I \models s \models a \implies \\
& \exists a' \cdot I \models s \models a' \land a' \star \models (\sigma, \pi^{-1}(\ell), \bar{a})'
\end{align*}

In Section 3.5.2, in the definition of \( B(\bar{a}, I, s) \), only \( r \)-inputs in \( \bar{a} \) need to be consistent with \( I \), and \( B(\bar{a}, I, s) \) turns both input and output on release channels.
into \( \bullet \). The definition of \( B(\alpha, I, s) \) becomes

\[
\begin{align*}
\frac{s \not\rightarrow s'}{B(\alpha, I, s) \not\rightarrow B(\alpha.rv, I, s')} \quad & \quad \frac{s \not\rightarrow s'}{B(\alpha, I, s) \not\rightarrow B(\alpha.rv, I, s')} \\
B(\alpha, I, s) & \not\rightarrow B(\alpha.a, I, s')
\end{align*}
\]

In Section 3.6, the definition of \( \iota \) can be simplified to

\[
(\iota(\alpha))_L = (\alpha|_{?,c} \cdot (?)^\infty).
\]

In Definition 3.20, the relation for comparing the input streams \( I_1 \) and \( I_2 \) should be \( \equiv \). The definition becomes

(3.25) **Definition** An attack \( \alpha \) is a 4-tuple \((\ell, I_1, I_2, \alpha_1)\) where \( I_1 \equiv \ell I_2 \) and \( I_1 \models \alpha_1 \). \( \alpha \) is an attack on \( s \) iff

1) \( I_1 \models s \overset{d_1}{\rightarrow}, \) and

2) \( \forall \alpha_2 \cdot I_2 \models s \overset{d_2}{\rightarrow} \implies \alpha_2 \not\equiv \ell \alpha_1. \)

In Figure 3.9b rule (L-a), \( i_L \) should be \( a_H \). The rule becomes

\[
\begin{align*}
\sigma \downarrow \sigma' \quad & \quad (\alpha, L) \models S(L) \overset{d_2}{\rightarrow} (\alpha_L, s_L) \quad \pi(a_L) = L \\
S(H) = (\alpha_H, \ldots) \quad & \quad |\alpha_L| + t \geq |\alpha_H| \quad (\alpha, H) \models S(H) \overset{d_2}{\rightarrow} \pi(a_H) = L \\
\text{if} \quad & \quad (\alpha, H) \models S(H) \overset{d_2}{\rightarrow} \text{then} \ a = a_H \wedge \alpha_1' = \alpha \\
\text{else} \quad & \quad a = a_L \wedge \alpha_1' = \alpha \quad \text{if} \ a_L = a_H \quad \\
\text{else} \quad & \quad a = a_L \wedge \alpha_1' = \alpha_1.(L, l(\alpha_H), l(\alpha_L), \alpha_L)
\end{align*}
\]

\[
\overset{d}{\rightarrow} (\sigma, S, \alpha_1, \alpha_2) \overset{d}{\rightarrow} (\sigma', S[L \mapsto (\alpha_L, s_L)], \alpha_1', \alpha_2)
\]

(L-a)

**Proofs**

**Proof of Lemma 3.6.** Let \( s, \sigma, \ell, I_1 \) and \( I_2 \) such that \( I_1 \equiv \ell I_2 \) be given. We prove that

\[
\forall \alpha_1, \sigma_1, S_1 \cdot I_1 \models \SME(\sigma, s) \implies (\alpha_1, \sigma_1, S_1) \implies \\
\exists \alpha_2, \sigma_2, S_2 \cdot I_2 \models \SME(\sigma, s) \implies (\alpha_2, \sigma_2, S_2) \wedge \tag{3.1}
\]

by induction in \( n = |\alpha_1| \). Let \( \sigma_1 \) and \( S_1 \) be given such that \( I_1 \models \SME(\sigma, s) \implies (\alpha_1, \sigma_1, S_1) \).

\( n = 0: \) Then \( \alpha_1 = \epsilon \). Set \( \alpha_2 = \epsilon \). Then for \((\alpha_2, \sigma_2, S_2) = \SME(\sigma, s) = (\alpha_1, \sigma_1, S_1)\), we get that \( \alpha_2 = \alpha_1 = \epsilon, S_2 = S_1 = \lambda \ell \rightarrow s \) and \( \sigma_2 = \sigma_1 = \sigma \).

Since \( \sigma_1 \) and \( S_1 \) such that \( I_1 \models \SME(\sigma, s) \implies (\alpha_1, \sigma_1, S_1) \) were arbitrary, (3.2) holds for \( n = \).
Given $n$: Assume (3.2) for all $a$ for which $|a| = n$; this is the induction hypothesis (IH). Let $\tilde{a}_1$ be such that $|\tilde{a}_1| = n + 1$. Then $\tilde{a}_1 = \tilde{a}_1',\tilde{a}_1$ for some $\tilde{a}_1'$ and $\tilde{a}_1$. Let $I_1 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_1',\sigma_1',S_1') \rightarrow (\tilde{a}_1,\sigma_1,S_1)$. By (IH), we have for some $\tilde{a}_2',\sigma_2'$ and $S_2'$ for which $\tilde{a}_1' = \tilde{a}_2', S_1' = \tilde{a}_2'$ and $\sigma_1' = \sigma_2'$ that $I_2 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_2,\sigma_2',S_2')$. Let $a_2, \sigma_2$ and $S_2$ be chosen such that such that $I_2 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_2,\sigma_2',S_2') \rightarrow (\tilde{a}_2,\sigma_2,S_2)$, where we set $\tilde{a}_2 = \tilde{a}_2'.a_2$. For some $\ell_1, \sigma_1', \tilde{a}_1' = \text{ct}_{\ell_1} \sigma_1$. Since $\sigma_2' = \sigma_1'$, $\sigma_2' \subseteq \sigma_1$. So $\sigma_1 = \sigma_2$.

It remains to be shown that $\tilde{a}_1 = \ell \tilde{a}_2$ and $S_1 = \ell S_2$. Case on $\ell_1$.

$\ell_1 \not\subseteq \ell$: Case on the derivation of the last step in trace $I_1 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_1,\sigma_1,S_1)$. By considering all possible derivations of a step in Figure 3.5 and by observing that $\sigma_1' \subseteq \sigma_1$ and that $\ell_1 \not\subseteq \ell$, we see that $a_1|_\ell = \bullet$ and for each $\ell' \subseteq \ell$, $S_1(\ell') = S_1'(\ell')$, and thus $S_1 = \ell S_1'$. By analogous reasoning, $a_2|_\ell = \bullet$ and for each $\ell' \subseteq \ell$, $S_2(\ell') = S_2'(\ell')$, and thus $S_2 = \ell S_2'$. By transitivity of $=_{\ell}$ on objects ranged by $S_1$, $S_1 = \ell S_2$. By definition of $=_{\ell}$ on traces, by definition of $\tilde{a}_1$ and $\tilde{a}_2$, and since $\tilde{a}_1\setminus_\ell \tilde{a}_2$, $\tilde{a}_1 = \ell \tilde{a}_2$.

$\ell_1 \subseteq \ell$: Since $S_1'(\ell_1) = S_2'(\ell_1')$, Case on the Figure 3.5b-rule used in the derivation of the last step in trace $I_1 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_1,\sigma_1,S_1)$.

(\bullet): Here, $(\tilde{a}_1',\ell_1) \models S_1'(\ell_1) \triangleright (\tilde{a}_1^1,\sigma_1^1, S_1^1)$ and $S_1 = S_1'[\ell_1 \mapsto (\tilde{a}_1^1,\sigma_1^1, S_1^1)]$ for some $(\tilde{a}_1^1,\sigma_1^1, S_1^1)$.

We show that $(\tilde{a}_2',\ell_1) \models S_2'(\ell_1) \triangleright (\tilde{a}_2^1,\sigma_2^1, S_2^1)$ (\dagger). Thus, by setting $S_2 = S_2'[\ell_1 \mapsto (\tilde{a}_2^1,\sigma_2^1, S_2^1)]$, we obtain a derivation of the last step in trace $I_2 \models \text{SME}(\sigma, s) \rightarrow (\tilde{a}_2,\sigma_2,S_2)$ using Figure 3.5b-rule (\bullet). Since $S_1' = \ell S_2'$, we will get by definition of $S_1$, $S_2$ and of $=_{\ell}$ on objects ranged by $S$ that $S_1 = \ell S_2$. Also, since $\tilde{a}_1' = \ell \tilde{a}_2'$ and $a_1 = a_2 = \bullet$, we will get by definition of $\tilde{a}_1$, $\tilde{a}_2$, and of $=_{\ell}$ on traces that $\tilde{a}_1 = \ell \tilde{a}_2$.

Case on the Figure 3.5a-rule used in the derivation of $(\tilde{a}_1',\ell_1') \models S_1'(\ell_1') \triangleright (\tilde{a}_1^1,\sigma_1^1, S_1^1)$.

(\dagger): Let $S_1(\ell_1') = (\tilde{a}_1^1,\sigma_1^1, S_1^1)$. Then for some $c, v$ and $s', S_1^1 \xrightarrow{c,v} S_1'_{\ell_1}$ and $\tilde{a}_1^1 \xrightarrow{c,v} S_1'_{\ell_1}$. Case on $\pi(c)$ and $\kappa(c)$.

$\pi(c) \not\subseteq \ell_1$: Then the conditional in the rule yields $v = d$ since $\pi(c) \subseteq \kappa(c)$. Also, $\tilde{a}_x \xrightarrow{c,v} S_1'_{\ell_1} \leq_{\bullet,\bullet,c} \tilde{a}_y$ for any $\tilde{a}_x$. 


and \( \hat{a}_y \) since both sides of \( \leq_{\ast,E_1 \ast,E_2,E_3} \) project to \( \epsilon \). Thus (†) holds.

\[ \pi(c) \subseteq \ell_1 \land \kappa(c) \not\subseteq \ell_1: \] Then the conditional in the rule yields \( \nu = d \). For \( a_{E_1} \text{?c}\nu \leq_{\ast,E_1 \ast,E_2,E_3} a_{E_2} \) to hold, \( a_{E_2} \) must have at least as many \( c \)-input actions (excluding blanks) as \( a_{E_1} \text{?c}\nu \). This follows from \( a_{E_1} \text{?c}\nu \leq_{\ast,E_1 \ast,E_2,E_3} a_{E_2} \), \( a_{E_2} = \ell \) and \( \ell_1 \subseteq \ell \). Thus (†) holds.

\[ \kappa(c) \subseteq \ell_1: \] For \( a_{E_1} \text{?c}\nu \leq_{\ast,E_1 \ast,E_2,E_3} a_{E_2} \) to hold, the \( c \)-input actions (excluding blanks) in \( a_{E_2} \text{?c}\nu \) must prefix the \( c \)-input actions (excluding blanks) in \( a_{E_2} \). By \( a_{E_2} = \ell \) and \( \ell_1 \subseteq \ell \), we get that \( a_{E_2} \) and \( a_{E_2} \) have exactly the \( c \)-input events (excluding blanks). Thus, by \( a_{E_1} \text{?c}\nu \leq_{\ast,E_1 \ast,E_2,E_3} a_{E_2} \), \( a_{E_2} \text{?c}\nu \leq_{\ast,E_1 \ast,E_2,E_3} a_{E_3} \) holds. Thus (†) holds.

(block-i): Here, \( (\hat{a}_1', \ell_1) \models S_1'(\ell_1) \xrightarrow{3c} (a_{E_1}, s_{E_1}) \) for some \( c \) for which \( \pi(c) \subseteq \ell_1 \), \( S_1' = S_1[\ell_1 \mapsto (a_{E_1}, s_{E_1})] \), and \( a_1 = \bullet \).

The only Figure 3.5a-rule which can constitute this derivation is (new-i). By this rule, \( a_{E_1}' = a_{E_1}, a_{E_2}' = a_{E_2} \) and \( s_{E_1}' = a_{E_1}, s_{E_2}' = s_{E_2} \), where \( S_1'(\ell_1) = (a_{E_1}', s_{E_1}') \). Since \( S_1' = S_2, S_2'(\ell_1) = (a_{E_2}', s_{E_2}') \). Since \( a_1' = a_2' \), \( a_1 = a_2 = \bullet \), by transitivity of \( =_{\ast,E_1 \ast,E_2,E_3} \), \( a_{E_1}' = a_{E_1}, a_{E_2}' = a_{E_2} \). Thus, by setting \( S_2 = S_2'[\ell_1 \mapsto (a_{E_1}, s_{E_1})] \), we obtain a derivation of \( (a_{E_2}', \ell_1) \models S_2'(\ell_1) \xrightarrow{3c} (a_{E_2}, s_{E_2}) \), and thus of the last step in trace \( I_2 \models \text{SME}(\sigma, s) \rightarrow (a_2, \sigma, s_2) \) using Figure 3.5b-rule (block-i). Since \( S_1' = S_2' \), we will get by definition of \( S_1 \), \( S_2 \) and of \( \models =_{\ell} \) on objects ranged by \( S \) that \( S_1 = S_2 \). Also, since \( a_1' = a_2' \), \( a_1 = a_2 = \bullet \), we will get by definition of \( a_1, a_2 \), and of \( \models =_{\ell} \) on traces that \( a_1 = a_2 \).

(i): Here, \( a_1 = \text{?c}\nu \) for some \( c \) and \( \nu = \ell_1 \). Let \( L_1 \) be the set of security levels \( \ell_1 \subseteq \ell' \) for which \( (a_{E_1}', \ell') \models S_1'(\ell') \xrightarrow{3c} (a_{E_1}', s_{E_1}') \) for some \( a_{E_1}' \), \( s_{E_1}' \). We then have \( \ell_1 \in L_1 \), and \( S_1 \) defined as \( S_1(\ell') = (a_{E_1}', s_{E_1}') \) if \( \ell' \in L_1 \), and \( S_1(\ell') = S_1'(\ell') \) otherwise. For each \( \ell' \in L_1 \), the only Figure 3.5a-rule which can constitute the \( (a_{E_1}', \ell') \models S_1'(\ell') \xrightarrow{3c} (a_{E_1}', s_{E_1}') \) derivation is (new-i). By this rule, \( s_{E_1}' \xrightarrow{3c} s_{E_1}' \), \( a_{E_1}' = a_{E_1}, s_{E_1}' = a_{E_1}, \) and \( a_{E_1}' = a_{E_1}, s_{E_1}' = a_{E_1}, \) where \( S_1'(\ell_1) = (a_{E_1}', s_{E_1}') \) and \( \nu_{E_1}' = \nu \) if \( \kappa(c) \subseteq \ell' \) or \( \nu = \ast \), and \( \nu_{E_1}' = d \) otherwise.

Since \( a_1' = a_2', I_1 \models \text{?c}\nu \) \( I_2 \models \text{?c}\nu \) and \( \pi(c) \subseteq \ell_1 \subseteq \ell_1 \), setting \( a_2 = \text{?c}\nu \) such that \( I_2 \models \text{?c}\nu \) gives us that \( a_1 = a_2 \), and thus by definition of \( a_1, a_2 \), and of \( \models =_{\ell} \) on traces that \( a_1 = a_2 \). Let \( L_2 \) be the set of security levels \( \ell_2 \subseteq \ell' \) such that \( (a_{E_2}', \ell' \models S_2'(\ell') \xrightarrow{3c} (a_{E_2}', s_{E_2}') \) for some \( (a_{E_2}', s_{E_2}') \). For each \( \ell' \in L_2 \), the only Figure 3.5a-rule which can constitute the \( (a_{E_2}', \ell') \models S_2'(\ell') \xrightarrow{3c} (a_{E_2}', s_{E_2}') \) derivation is (new-i). By this rule, \( s_{E_2}' \xrightarrow{3c} s_{E_2}' \), \( a_{E_2}' = a_{E_2}, \) and \( a_{E_2}' = a_{E_2}, s_{E_2}' = a_{E_2}, \).
\( \tilde{a}_2 \), where \( S_2(\ell_1) = (\tilde{a}_2^I, s_2^I) \) and \( v_2^I = v' \) if \( \kappa(c) \subseteq \ell' \) or \( v = \star \), and \( v_2^I = d \) otherwise.

When \( v_1^I \) and \( v_2^I \) are both defined and when \( \ell'' \subseteq \ell \), \( v_1^I = v_2^I \triangleq v'' \); this is seen by regarding the condition placed on \( v_1^I \) and \( v_2^I \) by (new-i), and that \( v' = v \) when \( \kappa(c) \subseteq \ell \).

Let \( S_2 \) be defined as \( S_2(\ell'') = (\tilde{a}_2^I, s_2^I) \) if \( \ell'' \in L_2 \), and \( S_2(\ell'') = S_2(\ell''') \) otherwise. Let \( L = \{ \ell'' \mid \ell_1 \subseteq \ell'' \subseteq \ell \} \). Since \( (\tilde{a}_2^I, s_2^I) = S_2'(\ell'') = (\tilde{a}_2^I, s_2^I) \) for each \( \ell'' \in L, \tilde{a}_2^I \xrightarrow{\text{SE}} s' \iff \tilde{a}_2^I \xrightarrow{\text{SE}} s' \) for each \( v'' \) and \( s' \).

We show that \( \forall \ell'' \cdot \ell'' \in L_1 \land L' \iff \ell'' \in L_2 \land L' \). That is, that for each \( \ell'' \in L, \)

\[ \tilde{a}_2^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \text{ iff } \tilde{a}_2^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I. \]

For the cases of \( \ell'' \) for which \( \tilde{a}_1^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \), we have \( \ell'' \notin L \land L_1 \), and since \( \tilde{a}_1^I = \tilde{a}_2^I \) and thus \( \tilde{a}_2^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \) for all \( v'' \), we get \( \ell'' \notin L \land L_2 \).

For the cases of \( \ell'' \) for which \( \tilde{a}_2^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \), where \( v'' = v \) if \( \kappa(c) \subseteq \ell' \) or \( v = \star \), and \( v'' = d \) otherwise, we have \( \ell'' \notin L \land L_1 \) since \( (\tilde{a}_1^I, \ell'') \vdash S_2'(\ell'') \xrightarrow{\text{SE}} \) cannot hold. Since \( S_2'(\ell'') = S_2(\ell'') \), by similar reasoning, \( \ell'' \in L \land L_2 \).

For the cases of \( \ell'' \) for which \( \tilde{a}_2^I \xrightarrow{\text{SE}} \land \tilde{a}_1^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \), where \( v'' = v \) if \( \kappa(c) \subseteq \ell' \) or \( v = \star \), and \( v'' = d \) otherwise, we can, by the same casing on \( \kappa(c) \) as the one performed in the proof of the (block-i)-case above, establish \( \tilde{a}_2^I = \star \cdot \ell'' \cdot \star \cdot \kappa \tilde{a}_1^I \).

Thus, \( L_1 \land L = L_2 \land L = \ell' \). Since \( \ell_1 \in L_1 \land L, \ell_1 \in \ell' \). Thus, \( \ell_1 \models \text{SME} \Rightarrow s \rightarrow (\tilde{a}_2, \sigma_2, S_2) \) is derivable with the above definition of \( \tilde{a}_2 \) and \( S_2 \) using Figure 3.5b-rule (i). It remains to be shown that \( S_1 \models \ell_1 \Rightarrow S_2 \). For each \( \ell'' \in L', (\tilde{a}_1^I, s_1^I) = (\tilde{a}_2^I, s_2^I) \).

Thus, for each \( \ell'' \in L', S_1(\ell'') = S_2(\ell'') \). Since \( S_1(\ell'') = S_2'(\ell'') \) and \( S_2(\ell'') = S_2'(\ell'') \) for each \( \ell'' \subseteq \ell \) for which \( \ell'' \notin L' \), we get by definition of \( S_1, S_2 \) and of \( =_{\ell_1} \) on objects ranged by \( S, S_1 =_{\ell_1} S_2 \).

(o): Here, \( (\tilde{a}_1^I, \ell_1) \vdash S_1'(\ell_1) \xrightarrow{\text{SE}} (\tilde{a}_1^I, s_1^I), S_1 = S_1'[\ell_1 (\ell_1 \Rightarrow (\tilde{a}_1^I, s_1^I)] \), and either \( \exists \tilde{a}_1^I, v_1^I \cdot \tilde{a}_1^I . c v_1^I = \star \cdot \ell_1 \cdot \star \cdot \kappa \tilde{a}_1^I, \) in which case \( v_1 = v_1^I \), or not, in which case \( v_1 = \ell_1 \), where \( S_1'(\kappa(c)) = (\tilde{a}_1^I, \_), \) and \( a_1 = \_ v_1 \). The only Figure 3.5a-rule which can constitute this derivation is (new-o). By this rule, \( \pi(c) = \ell_1 \), \( s_1^I \xrightarrow{\text{SE}} s_1^I \), and either \( \kappa(c) = \ell_1 \), in which case \( v_{\ell_1} = v_{\ell_1}, \) or not, in which case \( v_{\ell_1} = d \), where \( S_1'(\ell_1) = (_, s_1^I). \)
We show that $(a_2', \ell_1) \models S_2'(s_1) \xrightarrow{\cap \ell \hat{a}} (a_1', s')$. This follows immediately from $S_1'(s_1) = S_2'(s_2)$. Set $S_2 = S_2'(s_1) \mapsto (a_1, s_1)$. 

By definition of $S_1, S_2$ and of $= \ell$ on objects ranged by $S, S_1 = \ell_1, S_2 = \ell_2$. Let $v_2 = v_1 \times e^{-} \if v_2 = a_2', v_2 \times e^{-} \if v_2 = a_2$, and $v_2 = v_1$ otherwise, where $S_2(\kappa(c)) = (a_2, \ldots)$. Set $a_2 = ! \times v_2$. We thus obtain a derivation of the last step in trace $I_2 = \models \sigma(s) \rightarrow (a_2', \sigma, S_2)$ using Figure 3.5a-rule (new-o) and Figure 3.5b-rule (o).

It remains to show $a_1 = \ell a_2$. Case on $\kappa(c)$.

$\kappa(c) \not\subseteq \ell$: Regardless of the value of $\nu_1$ and $\nu_2$, $a_1 = \ell a_2$.

$\kappa(c) \subseteq \ell$: Then since $S_1' = S_2', S'_2(\kappa(c)) = S_2'(\kappa(c))$, and thus $a_1 = a_2$. We thus have $\exists \hat{a}', v_1 \times e^{-} \if v_1 \times e^{-} \if v_1 \times e^{-} \if v_2 \times e^{-} \if v_2$ when defined. Thus $v_1 = v_2$, and thus $a_1 = \ell a_2$.

Thus, $a_1 = \ell a_2$, as desired.

Since $\sigma_1$ and $I_1$ such that $I_1 = \models \sigma(s) \rightarrow (a_1, \sigma, S_1)$ were arbitrary, (3.2) holds for all $a_1$ with $|a_1| = n + 1$ assuming (3.2) holds for all $a_1$ with $|a_1| = n$.

Since (3.2) holds for arbitrary $s, \sigma, \ell, I_1$ and $I_2$ such that $I_1 = \ell I_2$, Lemma 3.6 follows.

**Proof of Theorem 3.5.** Follows from Lemma 3.6 since $a_1 = \ell a_2 \implies a_1 = \ell a_2$.

(3.26) **Lemma** $\forall I, s, \ell, \hat{a} \bullet$

$I \models s \hat{a} \implies \forall \sigma \bullet \exists \hat{a}'$.

$I \models \sigma(s) \rightarrow (\ldots, S) \wedge S(\ell) = (\hat{a}', \ldots) \wedge \hat{a} = \bullet \hat{a}'$.

**Proof.** Follows from the definition of $I \models \sigma$, (new-i), (old-i), (old-o) and (new-o), and from the assumption that $s$ is input-blocking.

(3.27) **Lemma** $\forall I, s, \sigma, S$

$I \models \sigma(s) \rightarrow (\ldots, S) \implies \forall \ell, \hat{a} \bullet$

$S(\ell) = (\hat{a}, \ldots) \implies \exists \hat{a}' \bullet \hat{a} = \bullet \hat{a}'$.

**Proof.** Follows from the definition of $I \models \sigma$, (new-i), (old-i), (old-o) and (new-o), and from the assumption that $s$ is input-blocking.

(3.28) **Lemma** $\forall I, \sigma, s, \hat{a}$

$I \models \sigma(s) \rightarrow (\hat{a}, \ldots, S) \implies \forall \ell, \hat{a} \bullet \hat{a} = \pi_{\ell}(\ell), \bullet \hat{a} \bullet$
Proof. Follows from (i) and (o). □

Proof of Theorem 3.8. We prove a); the proof of b) is similar. Assume \( s \in \text{PSNI} \). Let \( I \) and \( \bar{a} \) be arbitrary such that \( I \models s \overset{\bar{a}}{=} \). Since \( s \in \text{PSNI} \) and \( I \models I \overset{\ell}{=} \) for all \( \ell \), we have for all \( \ell \) that \( I \overset{\ell}{=} s \overset{\bar{a}}{=} \) and \( \bar{a} \overset{\ell}{=} \bar{a}_\ell \) for some \( \bar{a}_\ell \). By Lemma 3.26, for each \( \ell \), we have for any \( \sigma \) some \( S_\ell \) for which \( I \models \text{sme}(\sigma, s) \rightarrow (\ldots, S_\ell) \) that \( \bar{a}_\ell = \bar{a}_\ell' \) for some \( \bar{a}_\ell' \) for which \( S_\ell(\ell) = (\bar{a}_\ell', \ldots) \). Since \((\overset{\ell}{=} \subseteq (\approx)\), \( \bar{a}_\ell \overset{\ell}{=} \bar{a}_\ell' \). By transitivity, \( \bar{a} \overset{\ell}{=} \bar{a}_\ell' \). Since \( \text{sme}(\sigma, s) \) is deterministic, we have for some \( S_\ell' \) that \( \bar{a}_\ell' \leq \bar{a}_\ell'' \) for all \( \ell \), where \( S_\ell'(\ell) = (\bar{a}_\ell'', \ldots) \). Let \( \bar{a}' \) be such that \( I \models \text{sme}(\sigma, s) \rightarrow (\bar{a}_\ell', \ldots, S_\ell') \). By Lemma 3.28, \( \bar{a}_\ell'' = \pi^{-1}(\ell), \bar{a}' \). From \( \bar{a}_\ell' \leq \bar{a}_\ell'' \) we get \( \bar{a}_\ell'' \leq \bar{a}_\ell, \bar{a}_\ell' \). Since \( \bar{a} \overset{\ell}{=} \bar{a}_\ell' \), we get \( \bar{a} = \bar{a}_\ell = \bar{a}_\ell = \bar{a}_\ell' \). Together this yields \( \bar{a} \leq \bar{a}_\ell = \bar{a}_\ell'' \). □

Proof of Theorem 3.10. We prove a); the proof of b) is similar.

When no output is produced on channels \( c \) for which \( \pi(c) \subseteq \kappa(c) \), the result follows from Theorem 3.8 since both \( s \) and \( \text{SME}(\sigma, s) \) are run under the same input stream (and thus will read the same \( n \)th input (for all \( n \)) on all channels, including ones with \( \pi(c) \subseteq \kappa(c) \)). So we need only show that the \( n \)th output (for any \( n \)) on any channel \( c \) with \( \pi(c) \subseteq \kappa(c) \) in \( \bar{a} \) has the same value as the \( n \)th \( c \)’s output in \( \bar{a}' \). This follows from Definition 3.9, (o), and the observation that, in any trace of \( s \), after the first occurrence of an input action on a channel \( c \), the remainder of the trace is silent to all observers at levels \( \ell'' \) which do not satisfy \( \pi(c) \subseteq \ell'' \) (thus, the \( \kappa(c) \)-run cannot be delayed to produce a \( c \) output by reading blanks on channels \( c' \) for which \( \pi(c) \subseteq \pi(c') \subseteq \kappa(c) \)). □

In the following lemma, \( S_1 \overset{\ell}{=} S_2 \) iff \( \forall \ell' \rho \ell' \cdot S_1(\ell') = S_2(\ell') \).

(3.29) LEADMA \( \forall \rho, s, \sigma, \ell, I_1, I_2 \cdot I_1 \overset{\rho}{=} I_2 \implies \)
\( \forall \bar{a}_1, \sigma_1, S_1 \cdot I_1 \models \text{sme}(\rho, \sigma, s) \rightarrow (\bar{a}_1, \sigma_1, S_1) \implies \)
\( \exists \bar{a}_2, \sigma_2, S_2 \cdot I_2 \models \text{sme}(\rho, \sigma, s) \rightarrow (\bar{a}_2, \sigma_2, S_2) \land \)
\( \bar{a}_1 = \ell \bar{a}_2 \land S_1 = \ell S_2 \land \sigma_1 = \sigma_2 \)

Proof of Lemma 3.29. Let \( \rho, s, \sigma, \ell, I_1 \) and \( I_2 \) such that \( I_1 \overset{\rho}{=} I_2 \) be given. We prove that
\( \forall \bar{a}_1, \sigma_1, S_1 \cdot I_1 \models \text{sme}(\rho, \sigma, s) \rightarrow (\bar{a}_1, \sigma_1, S_1) \implies \)
\( \exists \bar{a}_2, \sigma_2, S_2 \cdot I_2 \models \text{sme}(\rho, \sigma, s) \rightarrow (\bar{a}_2, \sigma_2, S_2) \land \)
\( \bar{a}_1 = \ell \bar{a}_2 \land S_1 = \ell S_2 \land \sigma_1 = \sigma_2 \)

by induction in \( n = |\bar{a}_1| \). Let \( \sigma_1 \) and \( S_1 \) be given such that \( I_1 \models \text{sme}(\rho, \sigma, s) \rightarrow (\bar{a}_1, \sigma_1, S_1) \).

Since the semantics of \( \text{sme} \) consists of inference rules added to the semantics of SME, the remainder of this proof is an addition to the proof of Lemma 3.6, with the outermost casing replaced with \( \ell_1 \rho \ell \) contra \( \neg(\ell_1 \rho \ell) \). When \( \ell_1 \rho \ell \) and \( \ell_1 \not\rho \ell \), \( \bar{a}_1 = \ell \bar{a}_2 \) still holds since each rule in the SME semantics which
produces I/O produces I/O on channels with presence level $\ell_1$; therefore, $a_1 = _\ell$ $a_2 = _\rho$.

Add the following cases to the $\ell_1 \rho \ell$ case. In all these cases, $a_1 = $, and thus $a_1 = a'_1$.

($\bullet$): Add the following case to the Figure 3.7a rule casing:

(r-o): Since $S'_1(\ell_1) = S'_2(\ell_1) = (a'_1, s'_1)$ and since (r-o) only conditions on $s'_1$, we get ($\dagger$).

(r-not): Here, $(a_1, \ell_1) \models S'_1(\ell_1) \xrightarrow{\text{not}} (a'_1, s'_1)$ for some $(a'_1, s'_1)$, $\neg \text{releaseok}(r, \ell_1)$, and $S_1 = S'_1[\ell_1 \mapsto (a'_1, s'_1)]$. The only Figure 3.7a rule which can constitute this derivation is (r-i). Since (r-i) only conditions on states, and since $S'_1(\ell_1) = S'_2(\ell_1)$, we get that $(a_1, \ell_1) \models S'_2(\ell_1) \xrightarrow{\text{not}} (a'_1, s'_1)$. By setting $S_2 = S'_2[\ell_1 \mapsto (a'_1, s'_1)]$, we obtain a derivation of the last step in $I_2 \models \text{SME}_R(\rho, \sigma, s) \rightarrow (a_2, \sigma_2, S_2)$ through rules (r-not) and (r-i). Thus, $a_2 = = a_1$, and since $a_1 = _\ell a'_2$, we get $a_1 = _\ell a_2$. Since $S'_1 = _\ell S'_2$, we get by definition of $S_1$ and $S_2$ that $S_1 = _\ell S_2$.

(r-d): Here, $(a_1, \ell_1) \models S'_1(\ell_1) \xrightarrow{\text{d}} (a'_1, s'_1)$ for some $(a'_1, s'_1)$, $\text{releaseok}(r, \ell_1)$, $S_1 = S'_1[\ell_1 \mapsto (a'_1, s'_1)]$ and for $S'_1(\kappa(r)) = (a'_1, s'_1)$ and $S'_1(\ell_1) = (a'_1, s'_1)$, $|a'_1| \|_r \cdot | = |a'_1| \|_r \cdot |$. The only Figure 3.7a rule which can constitute this derivation is (r-i). Since (r-i) only conditions on states, and since $S'_1(\ell_1) = S'_2(\ell_1)$, we get that $(a_1, \ell_1) \models S'_2(\ell_1) \xrightarrow{\text{d}} (a'_1, s'_1)$. Since $\ell_1 \rho \ell$ and $\text{releaseok}(r, \ell_1)$, we get by definition of $\neg \rho$ that $\kappa(r) \rho \ell$. Thus, since $S'_1 = _\ell S_2$, $S'_1(\kappa(r)) = S'_2(\kappa(r))$. By setting $S_2 = S'_2[\ell_1 \mapsto (a'_1, s'_1)]$, we obtain a derivation of the last step in $I_2 \models \text{SME}_R(\rho, \sigma, s) \rightarrow (a_2, \sigma_2, S_2)$ through rules (r-d) and (r-i). Thus, $a_2 = = a_1$, and since $a'_1 = _\ell a'_2$, we get $a_1 = _\ell a_2$. Since $S'_1 = _\ell S'_2$, we get by definition of $S_1$ and $S_2$ that $S_1 = _\ell S_2$.

(r): Here, $(a_1, \ell_1) \models S'_1(\ell_1) \xrightarrow{\text{i} r} (a'_1, s'_1)$ for some $r, v_1, (a'_1, s'_1)$, $\text{releaseok}(r, \ell_1)$, $S_1 = S'_1[\ell_1 \mapsto (a'_1, s'_1)]$ and for $S'_1(\kappa(r)) = (a'_1, s'_1)$ and $S'_1(\ell_1) = (a'_1, s'_1)$, $|a'_1| \|_r v_1, \cdot | = |a'_1| \|_r v_1, \cdot |$. The only Figure 3.7a rule which can constitute this derivation is (r-i). Since (r-i) only conditions on states, and since $S'_1(\ell_1) = S'_2(\ell_1)$, we get that $(a_1, \ell_1) \models S'_2(\ell_1) \xrightarrow{\text{i} r} (a'_1, s'_1)$. Since $\ell_1 \rho \ell$ and $\text{releaseok}(r, \ell_1)$, we get by definition of $\rho$ that $\kappa(r) \rho \ell$. Thus, since $S'_1 = _\ell S_2$, $S'_1(\kappa(r)) = S'_2(\kappa(r))$. Depending on the “if”-statement in (r), then either $v_1 = d$ or $v_1 = v_k$. If the “if”-statement is true for some $\ell_d$ in the derivation with state $S'_1$, the “if”-statement will be true for the same $\ell_d$ in the derivation with state $S'_2$, since $S'_1 = _\ell S'_2$ and since $\text{releaseok}(r, \ell_4)$ implies that $\ell_d \rho \ell$. Thus, regardless of which value $v_1$ has, by setting $v_2 = v_1$, and by setting $S_2 = S'_2[\ell_1 \mapsto (a'_1, s'_1)]$, we obtain a derivation of the last step in $I_2 \models \text{SME}_R(\rho, \sigma, s) \rightarrow (a_2, \sigma_2, S_2)$ through rules (r) and (r-i). Thus, $a_2 = = a_1$, and since $a'_1 = _\ell a'_2$, we get $a_1 = _\ell a_2$. Since $S'_1 = _\ell S'_2$, we get by definition of $S_1$ and $S_2$ that $S_1 = _\ell S_2$. 
Proof of Theorem 3.14. Follows from Lemma 3.29 since $\bar{a}_1 \cong_{\ell} \bar{a}_2 \implies \bar{a}_1 \cong_{\ell} \bar{a}_2$. □

Proof of Theorem 3.15. Regardless of how input is fed on any $r$ in $s$, be it arbitrary, or systematically as in each of the $\ell$-runs in $\text{SME}_R(\rho, \sigma, s)$, since $s \in \text{TSNI}$, interaction on any $r$ in $s$ does not affect interaction on channels $c$ with $\kappa(r) \not\subseteq \pi(c)$. Thus $\text{SME}_R(\rho, \sigma, s) \in \text{TSNI}_\rho$. □

Proof of Corollary 3.16. Follows by comparison of Definitions 3.12 and 3.4 with $\rho = \emptyset$. □

Proof of Corollary 3.17. Since $s \in \text{LTS}_{\rho, \sigma}^C$, derivations of any action in any trace of $\text{SME}_R(\rho, \sigma, s)$ never use rules from Figure 3.7. Thus, $\text{SME}_R(\rho, \sigma, s)$ behaves like $\text{SME}(\sigma, s)$. By Theorem 3.5, $\text{SME}(\sigma, s) \in \text{TSNI}$. Therefore, $\text{SME}_R(\rho, \sigma, s) \in \text{TSNI}$. □

Proof of Theorem 3.19. We prove a); the proof of b) is similar. Assume $s \in (\text{LTS}_{\rho, \sigma}^C \cap \text{TSNI}_{mod} \cap \text{TSNI}_{\pi}(s))$. Let $I$ and $\bar{a}$ be arbitrary such that $I \models F(s) \xrightarrow{\bar{a}}$. Since $s$ is deterministic, any trace emitted by $s$ prefixes a unique (possibly infinite) sequence of actions. Let $I'$ be $I$ with all $r$ inputs, for any $r$, replaced by the $r$ inputs emitted (fed back) in the unique (possibly infinite) sequence of actions of $F(s)$. Then $I' \models s \xrightarrow{\bar{a}}$. Then $I' \models s \xrightarrow{\bar{a}}$; and $\bar{a} \cong_{\ell} \bar{a}_f$ for some $\bar{a}_f$. Then $I' \models B(I' \models s) \xrightarrow{\bar{a}_f}$ and $\bar{a} \cong_{\ell} \bar{a}_f$ for some $\bar{a}_f$. The remainder of this proof is obtained by comparing $B(I' \models s)$ to the $\ell$-run of $\text{SME}_R(\rho, \sigma, s)$ (due to high-lead scheduling, no derivation of any step of $\text{SME}_R(\rho, \sigma, s)$ uses rule (r-d), so the $r$-inputs are the same as in $I' \models s$) and by proceeding as in the proofs of Theorems 3.8 and 3.10, □

Proof of Theorem 3.21. Let $I_1, I_2$ such that $I_1 \cong_L I_2$ be given. Then $I_1 \models s \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $I_2 \models s \xrightarrow{\bar{a}'}$ and $\bar{a} \cong_L \bar{a}'$. Let $\sigma$ and $I$ be arbitrary. We show

1. for each $\bar{a}$ for which $I \models \text{SME}_T(\sigma, s) \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $I \models s \xrightarrow{\bar{a}'}$ and $\bar{a} \cong_L \bar{a}'$, and

2. for each $\bar{a}$ for which $I \models s \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $I \models \text{SME}_T(\sigma, s) \xrightarrow{\bar{a}'}$ and $\bar{a} \cong_L \bar{a}'$.

With the above, this gives, by transitivity of $\cong_L$ that, for each $\bar{a}$ for which $I_1 \models \text{SME}_T(\sigma, s) \xrightarrow{\bar{a}}$, we have a $\bar{a}'$ for which $I_2 \models \text{SME}_T(\sigma, s) \xrightarrow{\bar{a}'}$ and $\bar{a} \cong_L \bar{a}'$. The remainder of this proof establishes 1) and 2).

An L-observable action in an SME$_T$-step can only be derived using (L-a) or (L-timeout). These derivations are only possible when the L-run is in a state where its next action is a L-observable. The L-observable action made by the SME$_T$-step is either the same as the one the L-run performed, or, in the case of
an output on a $H^L$-channel, the outputted value can be replaced. Either way, the
$L$-observable action made by the SME$_T$-step is $\approx_L$-equivalent to the $L$-observable
action made by the $L$-run. When an SME$_T$-step is derived using ($L$-wait), then a $L$-
observable is forthcoming (derived using ($L$-a) or ($L$-timeout)). This follows from
fairness of the scheduler; eventually, the $H$-run gets scheduled often enough to
either reach an $L$-observable (in which case, the next time the scheduler produces
$L$, SME$_T$-step is derivable using ($L$-a)), or timeout (in which case, the next time
the scheduler produces $L$, SME$_T$-step is derivable using ($L$-timeout)).

Thus the sequence of $L$-observables performed by SME$_T(\sigma, s)$ is $\approx_L$-equivalent
to the sequence of $L$-observables performed by the $L$-run in SME$_T(\sigma, s)$. We est-
establish that the sequence of $L$-observables performed by the $L$-run in SME$_T(\sigma, s)$
(which is run under environment $I$) is $\approx_L$-equivalent to the sequence of $L$-observ-
ables performed by $s$ under environment $I |_L$. The two runs match actions (and
thus traverse the same sequence of states) until one performs an input. If one run
reads on a $H^L$-channel, then so can the other, and both runs read $d$. This can be
seen in the definition of $I |_L$ (for the $s$-under-$I |_L$-run), and by rule (old) (for the
$L$-run in SME$_T(\sigma, s)$). If one run reads on a $L$-presence channel $c$, then so can
the other run. Since both runs are input blocking, both runs will read a (possibly
empty) list of blanks. If there are more $c$-inputs forthcoming in $I$, then by defini-
tion of $I |_L$, there will also be more $c$-inputs forthcoming in $I |_L$, and vice versa.
So eventually, either both runs will read blanks infinitely, or both runs will read
the same $c$-input, and enter the same state. (both runs read the same $c$-input in
the case of $\kappa(c) = H$ by (new-i) and definition of $I |_L$ (value becomes $d$)).

**Proof of Theorem 3.22.** The sequence of actions performed by the run of $s$ under
$I$, and the sequence of actions performed by the $H$-run of SME$_T(\sigma, s)$ under $I$,
are $\approx$-equivalent (as long as no attack is discovered). The two runs match actions
(and thus traverse the same sequence of states) until one performs an input. If
one run reads on a channel $c$, then so can the other run. Since both runs are
input blocking, both runs will read a (possibly empty) list of blanks. So, either
both runs will read blanks infinitely, or eventually, both runs will read the same
$c$-input, and enter the same state.

We now show that the sequence of actions performed by the SME$_T(\sigma, s)$-
run and the $H$-run in SME$_T(\sigma, s)$ respectively are $\approx$-equivalent (as long as no
attack is discovered). While no attacks are discovered, ($L$-timeout) is not used in
derivations of steps. In all other rules for deriving $a$-steps where $a \neq \bullet$, the rule
requires that the $H$-run can do $a$ (particularly, in the case of rule ($L$-a), the last else
is never chosen while no attacks are discovered, and the $H$-run, will do $a$ the next
time the scheduler picks $H$). The only difference between the sequence of actions
performed by the SME$_T(\sigma, s)$-run and the $H$-run in SME$_T(\sigma, s)$ respectively is in
$\bullet$ and $\star$-read actions, which are insignificant when comparing for $\approx$-equivalence.

**Proof of Theorem 3.23.** Consider each L-observable $a$ in any trace $\bar{a}$ for which
$I \models SME_T(\sigma, s) \rightarrow (\bar{a}, S, \epsilon, \epsilon)$. The $a$-step was derived using rule ($L$-a), and
the $H$-run in the state before the $a$-step can do an $a_H$-step for some $a_H$ for which $a =_L a_H$. The next time the $H$-run is scheduled, the $H$-run does the $a_H$-step by (old). Since, for $S(L) = (\hat{a}_L, \_)$, $\hat{a} \approx_L \hat{a}_L$, we get by the above that the $H$-run can produce a trace $\hat{a}_H$ for which $\hat{a}_L \approx_L \hat{a}_H$. Now consider $\hat{a}$ and $L$-observable $a$ for which $I \models \text{SME}_T(\sigma, s) \rightarrow (\hat{a}, S, \epsilon, \epsilon) \rightarrow (\hat{a}.a, S', \alpha, \epsilon)$ for some $S$ and $\alpha$. The $a$-step was derived using rule (L-a), and the $H$-run in $S(H)$ can do a $L$-observable $a_H$-step, but cannot do one such that $a =_L a_H$. Since the $H$-run is deterministic, we get that the $H$-run cannot match $\hat{a}_L$ where $S'(L) = (\hat{a}_L, \_)$.

Thus, by definition of $\alpha$, $\alpha$ is an $L$-attack on $s$. $\square$
\[ o.\tilde{a} \uparrow_? = \tilde{a} \uparrow_? \]
\[ i.\tilde{a} \uparrow_? = i. (\tilde{a} \uparrow_?) \]

\[ i.\tilde{a} \uparrow_! = \tilde{a} \uparrow_! \]
\[ o.\tilde{a} \uparrow_! = o. (\tilde{a} \uparrow_!) \]

\[ \bullet.\tilde{a} \uparrow_c = \tilde{a} \uparrow_c \]
\[ \$c'v.\tilde{a} \uparrow_c = \tilde{a} \uparrow_c, \text{if } c' \neq c \]
\[ \$cv.\tilde{a} \uparrow_c = \$cv. (\tilde{a} \uparrow_c) \]
\[ \bullet.\tilde{a} \uparrow_\ell = \bullet. (\tilde{a} \uparrow_\ell) \]
\[ \$cv.\tilde{a} \uparrow_\ell = \bullet. (\tilde{a} \uparrow_\ell), \text{if } \pi(c) \nsubseteq \ell \]
\[ \$c\star.\tilde{a} \uparrow_\ell = \$c\star. (\tilde{a} \uparrow_\ell), \text{if } \pi(c) \subseteq \ell \]
\[ \$cv.\tilde{a} \uparrow_\ell = \$cv. (\tilde{a} \uparrow_\ell), \text{if } \pi(c) \subseteq \ell, v \neq \star \text{ and } \kappa(c) \nsubseteq \ell \]

\[ \tilde{a} \uparrow_\bullet = \epsilon, \text{if } \forall \exists a'.a.a'' \leq a \cdot a \neq \bullet \]
\[ \tilde{a}' . a . a'' \uparrow_\bullet = \tilde{a}' . a . (a'' \uparrow_\bullet), \text{if } a \neq \bullet \]

\[ \bullet.\tilde{a} \uparrow_\bullet = \tilde{a} \uparrow_\bullet \]
\[ \$cv.\tilde{a} \uparrow_\bullet = \$cv. (\tilde{a} \uparrow_\bullet) \]

\[ ?c\star.\tilde{a} \uparrow_\star = \tilde{a} \uparrow_c \]
\[ ?cv.\tilde{a} \uparrow_\star = ?cv. (\tilde{a} \uparrow_\star), \text{if } v \neq \star \]
\[ o.\tilde{a} \uparrow_\star = o. (\tilde{a} \uparrow_\star) \]

\[ ?c\star.\tilde{a} \uparrow_\star = \bullet. (\tilde{a} \uparrow_c) \]
\[ ?cv.\tilde{a} \uparrow_\star = ?cv. (\tilde{a} \uparrow_\star), \text{if } v \neq \star \]
\[ o.\tilde{a} \uparrow_\star = o. (\tilde{a} \uparrow_\star) \]

Figure 3.10: Projections
chapter 4

Limiting Information Leakage in Event-based Communication

ABSTRACT Event-based communication is a major source of power and flexibility for today’s applications. For example, in the context of a web browser, the dynamism of user experience is driven by events: fine-grained interaction of the user with a web application triggers events reactively handled by JavaScript code. This paper explores channels for leaking sensitive information through constructs in a reactive language. We propose a general and realizable security framework for preventing information leaks in a reactive setting with such features as new handler creation and hierarchical event structures. While prior work largely takes an all-or-nothing approach to information flows due to intermediate output, our framework tightly regulates the bandwidth of such flows: at most \( \log(n + 1) \) bits are allowed to be released, where \( n \) is the number of public inputs to the program. We gain flexibility from distinguishing between the security levels of message existence and content. A combination of flow-sensitive analysis and buffering output enables us to enforce security without being overly restrictive.

4.1 Introduction

Event-based communication is a major source of power and flexibility for today’s applications. For example, in the context of a web browser, the dynamism of user experience is driven by events: fine-grained interaction of the user with a web application triggers events reactively handled by JavaScript code. Unfortunately, the power of event-based communication opens up channels for leaking sensitive information. This is a concern where programs operate on data of different levels of sensitivity. For example, a web mashup is a web application that integrates several services into a new combined service. Typically, a web mashup contains JavaScript code from different Internet domains integrated into a single page. It is essential that sensitive information such as user clicks or input form data (say, in an online shopping part of the mashup) is not propagated to a third party (say, an advertisement part of the mashup). At the same time, separation and isolation based on safe language subsets and reference monitoring [35, 12, 16, 29, 28] is
often too restrictive: isolating Google Maps in a mashup from the rest of the web application renders the map-service mashup useless. Hence, a fine-grained approach is desirable, where information flow between inputs and outputs is tracked as it is propagated by program constructs [47, 30]. However, information flow in such a scenario is a delicate problem. In the presence of events, there are channels for leaking information that do not arise in standard programming languages [47]. We illustrate the intricacies with web-based examples, but note that the nature of this problem is general.

**Attacker model** We are interested in securing reactive programs that do not possess any secrets initially. However, a program interact with its environment by input and output events. Input events might carry secret information (e.g., reading the content of a cookie in JavaScript). Programs may generate output events that might carry public information (e.g., loading an image from a third-party server). Assuming the attacker observes (or controls) public input, the attacker’s goal is to deduce information about secret inputs from public outputs. In this model, the only attacker-observable behavior is public output. Internal program behavior such as variable assignment and (non)termination are invisible from outside.

**Tracking information flow** Some events are more secret than others, e.g., user clicks in an online banking application might need to be protected, while clicks in an online shopping application can be released to a statistics service. The challenge is not to release too much: the fact that a user has submitted a credit-card form can be released, but the credit card number must stay secret. We thus distinguish between the security level of event existence and content. In the former example, both are secret, but in the latter the existence is public and content is secret. Our model is similar to security labels for structured datatypes [36, 37].

In a standard reactive language, an event triggers a single handler. In a more general setting, a single event might lead to triggering several handlers in an event hierarchy. Coming back to the web setting, an event hierarchy is induced by the Document Object Model (DOM) [22] tree, a language-independent interface that regulates access to the tree structure of the underlying HTML document. For example, it is possible to set onclick handlers in both the body part and a div part inside the body. In the event of a click inside the div part, both will be triggered and run in sequence.

**Balancing security and permissiveness** Motivated by the scenario of running potentially malicious JavaScript in a browser, we assume the code is in the hands of the attacker. Hence, all possible channels of information leaks by malicious code need to be addressed. The baseline security condition of noninterference [10, 19] prescribes independence of public output from secret input. In a reactive setting, the possibility of observing intermediate outputs needs special attention.
as it allows high-bandwidth leakage of secrets to the attacker. To this end, existing baseline security conditions in a setting with communication primitives offer two choices of treating intermediate output. We use the terminology of progress-(in)sensitivity to highlight the difference. Progress-sensitive noninterference (PSNI) (e.g., [38, 3]) demands that the sequence of outputs produced by programs is fully independent of secrets. This is a strong guarantee, which comes at a price of restrictiveness when enforcing it: Typically, looping on secret data is disallowed [53]. At the other extreme is progress-insensitive noninterference (PINI) (e.g., [2, 3, 7]) that allows programs looping on secret data as long as there are no public side effects in the body loop. However, PINI is vulnerable to brute-force attacks. Consider the source for the following simple web page in Figure 4.1, where function brute is based on an example by Askarov et al. [2].

Assume \( h \), secret and save-clicks are secret and brute-clicks public. This web page lets the user save a secret value in variable \( h \), and then have the program brute-force the value stored on \( h \) by successively guessing from 0 to \( h \). Note that there is no explicit passing of sensitive data to the adversary in the code. Nevertheless, when this script diverges, it has already sent the value from \( h \) to a server-side script \( a.php \) (through GET-attribute \( guess \)) which can then log it for the world to see. This problematic program is deemed secure by PINI and enforcement mechanisms for it [2, 3, 7]). The overrestrictiveness of PSNI and entire-secret leaks of PINI currently leave no choice for anything in-between.

This motivates the need for deeper understanding of security specification and enforcement for reactive languages. While the main long-term motivation for our work is the reactive part of JavaScript in a browser, our results are general and applicable to languages with various flavors of intermediate output. Our results are particularly relevant to languages that feature events, like Erlang, Java, and Smalltalk. Once we gain fundamental understanding of the impact of events, we are in a good position to advance implementation and practical evaluation in a
The paper presents the following contributions to securing information flow in event-based systems:

**Security framework** We introduce a general framework for reasoning about security of reactive programs. A novel contribution is a security framework that addresses the challenge of adequately treating intermediate output. Our security condition occupies the sweet spot between the restrictive PSNI and leaky PINI. It is more restrictive than the latter (preventing brute-force leaks) and more permissive than the former (allowing loops on high data). The condition is a form of noninterference \[10, 19\], that builds insensitivity to computation progress into phases of computation between public inputs. The condition requires that once a public input is consumed, no matter what the secret inputs to the system are, there are only two outcomes until the system is ready to consume another public input: either silent divergence or convergence with the same public output. Thus, a reactive system that diverges while handling an observable phase handles that phase silently. Our approach enables tight control over the bandwidth of allowed leaks by connecting it to the number of processed inputs: we show at most \(\log(n + 1)\) bits are allowed to be released, where \(n\) is the number of public inputs to the program. Thus, by controlling the number of public inputs to be processed, we have full control of the amount of released information. This is a major improvement over PINI, where there is no bound on how much information is leaked when handling a single input. Further, the framework includes the possibility for each channel to distinguish between the security levels of message presence and message content. We then develop a JavaScript-like language with such features as new handler creation and hierarchical event handling, to model and analyze code-in-a-browser in this framework. We model a general notion of a hierarchy that includes such tree-like structures as the DOM tree in browsers.

**Permissive enforcement** We support the language with permissive enforcement based on a novel combination of static analysis and transformation. One source of permissiveness is flow sensitivity. Our static analysis computes a mapping from each sink (output channel) to the set of sources (input channels) from which input can leak on that sink. Another important source of permissiveness is output buffering, realized as a transformation that replaces outputs by appending to a queue and flushes the queue immediately before getting ready to receive new input. This transformation removes information leaks from intermediate observations as in the example in Figure 4.1. Further, we show that all potentially leaking programs that satisfy PINI are repaired by buffering. Buffering output to protect against brute-force attacks is the main thrust of our work, and we expect it have most practical consequences.

The set-inclusion diagram in Figure 4.2 illustrates the relative permissiveness of our enforcement. Bold circles correspond to the sets of programs that satisfy the increasingly liberal security conditions PSNI, IBNI, and PINI (where IBNI is our
4.2 Stream model

Our goal is to secure information flow in systems producing intermediate output. We address this issue in an incarnation of a gradually-maturing stream-based security model for reactive systems [38, 9, 2, 7]. Here, information can only enter and exit our systems through channel-based message passing. Each channel comes with a label expressing the confidentiality level of the information it carries. We then compare each possible input sequence to the resulting output sequence and ensure that confidential information in inputs does not leak into public outputs. An important issue this model deals with is that of feedback loops. Since some inputs can be generated as a function of outputs, it would seem that we have to consider the behavior of the environment when performing information flow security analysis on our systems, like in [38]. However, as proved in [9], for deterministic programs, it is sufficient to consider only input sequences
which are independent of outputs, as quantifying over all these in our security conditions will necessarily include dependent input sequences. This yields compositional results, as we do not have to take into account the behavior of the environment. A sequence of inputs or outputs can then be given as a single stream, i.e., a (possibly infinite) list of messages. We assume that the environment supplies an input and output buffer for the input and output stream, thus making the communication between our reactive system and its environment asynchronous. This greatly simplifies our framework, as a reactive system can be considered as a stream transducer, transforming a given input stream \( I \) into an output stream \( O \), much like a batch-job program transforms an initial memory to a final memory, a scenario thoroughly explored in information-flow security [47]. Still, there is a key difference from batch-job computation: the possibility of producing intermediate outputs. We will return to this difference and show how to secure the information-flow channel (progress) it introduces.

This model appears in its most mature form in [7], and it is this model ours resembles most. Like [7], we treat deterministic reactive systems which operate on streams. However, instead of defining streams and relations on these coinductively, our treatment of streams resembles the one used in Scheme and Haskell, and relations on streams are defined inductively. Furthermore, our security policies are more fine-grained, distinguishing the confidentiality level of message existence and content.

### 4.2.1 Reactive systems

Our computation model is that of reactive systems, in which computation occurs as a reaction to an external event. These events, which could e.g. represent a keystroke, GUI button click, network packet reception, sensor reading, or timer event, are triggered in the system by the environment in which the system runs. This environment could for instance consist of users, hardware, or other systems, such as a web browser, as in our setting where the reactive system is a JavaScript program. Indeed, as exemplified by a web browser running in an environment consisting of a user and other computers on the network, a reactive system can itself be a reactive system running in an even greater environment. While reacting to an event, a reactive system can change its state, as well as trigger zero or more events in its environment. This interaction of a reactive system with its environment is modeled by channel-based message-passing. Each event the system reacts to is associated an input channel, and the environment triggers a given event in the system by sending a message, containing a value, to the system on the associated channel. Likewise, the system triggers events in its environment by sending messages on output channels. Inputs \( i \), outputs \( o \), and messages \( s \) are then given by

\[
i ::= ch(v) \quad o ::= ch(v) \mid \bullet \quad s ::= i \mid o
\]

where \( ch(v) \) (resp. \( ch(v) \)) denotes a message received (resp. sent) on channel \( ch \) carrying value \( v \), and \( \bullet \) denotes that a silent, internal step, or “tick” occurred in
the source of the \( \bullet \) (e.g. an internal channel synchronization, memory assignment, etc.). These channels are the only external interface to the reactive system, and therefore, the only medium by which information can enter and exit our systems.

The behavior of a reactive system can now be given by a labeled transition system with actions ranged by \( i \) and \( o \). That is, a triple

\[(Q, A, \{ \Delta_a \mid a \in A \}),\]

where \( Q \) is a set of states, \( A \) a set of actions, and \( \Delta \subseteq Q \times Q \), for all \( a \in A \).

Intuitively, if \( q \) is a state which can, as its next computation step, input \( i \) and enter state \( q' \), then \( q \overset{a}{\rightarrow} q' \). Likewise, \( q \overset{o}{\rightarrow} q' \) if \( q \) can output \( o \) and enter state \( q' \) as its next step. Practical computation models native to this paradigm include event loops, actors in the Actor Model, and, of interest here, JavaScript programs.

### 4.2.2 Streams

Consider the classic list operators \( \text{cons} \), \( \text{head} \), and \( \text{tail} \), given by

\[
\text{cons}(x, X) = x :: X \quad \text{head}(x :: X) = x \quad \text{tail}(x :: X) = X.
\]

Here, \( :: \) is a data constructor, where \( x :: X \) represents \( X \) with \( x \) prepended (or “cons”ed). For languages with lists, we define the next operator \( \triangleright \) to evaluate a term \( X \) until it reaches the form \( x :: X' \) for some terms \( x \) and \( X' \). Without further evaluating \( x \) and \( X' \), \( \triangleright \) then yields \( x :: X' \). So, \( X \triangleright x :: X' \). For example, given

\[\text{inc}(n) = n :: \text{inc}(n + 1),\]

\( \text{head}(\text{inc}(3)) \) first evaluates to \( \text{head}(3 :: \text{inc}(3 + 1)) \) as \( \text{inc}(3) \triangleright 3 :: \text{inc}(3 + 1) \). At this point, with an appropriate evaluation strategy, \( \text{head}(3 :: \text{inc}(3 + 1)) \) can evaluate directly to \( 3 \), without first evaluating \( \text{inc}(3 + 1) \) to a value. This idea of reducing a term only as needed (i.e. lazily) to yield the head-and-tail of a list exists as streams in Scheme and lists in Haskell, with which you can express finite or infinite lists. Now, \( S \) is a (non-empty) message stream if \( S \triangleright s :: S' \) for some \( s \) and \( S' \). If \( \triangleright \) is not defined on \( S \), then \( S \) is an empty message stream, denoted by the empty list symbol \([\]. \) Input streams \( I \) and output streams \( O \) are defined similarly. Throughout the paper, we frequently denote by \( s :: S' \) any stream \( S \) for which \( S \triangleright s :: S' \).

How do we compare possibly infinite lists? We use the idea that two streams are equivalent if they cannot be distinguished. \( S_1 \) and \( S_2 \) are distinct, written \( S_1 \not\equiv S_2 \) if a component-wise equality check of \( S_1 \) and \( S_2 \) eventually\(^2\) fails. \( S_1 \equiv S_2 \) is defined in Figure 4.3. Throughout the paper, if a rule is labeled with \((*)\) on its right, then we have, for brevity, neglected to write the symmetric counterpart of the \((*)\)-labeled rule into the definition (to obtain the symmetric counterpart of

\(^1\)That is, \( x :: X' \) is the head normal form of \( X \).

\(^2\)After a finite number of equality checks.
the (\(*\))-labeled rule in Figure 4.3, swap \(s_1 \vdash S_1\) and \([\]\). We then define stream equivalence as
\[
S_1 \equiv S_2 \overset{\text{def}}{=} \neg (S_1 \ddoteq S_2).
\]

### 4.2.3 Runs as streams

Viewed externally, a run (trace) of a reactive system state \(q\) on an input stream \(I\), denoted \(q(I)\), is a sequence of messages consisting of the inputs in \(I\) interleaved with the outputs emitted while \(q\) reacts to each input. We interpret a run as a message stream in Figure 4.4 by defining \(\circ\) on runs. While this definition allows runs to be nondeterministic, we assume \(q(I)\), and thus \(q\), to be deterministic, as we are ultimately interested in the reactive part of JavaScript (which is deterministic and single-threaded). However, the step to nondeterminism in our results is small: Nondeterministic choice can be resolved through a labeled reduction. By giving a random choice stream to a run, we effectively “factor out” nondeterminism into streams, as per O’Neil et al. [38].

When we are only interested in the outputs in a stream, we re-interpret the stream, using the definition of \(\circ\) in Figure 4.5. The re-interpretation can be viewed as an operator (\(\circ\)) which “filters” (by need) the inputs from a stream\(^3\), yielding the outputs.

When defining security for reactive systems, we need only consider input streams of finite length. Assume a given infinite input stream causes our system to leak. Then there must be a finite input stream which causes the same leak. Otherwise, the “attack” requires infinite consumption to succeed, in which case the attack never finishes. Similarly, we only need to consider messages which are finitely far

\(^3\)Like “filter” in Python, Erlang and Haskell.
4.3 Security of reactive systems

We now formalize a notion of information security which rejects leaking systems. As mentioned earlier, the observables of a reactive system are its inputs and outputs. Intuitively, if an input is changed in a way that cannot be observed, then there must be no observable difference in the resulting output. This intuition corresponds to the notion of noninterference [10, 19, 9].

Whether a message on a channel is observable or not is indicated by a security label associated with the channel. We assume a lattice of security levels \((L, \subseteq)\) expressing levels of confidentiality. In our examples, we use the lattice given by \(L = \{H, L\}\) and \(\subseteq = \{(H, H), (L, H), (L, L)\}\), with \(H\) for “high” and \(L\) for “low” confidentiality. We let \(\text{lbl}(ch)\), the security label we associate with a channel \(ch\), be a pair of security levels from \(L\). Here, if \(\text{lbl}(ch) = l_c^e\), then \(l_c\) is the confidentiality of values (content) passed on \(ch\), and \(l_e\) the confidentiality of the existence of a message on \(ch\). For instance, a channel carrying secret values but where the presence of messages is public has label \(H_L\). We note that \(l_e \sqsubseteq l_c\), since being capable of observing values on \(ch\) necessarily implies being capable of observing that some message was transmitted on \(ch\). So \(H^H\) is impossible. We abbreviate channel labels \(H^H, H^L\) and \(L^L\) by \(H, M\) and \(L\), respectively. In our examples we denote a channel by its label when its name does not matter. To simplify notation, we let \(\text{lbl}(ch(v)) = \text{lbl}(\hat{ch}(v)) = \text{lbl}(\hat{ch})\). When \(\text{lbl}(\bullet) = l_c^e\), then \(l_c = l_e\), which in our examples equals \(H\). The distinction of existence and content levels is similar to that for general datatypes. For example, Jif [36, 37] allows arrays, where the length of the array is public but the individual elements are secret.

The security labels express who can observe what. An observer is associated a security label \(l\) from \(L\), indicating the observer is capable of observing the value in a message \(s\) with \(l_c \sqsubseteq l\), and the presence of a message with \(l_e \sqsubseteq l\), where
Chapter 4: Limiting Information Leakage in Event-based Communication

\[ \neg \text{obs}_I(s) \quad S_I \triangleright s' :: S' \quad \text{obs}_I(s) \]

(s :: S)_I \triangleright s :: S

Figure 4.6: Next \( l \)-observable in a stream

\[ \frac{-}{\text{fin}([])} \quad \frac{\text{fin}(S)}{\text{fin}(s :: S)} \]

Figure 4.7: Finite stream predicate

\( \text{lbl}(s) = l_c^l \). The \( l \)-observables in \( s \), \( \text{obs}_I(s) \), are thus

\[
\text{obs}_I(\bullet) = \begin{cases} 
\bullet & \text{if } l_c \sqsubseteq l \\
\cdot & \text{otherwise}
\end{cases}
\]

\[
\text{obs}_I(ch(v)) = \begin{cases} 
ch(v) & \text{if } l_c \sqsubseteq l \\
\cdot & \text{otherwise}
\end{cases}
\]

\[
\text{obs}_I(\tilde{ch}(v)) \text{ defined in the same manner as } \text{obs}_I(ch(v))
\]

where \( \cdot \) means nothing is observed\(^4\). We also use \( \text{obs}_I(s) \) as a predicate, where \( \text{obs}_I(s) \) is false when \( \text{obs}_I(s) = \cdot \) and true otherwise. Also, we define \( s_1 =_I s_2 \) iff \( \text{obs}_I(s_1) = \text{obs}_I(s_2) \).

We are most interested in observable messages in message streams in our security definitions. This motivates the stream re-interpretation in Figure 4.6, which can be viewed as an operator \(( \cdot I )\) which “drops” unobservables from a stream, until an observable is found\(^5\). We denote by \( s :: S \) any stream \( S \) for which \( S \triangleright \varnothing :: S' \).

We say \( S \) is \( l \)-silent, written \( \text{sil}_I(S) \), when \( S \) produces no \( l \)-observables, that is, when \( S_I \equiv [] \). The finite stream predicate, \( \text{fin}(S) \), is defined Figure 4.7.

If we were not interested in unobservables at all, we would use the re-interpretation \(( \cdot ! I )\) in Figure 4.8 which filters all unobservables from a stream. However, \(( \cdot ! I )\) hides whether a stream is silent and finite or silent and infinite. It is this subtle detail which is of key importance in the distinction between the security definitions we study in the next section.

4.3.1 Progress and Termination

The choice on how to deal with diverging runs which produce no observables leads to different security conditions. They differ in whether or not they secure

\(^4\)The observer only learns that a message occurred on \( ch \) by observing \( ch(\cdot) \) or \( \tilde{ch}(\cdot) \). The observer knows the specification of the reactive system, so this might enable the observer to infer on the reactive system state.

\(^5\)Like “dropwhile” in Python, Erlang and Haskell.
4.3 Security of reactive systems

progress and termination observations. An observer capable of observing termination will know, by inspecting its observables, whether a program is currently diverging or not. Such an observer would be capable of learning whether \( h \) is "true" or not in the following program.

\[
\text{in } M(h); \text{ while } h \{ \text{skip} \}
\]  

(4.1)

This program inputs a (secret) value on \( M \), binds it to \( h \), and then loops on \( h \). Noninterference definitions which secure such observations are termination-sensitive, and those that do not are termination-insensitive. As we mention in the introduction, the attacker in our reactive setting does not observe divergence because of its internal nature; observers can only reason about the behavior of our systems by observing message transmissions. An observer capable of observing progress, on the other hand, will know how far a program is in its computation. Such an observer would learn the value of \( h \) in the following program.

\[
\text{in } M(h); \\
\text{ let } l := 0; \\
\text{ while } l <= h \{ \\
\quad \text{out } L(l); \\
\quad l := l + 1 \\
\} 
\]  

(4.2)

After assigning an input \( \nu \) on \( M \) to \( h \), this program outputs the sequence of numbers \( 0..\nu \) on \( L \). Observers not capable of observing progress would, upon observing an output sequence \([L(0), L(1), L(2)]\) not know whether \( L(2) \) is the final output of the program (meaning \( h = 2 \)), or whether \( L(3) \) will follow. Thus, the observer would never know the exact value of \( h \). Noninterference definitions which secure these observations are progress-sensitive, and those that do not are progress-insensitive.

Bohannon et al. [7] define several noninterference notions, and note two of them to be of practical interest. As they coincide on finite streams, the interesting bit is how they treat infinite streams, in particular, streams which eventually become silent and infinite. The first definition, ID (indistinguishable) security, is given as

\[
\text{(4.1) \textbf{definition} } q \text{ is ID-secure iff, for all } l, I_1 \text{ and } I_2, \\
I_1 \sim_l I_2 \implies (q(I_1))_o \sim_l (q(I_2))_o, 
\]
where $S_1 \sim_l S_2 \overset{\text{def}}{=} \neg (S_1 \not\sim_l S_2)$.

(\(\oversim:\)) is given in Figure 4.9. Intuitively, for $S_1 \sim_l S_2$ to hold, the $l$-observables of $S_1$ and $S_2$ must be component-wise equal, until either a) both $S_j$ have no more $l$-observables, or b) one $S_j$ is silent and infinite. ID-security rejects programs like (4.2). One might therefore be lead to believe that ID-security is progress-sensitive. However, by exploiting the exception in b), (4.2) can leak all of $h$ when progress is observable, as follows.

\begin{verbatim}
in M(h);
l := 0;
while l <= h {
    out L(l);
    l := l + 1
};
while 1 {skip}
\end{verbatim}

In fact, program (4.3) has the same input-output behaviour as the brute-force attack in Figure 4.1, extracted in program (4.4).

\begin{verbatim}
in M(h);
l := 0;
while 1 {
    out L(l);
    while l = h {skip};
    l := l + 1
}
\end{verbatim}

ID-security is thus both progress- and termination-insensitive. One might disregard this issue, thinking that, while there are leaky PINI-secure programs, PINI-enforcements will surely reject them. However, the way programs like (4.4) exploit the “progress channel” cannot be detected by current PINI-enforcements, since they contain no explicit leaks of $h$ or $L$ effects in a $H$ context (loop/branch).

The other definition noted to be of practical interest in [7] is CP (co-productive) security, defined as follows.

(4.2) \textbf{Definition} \quad q \textit{is CP-secure iff, for all } l, I_1 \text{ and } I_2, 

\[ I_1 \oversim_l I_2 \implies (q(I_1))_o \oversim_l (q(I_2))_o, \]
where $S_1 \approx_l S_2 \overset{\text{def}}{=} \neg (S_1 \dot{=} l S_2)$.

$(\approx_l)$ is given in Figure 4.10. Observe the minute, yet key, difference between $(\approx_l)$ and $(\sim_l)$. Intuitively, for $S_1 \approx_l S_2$ to hold, the observables of $S_i$ must match exactly. Indeed, we have $S_1 \approx_l S_2 \iff S_1^{\uparrow} \equiv S_2^{\uparrow}$. CP-security rejects programs like (4.4), and is thus progress-sensitive. It will, however, accept programs like (4.1), where the only (possibly) observable behavioral difference is whether the program diverges or not. So CP-security is termination-insensitive.

We have now seen most of Figure 4.2. ID-security is PINI, the set of programs proven ID-secure through enforcement make up TPINI, CP-security is PSNI and the set of programs proven CP-secure through enforcement make up TPSNI. While ID-security can leak everything, CP-security leaks nothing in our setting since we do not consider the termination channel exploitable\(^6\). Since leaking arbitrarily on the progress channel is unacceptable in practice, CP-security is a much more reasonable property to aim for. However, CP-security is hard to enforce permissively [53]; typically, looping on high data is disallowed.

### 4.3.2 IB-security

What makes the brute force attack successful is that before the program reaches a point in its control flow where it will diverge, the program has already leaked its secret through intermediate outputs. We devise a new security notion, IB-security (input-bounded), which deals with this problem by requiring that a reactive system that diverges while handling an observable phase handles that phase silently. Phases arise from the idea that an observer might consider it possible for unobservables to appear before any observable he sees in a stream, and after the last observable he sees. If $S$ is silent, all of $S$ is one phase. Otherwise, the first phase of $S$ are all the messages in $S$ up to (and including) the first observable. The next phase is then the first phase in the rest of the stream. Figure 4.11 partitions a stream this way by placing a $\ast$ into the stream between phases. Let $o ::= \ast \mid o$, $s ::= i \mid o$ and $S$ be streams of $s$. We set $\text{lbl}(\ast) = \bot^\bot (= L$ in our examples), so $\text{obs}_l(\ast) = \ast$, for all $l$.

\(^6\)If the termination status of a program is observable, then CP-security will leak at most 1 bit (per execution) [2] in any case.
Chapter 4: Limiting Information Leakage in Event-based Communication

Figure 4.11: Observable phase stream partitioning. $S^p_l \overset{\text{def}}{=} S^p_0$

Figure 4.12: IB-difference of partitioned streams. $\approx_l \overset{\text{def}}{=} \approx_l^0$

(4.3) DEFINITION $q$ is IB-secure iff, for all $l$, $I_1$ and $I_2$,

$$I_1 \approx_l I_2 \implies (q(I_1))^{p^l}_0 \approx_l (q(I_2))^{p^l}_0,$$

where $S^{p^l}_0 \overset{\text{def}}{=} (S^p_0)_o$ and $S_1 \approx_l S_2 \overset{\text{def}}{=} \neg(S_1 \approx_l S_2)$.

( $\approx_l$ ) is given in Figure 4.12. IB-difference behaves like ID-difference (this is ( $\approx_l^0$ )) until an observable message is found in both $S_1$ and $S_2$; then it behaves like CP-difference (this is ( $\approx_l^1$ )). As soon as a * is observed in both $S_1$ and $S_2$, however, IB-difference goes back to behaving like ID-difference. In both these cases, observable messages, and *, have to match.

IB-security rejects Program (4.3). For instance, let $I_1 = [H(1)]$ and $I_2 = [H(2)]$, each an input stream with a single phase. Running (4.3) on these streams yields outputs that are IB-equivalent to the lists $O_1 = [L(1), \uparrow]$ and $O_2 = [L(1), L(2), \uparrow]$, respectively (where $\uparrow$ denotes silent divergence). However, $O_1 \approx_l O_2$ since, after matching $L(1)$, one stream is silent and the other is not.

It should not be too surprising that IB-security resides between ID-security and CP-security. The proofs of these and further formal results are given in the full version of this paper [40].

(4.4) PROPOSITION If $q$ is CP-secure, then $q$ is IB-secure.

(4.5) PROPOSITION If $q$ is IB-secure, then $q$ is ID-secure.
4.3 Security of reactive systems

IB-security does not stop progress leaks entirely. For instance, the “guess” attack in (4.5) is IB-secure while an observer can learn the correctness of his guess by probing the responsiveness of (4.5).

\[
\begin{align*}
in & \ H(h); \\
while \ 1 \{ \\
\ & \ in \ L(l); \\
\ & \ while \ l = h \ {\text{skip}}; \\
\ & \ out \ L(l) \\
\} \\
\end{align*}
\]

(4.5)

A JavaScript modeling of this program is given in Figure 4.13. Here, the guess is fed by the user through (public) clicks on a guess button\(^7\). The key difference between programs (4.3) and (4.5) is that program (4.5) leaks only “a little” as a reaction to each phase, and thus has a lower bandwidth on the progress channel. A crucial question arises: What is the maximum bandwidth of leaks IB-security permits on the progress channel? We answer this question in the following section.

4.3.3 Quantitative guarantee

Our security condition entails a tight quantitative security guarantee. We utilize Smith’s recent model for quantitative security. Smith [51] defines the notion of vulnerability \( V(X) \) as the worst-case probability of guessing the value of secret \( X \) by an adversary in one try. The measure of information quantity is then defined as \( - \log V(X) \), which corresponds to min-entropy. Based on the intuition

\[
\text{information leaked} = \text{initial uncertainty} - \text{remaining uncertainty},
\]

Smith defines information leakage, which for deterministic programs and uniformly distributed secrets amounts to \( \log |S| \), where \( |S| \) is the size of the set of possible public outputs \( S \) given that the public input is fixed. \( |S| \) translates to the

\(^7\)A similar example can be made where the guess comes from the network.
number of indistinguishability classes for the high input, which, in effect, is the number of different possibilities for the phases of an input stream. This is in line with Lowe [27], who measures the number of secret behaviors distinguished by an attacker in a nondeterministic setting.

Smith’s model allows us to obtain a quantitative guarantee without reasoning about probabilities. Indeed, it suffices to give an estimate on the number of possible public observations in order to derive min-entropy. For the quantitative results, we assume input streams are drawn from a finite universe \( U(I_L) \), where \( I_L \) is a (fixed) stream of observables where \( I \equiv I_l \) holds for each \( I \in U(I_L) \). Given that the number of input streams satisfying this criteria is infinite, and that we thereby seemingly lose precision by assuming a finite universe, we note that the result which is based on this assumption holds regardless of how we fix our finite universe.

Let \( E \) be an equivalence relation, \([a]_A^E\) the \( E \)-equivalence class of \( a \) in \( A \), and \( A/E \) the set of \( E \)-equivalence classes in \( A \). Formally,

\[
[a]_A^E \overset{\text{def}}{=} \{ b \in A \mid (a, b) \in E \} \\
A/E \overset{\text{def}}{=} \{ [a]_A^E \mid a \in A \}.
\]

(4.6) **Definition** (\( k \)-bit security) Let \( I_L \) and \( U(I_L) \) be given, \( U(I_L) \) uniformly distributed, and \( q \) be a system taking input from \( U(I_L) \). Then \( q \) is \( k \)-bit secure if \( k \leq \log_2 |S| \), where

\[
q(U(I_L)) \overset{\text{def}}{=} \{ (q(I))_o \mid I \in U(I_L) \} \\
S \overset{\text{def}}{=} q(U(I_L))/\simeq_l
\]

In our setting, a \( k \)-bit secure program leaks at most \( k \) bits. The following program leaks whether the first value received on the \( M \)-channel (if any) is even or odd.

\[
in \ M(h); \quad \text{while } (h \% 2) \{ \text{skip} \}; \quad \text{out } L(0)
\]

For fixed low inputs in an input stream with at least 1 \( M \)-message, the observer sees at most two kinds of outputs: those equivalent to \([\] \) and \([L(0)] \) (by \( \simeq_l \)) respectively. As \( \log_2 |S| = \log_2 2 = 1 \), Program (4.6) is at most 1-bit secure. Program (4.3), on the other hand, eventually outputs the exact value received last on the \( H \)-channel. In this case we have at most \( m \) possible outputs, where \( m \) is the number of integers in \( U(I_L) \). Since \( \log_2 |S| = \log_2 m \), and since all these bits come from a single secret input value, Program (4.3) leaks that whole value. At last, the number of bits leaked by Program (4.5) is a function of the length of the observables \( I_L \). If \( I_L \) has \( n \) messages, then \( I_L \) has \( n + 1 \) phases. Depending on the secret, the program can diverge when handling any of these phases, or in none of them. The last phase must be handled silently. We thus have \( n + 1 \) classes of
4.3 Security of reactive systems

outputs, so Program (4.5) is at most \( \log_2 (n + 1) \)-bit secure.

(4.7) \textbf{Theorem} \quad \text{If } q \text{ is IB-secure, then } q \text{ is at most } \log_2 (n + 1) \text{-bit secure, where } n \text{ is the nr. of observables in } q \text{'s input.}

4.3.4 Buffering improves security

The reader might wonder which reactive systems in general are IB-secure. It turns out that ID-secure systems which buffer outputs between handling of inputs are IB-secure\(^8\). We give a buffered re-interpretation of a stream in Figure 4.14, which buffers outputs between each input. Basically, if \( S \triangleright o :: S' \) for some \( o \) and \( S' \), then \( S_B \triangleright o :: S'' \) for some \( S'' \) only if an input follows \( o \) in \( S \), or \( S \) is a finite number of outputs. We realize this idea with a two-mode re-interpretation: buffer (annotation \( B \)), and flush (annotation \( F \)). \( (S,O)_B \) will, when the next operator is applied on it, queue non-\( \bullet \) outputs from \( S \) in \( O \) using the reverse “cons” constructor\(^9\). This constructor interacts with the next operator as follows.

\[
([], o :: O)_B \triangleright o :: O
\]

\[
\begin{array}{c}
(\bullet :: S,O)_B \triangleright \bullet :: (S,O)_B \\
(i :: S,O)_F \triangleright s :: S'
\end{array}
\]

\[
\begin{array}{c}
(S,o :: O)_F \triangleright o :: (S,O)_F \\
(i :: S,[])_F \triangleright i :: (S,[])_B
\end{array}
\]

Figure 4.14: Buffered stream. \( S_B \overset{\text{def}}{=} (S,[])_B \)

When an input is encountered, \( O \) is flushed. So, \( O \) practically takes over for \( S \) until exhausted.

Let \( q_B \) be like \( q \) in every way, except when run on an input stream \( I \). The resulting stream is then \( (q(I))_B \) instead of \( q(I) \).

(4.8) \textbf{Theorem} \quad \text{If } q \text{ is ID-secure, then } q_B \text{ is IB-secure.}

This theorem is central. It states that we can drastically reduce the leak on the progress channel by running the program in a context which buffers output. In practice, however, having the context do this buffering is not always an option; in JavaScript, for instance, this would require changing the JavaScript interpreter. However, in such a scenario, buffering can be realized through program

---

\(^8\)While it is sufficient to buffer output between handing of observable input phases, doing so is not viable in practise where there might be multiple (unknown) observers at different observation levels (for instance, in a Mashup).

\(^9\)“snoc” in Haskell.
transformation, by “inlining” the buffering into the JavaScript program. Then, provided the JavaScript program can be enforced to be ID-secure, applying the buffering transformation on the program will make it IB-secure. We now give a concrete example of an ID-security enforcement and a buffering program transformation in a JavaScript subset. The language extends the one given in [7], but the enforcement and program transformation are ours.

4.4 Language

We now present a simple core language for reactive imperative systems, given in Figure 4.15. The language is a subset of JavaScript, sharing many of its features and assumptions. In this language, when reacting to an event, a reactive system runs a handler associated with that event, as well as all handlers above it in its hierarchy of event handlers. Each such handler can change the state of the reactive system, and trigger zero or more events in its environment. Abstractly, our systems repeat the following: i) take the next available input, ii) produce zero or more outputs. Inputs are buffered, and then handled in the order they are received in. Our programs are single-threaded in the sense that it does not handle input messages concurrently. Input and output channels are disjoint, so our programs cannot send messages to themselves. This last restriction is not severe; in JavaScript, events generated procedurally are implemented as procedure calls. Besides, we are most interested in how our systems react to their environment.

4.4.1 Syntax

Let programs, handlers, commands and expressions be ranged by \( p, \ ha, \ c, \ e \), respectively, and let the sets \( \mathbb{C}, \ \mathbb{X}, \ \mathbb{V} \) of channels, variables and values respectively be ranged by \( ch, \ x, \ v \). A program \( p \) is a list of handlers. When \( p \) processes an input \( ch(v) \), it looks through its list of handlers for a \( ch \)-handler, \( ch(z)\{c\} \). If none is found, \( ch(v) \) is dropped. If found, \( p \) will execute the body of the handler, \( c \), with \( v \) in place of the formal parameter \( z \). A command is merely a program in a \textbf{while} language, extended with \textbf{output} and handler creation. Beyond memory inspection and modification, branching and looping, \( c \) can output messages, and add/replace a handler to/in \( p \). A memory, ranged by \( \mu \), is a \( \mathbb{X} \rightarrow \mathbb{V} \) mapping which, initially, is 0 for all \( x \). This memory is global, so when \( p \) processes an input, the change in memory can affect how other handlers process future input.

\begin{align*}
p &::= \cdot | ha;p \\
ha &::= ch(z)\{c\} \\
c &::= \text{skip} \\
&| \quad c; c \\
&| \quad x := e \\
&| \quad \text{if } e \{c\} \{c\} \\
&| \quad \text{while } e \{c\} \\
&| \quad \text{out } ch(e) \\
&| \quad \text{new } ha
\end{align*}

Figure 4.15: Syntax

\[^{10} \text{Timeout events are an exception. However, we can model these by considering setTimeout("s", ms) in JavaScript as a request to the browser to send a message on a reserved channel after time ms to the JavaScript, which stands ready with a handler which reacts by running s.}\]
After (if) $c$ terminates, $p$ consults a **hierarchy** of channels $H$, processing $ch(v)$ as if it were an input to the parent of $ch$ (effectively forwarding $ch(v)$ to the parent of $ch$). $H(ch)$ yields the parent channel of $ch$, or $\top$ if $ch$ has no parent. In effect, $H$ is a tree (or a forest) and can be used to model e.g. the DOM tree. Once a message has been forwarded to $\top$, $p$ will enter a state where it is ready to process a new input.

We assume the presence of an expression language, which can be more or less arbitrary, except the relation $\mu \vdash e \Downarrow v$, which under memory $\mu$ reduces $e$ to $v$, must be given. This relation must be side-effect free, deterministic and terminating. $X$ and $V$ must be disjoint, and $0 \in V$ as $0$ is treated as Boolean false in branching and looping instructions. In our examples we have arithmetic and conditional expressions over $X \cup \{z\} \cup V$, with $V = \mathbb{N}$ and operators defined as usual.

### 4.4.2 Semantics

The operational semantics of our language is given as a labeled transition relation on system states, ranged by $q$. There are two kinds of states. Consumer states denote a system ready to process new input, and are given as a memory-program pair. Producer states denote a system currently handling input, producing output as it goes. Such states are given as a 4-tuple consisting of the current memory, program definition, message being handled, and command being executed in response.

$$q ::= (\mu, p) | (\mu, p, i, c)$$

The labeled transition relation on $q$ is defined in terms of the following intermediate judgments.

$(\mu, p, c) \xrightarrow{\mu'} (\mu', p', c')$: A small-step labeled reduction stating that, in memory $\mu$, with program $p$, command $c$ produces $o$ in a single step, modifying $\mu$ and $p$ to $\mu'$ and $p'$ while doing so, and becoming $c'$. This reduction relation is given in Figure 4.16. The only non-standard rules are the out $ch(e)$ rule and the new $ha$ rule. The former emits output, and the latter adds a handler definition as the head of $p$.

$(p, i) \Downarrow c$: A big-step reduction for handler selection stating that, given program $p$ and input $i$, $c$ is the command to be executed in response to $i$. $c$ is the body of the first $ch$-handler in $p$, where $i = ch(v)$\footnote{new $ch(z)\{c\}$ thus effectively replaces the $ch$-handler in $p$.}. In $c$, any occurrence of the formal parameter $z$ has been replaced by $v$ (except those appearing in new $ha$ statements). When there is no handler for $i$ in $p$, command $\text{skip}$ is chosen. This reduction relation is given in Figure 4.17.

The labeled transition rules for system states, $q \xrightarrow{s} q'$, are given in Figure 4.18. The initial state of a reactive system defined by $p$ is the consumer state $(\mu_0, p)$. Here, $\mu_0(x) = 0$, for all $x \in X$. A $q \xrightarrow{s} q'$ transition corresponds to feeding
\[ \frac{\mu, \ldots \vdash e \downarrow v}{(\mu, p, \text{skip}; c) \triangleright (\mu, p, c)} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, c)}{\mu, p, c_1) \triangleright (\mu', p, c'_1)} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, c_1; c_2) \triangleright (\mu', p, c'_1; c_2)}{\mu, \ldots \vdash (\mu, p, c_1; c_2) \triangleright (\mu', p, c'_1; c_2)} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{skip})}{(\mu, p, x := e) \rightarrow (\mu, p, \text{skip})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, c_1)}{\mu, \ldots \vdash (\mu, p, c_1)} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, c_2)}{\mu, \ldots \vdash (\mu, p, c_2)} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{out ch}(e))}{(\mu, p, \text{out ch}(e)) \rightarrow (\mu, p, \text{skip})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{while e} \{c\})}{(\mu, p, \text{while e} \{c\}) \rightarrow (\mu, p, \text{skip})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{while e} \{c\})}{(\mu, p, \text{while e} \{c\}) \rightarrow (\mu, p, \text{while e} \{c\})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

Figure 4.16: Reduction relation for commands

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

\[ \frac{\mu, \ldots \vdash (\mu, p, \text{new ha})}{(\mu, p, \text{new ha}) \rightarrow (\mu, p, \text{new ha})} \]

Figure 4.17: Reduction relation for handler selection

input \( i \) to a system in consumer state \( q \) (which in turn enters producer state \( q' \)). Here, handler selection rules are used to determine which command \( c \) to execute in response to \( i \). A \( q \rightarrow q' \) transition corresponds to receiving output from a system in a producer state \( q \) (which in turn enters state \( q' \)). Output \( o \) is
the result of taking 1 transition in $c$, except when $c = \text{skip}$, in which case the channel hierarchy $H$ is consulted to check whether the last input channel has a parent. If so, the last input is forwarded to the handler for that parent. If not, the system enters a consumer state. In any case, $\bullet$ is emitted.

### 4.4.3 Examples

Program (4.7), upon receiving $\text{ch}_1(v)$, outputs $\text{ch}_0(5)$ when $v = 0$ and $\text{ch}_0(4)$ otherwise.

$$
\text{ch}_1(z)\{\text{if } z \{\text{out } \text{ch}_0(4)\} \{\text{out } \text{ch}_0(5)\}\}
$$

Given $I_1 = [\text{ch}_1(0)]$ and $I_2 = [\text{ch}_1(1)]$, Program (4.7) yields output streams $q_0(I_1) = [\bullet, \text{ch}_0(5), \bullet]$ and $q_0(I_2) = [\bullet, \text{ch}_0(4), \bullet]$. Here, $q_0$ denotes the initial state of the program under consideration. Program (4.8), upon receiving a message on $\text{ch}_1^2$, replace its $\text{ch}_1^1$-handler with a handler that, instead of forwarding $\text{ch}_1^1$ messages to $\text{ch}_0$ untouched, adds 1 to the transmitted value.

$$
\text{ch}_1^1(z)\{\text{out } \text{ch}_0(z)\}
\text{ch}_1^2(z)\{\text{new } \text{ch}_1^1(z)\} \{\text{out } \text{ch}_0(z + 1)\}\}
$$

Given $I_1 = [\text{ch}_1^1(0)]$ and $I_2 = [\text{ch}_1^2(5), \text{ch}_1^1(0)]$, Program (4.8) yields output stream $q_0(I_1) = [\text{ch}_0(0), \bullet]$ and $q_0(I_2) = [\bullet, \bullet, \text{ch}_0(1), \bullet]$. Program (4.9) models Pro-
Figure 4.19: Modeling JavaScript in our Framework

Finally, Figure 4.19 gives an impression of how JavaScript programs can be modeled in our framework. Here, just like in JavaScript where an onclick-event in paragraph 1 causes $X_1$, $Y$ and $Z$ to be executed in response (in that order (in Firefox)), sending $click_p1(v)$ to the corresponding reactive system will cause $X_1$ to be executed, whereafter $v$ gets forwarded to the parent handler of $click_p1$ (namely $click_body$), causing $Y$ to be executed, etc.

4.5 Enforcement

We now develop the static enforcement mechanism for ID-security given in Figures 4.20 and 4.21. Along with a program transformation which turns ID-secure programs into IB-secure programs through dynamic enforcement (given in Section 4.5.1), these two parts form a mechanized approach to rejecting IB-insecure...
programs\textsuperscript{12}. Although the enforcement is phrased as a type system, it is by no means a fundamental choice as there are several viable alternatives such as abstract interpretation \cite{11} for representing the analysis.

Each channel $ch$ has two sub-channels associated with it, one for existence of messages on $ch$, denoted $ch^e$, and one for content of messages on $ch$, denoted $ch^c$. If $lb(ch) = l^e_c$, then we set $lb(ch^e) = l_e$ and $lb(ch^c) = l_c$. The sources (resp. sinks) of our system are the input (resp. output) sub-channels. When analyzing information flow in $p$, we are interested in knowing how $p$ relates sources and sinks. We (over)approximate this relationship with a mapping $\Gamma$. $\Gamma(ch^e)$, resp. $\Gamma(ch^c)$, is the set of sources that an observer capable of observing existence, resp. content, of messages on $ch$ can obtain information from (by observing presence of messages, resp. values passed, on $ch$). $\Gamma(ch^e) \subseteq \Gamma(ch^c)$, for all $ch$, since being capable of observing values on $ch$ necessarily implies being capable of observing that some message was sent on $ch$. The type checker checks whether a $\Gamma$ correctly (over)approximates information flows in $p$, in which case $p$ has type $\Gamma$, written $\vdash p : \Gamma$.

We let $C^e = \{ch^e \mid ch \in C\}$ for any $C \subseteq C$. We define $C^c$ similarly. Then $\Gamma : \text{I} \rightarrow \mathcal{P}(\text{I})$, where $\text{I} = C^e \cup C^c$ is the set of sources and sinks, ranged by $a$. The set of sink types is the powerset of sources. These form a lattice, with $\subseteq$, $\cap$ and $\cup$ defined as $\subseteq$, $\cap$ and $\cup$. In this way, our enforcement mechanism resembles the flow-sensitive security-type system of \cite{24}. There the powerset of information sources is a “universal” flow lattice $\mathcal{L}_U$ which all other flow lattices $\mathcal{L}'$ can be defined in terms of, and that a principal type of $\mathcal{L}'$ can be derived from the principal type of $\mathcal{L}_U$. $\Gamma \subseteq \Gamma'$ if $\Gamma(a) \subseteq \Gamma'(a)$, $\forall a \in \text{I}$, so the $\Gamma$s themselves form a lattice. Any typable $p$ thus has a principal (that is, least) type.

The type system assumes that loops and handlers can run an arbitrary number of times, handlers can be run in any order, and any possible definition of a handler is considered possibly active at any time. Therefore, when a new command is encountered during typing, it is brought to the “top level” (in a sense “flattening” $p$), and typed there in the $pc$ the new command was discovered in. So we are in fact type checking a (slightly) richer syntactic category, $p ::= p \mid pc : ha; p$, of programs where handlers can be paired with the $pc$ they were discovered in. Notice that this (simplified) version of our type system infers nothing. It requires $\Gamma$s of the form $\Gamma : \text{I} \cup \text{X} \rightarrow \mathcal{P}(\text{I})$. We note, though, that a principal $\Gamma : \text{I} \rightarrow \mathcal{P}(\text{I})$ can indeed be inferred from $p$. This inference involves several fixed point computations to make sure the inferred $\Gamma$ is (over)approximative wrt. the above assumptions.

Let $T$ range over $\mathcal{P}(\text{I})$. Here, $\Gamma[x \mapsto T]$ replaces the set of sources $\Gamma$ says can leak into $x$, with $T$. This makes our type system flow-sensitive, taking into account the order of command execution. This is in sharp contrast to flow-insensitive type systems, such as those of \cite{54, 7}, which over-approximate by assigning the same type to $l := h$; out $L(l) : l := 0$ and $l := h$; $l := 0$; out $L(l)$, for instance. However, $\Gamma[\text{ch} \mapsto (T, T')]$ inserts the content of $T \cup T'$ (resp. $T'$)

\textsuperscript{12}You can in fact replace our type system with any sound enforcement of ID-security.
\[
\begin{align*}
\Gamma \vdash e : T \\
\Gamma \vdash e : p c' \quad pc \sqcup pc' \vdash \Gamma p \{c\} p' \quad i = 1, 2
\end{align*}
\]

\[
\begin{align*}
p c \vdash \Gamma p \{\text{skip}\} p \Gamma \\
p c \vdash \Gamma p \{\text{ha}\} p c' ; p \Gamma \\
p c \vdash \Gamma p \{c_1\} p' \Gamma' \\
p c \vdash \Gamma' p' \{c_2\} p'' \Gamma'' \\
\Gamma \vdash e : T \\
\Gamma \vdash e : T \\
\Gamma \vdash e : p c' \quad pc \sqcup pc' \vdash \Gamma p \{c\} p' \Gamma \\
\Gamma \vdash e : p c' \quad pc \sqcup pc' \vdash \Gamma p \{c\} p' \Gamma
\]

Figure 4.20: Command Type Rules

\[
\begin{align*}
p c \sqcup \check{\Gamma} \vdash \Gamma[z \mapsto \check{c}] p \{c\} p' \Gamma[z \mapsto \check{c}] \\
\Gamma \vdash p'
\end{align*}
\]

\[
\begin{align*}
\vdash \bot : \text{ha}; p : \Gamma \\
\vdash \bot : \text{ha}; p : \Gamma
\end{align*}
\]

Figure 4.21: Handler Type Rules

to the set of sources \( \Gamma \) says can leak into \( cb^e \) (resp. \( cb^e \)).

We let \( \check{ch} = \{ch' \in C | \exists n \in N. H^n(ch') = ch\} \), that is, the set of all descendants of \( cb \). \( c \) can be run as a reaction to receiving a message on any \( ch' \in \check{ch} \), so \( c \) runs in the context containing information about the existence of messages on all
channels in $\hat{\mathcal{C}}$. $\mathcal{Z}$ could contain content from any of the $\hat{\mathcal{C}}$ channels. We assume the presence of a typing relation for expressions $\Gamma \vdash e : T$, with the requirement that for each variable $x$ in $e$, $\Gamma(x) \subseteq T$.

The command typing judgment $pc \vdash \Gamma p \{c\} p' \Gamma'$ should be read “under context $pc$, command $c$ takes an initial flow approximation $\Gamma$ and program $p$ to $\Gamma'$ and program $p'$”. Most of the rules are standard, save the first rule in Figure 4.20, referred to as the “weakening rule”. While the other rules permissively approximate information flow in $c$, this rule allows us to conclude that more flows occur in $c$ (yielding a weaker guarantee). This is needed when typing while commands and whole handler bodies, as the typing must approximate an arbitrary number of executions of these. The program typing judgment $\vdash p : \Gamma$ should be read “$\Gamma$ (over)approximates information flows in $p$”. The only interesting rule in Figure 4.21 is the last one, which types a handler together with the $pc$ it was discovered in while traversing $p$. It types the handler body $c$ under context $pc \sqcup \hat{\mathcal{C}}$, where $\hat{\mathcal{C}}$ is the information conveyed by the existence of a message on $\mathcal{C}$, and any of its subchannels.

We close this section with the type soundness theorem. $\Gamma$ is consistent with the channel labeling if $\forall a \in \text{dom}(\Gamma). \forall a' \in \Gamma(a). \text{lbl}(a') \subseteq \text{lbl}(a)$. Also, $p$ is well-typed if $\vdash p : \Gamma$ for some consistent $\Gamma$. At last,

**Theorem** If $p$ is well-typed, then $p$ is ID-secure.

### 4.5.1 Buffering Output

Ideally, a reactive system should stay reactive. Thus, one would usually expect an event handler to always terminate, yielding finite output, and allowing the reactive system to process the next input symbol. For instance, in JavaScript, `timeOut` events are handled with a lower priority than other events to prevent procedurally-created events from starving other events. Also, when a JavaScript program enters an infinite loop, the browser asks the user whether he wants to terminate the reaction prematurely. One could argue that any diverging program is either the product of a programming error or a programmer with malicious intent, making the program diverge in the hope that doing so makes the program pass a static enforcement check and still leak.

We present an encoding of programs which makes a program buffer its output until it is ready to process a new input. Buffering output mitigates the bandwidth of leaks due to intermediate output. One downside is that this encoding will mute handlers that diverge while producing output. However, our justification for considering buffering useful is that programs with diverging handlers do not belong to the paradigm of reactive systems. The program transformation $\text{buff}(p)$, given in Figure 4.22, replaces each output command $\text{out}(v)$ with an “enqueue” command $\text{enq}$, queuing $\text{idx}(\mathcal{C})$ and $v$ in $q$. Here, $\text{idx}$ is a bijection from the $n$ channels occurring in $p$ and $\{1..n\}$. Control for flushing the queue, $c_o$, is then added at the end of each root handler. $\text{deq}$ yields the next element in a queue and
buff(·) is homomorphic for recursively defined objects, and leaves atomic objects unchanged, with these exceptions:

\[
\text{buff}(ch(z)\{c\}) = \begin{cases} 
  ch(z)\{c_i; \text{buff}(c)\; c_o\} & \text{if } H(cb) = \top, \\
  ch(z)\{c_i; \text{buff}(c)\} & \text{otherwise}.
\end{cases}
\]

\[
\text{buff}(\text{out } ch(e)) = q := \text{enq idx}(ch) \text{ enq } e \text{ q}
\]

Here,

\[
c_i = \text{if } q = 0 \{q := \text{emptyq}\} \{\text{skip}\}
\]

\[
c_o = \text{while } q \neq \text{emptyq} \{ \\
  ch_{out} := \text{deq } q; \\
  val := \text{deq drop } q; \\
  q := \text{drop drop } q; \\
  \text{if } ch_{out} = 1 \{\text{out } ch_1(val)\} \{\text{skip}\}; \\
  \vdots \\
  \text{if } ch_{out} = n \{\text{out } ch_n(val)\} \{\text{skip}\}\}
\]

Figure 4.22: Buffering Encoding

drop drops the next element in the queue. \(c_i\) initializes the queue \(q\) to the empty queue if \(q\) has the initial variable value 0.

The effect of this buffering is the same as that of running the original program in a wrapper which buffers outputs, like the one given in Figure 4.14. This leads to the following observation.

**(4.10) THEOREM** If \(p\) is ID-secure, then buff\((p)\) is IB-secure.

We summarize the quantitative implication in this theorem.

**(4.11) THEOREM** If \(\vdash p\), then buff\((p)\) is log\(_2\)(\(n + 1\))-bit secure where \(n\) is the number of observables in the input to buff\((p)\).

So, applying a JavaScript implementation of buff(·) on the script from Figure 4.1 yields an IB-secure program, thus limiting the bandwidth of the progress channel from arbitrary to log\(_2\)(\(n + 1\)).

### 4.6 Related work

Security of event-driven systems has been investigated in the context of process calculi [17, 45, 20, 44, 21, 39, 25] and event-based abstractions [31, 32, 46]. Connections with security models for more concrete programming languages have been made [18, 33]. However, relatively little has been done on exploring the flow of information through language constructs in reactive languages.
4.6 Related work

Sabelfeld and Mantel [46] investigate the impact of different types of channels (secret, encrypted, public) and different types of communication (synchronous and asynchronous) on information-flow security. The encrypted channel is similar to our existential channel, where only the presence (not the content) of messages is visible to attackers. The origins of existence and content levels are in security labels for datatypes. For example, Jif [36, 37] allows arrays, where the length of the array is public but the individual elements are secret.

O’Neil et al. [38] investigate the security of interactive programs. They focus on protecting secret user strategies from leaking to the adversary. Clark and Hunt [9] note that it makes no difference in a deterministic setting whether the input/output is represented by strategies or streams. As discussed in Section 4.1, O’Neil et al. [38], as well as Askarov and Sabelfeld [3], consider termination-sensitive noninterference. The price of termination-sensitivity is restrictiveness: loops with secret guards will likely break security and will hence be rejected by the respective enforcements.

Almeida Matos et al. [34] propose a type system for noninterference and nondisclosure properties. They focus on suspension features and leaks associated with them. Communication is modeled by streams in security formalizations by Askarov et al. [1] for a language with cryptographic primitives and by Askarov and Sabelfeld [3] for a language with dynamic code evaluation and declassification primitives.

Askarov et al. [2] clarify the impact of leaking information via (non)termination of programs in the presence of intermediate output. Restrictions on language constructs that might result in abnormal termination or divergence, originating in classical security analysis [13, 54] and supported in modern information-flow tools Jif [37], FlowCaml [50], and the SPARK Examiner [6, 8], are not strong enough to prevent brute-force attacks as Program 4.4.

As mentioned in Section 4.2, Bohannon et al. propose security definitions for reactive systems that correspond to four indistinguishability relations on streams. They emphasize (progress-sensitive) CP-security and (progress-insensitive) ID-security and choose to focus on the latter. Distinct feature of our approach compared to that of Bohannon et al. is (i) simple framework (finite inductive streams rather than infinite streams and coinductive definitions), (ii) new handler creation, (iii) strong security guarantees (the security definition of Bohannon et al. is similar in spirit to PINI [2] which allows leaking secrets entirely via the intermediate output, whereas we allow only one bit to be leaked at most per consumed public input), (iv) distinguishing the security level of message existence and content, (v) output buffering to guarantee strong security, and (vi) a more permissive flow-sensitive enforcement.

Askarov et al. [2] demonstrate that progress-insensitive noninterference allows leaking secrets in non-polynomial time in the size of the secret. In contrast, our security condition provides a tight quantitative guarantee: the number of leaked bits is bounded by $\log_2(n + 1)$, where $n$ is the number of public inputs. Quantitative information-flow security is a mature area by itself. Smith [51] provides an
excellent summary of the state of the art. We adopt Smith’s min-entropy based definition of quantitative security in our paper. To the best of our knowledge, quantitative security of reactive programs has not been explored previously.

Devriese and Piessens [14] suggest splitting the execution of a program onto threads operating at different security levels. Only the thread at a given level is allowed to consume input from a channel labeled with level. A similar mechanism is in place for output.

Tracking information flow in web applications is becoming increasingly important, e.g., recent highlights are a server-side mechanism by Huang et al. [23] and a client-side mechanism for JavaScript by Vogt et al. [52], although, like a number of related approaches, they do not discuss soundness. Mozilla’s ongoing project FlowSafe [15] aims at extending Firefox with runtime information-flow tracking, where dynamic information-flow monitoring [4, 5] lies at its core. Recently, Magazinius et al. [30] have proposed how to support decentralized policies with possible mutual distrust for dynamically tracking information flow in mashups.

4.7 Conclusion

We have proposed a framework for information-flow security of reactive programs. The framework tightly regulates the bandwidth of leaks due to intermediate output: at most \( \log(n + 1) \) bits are allowed to be released, where \( n \) is the number of public inputs to the program. This provides much-desired middle ground between the Draconian progress-sensitive and the brute-force attackable progress-insensitive security. The framework includes a flexible treatment of channels: it is possible to reveal the existence of messages and at the same time protect their content. We address features of reactive programs that are important in a dynamic environment (such as in a web browser): new handler creation and hierarchical event handling. Although our security requirement is strong, it is realizable: we have presented a combination of flow-sensitive static analysis and output buffering to guarantee security. The model scales up to handle exceptions due to the insensitivity to abnormal termination can be treated in the same way as nontermination. Thus, uncaught exceptions due to, say, partial operators, in high context correspond to looping in high context which is allowed by both our enforcement and security condition.

Future work includes explorations of further features of reactive languages, which will allow us to treat channels as first-class values. Another important direction of current and future work is integration of our approach with the larger research program [3, 41, 43, 30] and experiments with case studies. Of particular focus is supporting policies for intentional information release or declassification [3] (including decentralized policies such as in web mashups [30]), timeout events [41], and interaction with the DOM tree [43]. We are experimenting with an enforcement mechanism for JavaScript that is based on an inlining transformation.
In a malicious-code scenario, it is important to cover all possible channels of leaking information. This paper gives particular attention to the leaks via intermediate output because they can be magnified into brute-force attacks, as illustrated in the example in Section 4.1. Other information channels such as via timing [41] and resource exhaustion [2] are important directions of future work.

We are investigating dynamic enforcement by runtime monitoring along the lines of a recent series of work on dynamic information-flow tracking [26, 49, 48, 4, 3, 5, 42]. Dynamic enforcement provides immediate advantages for handling dynamic language constructs and extending our approach to dynamic channel hierarchies.

We anticipate it is straightforward to generalize our security framework to state-transition systems and parametrize on when buffering is done. We expect a generalization of Theorem 4.11 to guarantee that high-bandwidth leaks via progress of single events are mitigated into low-bandwidth leaks via progress of event chunks. However, as the focus of the present paper is on reactive systems, such a framework is subject to future investigation.

Finally, we are exploring the possibility of giving the programmer control over flushing the output buffer. When several public inputs can be processed until the output buffer is flushed, we have the potential of providing stronger guarantees on the number of leaked bits. The potential of this alternative depends on common usage patterns in existing applications, which we plan to roadmap.

Acknowledgments Thanks are due to Cédric Fournet for the suggestion of buffering output and to Daniel Hedin, Sebastian Hunt and anonymous reviewers for useful discussions. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR.

References


Chapter 4: Limiting Information Leakage in Event-based Communication


Chapter 4: Limiting Information Leakage in Event-based Communication


Appendix

IB-security

First we show, in Figures 4.23 and 4.24, how ID- and CP-difference can be trivially extended to partitioned streams.

The proofs of the following lemmas are trivial.

\( \text{(4.12) LEMMA } S_1^p \sim^p S_2^p \iff S_1 \sim^l S_2. \)

\( \text{(4.13) LEMMA } S_1^p \preceq^p S_2^p \iff S_1 \preceq^p S_2. \)
For instance, to obtain a proof \( S_1 \sim_I S_2 \) from a proof of \( S_1 \simp_I S_2 \), you simply remove each (ID4) node in the derivation tree of \( S_1 \sim_I S_2 \), and “glue” the tree back together, that is, setting the child of the parent of the (ID4) node, to the child of the (ID4) node.

We add labels to the rules defining IB-difference in Figure 4.25.

We are now ready to prove the propositions which position IB-security between ID-security and CP-security.

(4.5) **Proposition**  If \( q \) is IB-secure, then \( q \) is ID-secure.

**Proof.** Assume \( I_1 \siml I_2 \), and thus \( (q(I_1))_0 \simp_I (q(I_2))_0 \). We must show that then either \( I_1 \siml I_2 \) or that \( (q(I_1))_0 \siml (q(I_2))_0 \) when \( I_1 \siml I_2 \). Since \( \siml \) and \( \simp \) coincide on finite streams, we have \( I_1 \siml I_2 \), so we must show that \( (q(I_1))_0 \siml (q(I_2))_0 \). Proving this amounts to proving

\[
S_1^p \siml S_2^p \implies S_1^o \siml S_2^o
\]
\[
\begin{align*}
\frac{s_1 \neq s_2}{s_1 :: S_1 \approx^k l s_2 :: S_2} & \quad \text{(IB1)} \\
\frac{s_1 \neq s_2 \quad S_1 \approx^0 l S_2}{* :: S_1 \approx^k l * :: S_2} & \quad \text{(IB2)} \\
\frac{s_1 = s_2 \quad S_1 \approx^l l S_2}{s_1 :: S_1 \approx^0 l S_2} & \quad \text{(IB3)} \\
\frac{\text{sil}(S_2) \quad \text{fin}(S_2)}{s :: S_1 \approx^0 l S_2} & \quad \text{(IB4)} \\
\frac{\text{sil}(S_2)}{s :: S_1 \approx^l l S_2} & \quad \text{(IB5)}
\end{align*}
\]

Figure 4.25: IB-difference of partitioned streams. \( \approx^l \) def = \( \approx^0 l \)

which you get from proving
\[-(S_{1_0}^p l \approx^0 l S_{2_0}^p l) \implies -(S_{1_0} \approx^l l S_{2_0})\]

which you get from proving
\[S_{1_0}^p l \approx^0 l S_{2_0}^p l \iff S_{1_0} \approx^l l S_{2_0}\]

which you, by Lemma 4.12, get from proving
\[S_{1_0}^p l \approx^0 l S_{2_0}^p l \iff S_{1_0}^p l \approx^l l S_{2_0}^p l .\]

We prove this last implication now. Let \( O_j = q(I_j)^p l \), and assume (A) that \( O_1 \approx^p l O_2 \). We must show that \( O_1 \approx^l l O_2 \). We do this by strong induction in the height \( k \) of the derivation of \( O_1 \approx^p l O_2 \), to prove the stronger property \( O_1 \approx^0 l O_2 \land O_1 \approx^l l O_2 \).

\( k = 1 \): Two cases.

(ID1): Here, \( k = 1 \). By (IB1), \( O_1 \approx^0 l O_2 \) and \( O_1 \approx^l l O_2 \).

(ID3): Here, \( k = 1 \). By (IB4), \( O_1 \approx^0 l O_2 \). By (IB5), \( O_1 \approx^l l O_2 \).

\( k + 1 \), given \( \leq k \): (IH) = “\( O_1' \approx^p l O_2' \) \implies \( O_1' \approx^0 l O_2' \land O_1' \approx^l l O_2' \)”, for all \( O_1, O_2 \) with \( O_1' \approx^p l O_2' \) derivation \( \leq k \)” is our induction hypothesis.
\[ [(s :: S_2) \triangleright s :: S_2] \quad \text{and} \quad [(s :: S_1)S_2 \triangleright s :: (S_1S_2)] \]

Figure 4.26: Concatenation of streams

(ID2): By (IB3), (A), (IH), \( O_1 \approx^0 I_1 \) and \( O_1 \approx^1 I_1 \).

(ID4): Assume w.l.o.g. that \( O_1 \triangleright * :: O'_1 \). Case on \( O_2 \).

\( O_2 \equiv [] \): By (IB4), \( O_1 \approx^0 I_1 \). By (IB5), \( O_1 \approx^1 I_1 \).

\( O_2 \triangleright o :: O'_2 \): By (IB1), \( O_1 \approx^0 I_1 \). By (IB1), \( O_1 \approx^1 I_1 \).

\( O_2 \triangleright * :: O'_2 \): By (IB2), (A), (IH), \( O_1 \approx^0 I_1 \) and \( O_1 \approx^1 I_1 \).

\( \Box \)

(4.4) **PROPOSITION** If \( q \) is CP-secure, then \( q \) is IB-secure.

**Proof.** Strategy is the same as in the above proposition. \( \Box \)

**Quantitative Guarantee**

First we need to establish a lemma stating that when reacting to unobservable messages, no observables are emitted. The proof uses concatenated streams, which we re-interpret as streams in Figure 4.26.

(4.14) **LEMMA** If \( q \) is IB-secure, then for any \( l \), we have for each \( i \) with \( \neg \text{obs}_l(i) \) in any \( I \), if \( q \) finishes handling \( i \) while running on \( I \), then for any \( o \) produced while handling \( i \), \( \neg \text{obs}_l(o) \) holds.

**Proof.** Assume the opposite. You can construct \( I_1 = II' \) and \( I_2 = I[i]I' \) such that \( I_1 \approx^1 I_2 \) but \( q(I_1) \approx^1 q(I_2) \), meaning \( q \) is not IB-secure, a contradiction. \( \Box \)

(4.7) **THEOREM** If \( q \) is IB-secure, then \( q \) is at most \( \log_2(n + 1) \)-bit secure, where \( n \) is the nr. of observables in \( q \)'s input.

**Proof.** From Lemma 4.14, we get that (*) \( q \) never produces observables when handling a message \( i \) in a high part of its input \( I \), whether \( q \) terminates on \( i \) or not (the latter follows from the assumption that \( q \) is IB-secure). We proceed by induction in \( n \), the number of observables in \( I \).

\( n = 0 \): By (*) there is only one equivalence class for outputs for the case where nothing is observed. Thus \( q \) is \( \log_2(1) \)-bit secure, as desired.

\( n + 1 \), given the theorem holds for \( n \): We have for any \( I \) with \( n + 1 \) \( l \)-observables that for some \( I^H \) and \( i^L \), \( I = I'[i^L]I^H \). \( I' \) has \( n \) observables. If \( q \) diverges on \( i^L \), the observer cannot know whether the program really diverged on \( i^L \) or the last high part in \( I' \). Also, if \( q \) terminates on \( i^L \), by (*), it makes...
no difference to the number of equivalence classes whether \(q\) terminates or diverges on \(I^{kl}\). So, there is only 1 more equivalence class, representing that \(q\) finished handling \(i^k\). By induction hypothesis, the greatest number of equivalence classes for \(I'\) is \(n + 1\). This totals to \(n + 2\) equivalence classes. So \(q\) is \(\log_2(n + 2)\)-bit secure for \(I\).

We are done. \(\square\)

**Buffering Improves Security**

Let \(S_1 \sim^p_1 S_2 \overset{\text{def}}{=} \neg(S_1 \sim^p S_2)\). By Lemma 4.12, \(S_1 \sim I S_2 \iff S_1^p \sim^p S_2^p\).

(4.8) **THEOREM** If \(q\) is ID-secure, then \(q_B\) is IB-secure.

**Proof.** Let \(I_1 \sim I_2\). Then \(I_1 \approx I_2\) as \(\sim\) and \(\approx\) coincide on finite streams. Also, \((q(I_1))_o \sim (q(I_2))_o\) by the definition of ID-security. We show that

\[(q(I_1))_o^p \sim^p (q(I_2))_o^p \implies (q(I_1))_o^p \approx (q(I_2))_o^p.\]

Let \(I_j = i^p_{j_1} \cdots i^p_{j_{n+1}}\) and \(I_k^k = I^p_{j_1} \cdots I^p_{j_k}\). \(I_1\) and \(I_2\) must have the same number of observables; otherwise \(I_1 \sim I_2\) cannot hold. Let \(n\) be the number of observables in \(I_1\) and \(I_2\). \(I_1\) and \(I_2\) will therefore both have \(n + 1\) phases. \(I^p_{j_{n+1}}\) both contain only unobservables, while \(I^p_{j_k}\) have an observable as last element, and all other elements unobservable. So, \(I_1^k \sim I_2^k\); otherwise \(I_1 \sim I_2\) cannot hold. So \((q(I_1^k))_o^p \sim^p (q(I_2^k))_o^p\) by the definition of ID-security. If \(q(I_1^k)\) both terminate, then \(q(I_1^k)\) will be finite streams. Then \((q(I_1^k))_o^p \approx (q(I_2^k))_o^p\) since \(\approx\) coincide on finite streams.

If \(q(I_j)\) is diverging, then the divergence occurs in some phase. Assume w.l.g. that \(q(I_1)\) diverges (and if \(q(I_2)\) also diverges, that \(q(I_1)\) diverges after consuming at most as many inputs as \(q(I_2)\)). Let \(k\) be smallest such that \(q(I_1^k)\) diverges. By a corresponding Lemma 4.14 for ID-security (which proof is near-identical), \(q(I_1^k)\) outputs no observables when handling the unobservable inputs in \(I^p_{j_k}\), provided \(q(I_1^k)\) terminates while doing so. Eventually, \(q(I_1^k)\) diverges while handling some \(i\) where \(I_1^p_{j_k} = I[i]I'\). Regardless of whether \(i\) is observable or not, the outputs emitted while \(q(I_1^k)\) reacts to \(i\) are buffered, so \(q(I_1^k)\) remains silent for the rest of \(I_1^p_{j_k}\). Since \(q(I_1^k)\) reacted silently to \(I\) as well, all of \(I^p_{j_k}\) is reacted to silently. This rules out all rules for distinguishing \((q(I_1))_o^p\) by \(\approx\). So \((q(I_1))_o^p \approx (q(I_2))_o^p.\) \(\square\)

**Type Soundness**

Let \(a\) range over \(\Pi = \Pi^e \cup \Pi^c\), \(\operatorname{lbl}_e(ch) = \operatorname{lbl}(ch^e), \operatorname{lbl}_c(ch) = \operatorname{lbl}(ch^c)\) and \(\operatorname{lbl}_\Gamma(a) = \bigsqcup_{a' \in \Gamma(a)} \operatorname{lbl}(a')\).
(4.15) **Definition**  \( \text{lbl} \) is consistent with \( \Gamma \), written \( \text{ct}(\Gamma) \), iff \( \text{lbl}_\Gamma(a) \sqsubseteq \text{lbl}(a), \forall a \).

(4.16) **Definition**  \( p \) is well-typed, written \( \vdash p \), iff \( \Gamma \vdash p \) and \( \text{ct}(\Gamma) \), for some \( \Gamma \).

(4.17) **Theorem**  If \( \vdash p \), then \( p \) is ID-secure.

Theorem 4.9 is the type soundness theorem we wish to prove. The proof relies on the following auxiliary definitions.

(4.18) **Definition**  \( \text{lbl} \) is consistent with the typing of \( c \) under \( \Gamma \) and \( pc \), written \( \text{ct}(c, \Gamma, pc) \), iff, \( pc \vdash \Gamma \cdot \{c\} \ p \ \Gamma \) (for some \( p \) and \( \text{ct}(\Gamma) \)).

(4.19) **Definition**  \( \mu_1 \) and \( \mu_2 \) \( l \)-agree on \( x \) under \( \Gamma \), written \( \mu_1 =^\Gamma_{l,x} \mu_2 \), iff, \( \text{lbl}_\Gamma(x) \sqsubseteq l \implies \mu_1(x) = \mu_2(x) \).

(4.20) **Definition**  \( \mu_1 \) and \( \mu_2 \) are \( l \)-equivalent under \( \Gamma \), written \( \mu_1 =^\Gamma_{l} \mu_2 \), iff, \( \mu_1 =^\Gamma_{l,x} \mu_2 \), \forall x.

(4.21) **Definition**  \( c \) is secure under \( \Gamma \) and \( pc \), written \( \text{sc}(c, \Gamma, pc) \), iff, for all \( l \),

i) If \( l_{pc} \sqsubseteq l \), then for all \( \mu_1, \mu_2 \), if \( \mu_1 =^\Gamma_{l} \mu_2 \), then

a) \( (\mu_1, c) \sim_l (\mu_2, c) \) and  
b) if \( (\mu_1, c) \downarrow \mu' \) and \( (\mu_2, c) \downarrow \mu'_2 \) then \( \mu'_1 =^\Gamma_{l} \mu'_2 \).

ii) If \( l_{pc} \not\sqsubseteq l \), then for all \( \mu \),

a) \( \text{sil}_l((\mu, c)) \) and  
b) \( (\mu, c) \downarrow \mu' \), then \( \mu =^\Gamma_{l} \mu' \).

where \( pc \vdash \Gamma \cdot \{c\} \ p \ \Gamma' \) (for some \( \Delta \)) and \( l_{pc} = \text{lbl}_\Gamma(pc) \).

Here, \( (\mu, c) \) is the run of \( c \) in \( \mu \) (defined in the obvious way). We write \( (\mu, c) \downarrow \mu' \) when \( (\mu, c) \xrightarrow{\ast} (\mu', \text{skip}) \). \( \mu \) does not represent the full system state; for that we would also need a \( p \) that gets updated when \( \text{new} \ldots \) statements in \( c \) are executed. However, Definition 4.21 only concerns outputs emitted and changes on \( \mu \) during a \( (\mu, c) \) run, that is, a single handler execution. So we omit \( p \) in these runs.

For concatenated streams, the following useful lemma holds.

(4.22) **Lemma**  \( S_1 \sim_l S_2 \land S_1' \sim_l S_2' \implies S_1 S_1' \sim_l S_2 S_2' \).

We are ready to prove the key lemma used in the proof of Theorem 4.9.

(4.23) **Lemma**  For all \( c, \Gamma, pc \), \( \text{ct}(c, \Gamma, pc) \implies \text{sc}(c, \Gamma, pc) \).

**Proof.** Assume \( \text{ct}(c, \Gamma, pc) \). Let

\[
\begin{align*}
pc & \vdash \Gamma \cdot \{c\} \ p \ \Gamma \\
\Gamma & \vdash e : T \\
l_{pc} & = \text{lbl}_\Gamma(pc)
\end{align*}
\]

We prove, by induction in \( c \), that \( \text{sc}(c, \Gamma, pc) \) must then hold.
Chapter 4 : Limiting Information Leakage in Event-based Communication

Base cases

c \equiv \text{skip} : \text{sil}_i((\mu_i, c)), thus (\mu_1, c) \sim_I (\mu_2, c), and \text{sil}_i((\mu, c)), for all \ l. Also, 
\mu' = \mu and \mu'_i = \mu_i. So sc(c, \Gamma, pc) is a tautology.

c \equiv x := e: \text{sil}_i((\mu_i, c)), thus (\mu_1, c) \sim_I (\mu_2, c), and \text{sil}_i((\mu, c)), for all \ l. Also, 
\mu' = \mu[x \mapsto v] and \mu'_i = \mu'_i[x \mapsto v_i], where \mu \vdash e \Downarrow v and \mu_i \vdash e \Downarrow v_i.

There are two cases to consider.

l_{bl_\Gamma}(x) \not\subseteq l: \text{Then } \mu' =_l \mu \text{ and } \mu'_i =_l \mu_i. By transitivity, \mu'_1 =_l \mu'_2. So
sc(c, \Gamma, pc) holds in this case.

l_{bl_\Gamma}(x) \subseteq l: \text{Then } l_{bl_\Gamma}(T \cup pc) \subseteq l. Thus l_{bl_\Gamma}(pc) \subseteq l \text{ and } l_{pc} \subseteq l \text{ (so }
\mu =_l \mu' \text{ is not required). Also, } l_{bl_\Gamma}(T) \subseteq l \text{ and therefore } l_{bl_\Gamma}(x') \subseteq l, \forall x' \in T. \text{ Thus } v_1 = v_2, \text{ and therefore, } \mu'_1 =_l \mu'_2. So sc(c, \Gamma, pc) holds in this case.

c \equiv \text{out } cb(e): \text{Then } \mu' =_l \mu \text{ and } \mu'_i =_l \mu_i. By transitivity, \mu'_1 =_l \mu'_2. There are three cases to consider.

l_{bl_e}(cb) \not\subseteq l: \text{Then } \text{sil}_i((\mu_i, c)), thus (\mu_1, c) \sim_l (\mu_2, c), and \text{sil}_i((\mu, c)).

So sc(c, \Gamma, pc) holds in this case regardless of whether \ l_{pc} \subseteq l \ or \ not.

To prove the other two cases we need to show that \ l_{bl_\Gamma}(pc) \subseteq l_{bl_e}(cb). We have \ pc \subseteq \Gamma(cb^a) \ by \ the \ type \ rule \ for \ output \ and \ the \ weakening \ rule. \ Now \ assume \ l_{bl_\Gamma}(pc) \not\subseteq l_{bl_e}(cb). \ Then, \ for \ some \ a, a \in \Gamma(cb^a) \ and \ l_{bl_e}(cb). \ But \ this \ contradicts \ our \ ct(c, \Gamma, pc) \ assumption. \ Likewise \ we \ have \ pc \cup T \subseteq \Gamma(cb^a) \ and, \ by \ the \ same \ argument, \ l_{bl_\Gamma}(pc \cup T) \subseteq l_{bl_e}(cb).

l_{bl_e}(cb) \subseteq l, l_{bl_e}(cb) \not\subseteq l: \text{Then } l_{bl_\Gamma}(pc) \subseteq l \text{ by transitivity of } \subseteq. \ Recall \ that \ l_{pc} = l_{bl_\Gamma}(pc). \ Since \ l_{pc} \subseteq l, (\mu_1, c) \sim_l (\mu_2, c) \ must \ hold \ for \ sc(c, \Gamma, pc) \ to \ hold. \ Indeed, \ we \ have \ (\mu_1, c) \triangleright (o_2, (\mu_i, \text{skip}))) \ and \ obs_i(o_1) = obs_i(o_2) = cb(c). \ So \ (\mu_1, c) \sim_l (\mu_2, c), \ and \ therefore \ sc(c, \Gamma, pc) \ holds \ in \ this \ case.

l_{bl_e}(cb) \subseteq l: \text{Again, } l_{pc} \subseteq l. \ Also, \ l_{bl_\Gamma}(T \cup pc) \subseteq l \text{ by similar argument, and thus } l_{bl_\Gamma}(T) \subseteq l. \ Then \ l_{bl_\Gamma}(x') \subseteq l, \forall x' \in T. \ Thus \ v_1 = v_2, \ where \ mu_i \vdash e \Downarrow v_i, \ so \ o_1 = o_2, \ where \ (\mu_i, c) \triangleright (o_2, (\mu_i, \text{skip}))). Thus \ (\mu_1, c) \sim_l (\mu_2, c), \ and \ therefore, \ sc(c, \Gamma, pc) \ holds \ in \ this \ case.

c \equiv \text{new } ha: \text{As \ here \ only \ care \ about \ output \ emitions \ and } \mu \text{ \ updates, \ the \ proof \ for \ this \ becomes \ the \ same \ as \ that \ of the } c \equiv \text{skip} \ case.

Inductive step \ \text{Induction hypothesis (IH): “for any } c_j \text{ structurally smaller than } c, \text{ then for any } \Gamma_j \text{ and } pc_j, \text{ ct}(c_j, \Gamma_j, pc_j) \implies \text{sc}(c_j, \Gamma_j, pc_j).” \ Let \n
l_{pc_j} = l_{bl_\Gamma}(pc_j) \quad (\mu_j, c_j) \Downarrow \mu'_j \quad (\mu_i, c_j) \Downarrow \mu'_i,j.
where \( c_j \) is structurally smaller than \( c \). Then, for instance, if \( \text{ct}(c_j, \Gamma_j, p_{c_j}) \), then \( \mu_1 = l_j \mu_2 \) \( \implies \mu_1' = l_j \mu_2' \) by IH since \( \text{sc}(c_j, \Gamma_j, p_{c_j}) \) holds.

\[ c \overset{\text{def}}{=} \text{if } e \{ c_1 \} \{ c_2 \} : \text{Let } \Gamma_j = \Gamma \text{ and } p_{c_j} = p_c \cup T. \text{ Then, by the } \text{ct}(c, \Gamma, p_c) \text{ assumption, } \text{ct}(\Gamma), \text{ and thus } \text{ct}(\Gamma_j). \text{ Since } c \text{ is well-typed, } p_{c_j} \vdash \Gamma_j \cdot \{ c_j \} p_j \Gamma_j \} \text{ (for some } p_j). \text{ So } \text{ct}(c_j, \Gamma_j, p_{c_j}) \text{, and thus } \text{sc}(c_j, \Gamma_j, p_{c_j}) \text{ by IH. Here, } p = p_1p_2. \text{ It remains to be shown that } \text{sc}(c, \Gamma, p_c). \text{ Now, } (\mu_i, c) \text{ either take }

i) \text{ different branches, or }

ii) \text{ the same branch,}

in } c. \text{ There are two cases to consider.}

\[ \text{lbl}_\Gamma(T) \not\subseteq l: \text{ Then } l_{pc_j} \not\subseteq l. \text{ Either } i) \text{ or } ii). \text{ We consider each case in turn.}

i): \text{ Assume wlg. that } (\mu_j, c) \text{ takes branch } j, \text{ executing } c_j. \text{ Then } \mu_j = l_j \mu_j. \text{ For all } x, \text{ if } x \text{ is assigned to in } (\mu_j, c), \text{ then } \text{lbl}_\Gamma(x) \not\subseteq l \text{ since } T \subseteq \Gamma_j(x). \text{ Then } T \subseteq \Gamma_j(x) \text{ since } \Gamma_j = \Gamma. \text{ Thus } \text{lbl}_\Gamma(x) \not\subseteq l \text{ and therefore } \mu_j = l_x \mu_j. \text{ By transitivity, } \mu_1 = l_j \mu_2'. \text{ Likewise, by transitivity of } \mu_1 = l_j \mu_2', \text{ here, } \mu_1 = l_j \mu_2'. \text{ So } \text{sc}(c, \Gamma, p_c).

\[ \text{lbl}_\Gamma(T) \subseteq l: \text{ Then } l_{pc_j} \subseteq l. \text{ Thus } \mu_1(e) = \mu_2(e).

i): \text{ Impossible.}

ii): \text{ Same argument as in case } \text{lbl}_\Gamma(T) \not\subseteq l.

\text{Since } \text{sc}(c_j, \Gamma_j, p_{c_j}) \text{, } p_{c_j} = p_c \cup T, \text{ and since } (\mu_1, c) \text{ and } (\mu_2, c) \text{ take the same branch, } (\mu_1, c) \sim_l (\mu_2, c). \text{ So } \text{sc}(c, \Gamma, p_c).

\[ c \overset{\text{def}}{=} c_1; c_2: \text{ Let } \Gamma_j = \Gamma \text{ and } p_{c_j} = p_c. \text{ Like in the } \text{if } \ldots \text{ case, since } \text{ct}(c, \Gamma, p_c), \text{ } \text{ct}(c_j, \Gamma_j, p_{c_j}), \text{ and thus } \text{sc}(c_j, \Gamma_j, p_{c_j}) \text{ by IH. It remains to be shown that } \text{sc}(c, \Gamma, p_c). \text{ Since } \text{sc}(c_j, \Gamma_j, p_{c_j}), \text{ by setting } \mu_1 = l_j \mu_2 = l_j \mu_2' \text{ we have } \mu_1 = l_j \mu_2 \implies \mu_1' = l_j \mu_2', \text{ if } (\mu_1, c_1) \parallel l \mu_1 \text{ and } (\mu_2, c_2) \parallel l \mu_2', \text{ as then } (\mu_1, c_1; c_2) \parallel l \mu_2'. \text{ We get } (\mu_1, c) \sim_l (\mu_2, c) \text{ by } \text{sc}(c_j, \Gamma_j, p_{c_j}), \text{ the fact that all silent streams are equivalent under } \sim_l, \text{ and by Lemma 4.22.}

\[ c \overset{\text{def}}{=} \text{while } e \{ c_1 \}: \text{ Let } \Gamma_1 = \Gamma \text{ and } p_{c_1} = p_c \cup T. \text{ Like in the } \text{if } \ldots \text{ case, since } \text{ct}(c, \Gamma, p_c), \text{ct}(c_j, \Gamma_j, p_{c_j}), \text{ and thus } \text{sc}(c_j, \Gamma_j, p_{c_j}) \text{ by IH. It remains to be shown that } \text{sc}(c, \Gamma, p_c). \text{ There are two cases to consider.}

\[ \text{lbl}_\Gamma(T) \not\subseteq l: \text{ Then } l_{pc_1} \not\subseteq l. \text{ Since } \text{sc}(c_1, \Gamma_1, p_{c_1}), \text{ and thus } \text{silt}((\mu_i, c_1)), \text{ for any } \mu_i. \text{ Likewise, by transitivity of } \mu_1 = l_j \mu_2, \text{ then } \mu_1' = l_j \mu_2'.

\[ \text{lbl}_\Gamma(T) \subseteq l: \text{ Then } l_{pc_1} \subseteq l. \text{ Since } \text{sc}(c_1, \Gamma_1, p_{c_1}), \text{ and thus } \text{silt}((\mu_i, c_1)), \text{ for any } \mu_i. \text{ Likewise, by transitivity of } \mu_1 = l_j \mu_2, \text{ then } \mu_1' = l_j \mu_2'.

Let $\mu_i^n$ be $\mu_i$ after $n$ iterations of $\Gamma_1$. So $\mu_i^0 = \mu_i$ and $\mu_i^{n+1} = (\mu_i^n, c_1) \downarrow \mu_i^{n+1}$. Since $\text{sc}(c_j, \Gamma_j, pc_j)$, if $\mu_1 = \Gamma_{l \downarrow} \mu_2$, then $\mu_1^n, c \sim_l \mu_2^n, c$ and $\mu_1^{n+1} = \Gamma_{l \downarrow} \mu_2^{n+1}$, for all $n$, up to $k$ (possibly non-existing), where one of two things happens:

- $\mu_i^k(e) = 0$; $\mu_i^k$ defined: Then $\mu_i' = \mu_i^k, \mu_i'^{n+1} = \Gamma_{l \downarrow} \mu_i^n, c \sim_l (\mu_i, c)$, and $(\mu_i, c) = (\mu_i^0, c_1) \cdots (\mu_i^{k-1}, c_1)$.

- $\mu_i^k$ undefined: Then $\mu_i^{k-1}(c_1)$ diverged (so no constraints are placed on memories for establishing $\text{sc}(c, \Gamma, pc)$). Regardless, $(\mu_i^1, c_1) \sim_l (\mu_i^{k-1}, c_1)$. Now, Lemma 4.22 can be used to prove that if $S \sim_l S'$ and $S' \sim_l \infty$. From this it follows that $(\mu_i, c) \sim_l (\mu_2, c)$, where $(\mu_i, c) = (\mu_i^0, c_1) \cdots (\mu_i^{k-1}, c_1)$ and $(\mu_2, c) = (\mu_2^0, c_1) \cdots (\mu_2^{k-1}, c_1)$.

Thus $\text{sc}(c, \Gamma, pc)$.

\[\square\]

Obtain a list

\[\text{pc}_1', \text{ch}_1, c_1); \cdots; (\text{pc}_n', \text{ch}_n, c_n); \cdots \text{def} \hat{\Delta}\]

when typing $p$ by adding a side-effect $\hat{\Delta} := (\text{pc}, \text{ch}, c)\hat{\Delta}$ as a premise in the third rule in Figure 4.21. Let $\text{ha}_i = (z)\{c_i\}$. Each $\text{ha}_i$ is then a possibly active handler, and $pc_i$ is the context in which $\text{ha}_i$ was activated.

Let $\Gamma_i = \Gamma[z \mapsto \hat{\mathcal{C}}]$ and $pc_i = pc_i' \cup \hat{\mathcal{E}}$.

(4.24) **Lemma** If $\vdash p$, then $\text{ct}(c_i, \Gamma_i, pc_i), \forall i$.

**Proof.** Since $\vdash p$, by typing, $pc_i \vdash \Gamma_i \{c_i\} p_i \Gamma_i$ (for some $p_i$), $\forall i$. Also, $\text{ct}(\Gamma) \implies \text{ct}(\Gamma_i)$ since $\Gamma_i = \Gamma[z \mapsto \hat{\mathcal{C}}]$ and $z \not\in \Gamma$.

(4.25) **Lemma** If $\text{sc}(c_i, \Gamma_i, pc_i), \forall i$, then $p$ is ID-secure.

**Proof.** Let $I_1$, $I_2$ and $l$ s.t. $I_1 \sim_l I_2$ be given. Let $O_1 = (q_0(I_1))_0$ and $O_2 = (q_0(I_2))_0$. We show that $O_1 \sim_l O_2$. Let $\mu_j$ denote the current memory of $O_j$ (initially $\mu_0$). So initially $\mu_1 = \Gamma_{l \downarrow} \mu_2$. As $O_j$ processes unobservables, $O_j$ remains silent by Definition 4.21 (ii). Also, all $\mu_j$ are $l$-equivalent under $\Gamma$. If either $O_j$ diverges when handling an unobservable, we are done. Otherwise $O_j$ both eventually start processing observables $i_j$. $\text{obs}(i_1) = \text{obs}(i_2) = \text{ch}(\omega)$ for some $\text{ch}$ and $\omega \in \mathcal{V} \cup (\cdot)$, since $I_1 \sim_l I_2$. At that time, $\mu_1 = \Gamma_{l \downarrow} \mu_2$. So by Definition 4.21 (i), $O_1 \sim_l O_2$ while handling $i_1$ and $i_2$, and if $O_1$ and $O_2$ finish doing so, the resulting $\mu_1, \mu_2$ satisfy $\mu_1 = \Gamma_{l \downarrow} \mu_2$, and this argument repeats for the rest of the input streams, until one diverges, or they are both exhausted. Thus $O_1 \sim_l O_2$.

\[\square\]

Theorem 4.9 now follows from Lemmas 4.23, 4.24 and 4.25.
(4.10) **Theorem**  If $p$ is ID-secure, then $\text{buff}(p)$ is IB-secure.

**Proof.** Let $I_1 \sim_l I_2$. Then $q_0(I_1) \sim_l q_0(I_2)$ by assumption. Let $I_j = I_{j_1}^p \cdots I_{j_{n+1}}^p$ and $I_j^k = I_{j_1}^p \cdots I_{j_k}^p$. Then $q_0(I_1^k) \sim_l q_0(I_2^k)$, $\forall 1 \leq k \leq n + 1$ (else we get a contradiction since $I_{1_m} \sim_l I_{1_m}$, for all $m$). In particular, if $C_0$ terminates on $I_1^k$ and $I_2^k$, then $C_0(I_1^k)$ and $C_0(I_2^k)$ will match observables exactly. Thus $C_0(I_1) \approx_l C_0(I_1)$. Since $\text{buff}(p)$ has the same input-output behavior in this case, $\text{buff}(p)$ is IB-secure for those inputs.

Nonterminating reactions are yet to be considered. Since $p$ is ID-secure, by a corresponding Lemma 4.14 for ID-security (which proof is near-identical), we have that $p$ never produces observables when handling a message $i$ in a high part of the input stream, when $p$ terminates on $i$. Same holds for $\text{buff}(p)$. If $p$ diverges on some $i$, then $\text{buff}(p)$ outputs nothing while diverging on $i$. Thus, if $\text{buff}(p)$ diverges on a part, high or low, no outputs emerge. We are done. □
Securing Class Initialization in Java-like Languages

**ABSTRACT** Language-based information-flow security is concerned with specifying and enforcing security policies for information flow via language constructs. Although much progress has been made on understanding information flow in object-oriented programs, little attention has been given to the impact of class initialization on information flow. This paper turns the spotlight on security implications of class initialization. We reveal the subtleties of information propagation when classes are initialized, and demonstrate how these flows can be exploited to leak information through error recovery. Our main contribution is a type-and-effect system which tracks these information flows. The type system is parameterized by an arbitrary lattice of security levels. Flows through the class hierarchy and dependencies in field initializers are tracked by typing class initializers wherever they could be executed. The contexts in which each class can be initialized is tracked to prevent insecure flows of out-of-scope contextual information through class initialization statuses and error recovery. We show that the type system enforces termination-insensitive noninterference.

**5.1 Introduction**

Language-based concepts and techniques are becoming increasingly popular in the context of security [23, 36, 41, 34, 24, 27, 14, 18] because they provide an appropriate level of abstraction for specifying and enforcing application and languagesensitive security policies. Popular examples include: i) Java stack inspection [41], which enforces a stack-based access-control discipline, ii) Java bytecode verification [24], which traverses bytecode to verify type safety, and iii) web languages such as Caja [27], ADsafe [14], and FBJS [18], which use program transformation and language subsets to enforce sandboxing and separation properties.

Language-based information-flow security [34] is concerned with specifying and enforcing security policies for information flow via language constructs. There has been much recent progress on understanding information flow in languages of increasing complexity [34], and, consequently, information-flow security tools
for languages such as Java, ML, and Ada have emerged [29, 37, 39]. In particular, information flow in object-oriented languages has been an area of intensive development [28, 10, 12, 5, 11, 6, 2, 31, 9, 21]. However, it is surprising that the impact of class initialization, being an important aspect of object-oriented programs, has received scarce attention in the context of security. In a language like Java, class initialization is lazy: classes are loaded as they are first used. This introduces challenges for information-flow tracking, in particular when class initialization may trigger initialization of other classes, which, for example, may include superclasses. Additional complexity is introduced by exceptions raised during initialization, as these can be exploited to leak secret information.

Because of its power, Java’s class loading mechanism [25] is a target for our model. A class is loaded, linked and initialized lazily on demand upon first active use [26]. Moreover the programmer may define application-specific loading policies. Class loading constitutes one of the most compelling features of the Java platform.

This paper turns the spotlight on security implications of class initialization (and loading and linking – prerequisites for initialization). We discuss the subtleties of information propagation when classes are initialized. The key issue is that class initialization may perform side effects (such as opening a file or updating the memory). The side effects may be exploited by the attacker who may deduce from these side effects which classes have (not) been initialized, which is sometimes sufficient to learn secret information.

We propose a formalization that illustrates how to track information flow in presence of class initialization by a type-and-effect system for a simple language. By ensuring that the initialization (or success thereof) of a class containing public fields in no way depends on the evaluation of an expression (or success thereof) containing secret data, the type-and-effect system guarantees security in a form of noninterference [19]. Informally, noninterference guarantees that a program’s public outputs are independent of secret inputs. A key intricacy here is that of class dependencies: An initialization of one class can cause the initialization of other classes. The only approach we are aware of that actually considers class initialization in the context of information-flow security is Jif [28, 29]. However, Jif’s restrictions on initialization code are rather severe: only simple constant manipulations, which cannot raise exceptions, are allowed. Our treatment of class initialization is more liberal than Jif’s and yet we demonstrate that it is secure. We argue that this liberty is desirable in scenarios such as server-side code.

5.2 Background

This section presents informal considerations that lead to a formalization in following sections. For illustration purposes, we use a simple subset of Java with classes that contain static fields. We assume variables and fields are partitioned

---

\[^{1}\text{The JVM specification permits the large flexibility as to the timing of loading and linking. But these activities must appear as if they happen on the class’s (or interface’s) first active use.}\]
into high (secret) and low (public), depending on the confidentiality of values they store. We assume that \( l \) and \( h \) are low and high variables, respectively. The goal is to prevent programs from leaking initial values of secret variables and fields, into final values of public variables and fields. The context corresponds to a body of a conditional or loop. This context is high if the guard depends on a secret (i.e., contains a secret variable or field) and low otherwise. Consider the following class definitions, with \( D.x \) and \( C.y \) low.

\[
\begin{align*}
class C & \{ y = 1 \} \\
class D & \{ x = 1/C.y \}
\end{align*}
\]

Certainly the above definitions may be considered secure since no high data is involved. However, an attempt to instantiate an object of \( D \) may cause an information leak:

\[
C.y := 0; \\
\text{if } h \neq 0 \text{ then new } D \text{ else skip}
\]

Except when \( h \) has the value 0 initially, the above program results in an error, since an initialization of class \( D \) is attempted. The object creation, should it occur, is the first active use of \( D \). This triggers initialization of \( D \). When a class is initialized, all its field- (field-)initializer assignments are executed in the state in which the first active use of the class occurred. Here, \( D.x := 1/C.y \) is performed in a state where \( C.y = 0 \), so a division by 0 occurs, producing an error. Note that in the terminology we have introduced, the initialization occurs in a high context. The attacker learns about the secret value of \( h \) by observing the termination behavior. It is illustrative to compare the above program that leaks through termination behavior with the following one that does not:

\[
\text{new } D; \\
C.y := 0; \\
\text{if } h \neq 0 \text{ then new } D \text{ else skip}
\]

In this latter program, \( D \) is initialized before it is used in a high context, so running the second \texttt{new} \( D \) statement does not incur any initialization activities.

In Java, when initialization of a class has completed abnormally, an exception is thrown and the class is marked as erroneous. Initialization of a class in an erroneous state is not possible [26, Ch. 2]. This makes initialization failure persistent throughout a run in the sense that when initialization of a class failed on its first (active) use, then it will fail on any future use irrespective of the state in which the second initialization is attempted. Catching initialization errors introduces a delicate scenario of information leaks. For instance, consider the

\footnote{To be precise, the \texttt{Class} object representing the class is labeled as erroneous.}

\footnote{Initialization may recover for instance by resorting to garbage collection. But normally a class is eligible for unloading when the running application has no reference to the class.}
following program:

\[
C.y := 0;
\text{if } h \neq 0 \text{ then (try new } D \text{ catch skip) else skip;}
C.y := 1;
\text{new } D
\]

Again, the above program results in an error except when \( h \) has the value 0 initially. In effect, information from \( h \) flows out of the scope of the if, through the initialization status of \( D \), into the termination behaviour of the last statement. The next variation of the example shows how to exploit this flow so that the resulting program always terminates normally and reflects the initial value of \( h \) in the final value of \( l \). Standard security type systems (e.g., [28, 32, 22, 4]) allow liberal handling of exceptions raised by expressions that are independent of secret data, as long as these expressions are used in public context. Since seemingly neither class definitions of \( C \) nor \( D \) involves high variables, one may be tempted to consider possible errors caused by initializing \( D \) as low. However, Program \( P_{\text{main}} \), given below, illustrates the subtlety of the problem:

\[
C.y := 0;
\text{if } h \neq 0 \text{ then (try new } D \text{ catch skip) else skip;}
C.y := 1; l := 0;
\text{try new } D \text{ catch } l := 1
\]

The above program successfully terminates irrespective of the initial value at \( h \), and the final value at \( l \) indicates whether \( h \) was nonzero or not.

Leaks like these can easily occur in practise; consider the Java program in Figure 5.1. Here, class \( D \), originating from [13], implements the Singleton design pattern through the "initialization on demand holder" idiom, which is thread-safe (as opposed to using double-checked locking [35]). To achieve thread-safety, this implementation leaves it to the class loader to construct the instance of \( D \) by calling the constructor in a field initializer. Since the class initialization phase is guaranteed to be serial [20, Section 12.4], no race conditions occur, so only one instance of \( D \) is ever created. However, the class dependencies in this program are exactly the same as those in \( P_{\text{main}} \), so in exactly the same way that one bit of the initial value of \( h \) is reflected in \( l \) after execution of \( P_{\text{main}} \), one bit of the high input is reflected in the low output of the program in Figure 5.1.

Dependencies in class definitions can impact security: before a class \( C \) is initialized, for each field and initializer \( C.x := e \) in \( C \), for each \( D.y \) occurring in \( e \), \( D \) is initialized. We say \( C \) depends on all these classes. If, when initializing \( C \), initialization of any class \( D \) which \( C \) depends on fails, then initialization of \( C \)
5.2 Background

Source File

```java
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class SingletonExample{
    static class D{ // Singleton
        private static final D INST = new D();
        public int x;
        public D(){
            x = 1 / C.y; // fails if C.x=0
        }
        public static D getInstance(){
            return INST;
        }
    }
    static class C{
        public static int y = 1;
    }
    public static void main(String[] a) throws IOException{
        int l = 0;
        System.out.print("h = "); // input (high)
        BufferedReader in = new BufferedReader(
            new InputStreamReader(System.in));
        int h = Integer.parseInt(in.readLine());
        C.y = 0;
        if (h == 0){
            try{
                D.getInstance(); // fails
                C.y = 1;
                try{
                    D.getInstance(); // fails if h=0
                } catch (LinkageError e){
                    l = 1; // run if h=0
                }
                System.out.println("l = " + l); // output (low)
            } catch (LinkageError e){
                l = 1; // 1 iff h = 0
            }
        }
        System.out.println("l = " + l); // output (low)
    }
}
```

Program Execution

$ javac SingletonExample.java
$ java SingletonExample
h = 0
l = 1
$ java SingletonExample
h = 42
l = 0

Figure 5.1: Singleton Example in Java
fails. For instance, consider the class definitions below, involving only low fields.

```java
class C { y = 1 }
class D0 { x = 1/C.y }
class D1 { x = D0.x }
```

$P_{dep}$, given below, always terminates normally and reflects secret input values in public results.

```java
C.y := 0;
if h ≠ 0 then (try new D0 catch skip) else skip;
C.y := 1; l := 0;
try new D1 catch l := 1
```

Class hierarchies impact security as well, since if a class $C$ is a subclass of a class $D$, then $C$ depends on $D$. So replacing the definition of $D_1$ with the following definition of $D_1$ in the class table above yields the same insecure flow in Program $P_{dep}$.

```java
class D1 extends D0 {}
```

The bottom line is that class initialization may perform side effects, causing information to leak. Languages which lazily initialize static classes, and where initialization failure is persistent, have this information channel. This includes Java (as seen above) and C# (example in the associated technical report [33]), but excludes C++ (no non-constant field initializers), VB.NET (no static class fields), Smalltalk (no field initializers), Ruby and Python (classes are objects, and failure is not persistent). One rather conservative approach to securing class initialization is to eliminate any possibilities of side effects during initialization and disallow errors due to initialization to be caught, an approach taken in Jif [28, 29]. This approach rules out, among other, read and write access to instance as well as static fields, method calls and object creation during initialization. For example, a static field of a reference type may only be initialized to null, which would exclude some standard Java APIs [38], such as `(java.lang)Boolean` and `String`, etc. Indeed Jif restricts (class) field initializers to simple constant manipulation that may not raise any exceptions. While it is rarely good practice to catch errors within ordinary methods, such as methods in libraries, there are several scenarios where this is good practice, e.g. in server applications to avoid crashing the entire system due to third party applications or to log messages. An example from practice where errors are caught and rethrown as exceptions can be found in [17].

This paper proposes and formalizes a different approach: we allow side effects during initialization, as long as these do not cause insecure flows. This paper extends and improves an earlier conference version [30]. The enforcement mechanism in the conference version disallows class initializations in secret contexts altogether, does not consider class dependencies, and uses a fixed lattice of security levels. These limitations are resolved in the present version: we present a more permissive type-and-effect system which is furthermore parameterized by
an arbitrary lattice of security levels. We track flows through dependencies in the
class hierarchy and in field initializers, and track every context in which each class
can be initialized in, to prevent the kind of leak demonstrated in program $P_{\text{main}}$.
Finally, we prove that our type system soundly enforces a termination-insensitive
notion of noninterference (TINI).

Section 5.5 develops a type-and-effect system for tracking information flows
in a simple language with classes, defined in Section 5.3 (class hierarchies added
in Section 5.6). We show in Section 5.7 that the type-and-effect system enforces
TINI, given in Section 5.4. Sections 5.8 discusses related work, and Section 5.9
concludes.

5.3 Language

The language for our formal study is defined by the the following abstract syntax:

| Expression | $e ::= n \mid x \mid e \oplus e \mid C.x$ |
| Statement | $s ::= \text{skip} \mid s;s \mid x ::= e \mid C.x ::= e$
| Class definition | $c ::= C \{i\} \mid C :: C \{i\}$ |
| Field definitions | $i ::= (x = e)^*$ |
| Class table | $\tau ::= c^*$ |

Metavariables $x$, $n$, $\oplus$ and $C$ range over variables, integers, operators and class
names respectively. Notation $C.x$ denotes field $x$ of class $C$. Compound
expressions are built using binary (integer arithmetic) operators, which metavariable
$\oplus$ ranges over. We assume a collection of such operators, partitioned into
partial resp. total operators, which $\oplus_p$ resp. $\oplus_T$ range over. Total, resp. par-
tial, operators are represented as functions mapping into $\mathbb{Z}$, resp. $\mathbb{Z} \cup \{\bullet\}$,
if both operands are integers, and into $\{\bullet\}$ if either operand is $\bullet$. The case
$n_1 \oplus_p n_2 = \bullet$ represents the scenario where $\oplus_p$ is undefined on $(n_1, n_2)$. So,
here, $\bullet$ denotes an evaluation error as a result of an undefined partial operator
application. Metavariable $c$ ranges over class definitions. Our class definitions
are compact versions of the ones used in the previous section; a class definition
$C \{x_0 = e_0 \ldots x_k = e_k\}$ declares a class named $C$ consisting of the fields $x_i$
and field initializers $e_i$. We assume $i \neq j \implies x_i \neq x_j$ in the following.
When $C$ extends $C'$ ($C$ a subclass of $C'$, $C$ inherits from $C'$), we instead write
$C :: C' \{x_0 = e_0 \ldots x_k = e_k\}$ ($::$ is the subtype relation). We sometimes write
$x_0 = e_0 \ldots x_k = e_k$ as $x_0 = e_0; \ldots ; x_k = e_k$ when this helps readability. If $i = \epsilon$
in $C \{i\}$ and $C :: C' \{i\}$, then $C$ has no fields. We write these definitions as
$C \{\}$ and $C :: C' \{\}$. A class table $\tau$ is a (finite) list of class definitions. We inter-
pret $\tau$ as a function mapping a class name $C$ to the class definition in $\tau$ named $C$.
as follows.

\[(c \tau)(C) = \begin{cases} 
  c & \text{if } \exists C', i . c \in \{C \{i\}, C <: C' \{i\}\}, \\
  \tau(C) & \text{otherwise.}
\end{cases}\]

A program is a pair \((\tau, s)\) of a class table and a statement. We let \(P\) range over programs. To lighten notation in our formal study, we assume a fixed \(\tau\) hereafter.

The following syntactic categories, not part of the language syntax, arise during program execution.

\[
\begin{align*}
\text{Initialization status} & \quad S ::= u | b | 1 | \bullet \\
\text{Initialization result} & \quad I ::= b | 1 | \bullet \\
\text{Value} & \quad V ::= n | \bullet \\
\text{Termination result} & \quad T ::= \text{skip} | \bullet
\end{align*}
\]

Programs operate on a state (a.k.a. store, or memory). We let \(\sigma\) range over states. These map variables and fields to integers, and class names to (class) initialization statuses. A class initialization status \(S\) denotes the loaded status of a class named \(C\) in a state \(\sigma\): \(C\) is uninitialized in \(\sigma\) when \(\sigma(C) = u\); \(C\) is being initialized when \(\sigma(C) = b\), and is initialized successfully when \(\sigma(C) = 1\). \(C\) has failed to initialize when \(\sigma(C) = \bullet\). We use the following notation for updating the value of \(x\) in \(\sigma\), and analogous notation for a \(C.x \mapsto n\) and a \(C \mapsto \) update.

\[
\sigma[x \mapsto n](x') = \begin{cases} 
  n & \text{if, } x \equiv x' \\
  \sigma(x') & \text{otherwise}
\end{cases}
\]

We use \(\_\) as a wildcard, that is, any mathematical object, which we do not intend to refer to\(^4\). The (Big-step operational) semantics of our language is given by relations of the form \(\langle \sigma, \_ \rangle \Rightarrow \langle \sigma', \_ \rangle\). Here, \(\sigma\) is the state before evaluation, the former \(\_\) is that which is evaluated under \(\sigma\), the latter \(\_\) is the result of the evaluation, and \(\sigma'\) is the resulting state. Evaluation of expressions is given in Figure 5.2 (a). The relation \(\langle \sigma, e \rangle \Rightarrow \langle \sigma', V \rangle\) states that the expression \(e\) in the state \(\sigma\) evaluates to the value \(V\) with final state \(\sigma'\). If \(V = \bullet\), an error occurred during the evaluation of \(e\) under \(\sigma\). Otherwise, \(e\) evaluated successfully, in which case \(V = n\) for some \(n\). The inference rules in Figure 5.2 (a) are straightforward except those for reading from a field of a class, \(C.x\). Both read and write access to a field of a class \(C\) triggers initialization of \(C\). That is, the definition of \(C, \tau(C)\), is evaluated under \(\sigma\) using the rules in Figure 5.2 (b), using relation \(\langle \sigma, c \rangle \Rightarrow \langle \sigma', I \rangle\) with \(c = \tau(C) = C \{i\}\), which states that class definition \(c\) in state \(\sigma\) evaluates to initialization result \(I\) with final state \(\sigma'\). If \(\sigma(C) \neq u\), then initialization of \(C\) has already been triggered, and possibly failed, so \(I = \sigma(C)\) and \(\sigma' = \sigma\). If \(\sigma(C) = u\), then \(C\) needs to be initialized. This is done by executing the field definitions of \(C\) as assignments. We refer to this code as the (class) initializer of \(C\). Given field definitions \(i\) of \(C\), we define the

---

\(^4\)Multiple occurrences of \(\_\) in the same context can represent different mathematical objects.
5.3 Language

(a) Expressions

\[
\begin{align*}
\text{NUM} & : \langle \sigma, n \rangle \Rightarrow \langle \sigma, n \rangle \\
\text{VAR} & : \langle \sigma, x \rangle \Rightarrow \langle \sigma, \sigma(x) \rangle \\
\text{OP-EL} & : \langle \sigma, e_1 \rangle \Rightarrow \langle \sigma', \bullet \rangle \\
\text{OP-ER} & : \langle \sigma, e_1 \oplus e_2 \rangle \Rightarrow \langle \sigma'', \bullet \rangle \\
\text{OP-EP} & : \langle \sigma, e_1 \rangle \Rightarrow \langle \sigma', n_1 \rangle \quad \langle \sigma', e_2 \rangle \Rightarrow \langle \sigma'', n_2 \rangle \\
& \quad n_1 \oplus n_2 = \bullet \\
\text{OP-OK} & : \langle \sigma, e_1 \oplus e_2 \rangle \Rightarrow \langle \sigma'', \bullet \rangle \\
\text{FIELD-E} & : \langle \sigma, \tau(C) \rangle \Rightarrow \langle \sigma', \bullet \rangle \\
\text{FIELD-OK} & : \langle \sigma, C.x \rangle \Rightarrow \langle \sigma', \sigma'(C.x) \rangle \\
\text{INIT-A} & : \sigma(C) \neq u \\
& \quad \langle \sigma, C \{i\} \rangle \Rightarrow \langle \sigma, \sigma(C) \rangle \\
\text{INIT-U} & : \sigma(C) = u \\
& \quad \langle \sigma, C \{i\} \rangle \Rightarrow \langle \sigma'[C \mapsto I(T)] \rangle \\
& \quad \langle \sigma'[C \mapsto I(T)], I(T) \rangle
\end{align*}
\]

(b) Class Definitions

Figure 5.2: Big-step Operational Semantics for Expression Evaluation

initializer of \( C \), \( s(C, i) \), as follows.

\[ s(C, \epsilon) = \text{skip} \]
\[ s(C, (x = e)i) = C.x := e; s(C, i) \]

So, to initialize \( C \), we execute \( s(C, i) \) under \( \sigma \) with initialization status of \( C \) set to \( b \), using relation \( \langle \sigma, s \rangle \Rightarrow \langle \sigma', T \rangle \) with \( s = s(C, i) \), which states that statement \( s \) in state \( \sigma \) evaluates to termination result \( T \) with final state \( \sigma' \). If \( T = \bullet \), then an error occurred during the execution. Otherwise, \( T = \text{skip} \), signifying that the execution was error-free. Once execution of \( i \) has terminated with termination result \( T \), we set the initialization status of \( C \) in the resulting memory to \( I(T) \), where

\[ I(T) = \begin{cases} 1 & \text{if } T = \text{skip}, \\ \bullet & \text{otherwise (} T = \bullet ) \end{cases} \]
If \( I = \bullet \), then this means \( C \) failed to initialize (now or previously), so its field read will fail. Otherwise, the resulting value is the value \( C.x \) has in the resulting memory. Note that for readability, we have postponed all treatment of class hierarchies to Section 5.6.

Execution of statements is given in Figure 5.3. We let \( \tilde{0} \) denote a non-0, non-\( \bullet \) value. Again, the inference rules for the one-step reduction relation are straightforward, with the exception of the expression evaluation error rule (E-e⇒) and the class field assignment rules (FIELD-a-E⇒) and (FIELD-a-OK⇒). For the former,
Q[e] specifies a grammar with a “hole” in it. That is, e is a formal parameter, and, for instance, Q[4 + 5 * x] defines a grammar where e has been replaced by 4 + 5 * x (while 4 + 5 * x do skip is generated by said grammar). For the latter, note that an assignment to C.x triggers initialization of C. To initialize C in rules (FIELD-A-E⇒) and (FIELD-A-OK⇒), we simply read C.x, since doing this causes C to be initialized as a side-effect.

Most of our results apply for objects from multiple syntactic categories, yielding following meta-categories.

\begin{align*}
\text{Expressional} & : a ::= e \mid c \mid i \\
\text{Term} & : t ::= a \mid s \\
\text{Result} & : R ::= V \mid I \mid T
\end{align*}

Expressions and elements evaluated as a consequence of expression evaluation are collectively referred to as expressionals. We refer to any mathematical object in our syntactic categories as a term, and the result of term evaluation as a result.

## 5.4 Security

We now develop a security notion for our language, and give some running examples of programs that leak through class initialization statuses.

### 5.4.1 Lattices

We recap some basic definitions from order theory. A partially ordered set (poset) is a set A together with a binary relation \( \sqsubseteq \) over A which is reflexive, antisymmetric and transitive. For \( x, y, z \in A \), \( z \) is a join (least upper bound, lub) of \( x \) and \( y \) if \( x \sqsubseteq z, y \sqsubseteq z \) and \( \forall w \in A. w \sqsubseteq x \land w \sqsubseteq y \implies z \sqsubseteq w \). If a lub of \( x \) and \( y \) exists, it is (provably) unique, so we denote it \( x \sqcup y \), thus defining a join operator \( \sqcup \). When \( x \sqcup y \) is defined \( \forall x, y \), then poset A is a join-semi-lattice. For \( x, y, z \in A \), \( z \) is a meet (greatest lower bound, glb) of \( x \) and \( y \) if \( z \sqsubseteq x, z \sqsubseteq y \) and \( \forall w \in A. x \sqsubseteq w \land y \sqsubseteq w \implies w \sqsubseteq z \). If a glb of \( x \) and \( y \) exists, it is (provably) unique, so we denote it \( x \sqcap y \), thus defining a meet operator \( \sqcap \). When \( x \sqcap y \) is defined \( \forall x, y \), then poset A is a meet-semi-lattice. A poset is a lattice iff it is a join- and a meet-semi-lattice.

### 5.4.2 Noninterference

We assume an arbitrary lattice of security levels [15]. In our examples, we use a two-level lattice; low (public) and high (secret), with low \( \sqsubseteq \) high. We let metavariables \( \ell \) and pc range over security levels. We assume that each variable and class field has been assigned a fixed security level, and denote this mapping from variables and class fields to security levels by lvl. We extend this mapping to expressions by letting \( \text{lvl}(e) \) denote the least upper bound of the security levels of all
variables and class fields occurring in \( \text{lvl}(e) \). For notational simplicity, we assume a fixed \( \text{lvl} \) henceforth.

These security levels define who can observe what. An observer in our setting is assigned a security level expressing which variables and class fields we assume the observer has access to. If an observer has level \( \ell \), then the observer can read variables and class fields with level \( \preceq \ell \), and write to these before the program is run. This gives rise to an (observational) equivalence on memories.

\[ (5.1) \text{ DEFINITION} \ (\ell\text{-equivalence}) \ \sigma_1 \text{ and } \sigma_2 \text{ are } \ell\text{-equivalent, written } \sigma_1 =_{\ell} \sigma_2, \text{ iff,} \]

\[ \forall x. \text{lvl}(x) \preceq \ell \implies \sigma_1(x) = \sigma_2(x) \quad (1) \]

\[ \forall C. (\exists C.x. \text{lvl}(C.x) \preceq \ell) \implies \sigma_1(C) = \sigma_2(C) \quad (2) \]

\[ \forall C.x. \left( \text{lvl}(C.x) \preceq \ell \land \sigma_1(C) = \sigma_2(C) \in \{1,0\} \right) \implies \sigma_1(C.x) = \sigma_2(C.x) \quad (3) \]

It turns out \( =_{\ell} \) is an equivalence relation; this will be useful later.

We adopt a commonly-used baseline policy of termination-insensitive noninterference \([40, 34, 32]\). Intuitively, a program satisfies noninterference if for any two initial memories that agree on public data, whenever the program runs that start in these memories terminate, then these runs result in the memories that also agree on public data. This policy is an appropriate fit for batchjob programs, where leaks due to (non)termination are ignored because they may leak at most one bit per execution \([3]\). An initial memory \( \sigma_{\text{init}} \) in our setting satisfies \( \sigma_{\text{init}}(C) = u \) for all \( C \), and \( \sigma_{\text{init}}(C.x) = 0 \) for all \( C.x \). In this sense, the “input” to a program is a value assignment to global variables.

\[ (5.2) \text{ DEFINITION} \ (\text{TINI}) \ s \text{ satisfies termination-insensitive noninterference (TINI) if, for all } \ell \text{ and initial } \sigma_1, \sigma_2 \text{ for which } \sigma_1 =_{\ell} \sigma_2, \text{ if } \langle \sigma_j, s \rangle \implies \langle \sigma_j', \text{skip} \rangle, \text{ then } \sigma_1' =_{\ell} \sigma_2'. \]

### 5.4.3 Running Examples

We now present some challenges an enforcement mechanism of TINI must deal with, in the form of example programs, some of which are insecure, and others which are secure. While the programs are simple, their pattern can easily arise in larger programs, so a decent enforcement mechanism must reject the insecure programs, yet accept the secure programs, presented here. Our first program, \( P_{\text{main}} \), is the main example we saw in the introduction. Even with all fields in the class table labeled \( \text{high} \), this program leaks an out-of-scope secret context by (ab)use of the \texttt{try-catch} language construct. One way to reject this program in an enforcement would be to “taint” \( C \) by the security level of all contexts which dereferencing of \( C \) occurs in. However, this approach would also reject \( P_{\text{art}} \), obtained by replacing the definition of \( D \) with the following one.

\[ D \{ x = 1 + C.y \} \]
This program is secure. However, try new C catch l := 1 becomes insecure for some memories. Namely, the memories that state that C failed to initialize. However, these memories never arise when running $P_{\text{art}}$ on an initial memory (C cannot fail to initialize), meaning we cannot universally and unconditionally quantify memories when proving soundness of the enforcement mechanism, as artificial flows can then arise.

The next program, $P_{\text{LinH}}$, leaks since the point during control flow at which a class C with an observable field is initialized, depends on a secret, and observable parts of the memory change between these points.

\[
\begin{align*}
D.l &:= 0; \\
\text{if } h \text{ then new } C \text{ else skip;}
\end{align*}
\]

\[
\begin{align*}
D.l &:= 1; \\
\text{if } C.l \text{ then } l := 1 \text{ else } l := 0;
\end{align*}
\]

Class table of $P_{\text{LinH}}$:

\[
\begin{align*}
C \{l = 0 + D.l\} \\
D \{l = 1\}
\end{align*}
\]

Securing this program is an easy matter; make sure this cannot happen, for instance, by injecting new C; just before the first if statement in $P_{\text{LinH}}$. We refer to this modified $P_{\text{LinH}}$ as $P'_{\text{LinH}}$. The lesson here is that it is okay for a reference to a class C, containing a low field, to appear in a high context, as long as dereferencing C there cannot cause C to be initialized.

### 5.5 Enforcement

We now present a type system for enforcing noninterference. The type system formalizes a data-flow analysis for tracking information dependencies in a program. To guarantee secure class initialization, in addition to standard information flow tracking in imperative systems [16, 40, 34], we need to track two things during evaluations: i) which classes are initialized, and ii) what information is leaked when such an action does (not) produce an error. To track i), our enforcement performs a must analysis to soundly approximate at any given point in the program which class must be (in the process of being) initialized. To track ii), our enforcement maintains a security level for each class expressing information obtained by observing whether its initialization produced an error or not ($\perp$ for classes which cannot fail to initialize). The type system is algorithmic, in the sense that the type rules specify how compute the constituents of a type judgment. A type inference algorithm can thus easily be obtained from our type system.
5.5.1 Type Environment

The above-mentioned tracking is maintained by the *type environment* $\Gamma$. With $\mathcal{L}_{\mathbf{I}}$ denoting the lattice given by $u \sqsubseteq b$ and $b \sqsubseteq 1$, $\Gamma$ consists of two mappings,

$$\Gamma^s : \text{dom}(\tau) \to \mathcal{L}_{\mathbf{I}} \quad \Gamma^e : \text{dom}(\tau) \to \mathcal{L}.$$  

$\Gamma^s$ keeps track of the initialization status of classes. The second, $\Gamma^e$, tracks information conveyed by observing whether a previous initialization of a class yielded an error. The notation for updating a $\Gamma$ is given below (note the first case in $\Gamma[C \mapsto e]^{s}(\hat{C})$).

$$\Gamma[C \mapsto \mathbf{I}]^s(\hat{C}) = \begin{cases} I & \text{if } \hat{C} = C \\ \Gamma^s(\hat{C}) & \text{otherwise.} \end{cases}$$

$$\Gamma[C \mapsto \mathbf{I}]^e(\hat{C}) = \begin{cases} \Gamma^e(\hat{C}) \sqcup \mathbf{I} & \text{if } \hat{C} = C \\ \Gamma^e(\hat{C}) & \text{otherwise.} \end{cases}$$

For $\Gamma$ to be sound wrt. some term $t$, $\Gamma^s$ must be an underapproximation, and $\Gamma^e$ an overapproximation. Say you have $\Gamma_1$ and $\Gamma_2$ representing type environments for two different control flow paths in the evaluation of $t$. To obtain a $\Gamma$ for the join point of these two paths, we set $\Gamma = \Gamma_1 \odot \Gamma_2$, with $\odot$ defined

$$(\Gamma_1 \odot \Gamma_2)^s(C) = \Gamma_1^s(C) \sqcap \Gamma_2^s(C)$$

$$(\Gamma_1 \odot \Gamma_2)^e(C) = \Gamma_1^e(C) \sqcup \Gamma_2^e(C)$$

The rationale for $\Gamma^s$: We can only guarantee a class is initialized at the join point for all evaluations reaching said point if both $\Gamma_1$ and $\Gamma_2$ say the class is initialized. The rationale for $\Gamma^e$: We assume the attacker knows exactly which control flow path was taken by the program, and thus exactly where classes fail to initialize.

5.5.2 Type Rules

The type rules, for expressions and statements respectively, are given in Figures 5.4 and 5.5. The rules define a type judgement relation $pc \vdash \Gamma \{t\} \Gamma' : \ell$, where $pc, \ell \in \mathcal{L}$. This type judgement reads:

“Under context $pc$, $t$ maps type environment $\Gamma$ to type environment $\Gamma'$ and error level $\ell$”

We use the type judgement to gain insight into program behaviour in the following way.

“Assume $t$ is to be evaluated in a context containing information $pc$, with $\Gamma$ expressing which classes are already initialized and information leaked through initialization error observations. Then $\Gamma'$
records (at most) the classes which were initialized during (successful) evaluation of \( t \), and the new information that can leak through initialization error observations. Further, \( \ell \) expresses the information leaked to observations of an evaluation error of \( t \). This whole evaluation contains no insecure flows”.

That last remark entails that we can use our enforcement to reason about the security of programs. The type rules contain several conditions and constraints engineered to make this so. We will step through the type rules now, and highlight a proof that our type judgement carries this whole meaning in the next section.

We start with the rules for expression evaluation in Figure 5.4 (a). Evaluation of \( n \) and \( x \) cannot fail and does not initialize classes. Thus, for \( pc \vdash \Gamma \{ n \} \Gamma' : \ell \)
We now move on to Figure 5.4 (b). Recall from the semantics that when evaluating \( C.x \), either \( C \) is triggered or not. If \( C \) is triggered, no evaluation occurs, and if \( C \) is not triggered, the initializer is evaluated. This behaviour is reflected in \((\text{INIT-T}_t)\) and \((\text{INIT-F}_t)\). When typing \( C \{i\} \), we first consult \( \Gamma^s(C) \). If \( \Gamma^s(C) = i \), no flows occur since no evaluation can possibly occur here. However, if \( \Gamma^s(C) = u \), initializer \( i \) is typed using the initializer typing rule in Figures 5.4 (c). The context of this typing is raised by \( \Gamma^e(C) \), as this evaluation is only possible if \( C \) has not been initialized (and then possibly failed) previously. \( \Gamma \) is updated to \( \Gamma[C \mapsto^s b] \) since \( C \) is being initialized. As this initialization attempt can fail as a consequence of a previous initialization attempt failing, \( \Gamma^e(C) \) is reflected in \( \ell \). At last, the initialization status of \( C \) is set to \( i \) and the error level tracking of \( C \) is augmented by \( \ell_C \), in \( \Gamma' \). Notice that there is no rule for the case \( \Gamma^s(C) = b \). This means that our type system will reject any program which references a mutual dependency in the class table, such as classes \( C \) and \( D \) in the following class table.

\[
\begin{align*}
C & \{l = 1 + D.l; h = 1 + E.h\} \\
D & \{l = 0 + C.h\} \\
E & \{h = 4\}
\end{align*}
\]

This also means that between \( \Gamma[C \mapsto^s b] \) and \( \Gamma' \), the initialization status of \( C \) is not updated. So \( \Gamma^s(C) = b \), always. While rejecting programs with mutual dependencies does rule out some well-behaving programs, it has been pointed out e.g. in [42] section 3.5.1 that a) mutual dependencies should be considered a bug since they introduce hard-to-predict program behaviour, and b) there are tools which can detect these. Because of this, the added challenge in proving soundness, and since mutual dependencies do not increase the bandwidth of leaks through class initialization statuses, we chose not to consider this feature in our security analysis. To add this feature to the typesystem, however, it should be sufficient to replace the premise of \((\text{INIT-T}_t)\) with \( \Gamma^s(C) \in \{b,1\} \).
We now move on to Figure 5.4 (c). The one rule for initializer typing, (Init.),
types each initializer expression in the same order as they would be evaluated
in the semantics. Since the evaluation of an initializer expression depends on
the success of all prior initializer expressions in this order, the context of the \( e_i \) typing
is raised by \( \sqcup_{j=1}^{i-1} \ell_j \). Likewise, a future read of any field in \( C \) will succeed only
if none of the initializer expressions evaluated to error during initialization, so
\( \sqcup_{j=1}^{n} \ell_j \subseteq \text{lvl}(C.x_i) \) must hold.

In the statement typing rules in Figure 5.5, we prevent explicit flows (\( l := h \))
and implicit flows [16] via control structure (\( \text{if } h = 0 \text{ then } l := 0 \text{ else } l := 1 \))
in a standard fashion [16, 40, 34]. What is nonstandard here is how we con-
struct type environments for joins in control flow branches. The control flows
for the language constructs with join points (in successful evaluations) is illus-
trated in Figure 5.6. Compare (Ir.) to Figure 5.6 (a). Here there are two suc-
cessful control flow paths. The only classes we can guarantee are initialized at the
join point are those that are initialized after taking either control flow path, hence
the \( (\Gamma_1 \odot \Gamma_2)^6 \) in the join point. Now compare (Try.) to Figure 5.6 (b). There
are two control flow paths here, one for successful evaluation of \( s_1 \), the other for
successful evaluation of \( s_2 \) after an erroneous evaluation of \( s_1 \). The interesting
case is the latter, as \( s_2 \) is typed under \( \Gamma \odot \Gamma_1 \). Since \( s_1 \) can yield an error without
initializing a class, \( s_2 \) must be typed under \( \Gamma^3 \). Since \( s_1 \) can fail anywhere, and the

5.5 Enforcement

\[
\begin{align*}
\text{Skip} & : \quad pc \vdash \Gamma \{\text{skip}\} \Gamma : \bot \\
\text{VAR-A} \quad \text{if } x \triangleq e \quad \Gamma' : \ell & : \text{lvl}(x), pc, \ell \subseteq \text{lvl}(x) \\
\text{FIELD-A} \quad pc \vdash \Gamma \{e \} \Gamma' : \ell & \quad pc \vdash \{C.x := e\} \Gamma'' : \ell' \\
\text{SEQ} \quad pc \vdash \Gamma_0 \{s_1\} \Gamma_1 : \ell_1 & \quad pc \sqcup \ell_1 \vdash \Gamma_1 \{s_2\} \Gamma_2 : \ell_2 \\
\text{If} \quad pc \vdash \Gamma \{e\} \Gamma' : \ell & \quad pc \sqcup \ell \vdash \Gamma' \{s_i\} \Gamma_i : \ell_i \\
\text{Try} \quad pc \vdash \Gamma \{\text{try } s_1 \text{ catch } s_2\} \Gamma_1 \odot \Gamma \odot \Gamma_2 : \ell_2 \\
\text{While} \quad pc \sqcup \ell_i \vdash \Gamma_i \{e\} \Gamma' : \ell_i^e & \quad pc \sqcup \ell_i \sqcup \ell_i^e \sqcup \text{lvl}(e) \vdash \Gamma' \{s\} \Gamma_{i+1} : \ell_{i+1}^e \\
& \quad \ell_0 = \bot \quad \ell_{i+1} = \ell_i \sqcup \ell_i^e \sqcup \ell_{i+1}^e \quad i = 0..n \quad (\Gamma_n, \ell_n) = (\Gamma_{n+1}, \ell_{n+1}) \\
& \quad pc \vdash \Gamma_0 \{\text{while } e \text{ do } s\} \bigodot_{j=0}^n \Gamma_j \odot \Gamma_{j+1} : \ell_n
\end{align*}
\]

Figure 5.5: Typing of Statements
attacker could know exactly where $s_1$ fails, $s_2$ must be typed under $\Gamma^e_1$. It turns out that $\Gamma^s$ and $\Gamma^e_1$ together are exactly $\Gamma \odot \Gamma_1$. Besides this, our treatment of exceptions is standard (see [28, 32, 22, 4]). Now compare (\texttt{While}) to Figure 5.6 (c). Here there are (possibly) infinitely many control flow paths. Observe that loops are typed the same way as the infinitely long sequence $e; s; s; e; s; \cdots$. Since there are finitely many $\Gamma$ (as $\tau$ has finite classes, and policy has finite labels), by Lemma 5.4, eventually the typing of this sequence reaches a fixed point. Since $e$ might yield $0$ on first evaluation, the only classes we can guarantee are initialized are those that are initialized during evaluation of $e$, so the $^s$-part of the join environment should equal $\Gamma^s_0$. Since an attacker could know exactly which iteration of the loop failed, and whether $e$ or $s$ failed, the $^e$-part of the join environment must equal $\bigcup_{j=0}^n \Gamma^e_j \sqcup \Gamma^e_{j+1}$. It turns out that this, together with $\Gamma^s_0$, is exactly $\bigodot_{j=0}^n \Gamma_j \odot \Gamma_{j+1}$.

### 5.6 Hierarchies

So far, we have seen a complete semantics and set of type rules for our language, excluding the $C <: C' \{i\}$ language construct. We saw in $P_{\text{dep}}$ that hierarchical dependencies can be utilized to leak information. We now present the semantics of hierarchical dependencies, and how to analyze programs which utilize these. We handle hierarchical dependencies in the same way as dependencies in field initializers. While the procedure of initializing $C'$ as a consequence of initializing $C$ is the same as the procedure for initializing $C'$ as a consequence of reading a field of $C'$, we cannot emulate superclass initialization through field reads like we did in (\texttt{FIELD-A-E⇒}) and (\texttt{FIELD-A-OK⇒}), since a class does not necessarily have any
### 5.6 Hierarchies

#### Figure 5.7: Big-step Operational Semantics for Expression Evaluation, Class Hierarchies

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INIT-s-a</strong></td>
<td>If ( \sigma(C) \neq \text{uninitialized} ), then initialize ( C ) in ( \sigma ).</td>
</tr>
</tbody>
</table>
| **INIT-s-uf** | If \( \sigma(C) = \text{uninitialized} \), then:
- If \( \sigma(C') \in \{1, b\} \), then initialize \( C \) to \( B \), \( \sigma' \) to \( I(T) \).
- Otherwise, \( \sigma(C) = \text{uninitialized} \), \( \sigma(C') = \text{uninitialized} \), \( \sigma' \) to \( I(T) \). |
| **INIT-s-ui** | If \( \sigma(C') \in \{1, b\} \), then:
- If \( \sigma(C) = \text{uninitialized} \), then:
  - \( \sigma'[C \mapsto b, \tau(C')] \Rightarrow \langle \sigma'^{\prime \prime}[C \mapsto I(T)], I(T) \rangle \) is the case. |
| **INIT-s-uuf** | If \( \sigma(C) = \text{uninitialized} \), \( \sigma(C') = \text{uninitialized} \), \( \sigma'(C') \Rightarrow \langle \sigma''[C \mapsto I(T)], I(T) \rangle \) is the case. |
| **INIT-s-uui** | If \( \sigma(C) = \text{uninitialized} \), \( \sigma(C') = \text{uninitialized} \), \( \sigma'(C') \Rightarrow \langle \sigma''^{\prime \prime}[C \mapsto I(T)], I(T) \rangle \) is the case. |

While some rules can be merged to yield a more concise set of rules, having a rule for each possible combination of class and superclass initialization statuses simplifies case analysis of programs, and thus our proofs. As such, while these rules may be technical, they should be unsurprising, given what we have seen so far.

#### 5.6.1 Semantics

The semantic rules of class hierarchies are given in Figure 5.7. Just like for normal class initialization, if initialization of \( C \) has begun already, we do nothing, and simply return the initialization status of \( C \) in \( \sigma \). Hence (\( \text{INIT-s-a} \Rightarrow \)). If \( C \) is uninitialized, however, then the first thing to do when initializing \( C \) is to initialize its superclass, \( C' \), if needed. If initialization of \( C' \) has failed, then we cannot proceed initializing \( C \), so initialization of \( C \) will fail. Hence (\( \text{INIT-s-uf} \Rightarrow \)). If initialization of \( C' \) has already begun (and has not failed), we skip the initialization of \( C' \), so this case in rule (\( \text{INIT-s-ui} \Rightarrow \)) is just like the semantics of normal class initialization. If \( C' \) is uninitialized, we initialize it (setting the initialization status of \( C \) to being initialized, since we are currently initializing \( C \)). If this initialization of \( C' \) fails, then the initialization of \( C \) immediately fails (we don’t even run \( i \)), hence (\( \text{INIT-s-uuf} \Rightarrow \)). If initialization of \( C' \) succeeds, however, we run \( i \). This evaluation yields either \( \text{skip} \) (for success) or \( \bullet \) (for failure). Regardless of which, we return, with initialization of \( C \) set to the initialization status corresponding to the return value from the \( i \) evaluation, \( i \) (for success) or \( \bullet \) (for failure). Hence (\( \text{INIT-s-uui} \Rightarrow \)).
\[
\begin{array}{c}
\text{INIT-s-t} \quad \frac{\Gamma^s(C) = 1 \quad \text{pc} \vdash \Gamma \{C <: C' \{i\}\} \Gamma : \perp}{
\Gamma^s(C) = \top \quad \Gamma^s(C') = 1 \quad \text{pc} \sqcup \Gamma^e(C') \vdash_{C} \Gamma[C \mapsto \top \text{\{i\}}]} \Gamma : \ell_C}
\end{array}
\]

\[
\begin{array}{c}
\text{INIT-s-ft} \quad \frac{\Gamma^s(C) = \top \quad \Gamma^s(C') = \top \quad \text{pc} \sqcup \Gamma^e(C) \vdash_{C} \Gamma[C \mapsto \top \text{\{i\}}]} \Gamma : \ell_C \quad \text{pc} \sqcup \ell_C' \sqcup \Gamma^e(C) \vdash_{C'} \Gamma' \{i\} \Gamma'' : \ell_C}
\end{array}
\]

\[
\begin{array}{c}
\text{INIT-s-ff} \quad \frac{\Gamma^s(C) = \top \quad \Gamma^s(C') = \top \quad \text{pc} \sqcup \Gamma^e(C) \vdash_{C} \Gamma[C \mapsto \top \text{\{i\}}]} \Gamma''[C \mapsto \top \text{\{i\}}, C \mapsto \top \ell_C' \sqcup \ell_C] : \ell_C' \sqcup \ell_C \sqcup \Gamma^e(C)}
\end{array}
\]

(a) Class Definitions

Figure 5.8: Typing of Expressions, Class Hierarchies

### 5.6.2 Type Rules

The rules for typing class definitions with hierarchical dependencies are given in Figure 5.8. Again, like for normal class initialization, when the type environment asserts that \( C \) is initialized, since the semantics would do nothing in this case, no flows occur here, so when \( \text{pc} \vdash \Gamma \{c\} \Gamma' : \ell, \ell' = \perp \), hence (\text{INIT-s-t}). If \( C \) is uninitialized in \( \Gamma \), we check the status of \( C' \). If \( \Gamma \) asserts that \( C' \) is already initialized, then no flows arise from or through \( C' \), so the (\text{INIT-s-ft}) case is exactly like normal class initialization. The last rule, (\text{INIT-s-ff}), considers the case where both \( C \) and \( C' \) are uninitialized in \( \Gamma \). Since this might also be the case in the program semantics, we assume the worst, that neither class is initialized. First we type the \( C' \) class definition (under the assertion that \( C \) is being initialized). Since \( i \) would only be evaluated if \( C' \) initialized successfully, we raise the context under which we type \( i \) (evaluated after the \( C' \) definition) by \( \ell_C' \). Likewise, because \( C \) can fail to initialize as a consequence of either the \( C' \) initialization failing or the \( i \) evaluation failing, \( \Gamma''(C) \) is augmented with \( \ell_C' \sqcup \ell_C \).

### 5.7 Soundness

We now see how the differing aspects of the type system fit together to form a whole which rejects leaking programs such as \( P_{\text{main}} \) and \( P_{\text{LinH}} \) while at the same time staying permissive, accepting programs such as \( P_{\text{art}} \) and \( P'_{\text{LinH}} \). As evident in program \( P'_{\text{main}} \), which is \( P_{\text{main}} \) with all class fields labeled \( \text{high} \), the main flows of interest are not the typical explicit and implicit flows for simple imperative programs. For instance, the parts

\[
\begin{align*}
C.y & := 0; \\
\text{if } h \neq 0 \text{ then (try new } D \text{ catch skip) else skip;}
\end{align*}
\]
and

\[
C.y := 1; \ l := 0;
\begin{align*}
\text{try } & \text{new } D \\
\text{catch } & \ l := 1
\end{align*}
\]

are both secure; it is only when you put these in sequence that an insecurity arises. Typical type systems for information flow are (sequentially) compositional wrt. TINI, which makes proving soundness wrt. TINI a simple matter. As our type system does not have this property in general, our soundness proof is nonstandard. The key part of the soundness proof are the lemmas it makes use of, as these establish a) which semantic and typing invariants are required for compositionality, and that b) these invariants hold in initial states and typing environments. We motivate invariants with examples. Proofs of theoretical results presented in this paper can be found in the full version of this paper [33].

### 5.7.1 Monotonicity

The semantics and the type system both satisfy simple monotonicity properties wrt. their respective environments. Let \( L_I \) be the (meet-semi-)lattice given by \( U \sqsubseteq B, B \sqsubseteq 1 \) and \( B \sqsubseteq \bullet \), illustrated in Figure 5.9. This yields a (meet-semi-)lattice \( L_{\equiv} \) of memories, where \( \sigma \sqsubseteq \sigma' \) iff

\[
\forall C. \sigma(C) \sqsubseteq \sigma'(C).
\]

Notice that no rule for establishing \( \langle \sigma, \_ \rangle \Rightarrow \langle \sigma', \_ \rangle \), assigns \( C \) to an \( I \) in \( \sigma' \) such that \( \sigma(C) \nsubseteq I \). This yields the following monotonicity property.

(5.3) **Lemma** (Reduction relation monotone wrt. \( \sigma \)) For all \( \sigma, \sigma' \) and \( t \), if \( \langle \sigma, t \rangle \Rightarrow \langle \sigma', \_ \rangle \), then \( \sigma \sqsubseteq \sigma' \).

We likewise obtain a lattice \( L_\Gamma \) of type environments by defining \( \Gamma \sqsubseteq \Gamma' \) iff

\[
\forall C. \Gamma^s(C) \sqsubseteq \Gamma'^s(C) \land \Gamma^e(C) \sqsubseteq \Gamma'^e(C).
\]

Observe that no rule for establishing \( \_ \vdash \Gamma \{ t \} \Gamma' : \_ \) assigns \( C \) to an \( I \) and \( \ell^e \) in \( \Gamma' \) s.t. \( \Gamma^s(C) \nsubseteq I \) or \( \Gamma^e(C) \nsubseteq \ell^e \). This yields the following monotonicity property, which also helps motivate the definition of \( (\Gamma_1 \circ \Gamma_2)^e \).

(5.4) **Lemma** (Type judgement monotone wrt. \( \Gamma \)) For all \( \Gamma, \Gamma' \), and \( t \), if \( \_ \vdash \Gamma \{ t \} \Gamma' : \_ \), then \( \Gamma \sqsubseteq \Gamma' \).

### 5.7.2 Must-analysis

Recall that part of the type system tracks, at any given point in the control flow of a program, which classes must be initialized. This analysis is crucial if we want to accept programs such as \( P'_{\text{LinH}} \) since, without it, the enforcement would not
know anything about the status of classes possibly initialized in the past, meaning the enforcement would need to regard \( C \) as uninitialized when its reference is seen under \( h \). For this analysis to be sound, the following property must be preserved.

(5.5) **Definition (agreement)** \( \sigma \) agrees with \( \Gamma \), written as \( \Gamma \models_{\text{dep}} \sigma \), iff, for all \( C \),

1) \( \Gamma^s(C) = 1 \implies \sigma(C) = 1 \),

2) \( \sigma(C) = b \iff \Gamma^s(C) = b \).

Pt. 1) states what we expect; if the type environment states that a class is initialized, then that class is initialized in the corresponding memory. Pt. 2) states that a class is being initialized in a run if and only if the analysis tracks that said class is being initialized\(^5\). While this property may be straight-forward, a straight-forward proof that the property is preserved through typing and reduction fails. Consider the case where \( \tau(C) \) is to be evaluated under \( \sigma \), with \( \sigma(C) = 1 \), \( \Gamma^s(C) = u \), \( \Gamma \models_{\text{dep}} \sigma \), and \( \vdash \Gamma \{ \tau(C) \} \Gamma' : \). Here, when \( \langle \sigma, \tau(C) \rangle \Rightarrow \langle \sigma', 1 \rangle \), \( \sigma' = \sigma \). However, \( \Gamma' = \Gamma \) is not guaranteed. This scenario is not artificial or unrealistic, as demonstrated by the following program

\[
\text{try } x := C.x \text{ catch skip;}
\]

\[
x := C.x
\]

run in the presence of the following class table.

\[
\begin{align*}
C \{ x = 1 + D.x \} \\
D \{ x = 5 \}
\end{align*}
\]

In line 2 of this program, for the type environment \( \Gamma \) at this point, \( \Gamma^s(C) = u \), since the must analysis can only guarantee that a class is initialized after a \texttt{try-catch} if said class is initialized in both the \texttt{try} and the \texttt{catch} branch (and the \texttt{catch} branch initializes nothing). However, in the memory \( \sigma \) at this point, \( \sigma(C) = 1 \), since the \texttt{try} branch never fails. In the final type environment \( \Gamma' \), \( \Gamma'^s(D) = 1 \), and a quick review of the operational semantics will show that in the final memory \( \sigma' \), \( \sigma'^s(D) = 1 \), so \( \Gamma' \models_{\text{dep}} \sigma' \). But what is it, in the type system, and in the operational semantics, that makes this so?

The answer: Dependencies. Recall that initialization of a class \( C \) can fail if any of the classes \( C \) depends on fail to initialize, or have failed previously. Thus, if \( C \) is initialized successfully, then all the classes \( C \) depends on must also be successfully initialized, for the remainder of the run (last statement follows from

---

\(^5\)About the \( \iff \): Note that, since \( \Gamma \) overapproximates \( \sigma \), there will be \( \Gamma \)'s for which \( \Gamma^s(C) = b \) for which there is no corresponding \( \sigma \). But this is no problem, as we only need agreement to hold where such a \( \sigma \) exists.
Lemma 5.3). The set $\text{dep}(a)$ of classes $a$ depends on is defined as follows.

$$
\text{dep}(n) = \emptyset \\
\text{dep}(x) = \emptyset \\
\text{dep}(e_1 \oplus e_2) = \text{dep}(e_1) \cup \text{dep}(e_2) \\
\text{dep}(C.x) = \text{dep}(\tau(C)) \\
\text{dep}(C \{i\}) = \text{dep}(i) \cup \{C\} \\
\text{dep}(C <: C' \{i\}) = \text{dep}(C') \cup \text{dep}(i) \cup \{C\} \\
\text{dep}(\epsilon) = \emptyset \\
\text{dep}(x = e; i) = \text{dep}(e) \cup \text{dep}(i)
$$

Here, $\text{dep}(a)$ is the set of classes that must be initialized as a consequence of (successfully) evaluating $a$. Note that in the case of $x = e$ of class $C$, this assignment is always performed as a consequence of $C$ being initialized. Hence $\text{dep}(x)$ is not part of a union on the right-hand side of the definition of $\text{dep}(x = e; i)$. Note that $\text{dep}(\cdot)$ is only defined on expressionals; this is because classes are only initialized when evaluating expressionals.

(5.6) **DEFINITION (dep consistency)** For any function $f$ for which $\text{dom}(\tau) \subseteq \text{dom}(f)$, we say $f$ is $\text{dep}$-consistent, written $\vdash_{\text{dep}} f$, iff

$$
\forall C. f(C) = 1 \implies \forall C' \in \text{dep}(\tau(C)). f(C') = 1.
$$

The type system preserves $\text{dep}$-consistency.

(5.7) **LEMMA (dep-consistency preservation)** For all $\Gamma, \Gamma'$ and $t$,

1. $\vdash_{\text{dep}} \Gamma \{t\} \Gamma': \_,$ and $\vdash_{\text{dep}} \Gamma,$
2. then $\vdash_{\text{dep}} \Gamma'.$

Well-typed programs, run on a memory which agrees with the initial type environment, which terminate successfully, preserve $\text{dep}$-consistency and agreement.

(5.8) **LEMMA (agreement preservation)** For all $\Gamma, \Gamma', t, \sigma$ and $\sigma'$,

1. $\vdash_{\text{dep}} \Gamma \{t\} \Gamma': \_,$ $\vdash_{\text{dep}} \Gamma, \vdash_{\text{dep}} \sigma, \Gamma \models_{\text{dep}} \sigma,$ and
2. then $\vdash_{\text{dep}} \sigma', \langle \sigma, t \rangle \Rightarrow \langle \sigma', R \rangle$
3. $\Gamma' \models_{\text{dep}} \sigma'.$

5.7.3 **Errors**

The main facilitator of the information channel being considered in this paper is evaluation errors, as these can cause a class initialization to fail and thus be used to store information. Cases where errors come into play in our proofs are
therefore of key importance. However, as we have seen in $P_{\text{art}}$, this is not trivial; it is easy to construct memories which “lie”, in the sense that they have registered that a class which can never fail, has failed to initialize. This creates artificial flows which our type system cannot guarantee the absence of. To address this issue, we formalize a sufficient condition for a memory to be “honest”. Such memories are error consistent, that is, a class is only failed in the memory if, during runtime, that class could possibly fail. Since whether a class can fail or not depends on the same for the classes it depends on, this definition of error consistency is inductive in nature. Finally, our operational semantics preserves error consistency. This, together with the observation that initial memories are error consistent, means we only need to consider error consistent memories in our proofs.

(5.9) **Definition (error consistency)** $\sigma$ is error consistent, written $\vdash_{\text{err}} \sigma$, iff

\[
\forall C. \sigma(C) = \cdot \implies \exists \sigma'; (\sigma' \sqsubseteq \sigma), (\vdash_{\text{err}} \sigma'), (\sigma'(C) \neq \cdot), (\sigma', \tau(C)) \Rightarrow (\cdot, \cdot).
\]

(5.10) **Lemma (error consistency preservation)** For all $\sigma, \sigma'$ and $t$,

- if i) $\vdash_{\text{err}} \sigma$, and ii) $(\sigma, t) \Rightarrow (\sigma', \cdot)$,
- then iii) $\vdash_{\text{err}} \sigma'$.

Since our operational semantics preserves $\text{dep}$-consistency and agreement, we furthermore need only consider error consistent memories which are dependency consistent and agree with an appropriate type environment. Some terms evaluations can still yield error under such memories; we call these terms volatile.

(5.11) **Definition (volatile)** $t$ is volatile under $\Gamma$, written $\Gamma \models_{\text{err}} t$, iff there exists a $\sigma$ for which

1) $\vdash_{\text{err}} \sigma, \vdash_{\text{dep}} \sigma, \Gamma \models_{\text{dep}} \sigma$
2) $(\sigma, t) \Rightarrow (\cdot, \cdot)$

Finally we arrive at what we really needed: A well-typed volatile term leaks its context to its error level. Observe that in $P_{\text{art}}$, the context did not leak, since the type system did not encounter a partial operator application. Intuitively, volatile expressions have a partial operator application somewhere along the path the type system takes to analyze $t$. If you examine $(\text{Op-p})$, you will see that this rule raises the error level by $pc$.

(5.12) **Lemma (error leaks pc)** For all $\Gamma, t, pc$ and $\ell'$,

- if i) $pc \vdash \Gamma \{t\} : \ell'$, ii) $\vdash_{\text{dep}} \Gamma$, and iii) $\Gamma \models_{\text{err}} t$,
- then iv) $pc \sqsubseteq \ell'$.

Recall the leak in $P_{\text{main}}$. There, old confidential contextual information, out of scope at the time the “low” assignment occurs, manages to leak into said assignment. When comparing two $\ell$-equivalent memories during parallel runs, we need
a way to know, at the time of the low assignment, that the program, in a previous scope, branched on confidential information, thus causing different class initialization statuses (of classes with no observable fields). The notion of \( \ell \)-consistency gives us this.

\[(5.13) \textbf{definition} (\ell \text{-consistent}) \sigma_1, \sigma_2 \text{ are } \ell \text{-consistent under } \Gamma, \text{ written } \sigma_1 \sim_{\ell} \Gamma \sigma_2, \text{ iff, for all } C,
\sigma_1(C) \neq \sigma_2(C) \land \Gamma \models_{\text{err}} \tau(C) \implies \Gamma^e(C) \not\subseteq \ell.\]

It turns out this relation is an equivalence relation, which will be useful in the next subsection. To link this relation with Lemma 5.12, intuitively, at the point where two runs start disagreeing on the initialization status of a class, the runs start entering a context containing confidential information. This context carries over to the class initialization. If that initialization can fail, then by Lemma 5.12, the error level of the initialization contains confidential information. At last, this error level is recorded in a type environment, and this recording carries over to the join point of the two runs.

5.7.4 Noninterference

We now have enough tools to tackle the type soundness result. Recall that noninterference states that successfully-terminating runs on \( \ell \)-equivalent initial memories yields \( \ell \)-equivalent memories. If all runs on \( \ell \)-equivalent memories followed the same control-flow path in the program, proving soundness would be an easy matter. In reality, however, two \( \ell \)-equivalent runs can take different control-flow paths. It turns out that this only happens when an evaluation of \( \ell \)-unobservables is involved. For instance, for if \( e \) then \( s_1 \) else \( s_2 \), if the evaluation of \( e \) depended only on observables, then \( e \) would evaluate to the same value in the two runs on \( \ell \)-equivalent memories, thus resulting in the same control-flow path taken. We would like the memories at the join point to be \( \ell \)-equivalent, but we cannot compare the intermediate memories of the two branching runs, as that reasoning will not be local. To address this issue, we ensure in our enforcement that no observable effects are allowed when the context contains confidential information. This way, if the memories at the branching point are \( \ell \)-equivalent (and \( \ell \)-consistent), the memories at the join point will be \( \ell \)-equivalent (and \( \ell \)-consistent) by transitivity.

\[(5.14) \textbf{lemma} \text{ For all } t, \sigma, \sigma', \Gamma, \Gamma', \ell \text{ and } \text{pc,}
\begin{align*}
& i) \text{pc} \vdash \Gamma \{ t \} \Gamma', \cdot, \quad ii) \langle \text{r}_{\text{dep}} \Gamma \rangle, \langle \text{r}_{\text{dep}} \sigma \rangle, \langle \Gamma \models_{\text{dep}} \sigma \rangle, \quad iii) \langle \text{r}_{\text{err}} \sigma \rangle, \\
& iv) \langle \sigma, t \rangle \Rightarrow \langle \sigma', \cdot \rangle, \quad \text{and} \quad iv) \text{pc} \not\subseteq \ell, \\
& \text{then} \quad vi) \sigma \sim_{\ell} \sigma', \quad \text{and} \quad vii) \sigma =_{\ell} \sigma'.
\end{align*}\]

At last we get to the main lemma which makes use of all the results we have seen so far. Given the usual assumptions, but for two \( \ell \)-equivalent, \( \ell \)-consistent memories, Lemma 5.15 states that the final memories are \( \ell \)-equivalent and \( \ell \)-
consistent, and the success status of the evaluations differs only if confidential information is present in \( \ell' \).

**Lemma (Main)** For all \( t, \sigma_j, \sigma'_j, \Gamma, \Gamma', \ell, \ell' \) and \( R_j \),

\[
\begin{align*}
&\text{i) } \perp \vdash \Gamma \{ t \} \Gamma': \ell', \\
&\text{ii) } (r_{\text{dep}} \Gamma, (r_{\text{dep}} \sigma_j), (\Gamma \models_{\text{dep}} \sigma_j), \text{ iii) } (r_{\text{err}} \sigma_j), \\
&\text{iv) } (\sigma_j, t) \Rightarrow (\sigma'_j, R_j), \text{ v) } \sigma_1 \models_{\ell} \sigma_2, \text{ and vi) } \sigma_1 =_{\ell} \sigma_2, \\
&\text{then vii) } R_j \neq \bullet = R_j \implies \ell' \not\subseteq \ell, \text{ viii) } \sigma_1 \models_{\ell} \sigma'_2, \text{ and ix) } \sigma_1 =_{\ell} \sigma'_2.
\end{align*}
\]

Take special note of parts i), v) and vii) in the premise, and parts viii) and ix) in the conclusion of Lemma 5.15 — these imply type soundness! We show how to “bootstrap” Lemma 5.15 to obtain a proof of soundness for our type system now.

\( \text{Let } \Gamma_{\text{init}}(C) = \{ \} \text{ and } \Gamma_{\text{init}}^e(C) = \perp \text{ for each class in } \tau. \)

**Theorem (Type Soundness)** For all \( s \), if there exists a \( \Gamma' \) and \( \ell' \) for which \( \perp \vdash \Gamma_{\text{init}} \{ s \} \Gamma' : \ell' \), then \( s \) satisfies TINI.

Let initial \( \sigma_j \) be given such that \( (\sigma_j, s) \Rightarrow (\sigma'_j, R_j) \) and \( \sigma_1 =_{\ell} \sigma_2 \), for any \( \ell \). TINI then requires that \( \sigma'_1 =_{\ell} \sigma'_2 \). We get this by instantiating Lemma 5.15. We have i) and ii) of Lemma 5.15. Let \( \Gamma = \Gamma_{\text{init}}, t = s \) and \( pc = \perp \). This immediately gives us part i) of Lemma 5.15. Since no classes are initialized in \( \Gamma \) or \( \sigma_j \), and none have failed in \( \sigma_j \), parts ii), iii) and v) of Lemma 5.15 hold. This now immediately gives us part viii) of Lemma 5.15 (regardless of whether \( R_j = \text{skip} \) or \( R_j = \bullet \)), which is exactly what we needed. So Theorem 5.16 holds.

### 5.8 Related Work

A survey [34] on language-based information-flow security contains an overview of the area. Most related to ours is work on tracking information flow in object-oriented languages and on information-flow controls in the presence of exceptions.

**Objects:** To the best of our knowledge, the only information-flow mechanism that addresses class initialization is the one implemented by Jif [28, 29], a compiler for Java extended with security types. As discussed earlier, Jif is rather conservative about class initialization code. This code is restricted to simple constant manipulation that may not raise any exceptions. As mentioned earlier, sometimes it is desirable to lift these restrictions.

Much other work has been done on information-flow security for object-oriented languages. Although none of the approaches directly addresses problems with class initialization, we nevertheless discuss some recent highlights.


bytecode by a combination of program transformation and bytecode verification. These two approaches assume fixed security levels for classes. This might not be a flexible choice since it forces all instances and attributes to conform to the class level. Another concern is the scalability of this choice in presence of inheritance.


Hammer and Snelting [21] develop a flow-sensitive, context-sensitive, and object-sensitive framework for controlling information flow by program dependence graphs. This approach takes advantage of similarities of information-flow and slicing analyses.

Exceptions: As noted earlier, our treatment of exception handling draws on standard approaches from the literature (which we extend with the must-analysis). The intuition is if an occurrence of an exception in a statement may carry sensitive information, then there must be no publicly-observable side effects in either the code that handles the exception or in the code between the statement and the exception-handling block. Jif [28, 29] implements such a discipline. Based on a similar discipline, Pottier and Simonet [32] propose a sound treatment of exceptions for ML.


Hedin and Sands [22] prove a noninterference property for a type system that tracks information flow via class-cast and null-pointer exceptions. Askarov and Sabelfeld [4] show how to achieve permissive yet secure exception handling by providing the choice for each type of exception: either the traditional discipline discussed above or by consistently disallowing to catch exceptions. The actual choice for each kind of exception is given to the programmer.

5.9 Conclusion

Seeking to shed light on a largely unexplored area, we have presented considerations for and a formalization of secure class initialization. Our considerations highlight that class initialization poses challenges for security since controlling (the order of) side effects performed by class initialization is challenging. Hence, great care needs to be taken by information-flow enforcement mechanisms to guarantee security. One path, taken by Jif [28, 29], is to severely restrict class
initialization code so that it may only manipulate constants in an exception-free manner. Arguing that it is sometimes too restrictive, we have explored another path: allow powerful initialization code but track its side effects. The enforcement ensures that the side effects do not reveal anything about the differences in control-flow paths that the program might take depending on secret input. Our formalization demonstrates the idea by a type-and-effect system for a simple language that enforces noninterference. To the best of our knowledge, it is the first formal approach to the problem of secure class initialization in the presence of class hierarchies. (Soundness of Jif’s class initialization is yet to be established.)

Acknowledgements: This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR. K. Nakata acknowledges the support of action IC0701 of COST, the Estonian Centre of Excellence in Computer Science, EXCS, financed mainly by ERDF, and the Estonian Science Foundation grant no. 6940.

References


[38] Java 2 platform, standard edition 5.0, API specification. http://java.sun.com/j2se/1.5.0/docs/api/.


