Language-Based Information-Flow Security

Andrei Sabelfeld and Andrew C. Myers

***Abstract*— Current standard security practices do not provide substantial assurance that the end-to-end behavior of a computing system satisfies important security policies such as confidentiality. An end-to-end confidentiality policy might assert that secret input data cannot be inferred by an attacker through the attacker's observations of system output; this policy regulates *information flow*.**

Conventional security mechanisms such as access control and encryption do not directly address the enforcement of information-flow policies. Recently, a promising new approach has been developed: the use of programming-language techniques for specifying and enforcing information-flow policies. In this article we survey the past three decades of research on information-flow security, particularly focusing on work that uses static program analysis to enforce information-flow policies. We give a structured view of recent work in the area and identify some important open challenges.

***Index Terms*— Computer security, confidentiality, information flow, noninterference, security-type systems, covert channels, security policies, concurrency.**

I. Introduction

PROTECTING the confidentiality of information manipulated by computing systems is a long-standing yet increasingly important problem. There is little assurance that current computing systems protect data confidentiality and integrity; existing theoretical frameworks for expressing these security properties are inadequate, and practical techniques for enforcing these properties are unsatisfactory. In this article we discuss language-based techniques—in particular, program semantics and analysis—for the specification and enforcement of security policies for data confidentiality.

Language-based mechanisms are especially interesting because the standard security mechanisms are unsatisfactory for protecting confidential information in the emerging, large networked information systems. Military, medical, and financial information systems, as well as web-based services such as mail, shopping, and business-to-business transactions are applications that create serious privacy questions for which there are no good answers at present.

Analyzing the confidentiality properties of a computing system is difficult even when insecurity arises only from unintentional errors in the design or implementation. Additionally, modern computing systems commonly incorporate untrusted, possibly malicious hosts or code, making assurance of confidentiality still more difficult.

The standard way to protect confidential data is (discretionary) access control: some privilege is required in order to access files or objects containing the confidential data. Access control checks place restrictions on the release of information *but not its propagation*. Once information is released from its container, the accessing program may, through error or malice, improperly transmit the information in some form. It is unrealistic to assume that all the programs in a large computing system are trustworthy; security mechanisms such as signature verification and antivirus scanning do not provide assurance that confidentiality is maintained by the checked program. To ensure that information is used only in accordance with the relevant confidentiality policies, it is necessary to analyze how information flows within the using program; because of the complexity of modern computing systems, a manual analysis is infeasible.

Belief that a system is secure with respect to confidentiality should arise from a rigorous analysis showing that the system as a whole enforces the confidentiality policies of its users. This analysis must show that information controlled by a confidentiality policy cannot flow to a location where that policy is violated. The confidentiality policies we wish to enforce are, thus, *information-flow policies* and the mechanisms that enforce them are *information-flow controls*. Information-flow policies are a natural way to apply the well-known systems principle of end-to-end design [1] to the specification of computer security requirements; therefore, we also consider them to be specifications of *end-to-end security*. In a truly secure system, these confidentiality policies could be precisely expressed and translated into mechanisms that enforce them. However, practical methods for controlling information flow have eluded researchers for some time.

Recently, a promising new approach has been developed by the authors and others: the use of type systems for information flow (e.g., [2]–[14]). In a *security-typed language*, the types of program variables and expressions are augmented with annotations that specify policies on the use of the typed data. These security policies are then enforced by compile-time type checking, and, thus, add little or no run-time overhead. Like ordinary type checking, security-type checking is also inherently compositional: secure subsystems combine to form a larger secure system as long as the external type signatures of the subsystems agree. The recent development of *semantics-based security* models (i.e., models that formalize security in terms of program behavior) has provided powerful reasoning techniques (e.g., [3], [5], [6], [9]–[17]) about the properties that security-type systems guarantee. These properties increase security assurance because they are expressed in terms of end-to-end program behavior and, thus, provide a suitable

Manuscript received April 15, 2002; revised August 21, 2002. This work was supported in part by ONR Grant N00014-01-1-0968, in part by NSF CAREER Award 0133302, and in part by an Alfred P. Sloan Research Fellowship. Any opinions, findings, conclusions, or recommendations contained in this material are those of the authors and do not necessarily reflect the views of the Department of the Navy, Office of Naval Research, the National Science Foundation, or the Alfred P. Sloan Foundation.

A. Sabelfeld and A. C. Myers are with the Computer Science Department, Upson Hall, Cornell University, Ithaca, NY 14853, USA (e-mail: andr{ei, u}@cs.cornell.edu).

vocabulary for end-to-end policies of programs.

The rest of the article is organized as follows. Section II gives some background on the problem of protecting confidentiality, including why existing security techniques are unsatisfactory. Section III illustrates the basics of information-flow techniques by giving examples of a security condition and a security-type system. Section IV gives a structured overview of recent work in the field. Section V identifies important challenges for future work. The article closes in Section VI.

II. Background

Terminology for security properties relating to confidentiality is somewhat inconsistent. This article is about the language-based specification and enforcement of strong confidentiality policies based on information flow. In this framework, it is assumed that computation using confidential information is possible, and that it is important to prevent the results of the computation from leaking even partial information about confidential inputs. This kind of security was described in Lampson's seminal article [18] as information *confinement*, and has also been known as *secrecy*. However, both "confinement" and "secrecy" have been used to describe related but weaker security properties. In the context of capability systems, "confinement" refers to the ability to prevent capabilities (and hence authority) from being transmitted improperly. Similarly, work on cryptographic protocols often builds on the Dolev–Yao model [19], where secret information is assumed to be indivisible and can be leaked only by insertion in its entirety into a message. Finally, "privacy" is sometimes used to refer to the protection of the confidentiality of a principal, but is also sometimes used as a synonym for *anonymity*. For clarity we use the term confidentiality. Unless otherwise stated, the term security refers to confidentiality in this article.

A. Standard Security Mechanisms

Although the difficulty of strongly protecting confidential information has been known for some time, the research addressing this problem has had relatively little impact on the design of commercially available computing systems. These systems employ security mechanisms such as access control, capabilities, firewalls, and antivirus software; it is useful to see how the these standard security mechanisms fall short.

Access control, as embodied in access-control lists [20] and capabilities [21], [22], is an important part of the current security infrastructure. For example, a file may be assigned access-control permissions that prevent users other than its owner from reading the file; more precisely, these permissions prevent processes not authorized by the file owner from reading the file. However, access control does not control how the data is used after it is read from the file. To soundly enforce confidentiality using this access-control policy, it is necessary to grant the file access privilege only to processes that will not improperly transmit or leak the confidential data—but these are precisely the processes that obey a much stronger information-flow policy! Access-control mechanisms cannot identify these processes; therefore, access control, while useful, cannot substitute for information-flow control.

Other common security enforcement mechanisms such as firewalls, encryption, and antivirus software are useful for protecting confidential information. However, these mechanisms do not provide end-to-end security. For example, a firewall protects confidential information by preventing communication with the outside. In practical systems, however, firewalls permit some communication in both directions (e.g., [23]); whether this communication violates confidentiality lies outside the scope of the firewall mechanism. Similarly, encryption can be used to secure an information channel so that only the communicating endpoints have access to the information. However, this encryption provides no assurance that once the data is decrypted, the computation at the receiver respects the confidentiality of the transmitted data. Antivirus software is based on detecting patterns of previously known malicious behavior in code and, thus, offers limited protection against new attacks.

B. Related Work on Language-Based Security

Language-based mechanisms have been used for security goals other than protecting confidentiality. Perhaps the best-known language-based security mechanism is the Java runtime environment, which provides a well-known set of security mechanisms for Java applets, including the bytecode verifier [24], the sandbox model [25], and stack inspection [26]. All three of these mechanisms are language-based—that is, enforced through the Java language—although only the bytecode verifier uses static program analysis. None of these mechanisms is intended to control information flow, and, therefore, they are not effective at protecting confidential data. The bytecode verifier ensures only that applications respect the Java type system, so that object references cannot be forged and private fields cannot be directly accessed. Protection of private fields is important for confidentiality but because it is static, it is less powerful than access-control mechanisms. The sandbox model restricts what classes a Java applet can name, but a malicious applet may violate confidentiality by communicating with the host from which it was downloaded. Stack inspection is a dynamic access-control mechanism; although it helps protect integrity, it does not address confidentiality.

Language-based techniques are also used in other ongoing security research, where the goal is to use type safety to protect the machine against subversion by mobile code (e.g., [24], [27]–[29]), although some more general security policies can be enforced [30]–[33]. However, none of this language-based work addresses end-to-end security policies.

C. Covert Channels

Mechanisms for signaling information through a computing system are known as *channels*. Channels that exploit a mechanism whose primary purpose is not information transfer are called *covert channels* [18]; they pose the greatest challenge in preventing improper information leaks. Covert channels fall into several categories:

- *Implicit flows* signal information through the control structure of a program.

- *Termination channels* signal information through the termination or nontermination of a computation.
- *Timing channels* signal information through the time at which an action occurs rather than through the data associated with the action. The action may be program termination; that is, sensitive information might be obtained from the total execution time of a program.
- *Probabilistic channels* signal information by changing the probability distribution of observable data. These channels are dangerous when the attacker can repeatedly run a computation and observe its stochastic properties.
- *Resource exhaustion channels* signal information by the possible exhaustion of a finite, shared resource, such as memory or disk space.
- *Power channels* embed information in the power consumed by the computer, assuming that the attacker can measure this consumption.

Which covert channels are a concern depends on what attackers can observe of the computing system. For example, power channels are important for smart cards, because they must draw their power from the untrusted terminal into which they are inserted. A program that is secure on an abstract computer with no power requirements might be part of a larger, insecure system when it is run on a real computer. Thus, a computing system can be said to protect confidential information only with respect to a model of what attackers and users are able to observe of its execution.

D. Integrity

Biba [34] first observed that integrity can be treated as a dual to confidentiality, and enforced by controlling information flows. Confidentiality requires that information be prevented from flowing to inappropriate destinations; dually, integrity requires that information be prevented from flowing from inappropriate sources. Integrity has an important difference from confidentiality: a computing system can damage integrity without any interaction with the external world, simply by computing data incorrectly. Thus, strong enforcement of integrity requires proving program correctness, often a daunting obstacle.

E. Mandatory Access Control

Early work on the confidentiality problem, such as that of Fenton [35], [36] and Bell and LaPadula [37], [38], developed *mandatory access control*. In this approach, each data item is labeled with a corresponding *security level* that is a simple confidentiality policy. Information flow is controlled by augmenting the ordinary computation of data within a running program with a simultaneous computation of the corresponding label that controls its future dissemination. This approach, prescribed by the U.S. Department of Defense "orange book" [39] for secure systems, has proved to be too restrictive for general use.

Apart from the obvious computational and storage overhead, the weakness of purely run-time enforcement mechanisms is in identifying *implicit* information flows [40]. Implicit flows arise from the control structure of the program, as opposed to *explicit* flows which are caused by directly passing confidential data to a public variable. For simplicity, let us suppose that there are two sensitivity levels, "*high*" and "*low*," corresponding to greater and lesser confidentiality respectively. Consider the code of Figure 1, which contains a flow from the high variable h to the low variable l. This code is insecure, because it has exactly the same effect as the explicit flow in the assignment $l := h \ \mathsf{mod}\ 2$. Clearly, the insecurity in this code arises from the assignment $l := 1$ in a control context that is conditioned upon the confidential variable h. Mandatory access control can catch this assignment by introducing a process sensitivity label [39] that keeps track of the sensitivity of the data used to control the process. The assignment to l is then detected at run time because a high process updates a low variable. However, consider the case where $h \neq 1$: no assignment to l (and hence, no run-time check) occurs, yet the value of the high variable h can be learned by observing that $l = 0$ holds. In fact, any variable or data structure must be assumed to contain confidential information if it might have been modified within the if statement—or inside any function called from it. Determining which modifications are possible requires evaluating all possible execution paths of the program, which is not feasible at run time.

$$
\begin{array}{l}
h := h \ \mathsf{mod}\ 2;\\
l := 0;\\
\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\\
\qquad\qquad\quad \mathsf{else\ skip}
\end{array}
$$

Fig. 1. An implicit flow.

Confidentiality can be obtained in this example by ensuring that the process sensitivity label remains high throughout the rest of the program, effectively treating all values read from variables as confidential after the if statement completes. In this approach the process label must increase monotonically in sensitivity throughout execution. Any reduction in the sensitivity of the process label (for example, at the return from a procedure [41]), creates a possible security violation through an implicit flow like the one above.

This effect of monotonically increasing labels is known as *label creep*, and is a problem for dynamic enforcement mechanisms. When a variable or field in a data structure is assigned a value of different sensitivity than the one it currently contains, it is necessary to apply the maximum of the two labels to the stored value. Thus, data labels tend to creep upwards as well. Label creep makes dynamic labeling systems too restrictive to be generally useful, because the results of computation tend to be labeled too sensitively for their intended use [41].

F. Static Information-Flow Control

Denning and Denning [40] first observed that static program analysis can also be used to control information flow with increased precision and low run-time overhead. Static characterizations of information flow have been implemented using theorem provers [42], [43]. Information-flow analyses can also be performed by type checking, which is the focus of this article. The type-checking approach has been implemented in the Jif compiler [7], [44].

In the type-checking approach, every program expression has a *security type* with two parts: an ordinary type such as int, and a label that describes how the value may be used. Unlike the labels used by mandatory access-control mechanisms, these labels are completely static: they are not computed at run time. Because of the way that type checking is carried out, a label defines an information-flow policy on the use of the labeled data. Security is enforced by type checking; the compiler reads a program containing labeled types and in type-checking the program, ensures that the program cannot contain improper information flows at run time. The type system in such a language is a *security-type system* that enforces information-flow policies.

The ability to track implicit flows accurately is one of the major advantages of static enforcement of information-flow control. To control implicit flows correctly, we introduce a *program-counter label* (pc) that tracks the dependencies of the program counter. Consider the example of Figure 1 again. In the branches of the if statement, the static label pc is $high$, capturing the dependency of the program counter on h. An assignment is considered secure only if the label on the assigned variable is at least as restrictive as pc. The second assignment to l (whose label is low) occurs in a control context where pc is $high$, which justifies the rejection of the program as insecure.

The program-counter label corresponds to the dynamic process sensitivity label used in mandatory access control, but there is an important difference: the statement following the if statement, if any, can be assigned the same static pc label as the statement preceding it. This assignment is secure because arriving at the following statement tells us nothing about h: the statement is executed regardless of the value of h.

It may seem surprising that static checking can improve accuracy. The reason is that dynamic enforcement only has information about a single program execution, but compile-time type-checking can prove that no possible execution paths within the if statement contain insecure assignments. This is a consequence of the general fact that confidentiality is not a property of a single execution path, but rather a property of the set of all executable paths [45], [46].

Notice that if a branch of the if statement of Figure 1 did not terminate, an attacker might infer the value of h from the termination or nontermination of the program. This is an example of a termination covert channel. We discuss this and other covert channels in Section IV-C.

G. Noninterference

If a user wishes to keep some data confidential, he or she might state a policy stipulating that no data visible to other users is affected by confidential data. This policy allows programs to manipulate and modify private data, so long as visible outputs of those programs do not improperly reveal information about the data. A policy of this sort is a *noninterference policy* [47], because it states that confidential data may not interfere with (affect) public data.

$$\begin{aligned} C ::= {} & \mathsf{skip} \mid var := exp \mid C_1; C_2 \\ & \mid \mathsf{if}\ exp\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \mathsf{while}\ exp\ \mathsf{do}\ C \end{aligned}$$

Fig. 2. Command syntax.

An attacker (or unauthorized user) is assumed to be allowed to view information that is not confidential (that is public). The usual method for showing that noninterference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [48]. Noninterference can be naturally expressed by *semantic models* of program execution. This idea goes back to Cohen's early work on *strong dependency* [49], [50]. McLean [51] argues for noninterference for programs in the context of trace semantics. However, neither work suggests an automatic security enforcement mechanism.

III. Basics of Language-Based Information Flow

Recently, techniques for proving that a security-type system enforces noninterference have been developed for increasingly complex programming languages. In this section, we introduce this approach by exploring a simple security-type system that enforces a noninterference security policy.

Consider the simple imperative language with the syntax given in Figure 2. This language has skip, assignment, sequential composition, conditional, and while-loop constructs. As before, we write h and l for typical variables of *high* and *low* confidentiality, respectively. We assume that expressions exp are formed by applying total arithmetic operations to constants and variables.

A. Semantics-Based Security

Noninterference for programs essentially means that *a variation of confidential (high) input does not cause a variation of public (low) output*. This intuition can be rigorously formalized using the machinery of programming-language semantics. We assume that computation starts in an input state $s = (s_h, s_l)$, a pair consisting of the initial values of h and l, respectively. The program either terminates in an output state $s' = (s'_h, s'_l)$ with output values for the high and low variables, or diverges. Thus, the semantics $[\![C]\!]$ of a program (command) C is a function $[\![C]\!] : S \to S_\perp$ (where $S_\perp = S \cup \{\perp\}$ and $\perp \notin S$) that maps an input state $s \in S$ either to an output state $[\![C]\!]s \in S$, or to $\perp$ if the program fails to terminate. The variation of high input can be described as an *equivalence relation* $=_L$; two inputs are equivalent whenever they agree on low values (i.e., $s =_L s'$ iff $s_l = s'_l$). The observational power of an attacker can be characterized by a relation $\approx_L$ on behaviors (as defined by the semantics) such that two behaviors are related by $\approx_L$ iff they are indistinguishable to the attacker. Relation $\approx_L$ is said to reflect the *low view* of the system. It is often an equivalence relation but is at least symmetric and reflexive. For a given semantic model, noninterference is formalized as follows: C is secure iff

$$\forall s_1, s_2 \in S.\, s_1 =_L s_2 \Longrightarrow [\![C]\!]s_1 \approx_L [\![C]\!]s_2 \qquad (*)$$

$$[\text{E1–2}] \quad \vdash \mathit{exp} : \mathit{high} \qquad \frac{h \notin \mathit{Vars}(\mathit{exp})}{\vdash \mathit{exp} : \mathit{low}}$$

$$[\text{C1–3}] \quad [\mathit{pc}] \vdash \mathsf{skip} \qquad [\mathit{pc}] \vdash h := \mathit{exp} \qquad \frac{\vdash \mathit{exp} : \mathit{low}}{[\mathit{low}] \vdash l := \mathit{exp}}$$

$$[\text{C4–5}] \quad \frac{[\mathit{pc}] \vdash C_1 \quad [\mathit{pc}] \vdash C_2}{[\mathit{pc}] \vdash C_1; C_2} \qquad \frac{\vdash \mathit{exp} : \mathit{pc} \qquad [\mathit{pc}] \vdash C}{[\mathit{pc}] \vdash \mathsf{while}\ \mathit{exp}\ \mathsf{do}\ C}$$

$$[\text{C6–7}] \quad \frac{\vdash \mathit{exp} : \mathit{pc} \quad [\mathit{pc}] \vdash C_1 \quad [\mathit{pc}] \vdash C_2}{[\mathit{pc}] \vdash \mathsf{if}\ \mathit{exp}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2} \qquad \frac{[\mathit{high}] \vdash C}{[\mathit{low}] \vdash C}$$

Fig. 3. Security-type system.

which reads "if two input states share the same low values, then the behaviors of the program executed on these states are indistinguishable by the attacker." The particular model of the observable behavior depends on the desired security property. For example, in our language we may set $s \approx_L s'$ iff $s, s' \in S$ implies $s =_L s'$. Under this assumption, condition $(*)$ corresponds to the absence of *strong dependency* [49], [50] of the variable h on the variable l. According to such a condition, the program $h := l{+}4$ is secure because the low output (which is the same as low input) is unaffected by changes to the high input. The program ($\mathsf{if}\ l = 5\ \mathsf{then}\ h := h + 1\ \mathsf{else}\ l := l + 1$) is also secure because the final values of l only depend on the initial value of l. However, the programs $l := h$ and ($\mathsf{if}\ h = 3\ \mathsf{then}\ l := 5\ \mathsf{else\ skip}$) are clearly insecure: for example, taking 2 and 3 as initial high values gives different final values for l. For the former program with 0 as the initial value of l we have $(2, 0) =_L (3, 0)$, but $[\![l := h]\!](2, 0) = (2, 2) \not\approx_L (3, 3) = [\![l := h]\!](3, 0)$.

B. A Security-Type System

A security-type system is a collection of *typing rules* that describe what *security type* is assigned to a program (or expression), based on the types of subprograms (subexpressions). We write $\vdash \mathit{exp} : \tau$ to mean that the expression exp has type τ according to the typing rules. This assertion is known as a *typing judgment*. Similarly, the judgment $[\mathit{pc}] \vdash C$ means that the program C is *typable* in the *security context* pc. For our purposes, the security context is just the program counter label pc (cf. Section II-F).

Figure 3 presents the typing rules for our simple language. (This system is, in fact, equivalent to a type system by Volpano et al. [3].) Expression types and security contexts can be either high or low. According to the rules [E1–2], any expression (including constants, the variable h, and even l) can have type high; however, an expression can have type low only if it has no occurrences of h.

Consider the rules [C1–7]. The commands skip and $h := \mathit{exp}$ are typable in any context. The command $l := \mathit{exp}$ is only typable if exp has type low. This prevents explicit flows. Notice that $l := \mathit{exp}$ is only typable in the low context which, in combination with the rest of the (purely compositional) rules, disallows implicit flows. Indeed, notice that $[\mathit{high}] \vdash C$ for some command C ensures that there are no assignments to low variables in C. This justifies the requirement of the rules [C5–6] that given an if (loop) with a high guard, the branches (loop body) must be typable in a high context. Let us refer to loops with a high guard as *high loops*, and to conditionals with a high condition as *high conditionals*. The rule [C7] is a *subsumption* rule. It guarantees that if a program is typable in a high context then it is also typable in a low context. This rule allows us to reset the program counter to low after a high conditional or a loop, avoiding one source of label creep (cf. Sections II-E and II-F).

Examples of typed programs are $[\mathit{low}] \vdash h := l{+}4; l := l{-}5$ and $[\mathit{high}] \vdash \mathsf{if}\ h = 1\ \mathsf{then}\ h := h + 4\ \mathsf{else\ skip}$. As expected, the example programs with explicit and implicit insecure flows $l := h$ and $\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else\ skip}$ are not typable.

IV. Trends in Language-Based Information Flow

We have considered succinct examples of how to express a noninterference-style security policy using low-view relations and a static certification-style security analysis using a type system. However, the true value of these two representations lies in their connection. Indeed, the ultimate goal for formalizing confidentiality properties is to have tools that are not only expressive but also ensure rigorous end-to-end security policies. While work on information flow prior to the mid-nineties typically handled either policies or analyses in separation, Volpano et al. [3] were the first to establish an explicit connection. They cast a Denning-style analysis as a type system similar to Figure 3 and showed that if a program is typable then it is secure according to condition $(*)$. This result improves security assurance because it is based on an *extensional* security definition. Such a definition is expressed by the low view under a standard semantic model as opposed to the ad-hoc security semantics (or no security semantics at all) underlying previous approaches (e.g., [2], [52]–[61]).

We identify four directions of research in language-based security that branch off from this meeting point of noninterference and static certification: (i) enriching the *expressiveness* of the underlying programming language, (ii) exploring the impact of *concurrency* on security, (iii) analyzing *covert channels*, and (iv) refining *security policies*. In the rest of this section, we sketch recent work in these directions. The diagram in Figure 4 illustrates the structure of current work and is intended to serve as a road map for the interested reader to follow the evolution described (recommended references are provided in the diagram).

A. Language Expressiveness

A major line of research in information flow pursues the goal of accommodating the increased expressiveness of modern programming languages. We concentrate on progress on procedures, functions, exceptions, and objects.

Procedures: Volpano and Smith [63] give a type system for a language with first-order procedures and prove that it guarantees noninterference. The type system relies heavily on *polymorphism*, a well-studied concept in type systems. Polymorphism means that the type of commands or expressions may be generic, i.e., may depend on the context. For example, procedures may have polymorphic security types so that their

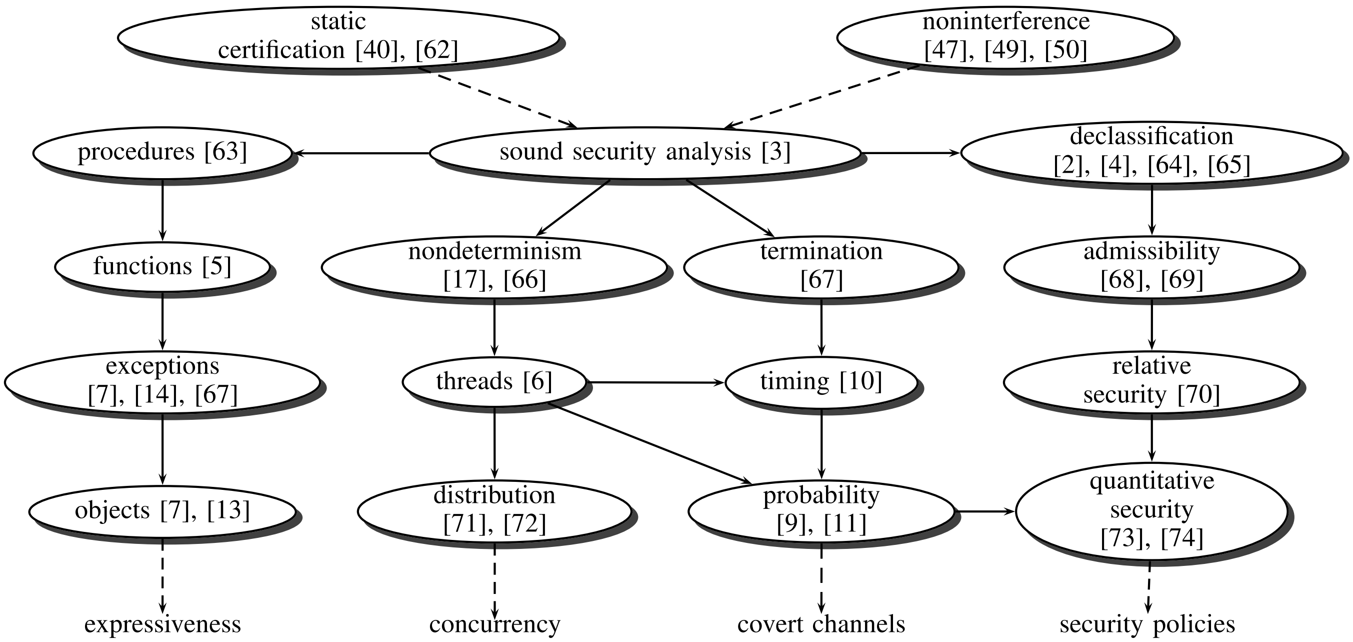

Fig. 4. Evolution of language-based information flow.

invocation can be adjusted to either low or high context. (Recall that a high context corresponds to a high program counter, i.e., a program point inside a high conditional or loop.) Notice that our example type system in Figure 3 also has simple polymorphic rules in pc, e.g., $h := h + 3$ can either be typed by $[high] \vdash h := h + 3$ or $[low] \vdash h := h + 3$ depending on the context.

Functions: Heintze and Riecke [5] consider information flow in the *SLam calculus*, a functional language, based on the λ-calculus [75], that supports first-class functions (functions usable as ordinary values). They propose a type system for confidentiality and integrity in SLam and prove a noninterference theorem using logical relations. They also extend SLam and the type system with state and concurrency, but do not prove noninterference.

Zdancewic and Myers [12], [76] define a secure calculus that has first-class continuations, state, and references, and prove that its type system enforces noninterference. Continuations are a more expressive control construct than functions. They also show that a fragment of SLam, augmented with state and references, can be encoded in their continuation-passing style calculus without any loss of precision.

Pottier and Conchon [16] show a systematic way to extend conventional type systems with flow control in a functional setting. This approach simplifies correctness proofs by allowing them to be extracted from the correctness proofs for the original type system. Pottier and Simonet [14], [77] prove noninterference for an extension of the purely functional part of SLam, extended with references, exceptions, and type polymorphism.

Exceptions: Exceptions (such as in Java) can be raised by the language in an event of a run-time error, such as division by 0. They may also be used by the programmer to signal exceptional conditions. Exceptions can be caught by a special language construct, resulting in a nonlocal transfer of control that can create implicit flows. If not caught, an exception may also create an implicit flow of information. Volpano and Smith [67] propose a simple but restrictive type system for handling exceptions. Myers [7] argues that this may lead to loss of precision in the analysis and introduces *path labels* that allow finer-grained tracking of the implicit flows caused by exceptions. Pottier and Simonet [14], [77], [78] suggest a similarly fine-grained analysis of exceptions and also give noninterference proofs for a functional setting.

Objects: Objects are another important language feature; they subsume first-class functions because first-class functions can be encoded as objects. The JFlow language [7] extends Java with a type system for tracking information flow. This language has also been implemented in the Jif compiler [44]. Barthe and Serpette [8] consider information flow in a simple object-oriented language based on the Abadi–Cardelli functional object calculi [79] and show that their type system enforces noninterference. Banerjee and Naumann [13] develop a security-type system for a Java-like imperative object-oriented language and show that it enforces noninterference.

B. Concurrency

In principle, concurrency could be considered one of the language extensions in the previous section. However, the nature of concurrent computation raises new concerns about the low-view model, making concurrency a major topic of its own.

Nondeterminism: A first step toward concurrency is nondeterministic computation. Noninterference as originally defined was a property of deterministic computations. A straightforward way to generalize it to nondeterminism is to consider the observable behavior of a program to be the *set* of its possible results. With this interpretation, the security condition $(*)$ of Section III-A means that high inputs may not affect the set of possible low outputs. This is known as a *possibilistic* security condition [80]. There is a substantial body of work on possibilistic generalizations of noninterference for a nondeterministic setting (e.g., [80]–[83]).

In the context of programming languages, Banâtre, Bryce, and Le Métayer [57] suggest an analysis that tracks dependencies between variables for a language with a nondeterministic choice operator.

Leino and Joshi [66], [84] define an elegant *equational* security property for nondeterministic programs. Define a program HH ("havoc on h") to have the (informal) semantics "set h to an arbitrary value". Now, a program C is considered secure iff

$$\forall s \in S.\, [\![HH; C; HH]\!]s \approx [\![C; HH]\!]s$$

for an appropriate equivalence relation $\approx$ on *sets of final values*. The occurrence of HH after C, in effect, equalizes the set of possible final values of h. This equation has the intuitive reading that scrambling the initial value of h does not reflect on the set of final values of l; thus, it is a possibilistic security condition. Among advantages of this approach is the flexibility in the choice of $\approx$ and verification conditions [66], [84] for proving equational security.

Sabelfeld and Sands [17], [85] formalize a number of security specifications by *partial equivalence relations* (PERs) of which the equational security condition above is an instance. Abadi et al. [15] were the first to adapt PERs from program analysis to reason about variations in the spirit of the low view $\approx_L$ in the deterministic setting. The extension of PERs to handle nondeterministic security [17], [85] develops a link between low-view relations $\approx_L$ and equivalence relations $\approx$ for programs that exhibit nondeterminism.

Thread concurrency: Consider multithreaded programs executed on a single processor. One complication with concurrent models is that the high part of program states has to be protected at all times during computation. For example, we might consider the program (thread) $h := 0; l := h$ secure because the initial secret value has been overridden by constant 0. However, security can be compromised in case another (secure) program runs in parallel. This other program may update the value of h with a secret (e.g., $h := h'$ for some high variable h') immediately before $l := h$ is executed by the first thread.

Another issue is that the security of multithreaded computation is tightly connected with *timing*- and *probability*-sensitive security. Indeed, assuming that the scheduler that determines what thread is selected at the next step exhibits (potentially probabilistic) behavior, this behavior is reflected on the choice of what thread is executed. Thus, the execution order of low-level computation may be affected. We will focus on timing- and probability-sensitive security in Section IV-C.

In contrast to earlier work on security for concurrent programs [52], [53], [56], [58], Smith and Volpano [6] prove noninterference for a multithreaded language. They show that two requirements—imposed in addition to those enforced for sequential languages—are sufficient for noninterference under a purely nondeterministic scheduler: no while loop may have a high guard, and no high conditional may contain a while loop in its branch.

However, some programs that this analysis determines to be secure—and that meet the possibilistic security condition—may be insecure in practice. For example, the program

$$(\mathsf{if}\ h = 1\ \mathsf{then}\ C_{long}\ \mathsf{else\ skip});\ l := 1\ \ \|\ \ l := 0$$

(where $\|$ denotes the parallel composition and C_{long} is a time-consuming series of skip commands) is considered secure. However, under many schedulers (such as round-robin), if h is 1 then the last subcommand to be executed is likely to be $l := 1$. This is an encoding of a *timing leak* into a direct leak (by assignments, sequential and parallel composition). Although the set of possible results of the program is independent of h assuming a nondeterministic scheduler, some *refinements* of the program, in which possible outcomes are eliminated by choosing a scheduler, are not secure. In a concurrent setting, possibilistic security is often subject to such *refinement attacks*.

Later work by Volpano and Smith [9], [86] investigates security in the presence of the *uniform* scheduler (such a scheduler selects a thread from the pool of live threads with the uniform probability distribution) and gives a probability-sensitive security specification that rejects the program above as insecure. Because scheduling policies may vary from implementation to implementation, Sabelfeld and Sands [11] argue for *scheduler-independent* security (robust with respect to a wide class of potentially probabilistic schedulers) and prove a noninterference result for a security-type system.

Smith [87] and, independently, Boudol and Castellani [88], [89] observe that a high while-loop can be considered secure in a concurrent setting, provided that there are no low assignments that follow the loop. This is enforced by respective type systems. Smith's type system guarantees probabilistic noninterference. Boudol and Castellani's typing rules are extended to schedulers with the result that any typed system (consisting of threads and a scheduler) must satisfy possibilistic noninterference.

Sabelfeld [90] extends both the definition of noninterference and a type system to handle thread *synchronization*. The effect of synchronization is similar to that of loops: the type system rules out synchronization that depends on high data in order to ensure noninterference.

Honda et al. [91], [92] have taken another approach to secure concurrent languages: security-type systems for the asynchronous π-calculus, a general model of concurrent computation in which threads are implicit. Channel types in these languages may be annotated with a number of attributes that describe possible communication patterns; notably, channel types may be *linear* or *affine*, meaning that the channel may be used for exactly one or at most one message, respectively. These complex security-type systems are able to enforce noninterference with about the same precision as the other type systems for concurrent languages.

Pottier [93] presents a syntactic technique that extends the π-calculus to a calculus of pairs of processes. Noninterference is reduced to a safety property for such a calculus, and this property is established by *subject reduction*, which is a standard technique for showing the soundness of type systems. This technique is also used in noninterference results for functional languages [14], [77], [78], [93].

Zdancewic [94] develops an alternate approach to checking security for concurrent systems based on the idea of *low-view determinism* developed by Roscoe [95] in the context of CSP. A program is considered to be secure only if its results, viewed through the low-view relation $\approx_L$, are deterministic despite considering high inputs as chosen nondeterministically. This security condition generalizes noninterference, but is not possibilistic and is not subject to refinement attacks. However, it may rule out useful nondeterminism. Zdancewic gives a security-type system for a concurrent language that supports an arbitrary number of processes and message-passing communication (including first-class channels); he shows that this type system, when combined with a suitable alias analysis, enforces the security condition.

Distribution: Understanding security in the distributed setting is one of the most pressing needs. Distributed systems are naturally concurrent, but three new issues are raised in this setting: first, distributed programs must have the ability to exchange messages, and these communications may be observed to some extent by attackers. Second, systems are often distributed precisely because they serve the needs of multiple principals who do not fully trust each other. A way is needed to provide security despite this mutual distrust. Third, distributed systems have components (such as host machines) that can fail; these failures may include the complete subversion of these components by malicious attackers. A subverted host may continue to simulate proper functioning, but improperly release data it receives; it may also attempt to compromise the behavior of other components of the system to cause them to violate confidentiality.

Early confidentiality definitions for language-based distributed systems were phrased in terms of a *security logic* (which, however, lacked a rigorous relation to program semantics). Reitman's logic [96] addresses the security of message-passing primitives commonly used in distributed programs. Banâtre and Bryce's logic [56], [58] formalizes security properties for a language with synchronous message passing.

Sabelfeld and Mantel [72] investigate the security implications of various communication primitives. They consider blocking vs. nonblocking and synchronous vs. asynchronous primitives for message passing and propose a type system that enforces timing-sensitive noninterference for a language that features both multithreading and message passing.

Relatively little work has addressed the problem of information flow in a distributed system that incorporates mutual distrust and arbitrary component failure. Zdancewic et al. [71] propose and implement an architecture called *secure program partitioning* to address these issues. The goal of this architecture is to protect the confidentiality of multiple principals who do not trust the other principals and have only partial (and differing) trust in the host machines available for computation. In this setting, the informal security condition is that the security of a principal is not threatened unless there is a failure of a host that the principal trusts. A sequential, security-typed program is automatically partitioned in a fine-grained manner into communicating subprograms that run securely on the available hosts and carry out the original computation. The security types in the language can specify both confidentiality and integrity policies; the latter are used to prevent untrusted hosts from subverting security-critical decisions. The system is intended to allow enforcement of end-to-end security policies that go beyond noninterference, but there is no proof that the system enforces a noninterference-like property. A theorem about the integrity of control flow in the partitioned program is proved, however.

Language-based techniques are useful in modeling and analyzing information flow in security protocols. Abadi [64], [97] shows how confidentiality can be achieved by typing in distributed security protocols in the presence of shared-key encryption. A recent work by Abadi and Blanchet [98] deals with types for protocols involving public-key encryption. Sumii and Pierce's work on protocols [99] employs logical relations for reasoning about the low view of systems in the presence of encryption.

C. Covert Channels

Recall from Section II-C that covert channels are ways to transfer information within a computing system using mechanisms not intended to function as information channels. For example, many implicit flows are examples of covert storage channels. There are several kinds of covert channels that are more difficult to identify: for example, information flows resulting from dependencies between sensitive data and observable behavior of the system such as timing or the system's stochastic behavior. Thus, the assumption that the attacker is capable of such observations must be reflected in the low view of the system.

Termination channels: Assuming that an attacker can observe program termination (or nontermination), program while $h = 1$ do skip is no longer secure. Volpano and Smith [67] argue for *termination-sensitive* noninterference. Such noninterference can be expressed as condition $(*)$ with the low-view relation $\approx_L$ that relates two behaviors iff both diverge or both terminate in low-equal final states ($s \approx_L s'$ iff either $s, s' \in S$ and $s =_L s'$ or $s = s' = \perp$). In order to prevent termination attacks, the type system of [67] disallows high loops and requires high conditionals have no loops in the branches.

Abadi et al. [15] establish a connection between security analysis and three types of program analyses: *binding-time analysis*, *call tracking*, and *program slicing*. These analyses are related because the properties they identify are all dependency properties. Abadi et al. express this dependency in terms of PERs for a calculus based on a variation of the λ-calculus. They show that PERs capture termination-sensitive security.

Binding-time analysis is particularly close to security analysis. In the field of *partial evaluation*, binding-time analysis divides program terms into *static* (known at partial-evaluation time) and *dynamic* (to be supplied later). The correctness condition for binding-time analysis states that no static term depends on a dynamic variable. Viewing dynamic as high and static as low we obtain the connection to security. The connection with partial evaluation has been explored by Sabelfeld and Sands [17], [85], Barthe and Serpette [8], and Thiemann [100].

Timing channels: In practice, nontermination cannot be distinguished from a very time-consuming computation. Thus, the termination channel can be viewed as an instance of the *timing channel*. Timing channels can present a serious threat (see [101] for a timing attack on RSA encryption). As we have seen in Section IV-B, timing channels are particularly dangerous in the presence of concurrent threads, as they may result in information leaks. *Timing-sensitive* noninterference is formalized by condition $(*)$ with the low-view relation $\approx_L$ that relates two behaviors iff both diverge or both terminate in the same number of execution steps in low-equal final states. A high conditional may generate a timing leak, e.g., $\mathsf{if}\ h = 1\ \mathsf{then}\ C_{long}\ \mathsf{else\ skip}$ (cf. Section IV-B). Volpano and Smith [9] suggest restricting high conditionals to have no loops in the branches and wrapping each high conditional in a $\mathsf{protect}$ statement whose execution is atomic. This discipline is enforced by an accompanying type system.

Agat's approach [10] to closing timing leaks is based on another example of a well-studied technique in programming languages, *program transformation*. The transformation can be represented as a type system with type assignments of the following form: $C \hookrightarrow C' : Sl$ where C is the original program, C' is the result of the transformation, and Sl is the *low slice* of C'. The low slice Sl is different from C' in that subcommands of C' that involve high data are replaced by dummy commands with no effect on high variables. This ensures $Sl \approx_L C'$. Either the original program C is rejected (in case of a potential explicit or implicit insecure information flow) or accepted and transformed into program C' free of timing leaks. The core rule of the type-and-transformation system is:

$$\frac{C_1 \hookrightarrow C_1' : Sl_1 \qquad C_2 \hookrightarrow C_2' : Sl_2 \qquad exp : high \qquad al(Sl_1) = al(Sl_2) = \mathsf{false}}{\mathsf{if}\ exp\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \hookrightarrow \mathsf{if}\ exp\ \mathsf{then}\ C_1'; Sl_2\ \mathsf{else}\ Sl_1; C_2' : \mathsf{if\text{-}skip}(Sl_1; Sl_2)}$$

where the predicate $al(C)$ is true iff there is an assignment to a low variable in a command C; and $\mathsf{if\text{-}skip}(C)$ is a command that acts as an if timing-wise except that there is only one possible branch C. This rule performs cross-copying of the slices of the branches of a high if to equalize the execution time of the branches. This approach has been adapted for transforming out timing leaks in languages with concurrency by Sabelfeld and Sands [11], [90] and distribution by Sabelfeld and Mantel [72].

Probabilistic channels: The low view $\approx_L$ can be enhanced with the probabilistic property that two behaviors are indistinguishable by the attacker iff the distribution of low output is the same. This formalization captures *probabilistic leaks*. Let us illustrate a probabilistic leak by an adaptation of an example from [102]. Assuming a high variable PIN stores a four-digit number, consider the following (intuitively insecure) program:

$$l := PIN \ [\!]_{\frac{9}{10}}\ l := rand(9999)$$

where $[\!]_p$ is a *probabilistic choice* operator that selects the left-hand side command with the probability p and the right-hand side with the probability $1 - p$; and $rand(n)$ is a function returning a random value from the range $0 \dots n$ according to the uniform distribution. According to purely possibilistic conditions (e.g., [6], [66]) the program above is secure. Indeed, varying PIN does not change the set of possible outcomes for l. However, it does change the *probability* of outcomes for l, which is reflected by probability-sensitive definitions. Such definitions in non-language-specific settings include McLean's *flow model* [102] and *probabilistic noninterference* [103], [104].

Sabelfeld and Sands [17], [85] lift the PER model to *probabilistic powerdomains* to characterize probabilistic noninterference. An important contribution of this work is the *compositionality* result that guarantees that if secure programs are plugged into an appropriate context the resulting program is secure. Due to this property the correctness proofs for security-type systems (which are also compositional) become straightforward, which is exemplified in [17].

As we observed in Section IV-B, a scheduler in a concurrent setting may be probabilistic. Volpano and Smith's type system [9], [86] captures probabilistic flows resulting from the uniform scheduler in a multithreaded language.

Sabelfeld and Sands [11] connect probabilistic security with *probabilistic bisimulation* [105] which is a standard semantic model for probabilistic computation. As a benefit of this connection, their security condition improves the precision of previous probability-sensitive definitions (e.g., [9]). They extend the language with dynamic thread creation and prove scheduler-independent timing-sensitive noninterference for a security analysis. Correctness proofs are accommodated by a compositional security definition that implies noninterference.

D. Security Policies

While a useful extensional property, noninterference imposes restrictions that are not always possible to meet in practice. In particular, noninterference rejects *downgrading* of the security level of information from high to low. However, such a declassification is necessary in order to allow a secret value that has been encrypted to be passed over a publicly observable medium. Another example is a password-checking program. Clearly, the result of this program depends on whether the secret password matches the publicly supplied data. Attempts to accommodate downgrading in end-to-end policies have been an active area of research.

Myers and Liskov [4], [106] introduce a *decentralized model* of security labels in which *selective declassification* [16] is permitted on the basis of a static analysis of process authority and relationships between principals. Security labels have additional structure that describes the entities capable of performing declassification. This model supports the labeling of computations performed on behalf of mutually distrusting principals.

Cryptographic protocols depend on encryption, and, thus, downgrading of information security levels is a necessity. Abadi [64], [97] devises type systems that guarantee confidentiality for a calculus of cryptographic protocols, the spi calculus [107]. Secret keys and their usage are hidden by the low view in an extensional security condition. Abadi and Blanchet's type system [98] analyzes Dolev–Yao secrecy for the spi calculus.

Dam and Giambiagi's *admissibility* [68] for JVM applets is a weakening of noninterference. They analyze an online payment protocol which involves encryption of secrets, and show that it has the desired admissibility property. Admissibility corresponds to the security policy that explicitly states what dependencies between data are allowed in a program (including those caused by downgrading). Giambiagi's subsequent work [69], [108] separates protocol specification from its implementation. The proposed security condition guarantees that an admissible program has no other information flows than those intended by the protocol specification (and explicitly recorded in a confidentiality policy). The security condition is termination and timing sensitive (agreeing with noninterference modulo flows allowed by the protocol specification). This approach is realized for a guarded-command implementation language that includes encryption, decryption, iteration, message-passing and exception constructs [69].

Systems containing intentional downgrading channels introduce the possibility that these channels will be exploited to downgrade more information than was intended by the programmer. Zdancewic and Myers [65] propose *robust declassification*, a security property that prevents exploitation of the channels. Robust declassification says that an active attacker (who can affect system behavior) cannot learn anything more than a passive attacker (who may only observe the system's behavior). The presumption is that information flows visible to the passive attacker are intended to be present.

Volpano and Smith [70] concentrate on the scenario of a password-checking program. They provide a type system that allows for operations similar to password queries and give a security assurance based on probabilistic complexity: first, no well-typed program can leak secrets in polynomial (in the length of the secret) time; and, second, secret leaks are only possible with a negligible probability. Continuing this line of work, Volpano [109] proves that leaking passwords in a system where passwords are stored as images of a one-way function is not easier than breaking the one-way function.

Laud's complexity-theoretic security specification [110] relies on *computational indistinguishability* by a polynomial-time (again in the length of the secret) adversary in an imperative language. The specification is accompanied by a sound analysis that, e.g., accepts the program $l := enc_k(h)$, i.e., encryption of h with a key k.

It is often useful to allow for a limited bandwidth of information leaks. For example, one may consider the program that queries a four-digit number and matches it to a secret PIN to be secure if the probability of leakage ($\frac{1}{10000}$ in this case) is less than a threshold value ϵ. Early ideas of *quantitative* security (as opposed to *qualitative*) go back to Denning's work [41] which, however, does not provide automated tools for estimating the bandwidth. Clark et al. [73] propose syntax-directed inference rules that aid in computing estimates on information flow resulted from if statements in an imperative language.

Di Pierro et al. [74] suggest *approximate noninterference*, which can be thought of as noninterference "up to ϵ." They also provide a probability-sensitive program analyses that ensures precise [111] and approximate [74] noninterference for a probabilistic constraint-programming calculus. An extension of this work [112] is concerned with a sound analysis geared toward the attacker that is able to make external observations about the system, such as the average state over a limited number of steps.

Lowe's quantitative definition of information flow [113] is intended to measure the capacity of covert channels in a process-algebra setting. Like other quantitative definitions, this definition is based on Shannon's information theory [114]. However, unlike other models, Lowe's definition is sensitive to nondeterminism. The amount of leaked information is based on the number of different behaviors of a high-level user that can be distinguished by a low-level user.

V. Open Challenges

This section discusses some challenges for language-based security researchers. Some challenges are natural goals emerging from the existing directions described in Section IV; others have been investigated less but are nonetheless crucial for practical enforcement of end-to-end security policies.

A. System-Wide Security

Computer systems are only as secure as their weakest point, so a *system-wide security* model is essential to guarantee that not only the system components are secure but also their combination. A challenging direction here is the integration of language-based information flow and system-wide information-flow control. The secure program partitioning approach of Zdancewic et al. [71] directly addresses this notion of end-to-end, system-wide security. It prevents attacks on a distributed system considered as a whole, and explicitly models distrust between principals and hosts. Mantel and Sabelfeld [115], [116] also make a step in this direction, proposing a way to integrate language-based confidentiality properties of local computation into an abstract framework of global properties. This link at the end-to-end level facilitates a modular end-to-end system design. Rigorous connections to areas such as security protocols and trust management are most desirable.

B. Certifying Compilation

One potential weakness of using a compiler to validate information flows is that it places both the type checker and the code generator of the compiler in the *trusted computed base* (TCB) [117]. It is clearly desirable to perform information-flow analysis on code that is as close to the executed code as possible, avoiding these trust dependencies. This is also important because much malicious code is distributed in the form of programs in a low-level machine language (not to be confused with the low level of confidentiality for data) such as Java applets or ActiveX components. *Certifying compilation* [118] is an attractive way to generate machine code that is annotated with the necessary information to perform static validation. Java bytecode verification [24] and typed assembly language [28] (primarily used to guarantee memory safety) are examples of this approach.

Low-level languages have not received much attention in studies of secure information flow. One difficulty with checking information flow in low-level languages is that useful information about program structure is lost during compilation. Consequently, typical source-language techniques do not generalize straightforwardly [12], [76], [119]. Zdancewic and Myers [12], [76] present a type system that ensures noninterference in low-level programs in which the only control construct is continuations (which correspond to indirect branches at the machine-code level [120].) *Ordered linear continuation* types enforce a stack discipline that permits a high-precision analysis.

Another worthwhile direction for future work is adapting techniques for the security of machine code to information flow: for example, typed assembly languages [28] that guarantee machine code does not violate type safety, and proof-carrying code [30], [121], where a proof that the program satisfies a security policy is distributed with the code and is checked before execution.

C. Abstraction-Violating Attacks

It is inevitable that the model of the attacker is an abstraction that removes possibly important details about the real attacker. This abstraction enables the real attacker to circumvent the security mechanisms by mounting an attack that lies outside the abstract model. One example of such an attack on timing-sensitive security is a *cache attack* [119]. Consider the following example:

$$(\mathsf{if}\ h = 1\ \mathsf{then}\ h' := h_1\ \mathsf{else}\ h' := h_2);\ h' := h_1$$

where all variables are high. Independently of sensitive data, the program executes the same number of instructions. However, in the presence of a cache, execution time is likely to be shorter when the initial value of h is 1: by the time the last assignment is executed, the value of h_1 will already be present in the cache.

While this attack can be prevented by a cross-copying transformation that ensures that the same memory cells are referenced in both branches of the if, possible attacks remain that are based on instruction cache, virtual memory and platform-dependent behavior [119]. As this example demonstrates, it is vital that the abstractions made in the attacker model are adequate with respect to potential attacks.

D. Dynamic Policies

It is a common assumption in language-based work on information flow that information-flow policies are known statically, at compile time. This is not a realistic assumption for a large computing system. For example, the files in a file system have attached security policies (permissions) that can be changed dynamically. If these permissions are to be enforced as end-to-end policies, programs accessing file system must be able to enforce dynamically changing security policies.

Dynamic security policies have been proposed in a language-based setting [4] and implemented in the Jif compiler [7], [44]. In the Jif security-type system, types may be annotated with confidentiality labels that refer to variables of the type label. Thus, labels may be used both as first-class values and as labels for other values. Types that depend on values computed at run time are *dependent types*, a topic long of interest in the programming-languages community (e.g., [122], [123]).

Dynamic security policies are an important area for future work; although dynamic labels are not known to introduce unsoundness into the Jif type system, currently there are no noninterference results for any fragment that supports them. The difficulty in adding dynamic labels is that because they are computed at run time, they create an additional information channel that must be controlled through the type system [7]. A type system that provably controls this information channel—without being unnecessarily restrictive—would be a welcome result.

E. Practical Issues

Efforts spent on accommodating richer languages and modeling elaborate attacks should be supported by the investigation of the impact on the restrictiveness for a programmer. The question is whether it is, in the first place, possible to write efficient secure programs that do not violate security requirements. For a timing-sensitive setting, a step has been made by Agat and Sands [124] who show that basic algorithmic building blocks (such as sorting and searching) that manipulate secret data can be securely implemented without a substantial loss of efficiency: for n objects, the asymptotic complexities of sorting and searching are $\mathcal{O}(n \log n)$ and $\mathcal{O}(\log n)$, respectively.

Only a few implementations exist that support security-type inference [44], [71], [119]. More experience is needed for deeper understanding on practical implications of secure information flow.

The general problem of confidentiality for programs is undecidable. For example, consider a program C that uses only low variables. Clearly, we can reduce the (undecidable) problem whether C always diverges to the problem whether the program $\mathsf{if}\ h = 1\ \mathsf{then}\ (C; l := 7)\ \mathsf{else\ skip}$ is secure. (While this example is specific to termination-insensitive security, the program $\mathsf{if}\ h = 1\ \mathsf{then}\ (C; l := 7)\ \mathsf{else}\ (\mathsf{while\ true\ do\ skip})$ can be used in the reduction under termination-sensitive security.) It is important that security static analyses do not reject too many secure programs, even though they are necessarily conservative. Research aimed at improving the precision of type systems deserves further attention (e.g., [10], [11], [76], [78], [87]–[89], [94]). Moreover, approaches other than type systems offer valuable alternatives for accurate and flexible security analyses. This is the focus of the following section.

F. Variations of Static Analysis for Security

Control- and data-flow analyses [125] are established areas of program analysis concerned with dependencies due to control and data flow; they are a natural match for tracking security dependencies in programs. While type systems are, in general, intuitive and well understood, one reason for using control- and data-flow analyses is that type systems sometimes lack the *principal type* (the most general type that can be given to a command or expression), which may result in the loss of precision [126].

Bodei et al. [127], [128] demonstrate the use of a control-flow analysis to establish Bell–LaPadula security properties for the π-calculus. In the context of firewalls, formalized by ambients [129], Nielson et al. [130] show how to statically reject firewalls that may accept the attacker that fails to provide the required password. Bodei et al. [131] devise a control-flow analysis that guarantees Dolev–Yao secrecy for the spi calculus. They suggest a conservative extension of the analysis that also enforces noninterference-based confidentiality.

Clark et al. [132] propose a high-precision control-flow-sensitive security analysis for a higher-order imperative language. In particular, the analysis traces global flows more accurately than many type systems. For example, it accepts as secure the program

$$(\mathsf{if}\ h = 1\ \mathsf{then}\ l := 1\ \mathsf{else}\ l := 0); l := 0$$

which is also secure by both termination-sensitive and termination-insensitive interpretations of condition $(*)$. Yet a typical security-type system (e.g., [3]) rejects the program.

The framework of *abstract interpretation* [133] is a powerful methodology for designing static analyses that are sound by construction. Malacaria and Hankin [134] develop an information-flow analysis by abstract interpretation in the setting of game semantics [135]. Zanotti [136] proposes an abstract-interpretation-based security analysis that generalizes the security-type system by Volpano et al. [3].

VI. Conclusion

We have argued that standard security practices are not capable of enforcing end-to-end confidentiality policies; mechanisms such as access control, encryption, firewalls, digital signatures, and antivirus scanning do not address the fundamental problem: tracking the flow of information in computing systems. Run-time monitoring of operating-systems calls is similarly of limited use because information-flow policies are not properties of a single execution; in general, they require monitoring all possible execution paths. On the other hand, there is clear evidence of benefits provided by language-based security mechanisms that build on technology for static analysis and language semantics. In this section, we summarize the benefits of security-type systems and semantic-based security models, and emphasize the compositional nature of both. We conclude by discussing related and future work.

Security-type systems: Type systems are attractive for implementing static security analyses. It is natural to augment type annotations with security labels. Type systems allow for compositional reasoning, which is a necessity for scalability when applied to larger programs. Many well developed features of type systems have been usefully applied to security analysis. Examples include *subtyping* (e.g., [3]), *polymorphism* (e.g., [7], [63]), *dependent* types (e.g., [7]), *linear* types (e.g., [12], [76]), and *type-directed translation* (e.g., [10], [71]).

Semantics-based security models: Semantics-based models are suitable for describing end-to-end policies such as noninterference and its extensions. These models allow for a precise formulation of the attacker's view of the system. This view is described as a relation on program behaviors where two behaviors are related if they are not distinguishable by the attacker. Attackers of varying capabilities can be modeled straightforwardly as different attacker views, and correspond to different security properties. In particular, it has been shown how to represent the *timing* (e.g., [10]) and *probabilistic* (e.g., [9], [11]) behavior of programs if the attacker is capable of making timing- and probability-sensitive distinctions.

Compositionality: A number of further advantages are associated with both security-type systems and semantics-based security. Compositionality is especially valuable in the context of security properties. While it is folklore in the security community that security properties do not compose (cf. [82], [137]), compositionality is fundamental in programming languages. Most type systems and many security conditions (e.g., [11], [17], [72], [90], [138]) are compositional, which ensures that plugging secure programs into a security-preserving context gives a secure program. Compositionality greatly facilitates correctness proofs for program analyses.

Related use of language-based techniques: There is a large body of work on noninterference in the setting of process algebras such as CCS (e.g., [139]), CSP (see [140] for an overview), π-calculus, the spi calculus, and other event-based systems (e.g., [83]). Notably, the idea of representing attackers as view relations is also common in studies of noninterference for process algebra [141]. Compositionality reasoning (e.g., [139], [142]) is an essential part of security investigations of event-based systems. Type systems are widely used for ensuring confidentiality properties, in the spi calculus (e.g., [97], [98]), and (variations of) π-calculus (e.g., [91], [92], [143]–[145]). Timing- and probability-sensitive confidentiality has been explored by, e.g., Focardi et al. [146] and, e.g., Aldini [147], respectively, for variations of CCS. Lowe [113] has explored quantitative information flow for CSP (cf. Section IV).

Toward a practical security mechanism: If the recent progress in language-based techniques for soundly enforcing end-to-end confidentiality policies continues, the approach may soon become an important part of standard security practice. However, there are three areas where further work is needed:

- Semantics of information flow are needed for concurrent and distributed systems so that useful end-to-end security guarantees are provided without ruling out useful, secure programs.
- New type systems or other static analyses are needed, for which the notion of a well-formed (typable) program closely approximates the semantic notion of security.
- Certifying compilers are needed for security-typed languages, because compilers for source languages (such as Jif) are too complex to be part of the trusted computing

base. However, current security-type systems are not expressive enough to support a security-typed low-level target language.

The inability to express or enforce end-to-end security policies is a serious problem with our current computing infrastructure, and language-based techniques appear to be essential to any solution to this problem.

ACKNOWLEDGMENT

The authors would like to thank M. Hicks for helpful comments and the anonymous reviewers for useful feedback.

REFERENCES

[1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[2] J. Palsberg and P. Ørbæk, "Trust in the λ-calculus," in *Proc. Symposium on Static Analysis*. Sept. 1995, number 983 in LNCS, pp. 314–329, Springer-Verlag.

[3] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *J. Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[4] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Proc. ACM Symp. on Operating System Principles*, Oct. 1997, pp. 129–142.

[5] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1998, pp. 365–377.

[6] G. Smith and D. Volpano, "Secure information flow in a multi-threaded imperative language," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1998, pp. 355–364.

[7] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 228–241.

[8] G. Barthe and B. Serpette, "Partial evaluation and non-interference for object calculi," in *Proc. FLOPS*. Nov. 1999, vol. 1722 of *LNCS*, pp. 53–67, Springer-Verlag.

[9] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," *J. Computer Security*, vol. 7, no. 2–3, pp. 231–253, Nov. 1999.

[10] J. Agat, "Transforming out timing leaks," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2000, pp. 40–53.

[11] A. Sabelfeld and D. Sands, "Probabilistic noninterference for multi-threaded programs," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 200–214.

[12] S. Zdancewic and A. C. Myers, "Secure information flow and CPS," in *Proc. European Symposium on Programming*. Apr. 2001, vol. 2028 of *LNCS*, pp. 46–61, Springer-Verlag.

[13] A. Banerjee and D. A. Naumann, "Secure information flow and pointer confinement in a Java-like language," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 253–267.

[14] F. Pottier and V. Simonet, "Information flow inference for ML," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2002, pp. 319–330.

[15] M. Abadi, A. Banerjee, N. Heintze, and J. Riecke, "A core calculus of dependency," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 147–160.

[16] F. Pottier and S. Conchon, "Information flow inference for free," in *Proc. ACM International Conference on Functional Programming*, Sept. 2000, pp. 46–57.

[17] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher Order and Symbolic Computation*, vol. 14, no. 1, pp. 59–91, Mar. 2001.

[18] B. W. Lampson, "A note on the confinement problem," *Comm. of the ACM*, vol. 16, no. 10, pp. 613–615, Oct. 1973.

[19] D. Dolev and A. Yao, "On the security of public-key protocols," *IEEE Transactions on Information Theory*, vol. 2, no. 29, pp. 198–208, Aug. 1983.

[20] B. W. Lampson, "Protection," in *Proc. Princeton Symposium on Information Sciences and Systems*, Princeton University, Mar. 1971, pp. 437–443, Reprinted in *Operating Systems Review*, vol. 8, no. 1, pp. 18–24, Jan. 1974.

[21] J. B. Dennis and E. C. VanHorn, "Programming semantics for multiprogrammed computations," *Comm. of the ACM*, vol. 9, no. 3, pp. 143–155, Mar. 1966.

[22] W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The kernel of a multiprocessor system," *Comm. of the ACM*, vol. 17, no. 6, pp. 337–345, June 1974.

[23] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer, "Simple object access protocol (SOAP) 1.1," http://www.w3.org/TR/SOAP/, May 2000.

[24] T. Lindholm and F. Yellin, *The Java Virtual Machine*, Addison-Wesley, Reading, MA, May 1996.

[25] J. S. Fritzinger and M. Mueller, "Java security," Tech. Rep., Sun Microsystems, Inc., Palo Alto, CA, 1996.

[26] D. S. Wallach, A. W. Appel, and E. W. Felten, "The security architecture formerly known as stack inspection: A security mechanism for language-based systems," *ACM Transactions on Software Engineering and Methodology*, vol. 9, no. 4, pp. 341–378, Oct. 2000.

[27] R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," in *Proc. ACM Symp. on Operating System Principles*, Dec. 1993, pp. 203–216.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew, "From System F to typed assembly language," *ACM TOPLAS*, vol. 21, no. 3, pp. 528–569, May 1999.

[29] D. Wagner, *Static analysis and computer security: New techniques for software assurance*, Ph.D. thesis, University of California at Berkeley, 2000.

[30] G. C. Necula, "Proof-carrying code," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1997, pp. 106–119.

[31] U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proc. of the New Security Paradigm Workshop*, Sept. 1999, pp. 87–95.

[32] D. Evans and A. Twyman, "Flexible policy-directed code safety," in *Proc. IEEE Symp. on Security and Privacy*, May 1999, pp. 32–45.

[33] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," in *Informatics—10 Years Back, 10 Years Ahead*, vol. 2000 of *LNCS*, pp. 86–101. Springer-Verlag, 2000.

[34] K. J. Biba, "Integrity considerations for secure computer systems," Tech. Rep. ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, Apr. 1977, (Also available through National Technical Information Service, Springfield Va., NTIS AD-A039324.).

[35] J. S. Fenton, *Information Protection Systems*, Ph.D. thesis, University of Cambridge, Cambridge, England, 1973.

[36] J. S. Fenton, "Memoryless subsystems," *Computing J.*, vol. 17, no. 2, pp. 143–147, May 1974.

[37] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," Tech. Rep. MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[38] L. J. LaPadula and D. E. Bell, "Secure computer systems: A mathematical model," Tech. Rep. MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973, Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[39] Department of Defense, *Department of Defense Trusted Computer System Evaluation Criteria*, DOD 5200.28-STD (The Orange Book) edition, Dec. 1985.

[40] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Comm. of the ACM*, vol. 20, no. 7, pp. 504–513, July 1977.

[41] D. E. Denning, *Cryptography and Data Security*, Addison-Wesley, Reading, MA, 1982.

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

[43] J. McHugh and D. I. Good, "An information flow tool for Gypsy," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1985, pp. 46–48.

[44] A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic, "Jif: Java information flow," Software release. http://www.cs.cornell.edu/jif, July 2001.

[45] J. McLean, "A general theory of composition for trace sets closed under selective interleaving functions," in *Proc. IEEE Symp. on Security and Privacy*, May 1994, pp. 79–93.

[46] D. Volpano, "Safety versus secrecy," in *Proc. Symposium on Static Analysis*. Sept. 1999, vol. 1694 of *LNCS*, pp. 303–311, Springer-Verlag.

[47] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1982, pp. 11–20.

[48] J. A. Goguen and J. Meseguer, "Unwinding and inference control," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1984, pp. 75–86.

[49] E. S. Cohen, "Information transmission in computational systems," *ACM SIGOPS Operating Systems Review*, vol. 11, no. 5, pp. 133–139, 1977.

[50] E. S. Cohen, "Information transmission in sequential programs," in *Foundations of Secure Computation*, R. A. DeMillo, D. P. Dobkin, A. K. Jones, and R. J. Lipton, Eds., pp. 297–335. Academic Press, 1978.

[51] J. McLean, "Proving noninterference and functional correctness using traces," *J. Computer Security*, vol. 1, no. 1, pp. 37–58, 1992.

[52] G. R. Andrews and R. P. Reitman, "An axiomatic approach to information flow in programs," *ACM TOPLAS*, vol. 2, no. 1, pp. 56–75, Jan. 1980.

[53] M. Mizuno and A. Oldehoeft, "Information flow control in a distributed object-oriented system with statically-bound object variables," in *Proc. National Computer Security Conference*, 1987, pp. 56–67.

[54] M. Mizuno, "A least fixed point approach to inter-procedural information flow control," in *Proc. National Computer Security Conference*, 1989, pp. 558–570.

[55] M. Mizuno and D. Schmidt, "A security flow control algorithm and its denotational semantics correctness proof," *Formal Aspects of Computing*, vol. 4, no. 6A, pp. 727–754, 1992.

[56] J.-P. Banâtre and C. Bryce, "Information flow control in a parallel language framework," in *Proc. IEEE Computer Security Foundations Workshop*, June 1993, pp. 39–52.

[57] J.-P. Banâtre, C. Bryce, and D. Le Métayer, "Compile-time detection of information flow in sequential programs," in *Proc. European Symp. on Research in Computer Security*. 1994, vol. 875 of *LNCS*, pp. 55–73, Springer-Verlag.

[58] J.-P. Banâtre, C. Bryce, and D. Le Métayer, "An approach to information security in distributed systems," in *Proc. IEEE International Workshop on Future Trends in Distributed Computing Systems*, 1995, pp. 384–394.

[59] P. Ørbæk, "Can you trust your data?," in *Proc. TAPSOFT/FASE'95*. May 1995, vol. 915 of *LNCS*, pp. 575–590, Springer-Verlag.

[60] P. Ørbæk and J. Palsberg, "Trust in the λ-calculus," *J. Functional Programming*, vol. 7, no. 6, pp. 557–591, 1997.

[61] P. Ørbæk, *Trust and Dependence Analysis*, Ph.D. thesis, BRICS, University of Aarhus, Aarhus, Denmark, 1997.

[62] D. E. Denning, "A lattice model of secure information flow," *Comm. of the ACM*, vol. 19, no. 5, pp. 236–243, May 1976.

[63] D. Volpano and G. Smith, "A type-based approach to program security," in *Proc. TAPSOFT'97*. Apr. 1997, vol. 1214 of *LNCS*, pp. 607–621, Springer-Verlag.

[64] M. Abadi, "Secrecy by typing in security protocols," in *Proc. Theoretical Aspects of Computer Software*, Sept. 1997, pp. 611–638.

[65] S. Zdancewic and A. C. Myers, "Robust declassification," in *Proc. IEEE Computer Security Foundations Workshop*, June 2001, pp. 15–23.

[66] R. Joshi and K. R. M. Leino, "A semantic approach to secure information flow," *Science of Computer Programming*, vol. 37, no. 1–3, pp. 113–138, 2000.

[67] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," *Proc. IEEE Computer Security Foundations Workshop*, pp. 156–168, June 1997.

[68] M. Dam and P. Giambiagi, "Confidentiality for mobile code: The case of a simple payment protocol," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 233–244.

[69] P. Giambiagi, "Confidentiality for implementations of security protocols," Unpublished manuscript, Feb. 2002.

[70] D. Volpano and G. Smith, "Verifying secrets and relative secrecy," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2000, pp. 268–276.

[71] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: Secure program partitioning," in *Proc. ACM Symp. on Operating System Principles*, Oct. 2001, pp. 1–14.

[72] A. Sabelfeld and H. Mantel, "Static confidentiality enforcement for distributed programs," in *Proc. Symposium on Static Analysis*. Sept. 2002, vol. 2477 of *LNCS*, pp. 376–394, Springer-Verlag.

[73] D. Clark, S. Hunt, and P. Malacaria, "Quantitative analysis of the leakage of confidential data," in *Quantitative Aspects of Programming Languages—Selected papers from QAPL 2001*. 2002, vol. 59 of *Electronic Notes in Theoretical Computer Science*, Elsevier.

[74] A. Di Pierro, C. Hankin, and H. Wiklicky, "Approximate non-interference," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 1–17.

[75] H. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*, North-Holland, 1984.

[76] S. Zdancewic and A. C. Myers, "Secure information flow via linear continuations," *Higher Order and Symbolic Computation*, vol. 15, no. 2–3, pp. 209–234, Sept. 2002.

[77] F. Pottier and V. Simonet, "Information flow inference for ML," *ACM TOPLAS*, 2002, To appear.

[78] V. Simonet, "Fine-grained information flow analysis for a λ-calculus with sum types," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 223–237.

[79] M. Abadi and L. Cardelli, *A Theory of Objects*, Monographs in Computer Science. Springer-Verlag, New York, 1996.

[80] J. McLean, "A general theory of composition for a class of "possibilistic" security properties," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 53–67, Jan. 1996.

[81] D. Sutherland, "A model of information," in *Proc. National Computer Security Conference*, Sept. 1986, pp. 175–183.

[82] D. McCullough, "Specifications for multi-level security and hook-up property," in *Proc. IEEE Symp. on Security and Privacy*, Apr. 1987, pp. 161–166.

[83] H. Mantel, "Possibilistic definitions of security – An assembly kit –," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 185–199.

[84] K. R. M. Leino and R. Joshi, "A semantic approach to secure information flow," in *Proc. Mathematics of Program Construction*, June 1998, vol. 1422 of *LNCS*, pp. 254–271.

[85] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," in *Proc. European Symposium on Programming*. Mar. 1999, vol. 1576 of *LNCS*, pp. 40–58, Springer-Verlag.

[86] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language," in *Proc. IEEE Computer Security Foundations Workshop*, June 1998, pp. 34–43.

[87] G. Smith, "A new type system for secure information flow," in *Proc. IEEE Computer Security Foundations Workshop*, June 2001, pp. 115–125.

[88] G. Boudol and I. Castellani, "Noninterference for concurrent programs," in *Proc. ICALP*, July 2001, vol. 2076 of *LNCS*, pp. 382–395.

[89] G. Boudol and I. Castellani, "Non-interference for concurrent programs and thread systems," *Theoretical Computer Science*, vol. 281, no. 1, pp. 109–130, June 2002.

[90] A. Sabelfeld, "The impact of synchronisation on secure information flow in concurrent programs," in *Proc. Andrei Ershov International Conference on Perspectives of System Informatics*. July 2001, vol. 2244 of *LNCS*, pp. 227–241, Springer-Verlag.

[91] K. Honda, V. Vasconcelos, and N. Yoshida, "Secure information flow as typed process behaviour," in *Proc. European Symposium on Programming*. 2000, vol. 1782 of *LNCS*, pp. 180–199, Springer-Verlag.

[92] K. Honda and N. Yoshida, "A uniform type structure for secure information flow," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 2002, pp. 81–92.

[93] F. Pottier, "A simple view of type-secure information flow in the pi-calculus," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 320–330.

[94] S. Zdancewic, *Programming Languages for Information Security*, Ph.D. thesis, Cornell University, July 2002.

[95] A. W. Roscoe, "CSP and determinism in security modeling," in *Proc. IEEE Symp. on Security and Privacy*, May 1995, pp. 114–127.

[96] R. P. Reitman, *Information flow in parallel programs: An axiomatic approach*, Ph.D. thesis, Cornell University, 1978.

[97] M. Abadi, "Secrecy by typing in security protocols," *J. ACM*, vol. 46, no. 5, pp. 749–786, Sept. 1999.

[98] M. Abadi and B. Blanchet, "Secrecy types for asymmetric communication," in *Proc. Foundations of Software Science and Computation Structure*. Apr. 2001, vol. 2030 of *LNCS*, pp. 25–41, Springer-Verlag.

[99] E. Sumii and B. Pierce, "Logical relations for encryption," in *Proc. IEEE Computer Security Foundations Workshop*, June 2001, pp. 256–269.

[100] P. Thiemann, "Enforcing security properties by type specialization," in *Proc. European Symposium on Programming*. Apr. 2001, vol. 2028 of *LNCS*, Springer-Verlag.

[101] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proc. CRYPTO'96*. 1996, vol. 1109 of *LNCS*, pp. 104–113, Springer-Verlag.

[102] J. McLean, "Security models and information flow," in *Proc. IEEE Symp. on Security and Privacy*, May 1990, pp. 180–187.

[103] J.W. Gray III, "Probabilistic interference," in *Proc. IEEE Symp. on Security and Privacy*, May 1990, pp. 170–179.

[104] P. Syverson and J. W. Gray III, "The epistemic representation of information flow security in probabilistic systems," in *Proc. IEEE Computer Security Foundations Workshop*, June 1995, pp. 152–166.
[105] K. G. Larsen and A. Skou, "Bisimulation through probabilistic testing," *Information and Computation*, vol. 94, no. 1, pp. 1–28, Sept. 1991.
[106] A. C. Myers and B. Liskov, "Complete, safe information flow with decentralized labels," in *Proc. IEEE Symp. on Security and Privacy*, May 1998, pp. 186–197.
[107] M. Abadi and A. D. Gordon, "A calculus for cryptographic protocols: The Spi calculus," *Information and Computation*, vol. 148, no. 1, pp. 1–70, Jan. 1999.
[108] P. Giambiagi, "Secrecy for mobile implementations of security protocols," Licentiate Thesis, Royal Institute of Technology, Stockholm, Oct. 2001.
[109] D. Volpano, "Secure introduction of one-way functions," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 246–254.
[110] P. Laud, "Semantics and program analysis of computationally secure information flow," in *Proc. European Symposium on Programming*. Apr. 2001, vol. 2028 of *LNCS*, pp. 77–91, Springer-Verlag.
[111] A. Di Pierro, C. Hankin, and H. Wiklicky, "Probabilistic confinement in a declarative framework," in *Declarative Programming—Selected papers from AGP 2000*. 2001, vol. 48 of *Electronic Notes in Theoretical Computer Science*, Elsevier.
[112] A. Di Pierro, C. Hankin, and H. Wiklicky, "Analysing approximate confinement under uniform attacks," in *Proc. Symposium on Static Analysis*. Sept. 2002, vol. 2477 of *LNCS*, pp. 310–325, Springer-Verlag.
[113] G. Lowe, "Quantifying information flow," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 18–31.
[114] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*, University of Illinois Press, 1963.
[115] H. Mantel and A. Sabelfeld, "A generic approach to the security of multi-threaded programs," in *Proc. IEEE Computer Security Foundations Workshop*, June 2001, pp. 126–142.
[116] H. Mantel and A. Sabelfeld, "A unifying approach to the security of distributed and multi-threaded programs," *J. Computer Security*, 2002, To appear.
[117] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proc. of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Sept. 1975.
[118] G. Morrisett, *Compiling with Types*, Ph.D. thesis, Carnegie Mellon University, Dec. 1995, Published as CMU Tech Report CMU-CS-95-226.
[119] J. Agat, *Type Based Techniques for Covert Channel Elimination and Register Allocation*, Ph.D. thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, Dec. 2000.
[120] A. Appel, *Compiling with Continuations*, Cambridge University Press, 1992.
[121] D. Kozen, "Language-based security," in *Proc. Mathematical Foundations of Computer Science*. Sept. 1999, vol. 1672 of *LNCS*, pp. 284–298, Springer-Verlag.
[122] M. A. Sheldon and D. K. Gifford, "Static dependent types for first class modules," in *Proc. Lisp and Functional Programming*, June 1990, pp. 20–29.
[123] H. Xi and F. Pfenning, "Dependent types in practical programming," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1999, pp. 214–227.
[124] J. Agat and D. Sands, "On confidentiality and algorithms," in *Proc. IEEE Symp. on Security and Privacy*, May 2001, pp. 64–77.
[125] F. Nielson, H. Riis Nielson, and C. Hankin, *Principles of Program Analysis*, Springer-Verlag, 1999.
[126] C. Bodei, P. Degano, H. Riis Nielson, and F. Nielson, "Security analysis using flow logics," in *Current Trends in Theoretical Computer Science*, G. Paun, G. Rozenberg, and A. Salomaa, Eds., pp. 525–542. World Scientific, 2000.
[127] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson, "Static analysis of processes for no read-up and no write-down," in *Proc. Foundations of Software Science and Computation Structure*. Apr. 1999, number 1578 in LNCS, pp. 120–134, Springer-Verlag.
[128] C. Bodei, P. Degano, F. Nielson, and H. Riis Nielson, "Static analysis for the π-calculus with applications to security," *Information and Computation*, vol. 168, pp. 68–92, 2001.
[129] L. Cardelli and A. D. Gordon, "Mobile ambients," in *Proc. Foundations of Software Science and Computation Structure*. Apr. 1998, vol. 1378 of *LNCS*, pp. 140–155, Springer-Verlag.
[130] F. Nielson, H. Riis Nielson, R. R. Hansen, and J. G. Jensen, "Validating firewalls in mobile ambients," in *Proc. CONCUR'99*. 1999, number 1664 in LNCS, pp. 463–477, Springer-Verlag.
[131] C. Bodei, P. Degano, H. Riis Nielson, and F. Nielson, "Static analysis for secrecy and non-interference in networks of processes," in *Proc. PACT'01*. Sept. 2001, vol. 2127 of *LNCS*, pp. 27–41, Springer-Verlag.
[132] D. Clark, C. Hankin, and S. Hunt, "Information flow for Algol-like languages," *Journal of Computer Languages*, 2002, To appear.
[133] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1977, pp. 238–252.
[134] P. Malacaria and C. Hankin, "Non-deterministic games and program analysis: An application to security," in *Proc. IEEE Symp. on Logic in Computer Science*, 1999, pp. 443–452.
[135] S. Abramksy and G. McCusker, "Game semantics," in *Logic and Computation: Proc. 1997 Marktoberdorf Summer School*, U. Berger and H. Schwichtenberg, Eds., NATO Science Series. Springer-Verlag, 1998.
[136] M. Zanotti, "Security typings by abstract interpretation," in *Proc. Symposium on Static Analysis*. Sept. 2002, vol. 2477 of *LNCS*, pp. 360–375, Springer-Verlag.
[137] D. McCullough, "Noninterference and the composability of security properties," in *Proc. IEEE Symp. on Security and Privacy*, May 1988, pp. 177–186.
[138] A. Sabelfeld, *Semantic Models for the Security of Sequential and Concurrent Programs*, Ph.D. thesis, Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, May 2001.
[139] R. Focardi and R. Gorrieri, "A classification of security properties for process algebras," *J. Computer Security*, vol. 3, no. 1, pp. 5–33, 1995.
[140] P. Ryan, "Mathematical models of computer security—tutorial lectures," in *Foundations of Security Analysis and Design*, R. Focardi and R. Gorrieri, Eds., vol. 2171 of *LNCS*, pp. 1–62. Springer-Verlag, 2001.
[141] P. Ryan and S. Schneider, "Process algebra and non-interference," in *Proc. IEEE Computer Security Foundations Workshop*, June 1999, pp. 214–227.
[142] H. Mantel, "On the composition of secure systems," in *Proc. IEEE Symp. on Security and Privacy*, May 2002, pp. 81–94.
[143] P. Sewell and J. Vitek, "Secure composition of untrusted code: Wrappers and causality types," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 269–284.
[144] M. Hennessy and J. Riely, "Information flow vs resource access in the asynchronous pi-calculus (extended abstract)," in *Proc. ICALP'00*. July 2000, vol. 1853 of *LNCS*, pp. 415–427, Springer-Verlag.
[145] D. Duggan, "Cryptographic types," in *Proc. IEEE Computer Security Foundations Workshop*, June 2002, pp. 238–252.
[146] R. Focardi, R. Gorrieri, and F. Martinelli, "Information flow analysis in a discrete-time process algebra," in *Proc. IEEE Computer Security Foundations Workshop*, July 2000, pp. 170–184.
[147] A. Aldini, "Probabilistic information flow in a process algebra," in *Proc. CONCUR'01*. Aug. 2001, vol. 2154 of *LNCS*, pp. 152–168, Springer-Verlag.

Andrei Sabelfeld received the Ph.D. degree in computer science from Chalmers University of Technology and Gothenburg University, Gothenburg, Sweden, in 2001.

He is a Research Associate in the Computer Science Department, Cornell University, Ithaca, NY. His research has developed the link between two areas of computer science: programming languages and computer security. He has pursued the certification of confidentiality according to established principles of programming languages.

Andrew C. Myers received the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, in 1999.

He is an Assistant Professor in the Computer Science Department, Cornell University, Ithaca, NY. His research interests include computer security, programming languages, and distributed object systems. His work on language-based information flow has focused on systems and languages that are expressive and practical.