Unifying Facets of Information Integrity

Arnar Birgisson, Alejandro Russo, and Andrei Sabelfeld

Chalmers University of Technology, 412 96 Gothenburg, Sweden

Abstract. Information integrity is a vital security property in a variety of applications. However, there is more than one facet to integrity: interpretations of integrity in different contexts include *integrity via information flow*, where the key is that trusted output is independent from untrusted input, and *integrity via invariance*, where the key is preservation of an invariant. Furthermore, integrity via invariance is itself multi-faceted. For example, the literature features formalizations of invariance as predicate preservation (*predicate invariance*), which is not directly compatible with invariance of memory values (*value invariance*). This paper offers a unified framework for integrity policies that include all of the facets above. Despite the different nature of these facets, we show that a straightforward enforcement mechanism adapted from the literature is readily available for enforcing all of the integrity facets at once.

1 Introduction

Information integrity is a vital security property in a variety of applications. However, there is clearly more than one facet to integrity. Indeed, security textbooks [40, 25] agree that it is hard to pin down the essence of integrity, and surveys [33, 45, 42] and tutorials [26] identify a range of integrity flavors.

Integrity in the area of information flow often means that trusted output is independent from untrusted input [10]. This is dual to the classical models of confidentiality [9, 30, 17, 24], where public output is required to be independent from secret input. Integrity in the area of access control [45] is concerned with improper/unauthorized data modification. The focus is on preventing data modification operations, when no modification rights are granted to a given principal. Integrity in the context of fault-tolerant systems is concerned with preservation of actual data. For example, a desired property for a file transfer protocol on a lossy channel is that the integrity of a transmitted file is preserved, i.e., the information at both ends of communication must be identical (which can be enforced by detecting and repairing possible file corruption). Integrity in the context of databases often means preservation of some important invariants, such as consistency of data and uniqueness of database keys.

The list of different interpretations of integrity can be continued, including rather general notions as integrity as *expectation of data quality* and integrity as guarantee of *accurate data* and *meaningful data* [45, 40].

Sabelfeld and Myers [42] observe that integrity has an important difference from confidentiality: a computing system can damage integrity without any external interaction, simply by computing data incorrectly. Thus, strong enforcement of integrity requires proving program correctness.

Seeking to clarify the area of integrity policies, Li et al. [31] suggest a classification for data integrity policies into *information-flow*, *data invariant*, and *program correctness* policies. In a similar spirit, Guttman [26] identifies *causality* and *invariance* policies as two major types of data integrity policies.

With the classification by Li et al. [31] as a point of departure, we present a general framework for the different facets of integrity that include information-flow, invariance, and correctness aspects. Furthermore, we argue that integrity via invariance is itself multi-faceted. For example, the literature (cf. [31]) features formalizations of invariance as predicate preservation (*predicate invariance*), which is not directly compatible with invariance of memory values (*value invariance*).

This paper offers a unified framework for integrity policies that include all of the facets above. A key feature of the framework is generalized invariants that can represent a range of properties from program correctness to predicate and value invariance. Our formalization shows that program correctness (which was previously identified as a separate type of integrity [31]) in fact subsumes invariance-based integrity.

Figure 1 illustrates the policy set inclusion. We comment on the characteristic policy examples that correspond to points in the diagram (the formal definitions of these policies are postponed to Section 2). Notation x and x' denotes the values of the corresponding variable before and after program execution. An example of a value invariance policy is $x = x'$, i.e., the value of the variable stays unchanged. An example of a predicate invariance policy is $x > 0 \Rightarrow x' > 0$, i.e., if the variable is positive initially, it must stay positive at the end of execution. Value invariance is inherently about the relation of some expression before and after the execution. On the other hand, predicate invariance is inherently about preservation of some predicate on the current memory. As we explain in detail in Section 2, these policies are not directly compatible because (i) in order to mimic value invariance (as in $x = x'$) by predicate invariance, the final memory needs to explicitly include the initial memory, and (ii) in order to mimic predicate invariance (as in $x > 0 \Rightarrow x' > 0$) by value invariance, the predicate to be preserved needs to be encoded, if at all possible, into expression equality.

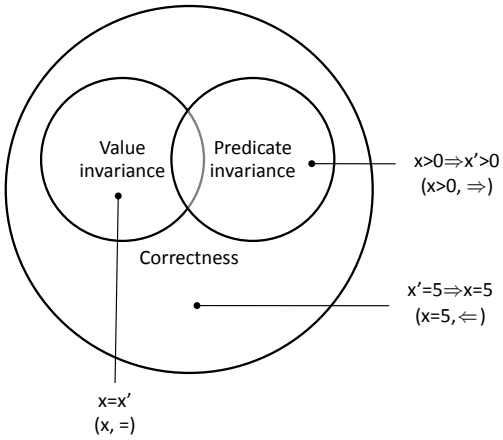

Fig. 1. Generalized invariance

Further, there are properties beyond invariance that are integrity properties. For example, $x' = 5 \Rightarrow x = 5$ is a property that assures that if the final value of the variable is 5, then it has not been modified compared to its initial value. This corresponds to a general class of properties, called *program correctness* properties. Thanks to its generality, program correctness can model all of the integrity flavors, including meaningfulness and consistency. In fact, any program property can be represented as long as it can be described by a generalized predicate that has access to initial and final memories. (As we remark in Section 8, an extension of the framework to handle intermediate states appears natural.)

Note that the goal of the paper is not to achieve as much expressiveness as possible. Indeed, a wide range of formalisms exists for reasoning about program correctness from Hoare logic [29] to refinement types [23], and a large body of work in-between [37].

Furthermore, logic-based mechanisms have been explored for reasoning about confidentiality [18, 8, 2, 6]. Instead, we aim at a treatment of integrity that allows us to express the different flavors in a uniform and convenient fashion that is directly connected to enforcement.

Indeed, despite the different nature of the integrity facets, we show that a straightforward enforcement mechanism adapted from the literature is readily available for enforcing all of the integrity facets at once. This mechanism, as proposed by Askarov and Sabelfeld [5], is originally for enforcing an information release (or *declassification*) policy of *delimited release* [43]. It guarantees that the values of declassification expressions (called *escape hatches*) have not changed compared to their initial values by performing a dynamic check at declassification time. We observe that the dual of this mechanism allows tracking both safe *endorsement* (i.e., intentional increase in trust to a given expression), and it is readily available to track correctness and therefore invariance properties. Indeed, the latter facets of integrity can be straightforwardly guaranteed by checking immediately before termination whether the desired correctness/invariance property is satisfied and terminating normally only in the case of positive outcome.

The possibility of easily deploying Askarov and Sabelfeld's enforcement [5] for a wide range of integrity policies (for which the enforcement was not originally designed) is a one of the greatest benefits of our approach. It liberates us from the necessity of designing a multi-dimensional enforcement framework of complexity similar to the policy framework.

A summary on the tightness of integration offered by our approach follows. We achieve tight integration on the enforcement side: a single enforcement mechanism is suitable to support all facets of integrity, including those that it has not been designed to support. On the policy side, the integration between information flow and correctness facets is not tight as these facets are inherently distinct. Nevertheless, within the correctness facet, we achieve tight integration of various flavors of invariants into our generalized invariant framework.

In the rest of the paper, we present a generalized definition for integrity as invariance (Section 2), recap a standard definition of integrity as information flow (Section 3), show how to enforce all facets of integrity with a single enforcement mechanism (Section 4), discuss endorsement (Section 5), extensions and practical aspects (Section 6), related work (Section 7), and offer some concluding remarks (Section 8).

To clarify the scope of this paper, we note that the focus is on *information integrity* (or data integrity), i.e., the integrity of data (in contrast to *system integrity* that addresses the integrity of the processing software and hardware units). Hence, integrity refers to information integrity throughout the paper.

2 Integrity via invariance

Before we launch into formal definitions of the concepts described above, we need some preliminaries. In particular, we must define what it means for a program to terminate. We use the term *memories* for mappings from variables to values. We work with semantics given as a small-step transition system with configurations of some form $\mathcal{C}$, where the transition system defines transitions of the forms

$$\mathcal{C} \longrightarrow \mathcal{C} \quad \text{and} \quad \mathcal{C} \longrightarrow m$$

where m is a memory. A transition of the second kind represents the terminating transition. If such a transition is contained in a trace, then it will always be the last one given

that there are only transitions of the above forms. An example of a configuration is the tuple $\langle c, m\rangle$ where c is a syntactic term (*command* or *program*) and m is a *memory*.

Definition 1 (Termination) *We say that configuration* $\mathcal{C}_0$ terminates *in a memory* m*, written* $\mathcal{C}_0 \downarrow m$ *if and only if there exists a trace*

$$\mathcal{C}_0 \to \mathcal{C}_1 \to \ldots \to \mathcal{C}_n \to m$$

(according to some particular semantics which is usually clear from the context.) If no such trace exists, we write $\mathcal{C}_0 \not\downarrow$*.*

Note that $\mathcal{C}_0 \not\downarrow$ covers both the cases when programs diverge, i.e., they have an infinite execution trace, or when they get stuck before reaching a terminal state.

2.1 Value invariance

A value invariant states that the value of an expression should not change by executing a program. We define value invariants to be expressions which are required to evaluate to the same value only in the initial and the final memory of a terminating program. We write $m(e)$ to denote the value of an expression e with respect to a memory m.

Definition 2 (Value invariant) *Let* e *be an expression. We say that a program* c *satisfies the* value invariant e *if and only if*

$$\forall m\,. \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad m(e) = m'(e).$$

A simple example of a value invariant would be the expression x, corresponding to $x = x'$ in Figure 1. This value invariant states that the variable x is not modified by running the program. Note that it may be modified *during* the execution of the program, as long as its original value is restored in the end. Another simple example is the expression $x + y$, which allows x and y to change, as long as their changes are balanced so that their sum stays constant.

On the other hand, there are some interesting "invariants" which we cannot describe by value invariants. This includes, for example, the invariant $x > 42$, which in some ways resembles a pair of a pre- and a postcondition. Treating this boolean expression as a value invariant requires that if the expression is false in the initial memory, it must also be false in the final memory. However, by our intuition, starting from a memory where the expression is false, we would like the program to be valid no matter the final value of the expression. This leads us to another notion of invariance from the literature.

2.2 Predicate invariance

Predicate invariance [31] resembles very much pre- and postconditions from Hoare logic [29, 37]. A predicate invariant consists of a boolean predicate on memories that programs must preserve.

Definition 3 (Predicate invariant) *For a predicate* φ *on memories, a program* c *satisfies the* predicate invariant φ *if and only if*

$$\forall m\,. \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \varphi(m) \Rightarrow \varphi(m').$$

Predicate invariants allow us to easily describe invariants such as $x > 0$ (see Figure 1) with the intuitive semantics described above. The intuitive idea described by Li et. al. is that φ can be used to describe when a memory has a *good* property, where it is desirable that programs preserve that property in the final memory.

However, there are also important examples of invariants which are not captured by predicate invariance. For example, the simple value invariant x, i.e., a given variable maintains it value, cannot be modeled as a predicate invariant without passing the initial value of the variable to the final memory. Thus, the two types of invariants are incompatible. In the next section we define a new notion of invariance which unifies the two.

2.3 Generalized invariance

We can observe that both of the above notions of invariance quantify over all initial memories, which for deterministic languages corresponds to quantifying over all runs of a particular program. If we treat (terminating) programs purely as a transformation on memories, then a possible general notion of invariance is simply a predicate on the initial and final memories. A given program satisfies such an invariant if all pairs of initial and final memories that it relates satisfy the predicate. Obviously, this captures the two notions of invariance above.

We provide a particularly convenient policy language that is equivalent in expressiveness to a general predicate on initial and final memory. The goal is that an invariant can be easily specified by the programmer and enforced by, e.g., a runtime monitor. Thus, we specify invariants by two expressions, one to be evaluated in initial memory and one in final memory, along with a binary predicate on those values. As we will see in Section 5, this is a particularly beneficial way to specify invariants because of smooth integration with *endorsement*.

Definition 4 (Generalized invariant) *A* generalized invariant *is a triple* (e_1, e_2, P)*, where* e_1*, and* e_2 *are expressions, and* P *is a binary predicate on values. A program* c *satisfies such an invariant if and only if*

$$\forall m\,. \qquad \langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad P\big(m(e_1), m'(e_2)\big)$$

We can explore the expressiveness of this notion of invariance. We can immediately observe that it captures the previously defined notions of invariance. Any value invariant e is represented by the generalized invariant $(e, e, =)$. Similarly, any predicate invariant φ is represented by $(\varphi, \varphi, \Rightarrow)$. These observations are depicted in Figure 2.

Generalized invariants can also describe more general notions of correctness. Our example of $x' = 5 \Rightarrow x = 5$ from Section 1 can be described by $(x = 5, x = 5, \Leftarrow)$. If we want to make sure a certain variable increases by running a program, we can write $(x, x, <)$.

Generalized invariance	e_1	e_2	P
Value invariance	e	e	$=$
Predicate invariance	φ	φ	$\Rightarrow$

Fig. 2. Kinds of invariance

It may not be clear why we would call such a condition as the last one an *invariant*, as it appears to state that something *has to* change. However, the property $m(x) < m'(x)$ must hold for all initial and final memories m and m', if we are to say that the program in question satisfies it. In other words, the property predicate by itself is an invariant for all runs of a program.

Another important facet of integrity is that we do not want untrusted inputs to have any influence on trusted outputs. This facet cannot be described by generalized invariants [34, 46], and is the topic of the next section.

3 Integrity via information flow

Information-flow integrity policies restrict how untrustworthy data flows inside programs. These policies seek to prevent corrupting critical information. For example, the (untrusted) data input of an in-flight entertainment system must not affect the auto-pilot control system (critical component), but the auto-pilot control system might be allowed to display information in the in-flight entertainment systems, such as estimated time of arrival. For simplicity, we only consider two integrity levels: H_i (high integrity) for trustworthy and L_i (low integrity) for untrustworthy data. A common baseline policy for information flow is the *noninterference* policy [17, 24]. This policy states that trustworthy data cannot be affected by untrustworthy values (written as $L_i \not\sqsubseteq H_i$). However, there is no risk for untrusted data to be affected by trusted data. In this case, we indicate $H_i \sqsubseteq L_i$. The integrity levels H_i and L_i form a two-point security lattice [19] that indicates how information can flow inside programs.

As before, we write $\langle c, m\rangle \downarrow m'$ for a terminating execution of program c under the initial memory m and final memory m'. We assume that every variable in memory is assigned an integrity level. Memories m_1 and m_2 are *high-integrity equivalent*, written $m_1 =_{H_i} m_2$, if they agree on high integrity values. The following definition captures the noninterference security policy.

Definition 5 (Noninterference) *A program c satisfies noninterference if for any memories m_1 and m_2 such that $\langle c, m_1\rangle \downarrow m_1'$, and $\langle c, m_2\rangle \downarrow m_2'$, then*

$$m_1 =_{H_i} m_2 \quad \Longrightarrow \quad m_1' =_{H_i} m_2'.$$

The definition above ignores nonterminating executions of programs. This kind of definition is known as *termination-insensitive* noninterference [47, 42, 1]. In some cases, attackers can still affect the termination behavior of the program. However, we ignore the termination channel because its bandwidth is negligible [1] in our setting.

Information-flow integrity can be seen as the dual to confidentiality. To illustrate this connection, we assume confidentiality levels L_c and H_c for public and secret information, respectively. Observe that the integrity requirements $L_i \not\sqsubseteq H_i$ and $H_i \sqsubseteq L_i$ are the duals to the ones $L_c \sqsubseteq H_c$ and $H_c \not\sqsubseteq L_c$, which indicate that secret information cannot be leaked to public recipients. This confidentiality policy underlies the original definitions of noninterference [17, 24]. Due to this duality, the techniques developed for confidentiality can also be used to guarantee information-flow integrity. In the next section, we extend a runtime monitor for enforcing information-flow confidentiality to enforce both information-flow and invariance integrity.

4 Enforcement

To illustrate the idea behind enforcement, we consider a simple imperative language with the syntax and semantics given in Figures 3 and 4, respectively. The syntax and semantic rules are mostly standard [48] except for minor extensions. We include a pseudo-term end that indicates leaving the scope of an `if` or a `while`. This term generates a

$$n \in \mathbb{Z}, \quad x \in \mathit{Vars}, \quad \mathit{op} \in \{+, -, \dots\}$$
$$e ::= n \mid x \mid e \mathit{op} e$$
$$c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \mid \texttt{while } e \texttt{ do } c$$

Fig. 3. Syntax

$$\text{SKIP}\ \langle \texttt{skip}, m\rangle \xrightarrow{nop} \langle \varepsilon, m\rangle \qquad \text{ASSIGN}\ \langle x := e, m\rangle \xrightarrow{a(x,e)} \langle \varepsilon, m[x \mapsto m(e)]\rangle$$

$$\text{SEQ}_1\ \frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle c_1', m'\rangle \qquad c_1' \neq \varepsilon}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_1'; c_2, m'\rangle} \qquad \text{SEQ}_2\ \frac{\langle c_1, m\rangle \xrightarrow{\beta} \langle \varepsilon, m'\rangle}{\langle c_1; c_2, m\rangle \xrightarrow{\beta} \langle c_2, m'\rangle}$$

$$\text{IF}_1\ \frac{m(e) \neq 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_1; \mathit{end}, m\rangle}$$

$$\text{IF}_2\ \frac{m(e) = 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m\rangle \xrightarrow{b(e)} \langle c_2; \mathit{end}, m\rangle}$$

$$\text{WHILE}_1\ \frac{m(e) \neq 0}{\langle \texttt{while } e \texttt{ do } c, m\rangle \xrightarrow{b(e)} \langle c; \mathit{end}; \texttt{while } e \texttt{ do } c, m\rangle} \qquad \text{WHILE}_2\ \frac{m(e) = 0}{\langle \texttt{while } e \texttt{ do } c, m\rangle \xrightarrow{b(e)} \langle \mathit{end}, m\rangle}$$

$$\text{END}\ \langle \mathit{end}, m\rangle \xrightarrow{f} \langle \varepsilon, m\rangle \qquad \text{TERM}_c\ \langle \varepsilon, m\rangle \xrightarrow{term(m)} m$$

Fig. 4. Command semantics

transition described by the rule END. The rule TERM_c is also nonstandard and, together with the empty term ε, it guarantees that a terminating run of any program ends with a transition generated by this rule. Transitions in the semantics are labeled with an event β. The purpose of labeled events as well as rules END and TERM_c is communication with the runtime monitor, which is described next.

We present an extension to the dynamic monitor found in [5] in order to enforce both information-flow integrity and generalized invariants.

Figure 5 gives the monitor semantics. The monitor is a separate transition system whose transitions are labeled with the same kind of events β as the command transitions. This is used to synchronize the two executions. Furthermore, the monitor may block progress of the program, in case the program can do a transition with a certain event but the monitor is not able to match it.

The monitor enforces information-flow integrity with the rules FLOW, BRANCH and FINISH, in the same way as [5]. The first rule allows direct assignments of an expression e to a variable x, indicated by the event $a(x, e)$, only if e has the same or higher integrity than x (Γ maps variables to their integrity levels.) The rule also ensures that the minimum level $lev(st)$ on the *context stack* st is at least as high as x's level. This is to prevent *implicit flows* [20], i.e., flows via control flow. The stack contains the levels

of expressions affecting control flow. It is maintained by the rules BRANCH and FINISH, which synchronize with the program entering or leaving an `if` or `while` block, as indicated by the events $b(e)$ and f, respectively.

The rule TERM_m synchronizes with program termination and enforces the invariance integrity policy. The monitor carries in its state a set of generalized invariants, as well as a snapshot of the initial memory, and uses these to ensure all the invariants are satisfied by the execution. If not, this rule blocks the program from terminating.

$$\text{NOP}\ \frac{}{\langle i, st, \mathcal{I}\rangle \xrightarrow{nop} \langle i, st, \mathcal{I}\rangle}$$

$$\text{FLOW}\ \frac{lev(e) \sqsubseteq \Gamma(x) \qquad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st, \mathcal{I}\rangle \xrightarrow{a(x,e)} \langle i, st, \mathcal{I}\rangle}$$

$$\text{BRANCH}\ \frac{}{\langle i, st, \mathcal{I}\rangle \xrightarrow{b(e)} \langle i, lev(e) : st, \mathcal{I}\rangle}$$

$$\text{FINISH}\ \frac{}{\langle i, hd : st, \mathcal{I}\rangle \xrightarrow{f} \langle i, st, \mathcal{I}\rangle}$$

$$\text{TERM}_m\ \frac{\forall (e_1, e_2, P) \in \mathcal{I} : P(i(e_1), m(e_2))}{\langle i, st, \mathcal{I}\rangle \xrightarrow{term(m)} \langle i, st, \mathcal{I}\rangle}$$

Fig. 5. Monitor semantics

Before proving the desired properties of our monitor we should make a small note about its practicality. While it is certainly infeasible to store a snapshot of the initial memory of a program, this is only a feature of our theoretical model. In practice the only additional state required to enforce a set of invariants $\mathcal{I}$, are the values of the first expression of each one, as evaluated in initial memory. A monitor needs only evaluate these expressions at the start, store their values and then at the end evaluate the second expression as well as the predicate of each invariant. Since we expect the set of invariants to be relatively small given their expressiveness, the overhead of adding invariant enforcement is small compared to the information flow enforcement overhead of the original monitor from [5]. Controlling the complexity of the expressions and predicates of course remains the responsibility of the policy writer.

In the rest of the section, we will talk about *monitored programs* which refers to a program which is run in lockstep with a monitor. For convenience, we represent monitored programs with a monitor combination operator $\sharp$, whose semantics is defined with the following two rules, where $\mathcal{C}_c$ is the configuration of program semantics, and $\mathcal{C}_m$ is that of the monitor semantics.

$$\frac{\mathcal{C}_c \xrightarrow{\beta} \mathcal{C}'_c \qquad \mathcal{C}_m \xrightarrow{\beta} \mathcal{C}'_m}{\mathcal{C}_c \sharp \mathcal{C}_m \longrightarrow \mathcal{C}'_c \sharp \mathcal{C}'_m} \qquad \frac{\mathcal{C}_c \xrightarrow{\beta} m \qquad \mathcal{C}_m \xrightarrow{\beta} \mathcal{C}'_m}{\mathcal{C}_c \sharp \mathcal{C}_m \longrightarrow m} \tag{1}$$

Note that $\sharp$ is a meta-operator, it works on configurations rather than syntactic terms.

We can immediately state and prove one useful property of such monitored processes. If an unmonitored program does not terminate, then adding a monitor can not make it terminate. This is obvious from the right rule above, a terminating transition of the unmonitored program is a premise for proving a terminating transition of the monitored one. Nevertheless it is useful to state this explicitly as a lemma.

Lemma 1 (Failstop correctness) *For any monitored program $\mathcal{C}_c \sharp \mathcal{C}_m$, we have*

$$\mathcal{C}_c \sharp \mathcal{C}_m \downarrow m \quad \Longrightarrow \quad \mathcal{C}_c \downarrow m.$$

Proof. If the monitored program terminates, there is a terminating trace with transitions proved by the rules (1). By taking the left premise of each transition proof, it is straightforward to construct a terminating trace for the unmonitored program $\langle c, m\rangle$.

Since the terminating transition must be due to the right rule, it is obvious that both traces terminate in the same memory. □

Throughout the paper, we assume finite sets of generalized invariants $\mathcal{I}$. Given a set of generalized invariants $\mathcal{I}$, we can now prove that the monitor presented in Figure 5 is sound, in the way that a monitored program that terminates satisfies all invariants of $\mathcal{I}$ and satisfies noninterference. An important ingredient to this is that all invariants are decidable. We assume evaluation of their expressions is decidable, but we also require that checking each invariant's predicate is decidable as well.

Theorem 1 (Soundness) *Let c be a command and $\mathcal{I}$ a set of (generalized) invariants with decidable predicates. Then, for all memories m it holds that*

$$\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \downarrow m' \quad \Longrightarrow \quad \forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

i.e., the monitored program satisfies all of the invariants in $\mathcal{I}$. Furthermore, if m_1 and m_2 are high-integrity equivalent memories, and $\langle c, m_i\rangle \sharp \langle m_i, [], \mathcal{I}\rangle \downarrow m_i'$, with $i \in \{1, 2\}$, then m_1' is high-integrity equivalent to m_2', i.e., the monitored program satisfies noninterference.

Proof. We note that $\mathcal{I}$ stays unchanged by the monitor. Since $\langle c, m\rangle$ terminates in m', there exists a trace

$$\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \longrightarrow \cdots \longrightarrow \mathcal{C}_c' \sharp \mathcal{C}_m' \longrightarrow m'$$

The last rule of a monitored execution can only be the right rule of (1), which in turn means the rule used to prove the left premise is TERM_c and that $\mathcal{C}_c' = \langle \varepsilon, m'\rangle$. Consequently, the last transition of this trace must have the following proof tree:

$$\dfrac{\text{TERM}_c \ \dfrac{}{\langle \varepsilon, m'\rangle \overset{term(m')}{\longrightarrow} m'} \qquad \text{TERM}_m \ \dfrac{\forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2))}{\langle m, st, \mathcal{I}\rangle \overset{term(m')}{\longrightarrow} \langle m, st, \mathcal{I}\rangle}}{\langle \varepsilon, m'\rangle \sharp \langle m, st, \mathcal{I}\rangle \longrightarrow m'}$$

The only premise in this proof must thus hold, which concludes the proof of the invariance part.

For proof of the noninterference part we refer to [5]. □

It is a natural question to ask also if the monitor is complete. Informally, we would formulate this in the following way: If a program satisfies a set of invariants to begin with, a monitored version will not diverge unless the program does also. The presented monitor enforces both information flow integrity as well as invariant integrity. The monitor is not complete in enforcing noninterference. For example, the program $h := l; h := 0$, where h and l are high- and low-integrity variables, respectively, is blocked by the monitor although it satisfies noninterference. However, we can prove that if the information-flow integrity is set aside, then the monitor is complete in enforcing invariance integrity.

We will use the following fact (that is straightforward to prove): If all variables used in a program have the same integrity level, then no execution of the monitored version $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$ (where $\mathcal{I}$ is arbitrary) will get stuck due to the rule FLOW being disabled. This is obvious since the premises of the rule are always true if all integrity levels are equal. We can now state and prove the completeness of the monitor with respect to invariant integrity policies.

Theorem 2 (Completeness of invariance enforcement) *Let c be a command, m some memory, and $\mathcal{I}$ a set of generalized invariants with decidable predicates. Assume all variables used in c have the same integrity level. Then, if the (unmonitored) program $\langle c, m\rangle$ satisfies the invariants in $\mathcal{I}$, i.e.,*

$$\langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

then the program either diverges by itself or the monitored version also terminates (in some memory):

$$\langle c, m\rangle \not\downarrow \quad \vee \quad \langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \downarrow m''.$$

Proof. If the premise holds because $\langle c, m\rangle$ does not terminate, then the conclusion holds trivially. In the other case, when $\langle c, m\rangle \downarrow m'$ and

$$\forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)), \tag{2}$$

then we consider the terminating trace

$$\langle c, m\rangle = \langle c_0, m_0\rangle \longrightarrow \ldots \longrightarrow \langle c_n, m_n\rangle \longrightarrow m'. \tag{3}$$

From the command semantics we can see that the last transition is due to rule TERM_c.

Now consider the monitored version $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$. If this does not terminate, it must be because the monitor blocks the execution at some point. This can only happen if rules FLOW or TERM_m are disabled. However, the rule FLOW is never disabled since there is no violation of information-flow integrity, and so the monitor can only block due to the termination rule being disabled. This would mean that the monitored program gets stuck just before the last transition of (3), since this is the only transition that can potentially synchronize with TERM_m. This means $m_n = m'$ and thus we are already in final memory at this point. Since TERM_m is disabled, its premise is false. However being in final memory, its premise is exactly (2), which we assumed true. Thus, the monitored program must terminate. □

The completeness theorem states that our monitor will never stop an otherwise terminating and correct program. In other words, the monitor does not raise false alarms.

However, completeness alone is not enough, since the monitor could potentially terminate in a different final memory than the original, correct program does. Of course, this is not desirable, so we follow with a proof that our monitor is *transparent*, i.e., it does not alter the semantics of correct programs.

Theorem 3 (Transparency of invariance enforcement) *Let c be a command, m a memory, and $\mathcal{I}$ a set of generalized invariants with decidable predicates. We assume that all variables in c have the same integrity level. If the (unmonitored) program satisfies the invariants in $\mathcal{I}$, formally*

$$\langle c, m\rangle \downarrow m' \quad \Longrightarrow \quad \forall (e_1, e_2, P) \in \mathcal{I} : P(m(e_1), m'(e_2)),$$

then, the following implications hold:

$$\begin{array}{lll} \langle c, m\rangle \downarrow m' & \Longrightarrow & \langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \downarrow m', \quad \textit{and} \\ \langle c, m\rangle \not\downarrow & \Longrightarrow & \langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle \not\downarrow \end{array}$$

Proof. First, assume that $\langle c, m\rangle \downarrow m'$. By the completeness theorem above, the monitored version terminates in some memory m''. To see that $m' = m''$, observe the last transition of the monitored trace. This transition is due to the right rule of (1) whose first premise can only be met by the last transition of (3) from the last proof. By the definition of that rule, the conclusion indeed "returns" the same memory m'. This proves the first implication. The second implication is a simple contrapositive of Lemma 1. □

$$\frac{}{\langle x := \texttt{endorse}(e), m\rangle \xrightarrow{end(x,e,m)} \langle \varepsilon, m[x \mapsto m(e)]\rangle} \qquad \frac{i(e) = m(e) \quad lev(st) \sqsubseteq \Gamma(x)}{\langle i, st, \mathcal{I}\rangle \xrightarrow{end(x,e,m)} \langle i, st, \mathcal{I}\rangle}$$

Fig. 6. Rules for endorsement

5 Endorsement

When dealing with confidentiality, it is sometimes necessary to intentionally release, or *declassify*, some confidential information [44]. Analogously for integrity, it is sometimes necessary to boost the integrity of some piece of untrustworthy data to trustworthy. For example, the integrity of user-provided data can be raised after the data is sanitized. Dually to declassification, *endorsement* converts low integrity into high integrity data.

This section introduces a security condition and an enforcement mechanism for endorsement that can be seen as the dual of *delimited release* [43, 5]. We include the command $x := \texttt{endorse}(e)$ in our language for boosting the integrity of expression e from low to high. The semantic rule, depicted in Figure 6, simply performs the assignment and triggers the event $end(x, e, m)$ for communication with the monitor.

The security condition, dubbed *delimited endorsement*, captures what it means to be secure for programs involving endorsements.

Definition 6 (Delimited endorsement) *Consider a program c containing exactly n endorsement commands $x_1 := \texttt{endorse}(e_1), \ldots, x_n := \texttt{endorse}(e_n)$, where expressions $e_1, \ldots, e_n$ are called* escape hatches*. Command c is secure if for all memories m_1 and m_2 such that $m_1 =_{H_i} m_2$, $\forall i. m_1(e_i) =_{H_i} m_2(e_i)$, $\langle c, m_1\rangle \downarrow m_1'$, and $\langle c, m_2\rangle \downarrow m_2'$, we have $m_1' =_{H_i} m_2'$.*

Intuitively, delimited endorsement establishes that a program is secure if whenever two high-integrity equivalent memories are indistinguishable by escape hatches, then they must also be indistinguishable by the program itself: terminating runs of the program in these memories leads to high-integrity equivalent final states. One way to enforce this condition is by checking whether the value of any escape-hatch expression at the time of endorsement is the same as it was at the beginning of computation. This brings us to the enforcement.

The monitor rule for endorsement is also given in Figure 6. It checks that the endorsed value $m(e)$ of expression e in memory m is indeed the same in the initial and current memory ($i(e) = m(e)$). This restriction avoids laundering, i.e., abusing the endorsement mechanisms to endorse other data than the one indicated by $x := \texttt{endorse}(e)$ [43, 5]. Similar as for regular assignments, restriction $lev(st) \sqsubseteq x$ is used to avoid implicit flows.

The mechanisms to enforce invariants in Figure 5 can be easily reused for enforcing endorsement. Observe that $i(e) = m(e)$ can be interpreted as a particular kind of invariant $P(i(e), m(e'))$, where P is the equality predicate $=$, and expressions e and e' are the same.

The following theorem establishes the formal guarantees obtained by the enforcement rules.

Theorem 4 *For any program c, the monitored execution of c (with the initial configuration $\langle c, m\rangle \sharp \langle m, [], \mathcal{I}\rangle$ for memory m) satisfies delimited endorsement.*

Proof. It follows by an adaptation of Askarov and Sabelfeld's proof [5] for delimited release. The failstop property of the monitor allows for a straightforward adaption of the proof: the invariant-checking part is largely orthogonal since all the monitor can do is to block the execution, in which case the high-integrity equivalence does not need to be tracked. □

As it was for the information-flow part of the monitor in Section 4, the delimited endorsement monitor is incomplete in the information-flow part for the same reason.

6 Extensions and practical aspects

The enforcement mechanisms presented in Section 4 and 5 can be naturally extended to support I/O operations and a form of access control. We briefly outline the principles behind such extensions and discuss practical aspects.

6.1 I/O

Programs often require to take inputs as well as produce outputs during execution. Defining and tracking delimited release in the presence of communication primitives is described in [5]. When considering inputs, the restriction $i(e) = m(e)$ needs to be revised because it does not allow to declassify (endorse in our case) variables that have been updated by inputs.

Askarov and Sabelfeld [5] remark that inputs may introduce fresh data into programs and, therefore, they distinguish them from regular updates. They propose a monitor that allows to declassify information when the value being declassified $(m(e))$ matches the value of the expression in a memory that records most recent inputs. If no inputs where performed for a given variable e, the value considered for that variable is the one found in the initial memory. In a similar fashion, it is possible to modularly extend the rules in Figures 5 and 6 to consider a *context level input* label *ct*, which records if there has been an input in a high context, and update memory i in the monitor's state every time that an input is produced. The extended monitor then disallows endorsement if the input context label ct has low integrity. This is necessary because inputs, unlike branch/loop guards, are not lexically-bounded in their impact. The update of memory i on every input allows the monitor to have a memory where each variable's value refers either to its last input or its value at the initial memory (i.e., no inputs are performed for that variable).

In the presence of outputs, checking invariants at the end of program execution needs to be revised. Data invariants could refer to outputs produced by programs, e.g., every credit-card number sent to a server must be formed by 16 digits. To express this, it is sufficient to apply rule TERM_c at every output produced by the program. In principle, it is possible to allow programmers to indicate what invariants must be checked at what outputs.

When considering inputs and outputs, the security condition for declassification in [5] is based on the attacker's knowledge [21, 4, 6]. With this in mind, it is possible to use the same semantics techniques to handle endorsement in presence of communication primitives. In fact, the dual of the attacker knowledge in [5] can be interpreted as the attacker capabilities to control or affect computations regarding high integrity data [3].

6.2 Access control

As mentioned in Section 1, integrity in the area of access control [45] focuses on preventing data modification operations when no modification access is granted to a given principal. Policies of the kind "resource R cannot be written by principal P" cannot be naturally enforced by noninterference. The main reason is the degree of freedom that noninterference allows regarding entities at the same security level. Noninterference only restricts how information flows among different security levels. To illustrate this, assume an information-flow enforcement mechanism is in place. Whatever security level variable R is assigned to, it is still possible to read its content, concatenated with itself, and save it back to R. Observe that these operations only manipulate data at security level R. In contrast, our monitor can be easily adapted to enforce that no write operation is invoked on R by P or, more generally, no changes are performed on resource R by just establishing, through an invariant, that the content of R is the same at the beginning and at the end of the program. Moreover, if considering endorsement as given in Figure 6, it is possible to enforce no changes on R by endorsing it at the end of the program. Direct enforcement of no unauthorized write operations is of course also possible when the monitor has access to the entire trace.

6.3 Practical aspects

Preliminary results from a Haskell-based library for integrity [22] suggest light implementation overhead to enforce integrity policies in presence of I/O and access control requirements. Diserholt [22] shows how to build a secure password administrator that preserves confidentiality of passwords as well as several facets of integrity policies, e.g., password must be difficult to guess (integrity via invariance), certain operations should not write the contents of some files (access control), and user input cannot determine the utilized hash function (integrity via information flow). We argue that it is not difficult to reformulate the concrete case study in [22] using our approach and obtain similar results.

7 Related work

Being one of the most fundamental security properties, integrity is subject to a vast area of research. We refer to security textbooks [40, 25] that discuss assorted flavors of integrity, and integrity surveys [33, 45] and tutorials [26] that develop integrity classifications. Section 1 also contains pointers to various interpretations of integrity in various disciplines.

To the best of our knowledge, our framework is the first to unify information integrity for programs. As mentioned previously, our departure point is the classification by Li et al. [31]. Our contribution compared to this classification is a more general model of invariants (Li et al. only discuss predicate invariants), a more general model of information flow (Li et al. do not consider endorsement), and a unified view, where we show that program correctness subsumes invariance policies. In addition, we also offer a unified enforcement mechanism that guarantees all aspects of integrity at once.

Information-flow integrity dates back to Biba's integrity model [10], which dualizes Bell and LaPadula's model [9, 30] for mandatory access control. The Clark-Wilson integrity model [15] is a classical model that focuses on separation of duties and transactions.

Although information integrity for programs has been unexplored compared to confidentiality, it has recently received increasing attention. Languages such as Perl, PHP, and Ruby offer dynamic integrity checks that are based on *tainting*, a runtime mechanism for tracking explicit flows.

Ørbæk and Palsberg [38, 39] define instrumented information-flow semantics for integrity in λ-calculus. The semantics is based on integrity label manipulation. An unsoundness related to the impact of flow sensitivity on information flow has been recently uncovered [41].

Heintze and Riecke [28] consider integrity as dual to confidentiality in their study of information flow for a language based on λ-calculus. Li and Zdancewic unify confidentiality and integrity policies [32] in the context of information downgrading.

A line of work on robust declassification [49, 35, 3] is based on an interplay between confidentiality and integrity, where information release (of high confidentiality data) is allowed only if it cannot be manipulated by the attacker (through attacker-controlled low integrity data) to release additional information. The Java-based Jif tool [36], as well as its web-based extensions [14, 13], implement robustness policies.

Sabelfeld and Sands [44] introduce dimensions of declassification, with the main focus on declassification of confidentiality levels. They informally discuss the dual of dimensions of declassification: dimensions of endorsement.

We draw on delimited release [43] when it comes to enforcement of integrity policies. Although delimited release is a confidentiality property, its enforcement includes information-flow aspects and is capable of enforcing generalized invariants. This paper builds on a runtime mechanism for delimited release by Askarov and Sabelfeld [5]. A static static alternative to tracking delimited release has been explored by Sabelfeld and Myers [43].

Boudol and Kolundzija [11] combine programming constructs for expressing access-control and declassification policies. Access control is represented at language level, with explicit granting, restricting, and testing access rights. Information-flow policies and access control have been also integrated at language level by Banerjee and Naumann [7], although without considering declassification.

Haack et al. [27] explore reasoning about explicit flows in program logic. They arrive at two kinds of integrity notions: *flow-based* and *format-based*. The former is an information-flow policy, and the latter is concerned with proper formatting (they give an example policy such as "a phone number field should only contain numbers"). This latter type of integrity is subsumed by generalized invariance.

Cheney et al. [12] investigate semantic foundations for *data provenance* in databases. Provenance is concerned with tracking the origin of information, and so Cheney et al. model it as a dependency analysis.

Diserholt [22] proposes a library that handles confidentiality and integrity policies in Haskell. Besides handling confidentiality, the library is also able to combine information-flow integrity, predicate invariants, and some means for access control. Similarly to this paper, their work is inspired by the classification of integrity policies in [31].

Clarkson and Schneider [16] propose contamination and suppression as quantitative definitions of integrity. The former is dual to quantitative information leakage, whereas the later measures how much information is lost from outputs. The study of suppression includes program suppression due to malicious influence and implementation errors as well as channel suppression due to information loss about inputs to a noisy channel.

8 Conclusions

We have presented a uniform framework for information integrity. The framework incorporates a range of integrity aspects from information-flow integrity to program correctness. The framework integrates different types of integrity as invariance. We show that some of the invariant-based policies are not compatible with each other (cf. value vs. predicate invariance). Nevertheless, they are naturally represented in our framework as program correctness properties. Endorsement policies naturally extend information-flow policies and also fit into the framework.

Despite being general, our integrity framework is realizable. A single enforcement mechanism [5] (for tracking delimited information release) turns out to be an excellent match for enforcement of integrity. It supports both information-flow integrity, including extensions with endorsement policies, as well as correctness properties, including the various flavors of invariance. This mechanism is scalable to handling communication primitives.

Future work is focused on the directions outlined in Section 6. We explore both formal aspects of policies in the presence of communication and access control and practical aspects of enforcement, with inlining transformation and library-based enforcement as our main goals. Another direction of work is an extension of the framework to represent trace properties, i.e., properties of sequences of intermediate states. We expect the extension of the framework and monitor rather straightforward: generalized invariants can just as well refer to the full traces, and enforcement corresponds to enforcing *safety* [46] properties.

It is important to support our results with practical findings from case studies. Preliminary results from a Haskell-based library for integrity [22] suggest light implementation overhead.

Acknowledgments Thanks are due to Michael Clarkson for useful comments. This work was funded by the European Community under the WebSand project and the Swedish research agencies SSF and VR. Arnar Birgisson is a recipient of the Google Europe Fellowship in Computer Security, and this research is supported in part by this Google Fellowship.

References

[1] A. Askarov and S. Hunt and A. Sabelfeld and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. European Symp. on Research in Computer Security*, volume 5283 of *LNCS*, pages 333–348. Springer-Verlag, Oct. 2008.

[2] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 91–102, 2006.

[3] A. Askarov and A. C. Myers. A semantic framework for declassification and endorsement. In *Proc. European Symp. on Programming*, LNCS, Mar. 2010.

[4] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[5] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *Proc. IEEE Computer Security Foundations Symposium*, July 2009.

[6] A. Banerjee, D. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *Proc. IEEE Symp. on Security and Privacy*, May 2008.

[7] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[8] G. Barthe, P. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. IEEE Computer Security Foundations Workshop*, pages 100–114, June 2004.

[9] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

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

[11] G. Boudol and M. Kolundzija. Access-control and declassification. In *Proc. Mathematical Methods, Models, and Architectures for Computer Networks Security*, volume 1 of *Communications in Computer and Information Science*, pages 85–98. Springer-Verlag, Sept. 2007.

[12] J. Cheney, A. Ahmed, and U. Acar. Provenance as dependency analysis. In *Proc. Database Programming Languages*, 2007.

[13] S. Chong, J. Liu, A. C. Myers, X. Qi, K. Vikram, L. Zheng, and X. Zheng. Secure web applications via automatic partitioning. In *Proc. ACM Symp. on Operating System Principles*, pages 31–44, Oct. 2007.

[14] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *Proc. USENIX Security Symposium*, pages 1–16, Aug. 2007.

[15] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 184–193, May 1987.

[16] M. Clarkson and F. B. Schneider. Quantification of integrity. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

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

[18] A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *Proc. Workshop on Issues in the Theory of Security*, Apr. 2003.

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

[20] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[21] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[22] A. Diserholt. Providing integrity policies as a library in Haskell. Master Thesis, Chalmers University of Technology, Gothenburg, Mar. 2010. `http://www.cse.chalmers.se/~russo/albert.htm`.

[23] T. Freeman and F. Pfenning. Refinement types for ml. In *Proc. ACM SIGPLAN Conference on Programming language Design and Implementation*, pages 268–277, 1991.

[24] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, Apr. 1982.

[25] D. Gollmann. *Computer Security (2nd Edition)*. Wiley, 2006.

[26] J. Guttman. Invited tutorial: Integrity. Presentation at the Dagstuhl Seminar on Mobility, Ubiquity and Security, Feb. 2007. `http://www.dagstuhl.de/07091/`. Slides at `http://web.cs.wpi.edu/~guttman/`.

[27] C. Haack, E. Poll, and A. Schubert. Explicit information flow properties in JML. In *Proc. WISSEC*, 2008.

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

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, 1969.

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

[31] P. Li, Y. Mao, and S. Zdancewic. Information integrity policies. In *Workshop on Formal Aspects in Security and Trust (FAST'03)*, 2003.

[32] P. Li and S. Zdancewic. Unifying confidentiality and integrity in downgrading policies. In *Workshop on Foundations of Computer Security*, pages 45–54, June 2005.

[33] T. Mayfield, J. E. Roskos, S. R. Welke, J. M. Boone, and C. W. McDonald. Integrity in automated information systems. Technical Report P-2316, Institute for Defense Analyses, 1991.

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

[35] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing robust declassification and qualified robustness. *J. Computer Security*, 14(2):157–196, May 2006.

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

[37] D. Naumann. Theory for software verification. Draft, `http://www.cs.stevens.edu/~naumann/pub/theoryverif.pdf`, Jan. 2009.

[38] P. Ørbæk. Can you trust your data? In *Proc. TAPSOFT/FASE'95*, volume 915 of *LNCS*, pages 575–590. Springer-Verlag, May 1995.

[39] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *J. Functional Programming*, 7(6):557–591, 1997.

[40] C. P. Pfleeger and S. L. Pfleeger. *Security in Computing (4th Edition)*. Prentice Hall, 2006.

[41] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. IEEE Computer Security Foundations Symposium*, July 2010.

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

[43] A. Sabelfeld and A. C. Myers. A model for delimited information release. In *Proc. International Symp. on Software Security (ISSS'03)*, volume 3233 of *LNCS*, pages 174–191. Springer-Verlag, Oct. 2004.

[44] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Computer Security*, 17(5):517–548, Jan. 2009.

[45] R. S. Sandhu. On five definitions of data integrity. In *Proceedings of the IFIP WG11.3 Working Conference on Database Security VII*, pages 257–267, 1994.

[46] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

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

[48] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[49] S. Zdancewic and A. C. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, June 2001.