



Sound Enforcement of Dynamic Release Information Flow Policy

JEFFREY CHING, Duke University, USA

DANFENG ZHANG[✉], Duke University, USA

Information flow analysis is the de facto method of assessing confidentiality and integrity issues. However, the widespread adoption of information flow analysis in real-world systems is still lacking, partly due to a fundamental gap between theory and practice: the dynamic nature of security concerns in real-world systems goes beyond the scope of existing techniques that assume a static policy (i.e., data secrecy does not change).

Recognizing the fundamental gap, a substantial amount of research has studied various aspects of it (e.g., enabling declassification, endorsement, and revocation policies). A recent work takes a step further by formalizing a promising end-to-end policy called dynamic release that unifies prior formalizations by allowing information flow restrictions to downgrade and upgrade in arbitrary ways. However, how to soundly enforce the powerful dynamic release policy is still an open question.

In this paper, we present the first type system that enforces dynamic release policy and formally prove its soundness. More specifically, we (1) formalize a core language that enables dynamic release policy, (2) develop a type system that checks dynamic release policy, (3) develop new proof techniques and formally prove that the type system enforces dynamic release policy, and (4) implement a prototype of the type system as an extension to the Rust language, along with case studies on a conference reviewing system and Civitas.

CCS Concepts: • **Security and privacy** → **Formal security models**; • **Theory of computation** → **Program analysis**.

Additional Key Words and Phrases: Information flow; Dynamic policy; Type system

ACM Reference Format:

Jeffrey Ching and Danfeng Zhang. 2026. Sound Enforcement of Dynamic Release Information Flow Policy. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 360 (October 2026), 28 pages. <https://doi.org/10.1145/3839492>

1 Introduction

Many security concerns in computer systems can be understood in terms of information flows. For example, it is crucial to ensure that private and untrusted data, as well as any data derived from them, never influence unintended channels (e.g., a public channel) within a computer system. For decades, noninterference [24] has been the de facto security policy for specifying and checking information flow requirements. However, a fundamental assumption of noninterference is that the confidentiality and integrity of information *do not* change over time, whereas security concerns in real-world applications are rarely static, making it infeasible to safeguard them via noninterference. This critical gap prevents widespread adoption of information flow analysis in real-world systems.

Therefore, to better describe information flow restrictions in real-world applications, researchers have introduced policies that allow sensitivity to change over time, known as dynamic policies. For example, declassification policy [35] allows the release of sensitive information to the public under certain conditions. Other dynamic policies have been explored, such as gradual release [7]

Authors' Contact Information: Jeffrey Ching, Duke University, Durham, USA, jeffrey.ching@duke.edu; Danfeng Zhang, Duke University, Durham, USA, danfeng.zhang@duke.edu.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/10-ART360

<https://doi.org/10.1145/3839492>

and its extended version called tight gradual release [8]. Erasure policy [20], on the other hand, requires public information to become more sensitive or be completely removed. It is required in applications on untrustworthy storage [6, 50] and secure voting systems [21].

While prior work studies *instances* of dynamic information flow policies, the field is overloaded with inconsistent terminology, formalisms, and sometimes contradictory semantics [34]. These obstacles make it difficult even for veteran researchers to follow, let alone compare, different policies. For instance, existing definitions of dynamic policies can have vastly different syntax for defining policies [44, 45], differing natures of security conditions, and even disagreements on whether or not a program is secure based on different variants of the same policy [13]. Moreover, many policies lack robust enforcement mechanisms, and only a limited number, notably the erasure policy [20], have been implemented as an extension to Jif [38].

A recent work, Dynamic Release [34], provides a promising solution: it generalizes various kinds of dynamic policies for the first time via a novel formalization. Beyond expressing various dynamic policies, the original work also allows apples-to-apples comparison between different dynamic policies and creates new insights on existing definitions. However, as a pioneering work, it comes without any enforcement mechanism. In this paper, we tackle the following goal: *to develop a sound and general enforcement mechanism for various kinds of dynamic policies*. While dynamic release policy provides a general platform for reasoning about various dynamic policies, we need to tackle several technical challenges for its enforcement. First, we need a metalanguage with its security specification, along with its operational semantics. A challenge in language design is to strike a balance between expressiveness and simplicity. To do so, we carefully design some language features tailored for dynamic release policy, including (1) a distinguished set of *security events* along with distinguished commands to set and unset those events during program execution, (2) controlled output commands in the form of $\text{output}(L, e, \text{using } s_0 \dots s_k)$, which reveal the value of e to security level L *only if* all security events $s_0 \dots s_k$ are met, and (3) a novel *relabel* command that performs permissive yet sound conversion between various kinds of labels.

Second, we must enforce the specified dynamic release policy on the program. To achieve this, we develop a novel type system that primarily controls information flow at compile time. One challenge is to achieve a balance between soundness and permissiveness, given the fact that the meaning of each label can change throughout program execution. To tackle the challenge, we define two sets of static semantics that we show soundly control information flows according to dynamic release semantics. In particular, the “flows-to” relation specifies which label is allowed to flow to another, including flows between dynamic labels and static levels, and even flows between different kinds of policies (e.g., declassification and endorsement). The “release-to” relation checks if we can safely release information to a security level at a given program point. Thanks to the simplicity of the type system, we also implement it as an extension to the Rust language.

Third, we formally prove that the type system enforces the specified dynamic release policy for all program executions. For noninterference with a static policy, various proof techniques (e.g., [33, 40, 49]) have been developed. However, these techniques are not applicable for dynamic release for two major reasons. First, standard noninterference proofs are built on an *indistinguishability* relation on memory states: two memory states are said to be indistinguishable when they agree on the public set of variables. As the set of public variables stays the same throughout program execution, a standard proof then proceeds by showing that the indistinguishability relation is an invariance throughout program execution. However, due to the dynamic nature of dynamic release policy, the set of public variables changes over time, making it impossible to establish such an invariance. To address the challenge, we introduce a novel indistinguishability relation parameterized on a security event trace (Definition 10). Moreover, dynamic release is built on the notion of *consistency* (Definition 7), which is absent in standard noninterference. To accommodate

that, we introduce unique conditions in several lemmas and theorems to reflect when intentional leakage is allowed.

In summary, we make the following contributions.

- We present a metalanguage together with its security specification for dynamic release policies (Section 3). We further formalize the semantics of dynamic security labels, as well as the semantics of several new constructs tailored to dynamic release policies.
- We present a type system that employs two sets of static inference rules to reason about and verify information flows within a program (Section 4).
- We develop novel proof techniques (Section 5) to establish the soundness of our type system. In particular, we prove that any well-typed program satisfies its specified dynamic release policy across all possible executions.
- We implement a prototype of our language and its type system as an extension to the Rust language. Moreover, we incorporate all the examples presented in the paper, and port Civitas [21, 30], originally implemented in Jif [38], as well as a conference management system developed in Lifty [39], to our Rust-based implementation (Section 6). We demonstrate that these applications, which embody realistic security requirements, can be successfully verified using our enforcement mechanism, while incurring only minimal runtime overhead.

2 Background and Overview

Noninterference. Noninterference [24] is the de facto policy enforced by information flow control [43]. Intuitively, noninterference requires that for any two runs of a program such that only secret inputs change, the program always produces the same publicly observable outputs (i.e., confidential inputs do not affect public outputs). However, noninterference is well-recognized [15, 26, 29, 43] to be too strict for real-world applications. One fundamental gap there is that noninterference assumes that the secrecy and integrity of information are *static* (i.e., they do not change). But in reality, the dynamic nature of security concerns demands information flow policies that can accommodate evolving secrecy and integrity constraints.

Dynamic Policy. Dynamic policies allow the sensitivity and/or integrity of information to change throughout its lifetime. There are mainly three kinds of such dynamic policies in the literature:

- Declassification/Endorsement [7, 10, 15]: a declassification (resp. endorsement) policy allows the sensitivity (resp. integrity) of a piece of information to be downgraded. For example, consider a bidding game illustrated in Figure 1-i, which can release the highest bid to the public only if the bidding is over, indicated by setting release at Line 7. The bid cannot be released before the bidding concludes, making both Line 5 and Line 6 insecure. However,

<pre> 1 //wbid: P 2 //bid, bid1, bid2: ¬release?S →_t P 3 eventoff(release); 4 bid = bid1 > bid2 ? bid1 : bid2; 5 wbid = bid; //insecure 6 output(P, wbid, using release); //insecure 7 eventon(release); 8 wbid=relabel 9 (bid, ¬release?S →_t P to P using release); 10 output(P, wbid, using release); </pre> (i). Bidding Game	<pre> 1 //store: M 2 //copy, credit_card: (¬trans?M →_t T) 3 eventoff(trans); 4 copy = credit_card; 5 // use copy to finish transaction 6 output(M, copy, using trans); 7 eventon(trans); 8 store = credit_card; //insecure 9 output(M, copy, using trans); //insecure </pre> (ii). Credit Card	<pre> 1 //Alice: P 2 //notes, book: (¬return?P →_p S) 3 eventoff(return); 4 notes= chap(book); 5 output(P, notes, using return); 6 eventon(return); 7 Alice = relabel //insecure 8 (book, return?P →_p S to S using return); 9 output(P, book, using return); //insecure 10 output(P, notes, using o_{notes}@P); </pre> (iii). Library System
--	--	--

Fig. 1. Examples of the dynamic policies in the order of declassification, erasure and revocation. The shaded lines use novel language features that we introduce in this work. Insecure code is marked in red. Predicates trans and return indicate that the corresponding security events trans and return are never set to true.

after the previously sensitive bids are *declassified* to the public at Line 7, the highest bid can be released to the public as shown in Line 10.

- Erasure [6, 19, 20]: an erasure policy allows the sensitivity of a piece of information (and its derivatives) to be upgraded. One important use case of the erasure policy is to upgrade sensitivity so that no memory can hold the information anymore (effectively enforces it to be removed from the whole system). For example, consider an online payment system illustrated in Figure 1-ii, which takes credit card information from customers to complete the purchase as in Lines 4–6. However, it also needs to upgrade the sensitivity of `credit_card` and copy after Line 7 to avoid information leakage after the transaction is complete. Hence, Lines 8 and 9 are considered insecure as they leak the credit card after the transaction is complete.
- Delegation and Revocation [23, 37]: delegation and revocation are usually expressed in a role-based system to add and remove access permission for each role respectively. For example, consider a library system illustrated in Figure 1-iii, where Alice is allowed to access the book and take notes during the borrowing period. However, once the book is returned, she cannot access the book anymore as shown in Lines 7–9. While revocation also upgrades the sensitivity of information, it differs from erasure that Alice can still access information released during the borrowing period (e.g., her notes) as shown in Line 10.

Dynamic Release Policy. To relieve system developers from having to understand different kinds of dynamic policies as well as the nuances among policies that appear similar on the surface but differ in important ways, a recent work by Li and Zhang [34] presents a promising end-to-end policy called *dynamic release* that unifies prior formalizations by allowing information flow restrictions to downgrade and upgrade in arbitrary ways. Furthermore, dynamic release is formalized in a framework that allows apples-to-apples comparisons across existing dynamic policies.

More specifically, a dynamic release policy is specified by a *dynamic security label* in the form of $\text{cnd}?p_1 \diamond p_2$ (see the full syntax in Section 3.3), where the condition *cnd* specifies when data sensitivity (or integrity) changes, and $\diamond$ specifies both the direction and kind of change. Each policy can either be *transient* or *persistent*, where intuitively a transient policy (resp. a persistent policy) disallows (resp. allows) information that was leaked in the past to be revealed afterwards if information flow restrictions upgrade. In this paper, we use $\rightarrow_t$ and $\leftrightarrow_t$ (resp. $\rightarrow_p$ and $\leftrightarrow_p$) to denote possible directions for transient (resp. persistent) policies. We simply use arrows without subscripts when the kind of policy is irrelevant.

With dynamic release policy, we can concisely and precisely specify the information security requirements for each of the examples in Figure 1 as dynamic security labels (annotated in gray):

- In Figure 1.i, we can specify the dynamic security label on bids as $\neg\text{release}?S \rightarrow_t P$, meaning that they are initially **S** and hence, cannot be released during the bidding phase. However, once the bidding has ended (signaled by the condition **release**), the sensitivity of `wbid` can be downgraded to **P**, making it publicly available. Moreover, according to the policy, the commands from Lines 8–10 are secure as bids are already declassified at those points. But the commands on Lines 5–6 releasing the bids before the bidding period ends are insecure.
- In Figure 1.ii, we can specify the dynamic security label on the credit card and its copy as $\neg\text{trans}?M \rightarrow_t \top$, meaning that before the transaction is complete, the merchant can use both information at level **M**. But once the transaction is complete, their sensitivity becomes the highest level $\top$, to ensure that they cannot be used in the rest of the system. Therefore, the commands on Lines 4–5 that take credit card information from customers to complete the purchase are secure according to the policy, while the commands on Lines 8–9, which leak credit card information after the transaction is complete, are insecure.

- In Figure 1.iii, we can specify the dynamic security label on book and notes as $\neg\text{return?P} \rightarrow_p \text{S}$ meaning that information cannot be released after **return** is set, unless the same information was released before the change (as this is a persistent policy with $\rightarrow_p$). Therefore, the command at Line 10 is secure according to the policy, since the notes were released to Alice before the book is returned. But releasing the book after it is returned on Lines 7–9 is insecure, as it was not (completely) released before the book is returned.

Note that Figures 1.ii and 1.iii both employ an upgrade policy, but differ in a subtle yet important way. In the former, information released prior to the sensitivity upgrade cannot be revealed after the upgrade, whereas in the latter such release remains permitted. This distinction also arises in other policies: for instance, gradual release [7] is a persistent policy, while cryptographic erasure [6], along with other erasure-based approaches, follows a transient policy semantics.

Sound Enforcement Overview. As illustrated in Figure 1, dynamic release policy provides a unified framework to specify and reason about a broad range of dynamic policies. However, prior work [34] formulates security as an end-to-end semantic property quantified over *all possible pairs of program executions* (Section 3.4). Hence, it remains an open question of how to statically determine whether a given program (e.g., the ones in Figure 1) complies with its specified policy.

The goal of this work is to develop a programming language that enables programmers to (i) implement programs, (ii) specify desired dynamic release policies, and (iii) verify that the source code satisfies its claimed dynamic release policy. So far, we have introduced several key components of the language. First, the core is an imperative language with standard constructs, including assignments, conditionals, loops, and output operations. Second, dynamic release policies are specified via annotations on variables at the beginning of the program. Third, policy changes are governed by *security events*, which can be toggled on and off through the distinguished functions `eventon` and `eventoff`. Finally, the language features `relabel` and output commands that enable permissive checking of information flows across different dynamic release labels.

To illustrate how to check the dynamic release policy in a static way, we revisit the code in Figure 1.i. The bidding game has a dynamic policy $\neg\text{release?S} \rightarrow_t \text{P}$. To check if the code satisfies this policy, we first need a sound mechanism to directly compare dynamic labels at each program point, such as Lines 8–9. This is achieved via a “flows to” relation $\sqsubseteq$ on labels, which essentially defines a security lattice on dynamic release labels. Compared with (static) lattices that are typically used in information flow systems, the security lattices for dynamic release policies are more complicated as the interpretation of a label changes over time. Hence, the lattice over the “flows to” relation is parameterized by some *facts* of security events that must hold at each program point. For example, on Lines 8–9, the label of `bid`, namely $\neg\text{release?S} \rightarrow_t \text{P}$, can safely flow to P whenever **release** has been true in the event trace (intuitively, since it has been declassified). We formalize the “flow to” relation on labels in Section 4 and prove its soundness: for any two labels $S \vdash p_1, p_2$ and security event trace σ , $p_1 \sqsubseteq_\sigma p_2$ if and only if p_2 is *always* at least as restrictive as that of p_1 on any extension of σ (Theorem 1), assuming that σ meets all security events in S . A type system built on the relation can reject insecure assignments, such as Line 5, where the relation cannot be statically verified.

Moreover, the command `output( $L, e$ , using  $s_0 \dots s_k$ )` reveals the value of e to the security level L . To check if the policy on e allows its value to be released at that point, such as Line 6 and Line 10 in Figure 1.i, we formalize the “release to” relation on labels in Section 4.3 and prove its soundness: for any label p , level L and a set of security events S such that $S \vdash p \leq L$, information with label p can be released on a channel at level L under any execution trace that satisfies S (Theorem 2). With the relation, we develop a type system that accepts the secure release in Line 10, but also rejects the insecure release in Line 6. One subtlety of checking an output command is that a persistent

Variables x, y, z
Expressions $e ::= x \mid n \mid e \text{ op } e$
Levels $L \in \mathcal{L}$
Sec. Events($\mathbb{S}$) $s \mid \neg s \mid \mathbf{o}_x @ L$
Commands $c ::= \text{skip} \mid x := e \mid \text{while } (e) \text{ do } c \mid$
 $\quad \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid c_1; c_2 \mid \text{eventon}(s) \mid$
 $\quad \text{eventoff}(s) \mid \text{output}(L, e, \text{using } s_0 \dots s_k) \mid$
 $\quad \text{output}(L, e, \text{using } \mathbf{o}_x, s_1 \dots s_k) \mid$
 $\quad x := \text{relabel}(e, p_f \text{ to } L_t \text{ using } s_0 \dots s_k)$
Sec. Labels($\mathbb{B}$) $p, q ::= L \mid \text{cnd? } p \diamond q$
Conditions $\text{cnd} ::= s \mid \widehat{s} \mid \text{cnd} \wedge \text{cnd} \mid \text{cnd} \vee \text{cnd} \mid \neg \text{cnd}$
Mut. Dir. $\diamond ::= \rightarrow_t \mid \leftrightarrow_t \mid \rightarrow_p \mid \leftrightarrow_p$

Fig. 2. Language syntax with security specification.

policy allows information revealed in the past to be released again regardless of the current policy. To do so, the security event trace also tracks a history of released information (Section 3.2).

Finally, we formally prove that the type system enforces the end-to-end dynamic release policy: for any well-typed program with annotated dynamic release labels, the knowledge gained from observing each program output does not exceed the “allowance” prescribed by the policy. A central technical challenge in this proof arises from the dynamic nature of release labels, which causes both the security lattice and the release-to relation to evolve during execution. Consequently, existing proof techniques (e.g., [33]) cannot be applied directly. We address these challenges and present a proof sketch in Section 5, with the full formal proof provided in [17].

3 Language

We first present a simple imperative language with its security specification, along with the operational semantics and the semantics on dynamic release labels.

3.1 Syntax

The language in Figure 2 provides standard features such as variables and expressions, along with commands such as branches, loops, sequential commands, and assignments. The remaining features are designed for security purposes:

- We assume a predefined lattice $\mathcal{L}$ consists of a set of static security levels. Here, $\mathcal{L}$ is an arbitrary security lattice, which can represent confidentiality levels, integrity levels, or a combination of both. For simplicity, though, we assume a confidentiality lattice throughout the paper. We further assume at least two levels in most examples: $\mathbf{P}$ for public and $\mathbf{S}$ for secret, while the language supports an arbitrary lattice. The assumption is general enough to support both Denning-style lattice [22] (e.g., with $\mathbf{P} \sqsubset \mathbf{S}$), and principal/role-based model [3, 23, 37] (e.g., with $\{\text{alice}, \text{bob}\} \sqsubset \{\text{alice}\}$, stating that information accessible to both Alice and Bob is less restrictive than information accessible to Alice only).
- In the simplest case, a security label $p \in \mathbb{B}$ is a security level taken from the security lattice. In general, a security label can be *dynamic*, as specified in the form of $\text{cnd? } p \diamond q$ where the condition cnd controls whether p or q takes effect, and the mutation $\diamond$ controls the direction and kind of the transition. In particular, there are three possible directions of mutation, $\rightarrow$, $\leftarrow$ and $\leftrightarrow$. $p \rightarrow q$ (resp. $p \leftarrow q$) specifies a one-time transition from p to q (resp. from q to p)

when its corresponding condition cnd evaluates to false (resp. true) the first time. Since $\text{cnd}?p \leftarrow q$ is semantically identical to $\neg \text{cnd}?q \rightarrow p$, our syntax omits the $\leftarrow$ form and treats it as syntactic sugar for the latter. Direction $\leftrightarrow$ means that the transition can happen in both directions depending on the value of cnd . Moreover, there are two types of policies: transient and persistent, specified with subscripts t and p respectively. The main difference between them is that a persistent policy always allows to reveal information that has been revealed in the past, while a transient policy does not [34].

To strike a balance between expressiveness and enforcement, we carefully designed a few language features in Figure 2, making it substantially different from the one presented in the dynamic release paper [34]:

- Rather than allowing the condition cnd to be an arbitrary program expression, we restrict it to a set of distinguished, predefined *security events* $s \in \mathbb{S}$. These security events can be viewed as special Boolean variables, distinct from ordinary program variables such as x, y, z , and may only be modified via the distinguished functions eventon and eventoff . This restriction enables tractable reasoning about event status within a type system. Besides user-defined security events above, the language also supports predicates over them, such as standard logical operations on security events and $\widehat{s}$, which holds when the security event s has never been set to true. Moreover, to enable reasoning about whether a variable's value has been released in the past, we introduce a new class of security events, denoted by $\mathbf{o}_x@L$, indicating that the variable x has previously been released at security level L .
- Rather than associating each output channel with a dynamic release label, we restrict output commands to fixed security levels, written as $\text{output}(L, e, \text{using } s_0 \dots s_k)$. As is standard, we assume that only observers at or above level L can access the value of e produced by this command. Furthermore, the expression e is revealed only if all security events in the sequence $s_0, \dots, s_k$ are enabled at the time the command is executed. Otherwise, the output command has no effect (i.e., it behaves as a nop). Consistent with prior works on dynamic policies [7, 20, 34], we assume that information is only released with output commands in the language; assignments to variables do not reveal any information directly.
- We introduce a `relabel` command that can also “peek” at the current status of security events and utilize more permissive information flow reasoning rules when possible. In particular, $x := \text{relabel}(e, p \text{ to } L \text{ using } s_0 \dots s_k)$ updates x to the value of e if the events $s_0 \dots s_k$ are all enabled at the time the command is executed. Otherwise, the command has no effect (i.e., it behaves as a nop). Note that the command is also annotated with a security label p , and a security level L ; the idea is that since information only flows from e to x when all security events $s_0 \dots s_k$ are enabled, a type system can rely on this assumption to reason more permissively about whether a flow from e to x is secure under this assumption.

For example, consider Lines 8–9 in Figure 1.i, where **release** is a security event that turns true after bidding is complete. At runtime, we know that **release** holds, and hence, the `relabel` command writes the value of `bid` to `wbid`. Staticly, the type system relies on the `using` annotations to determine which security events hold at each program point, which in this example is **release**. With the assumption, the dynamic label on `wbid`, which is $\neg \text{release}?S \rightarrow_t P$, can be leaked to **P** as the information is already declassified whenever **release** holds. Hence, the novel design of `relabel` command allows a static type system to soundly prove security without deep knowledge of the program state at each program point. We formalize and enforce such reasoning in our type system (Section 4.4).

One caveat is that the design also relies on a programmer, or an advanced program analysis, to properly annotate the clause after `using` to preserve the original semantics of a program. We leave

$$\begin{array}{c}
\text{OS-EVENTON} \qquad \qquad \qquad \text{OS-EVENTOFF} \\
\langle m, \text{eventon}(s), \sigma \rangle \rightarrow \langle m, \text{skip}, \sigma \cdot s \rangle \quad \langle m, \text{eventoff}(s), \sigma \rangle \rightarrow \langle m, \text{skip}, \sigma \cdot \neg s \rangle \\
\\
\text{OS-OUTPUT-SUCC} \\
\frac{\langle m, e \rangle \Downarrow v \quad \forall i \in 0 \dots k. \quad \text{eval}(s_i, \sigma) = \text{true} \quad \sigma' = \begin{cases} \sigma \cdot \mathbf{o}_x @ L, & e \text{ is a variable } x \\ \sigma, & \text{otherwise} \end{cases}}{\langle m, \text{output}(L, e, \text{using } s_0 \dots s_k), \sigma \rangle \xrightarrow{\langle L, v, \sigma \rangle} \langle m, \text{skip}, \sigma' \rangle} \\
\\
\text{OS-OUTPUT-FAIL} \\
\frac{\langle m, e \rangle \Downarrow v \quad \exists i \in 0 \dots k. \quad \text{eval}(s_i, \sigma) = \text{false}}{\langle m, \text{output}(L, e, \text{using } s_0 \dots s_k), \sigma \rangle \rightarrow \langle m, \text{skip}, \sigma \rangle} \\
\\
\text{OS-RELABEL-SUCC} \\
\frac{\langle m, e \rangle \Downarrow n \quad \forall i \in 0 \dots k. \quad \text{eval}(s_i, \sigma) = \text{true}}{\langle m, x := \text{relabel}(e, p_f \text{ to } L_t \text{ using } s_0 \dots s_k), \sigma \rangle \rightarrow \langle m[x := n], \text{skip}, \sigma \rangle} \\
\\
\text{OS-RELABEL-FAIL} \\
\frac{\exists i \in 0 \dots k. \quad \text{eval}(s_i, \sigma) = \text{false}}{\langle m, x := \text{relabel}(e, p_f \text{ to } L_t \text{ using } s_0 \dots s_k), \sigma \rangle \rightarrow \langle m, \text{skip}, \sigma \rangle}
\end{array}$$

Fig. 3. Small-Step Semantics of Selected Commands, which eval checks the value of s on trace σ .

this as orthogonal future work, as our primary goal of this work is to develop a core language with sound enforcement of dynamic release policies, assuming that an oracle is provided.

3.2 Semantics of Commands

We model a program state as memory, written as m : a mapping from program variables to their corresponding values. For an expression e , its big-step semantics is written as $\langle m, e \rangle \Downarrow n$, which means that e evaluates to value n under memory m . For commands, we use small-step semantics since the dynamic nature of information flow policy requires step-by-step details during a program execution. The small-step semantics is written as:

$$\langle m, c, \sigma \rangle \xrightarrow{\langle L, v, \sigma \rangle} \langle m', c', \sigma' \rangle$$

where σ and σ' are sequences of triggered security events (e.g., $s_1 \cdot \neg s_2 \cdot \mathbf{o}_x @ P$).

Each step emits a (potentially empty) output event $\langle L, v, \sigma \rangle$, which indicates the output of value v to an output channel at level L under the security event trace σ . We assume that all security events are initialized. By default, the condition is set to true in right-arrow and bi-directional policies. Only output commands reveal information to entities and, hence, generate output events, as formalized in Rule (OS-OUTPUT-SUCC) in Figure 3. Moreover, changes to security events are tracked in the security event trace, by rules (OS-EVENTON), (OS-EVENTOFF) and (OS-OUTPUT-SUCC).

It is worth noting that both the output and relabel commands are syntactically associated with using clauses. These using annotations specify sufficient conditions on security events under which the corresponding commands are considered secure. For example, the output command at Line 6 in Figure 1.i can reveal `wbid` to the public *only* when `release` is set. Accordingly, when these using clauses are satisfied, the corresponding commands behave identically to their standard semantics.

$$\begin{aligned}
\llbracket L \rrbracket_\sigma &= L \\
\llbracket \text{cnd}?p \rightarrow q \rrbracket_\sigma &= \begin{cases} \llbracket p \rrbracket_\sigma & \text{first}(\text{cnd}, \text{false}, \sigma) = -1 \\ \llbracket q \rrbracket_{\sigma[i:]} & i = \text{first}(\text{cnd}, \text{false}, \sigma) \end{cases} \\
\llbracket \text{cnd}?p \leftrightarrow q \rrbracket_\sigma &= \begin{cases} \llbracket p \rrbracket_{\sigma[i+1:]} & i = \text{last}(\text{cnd}, \text{false}, \sigma) \neq \|\sigma\| \\ \llbracket q \rrbracket_{\sigma[i+1:]} & i = \text{last}(\text{cnd}, \text{true}, \sigma) \neq \|\sigma\| \end{cases}
\end{aligned}$$

Fig. 4. Label semantics.

Otherwise (namely, when any security event appearing after the using keyword evaluates to false) the command instead behaves as a nop. For example, the output command at Line 6 in Figure 1.i produces no outputs due to Rule (OS-OUTPUT-FAIL), as **release** is not set at that point.

As we elaborate in Section 4.4, a type system can statically verify that these programmer-provided annotations are sufficient conditions for end-to-end security. Hence, the using annotations affect the observable behavior of the original program only in two cases: either the annotations are overly restrictive, preventing an otherwise secure command from executing, or the corresponding commands in the original program would violate the security policy (e.g., Line 6 in Figure 1.i) and must therefore be suppressed.

Finally, as introduced informally earlier, the **relabel** command has the same semantics as a normal assignment $x := e$ when all the security events' values are as expected; otherwise, the command is the same as a nop instruction. The semantics of other commands are standard and are included in [17].

One program execution under initial state $\langle m_0, c_0, \sigma_0 \rangle$ produces an *execution trace* τ as follows:

$$\langle m_0, c_0, \sigma_0 \rangle \xrightarrow{\langle L_0, v_0, \sigma_0 \rangle} \langle m_1, c_1, \sigma_1 \rangle \cdots \xrightarrow{\langle L_{n-1}, v_{n-1}, \sigma_{n-1} \rangle} \langle m_n, c_n, \sigma_n \rangle$$

We use $\langle m, c, \sigma \rangle \hookrightarrow \tau$ to denote that execution from $\langle m, c, \sigma \rangle$ produces an execution trace τ . We use $\tau^{[i]}$ to denote the configuration after the i -th evaluation step in τ , $\tau^{[i]}$ to denote a sub-trace from the initial point to the i -th evaluation step, and $\tau^{[i:j]}$ to denote a sub-trace between i -th and j -th evaluation steps. We use $\|\tau\|$ to denote the number of evaluation steps in the trace. Since σ is a sequence of events, we use similar annotation for its subsequences.

3.3 Semantics of Security Labels

To interpret dynamic release labels, we treat each label $\text{cnd}?p \diamond q$ as an “expression” and evaluate its value (i.e., a security level from a predefined lattice in our setting) at the end of a trace σ as shown in Figure 4. In the definition, taken from [34], the helper function **first** returns the first index of σ such that **cnd** evaluates to the second parameter, or -1 if no such index is found; and **last** returns the last index of σ such that **cnd** evaluates to the second parameter, or -1 if no such index is found. As for $\leftrightarrow$, the labels may transit an arbitrary number of times whenever the value of **cnd** changes, which is mainly used in the revocation dynamic policy.

Intuitively, $\llbracket p \rrbracket_\sigma$ computes the *least* restrictive security level that p can flow to at the end of σ . More specifically, for a static level L , its interpretation is simply L . For dynamic labels like $\text{cnd}?p \rightarrow q$, a one-time sensitivity change from p to q occurs when **cnd** first evaluates to false. Let i be the first index in σ such that **cnd** evaluates to false. $\llbracket \text{cnd}?p \rightarrow q \rrbracket_\sigma$ reduces to $\llbracket p \rrbracket_\sigma$ if no such i exists; otherwise, it reduces to $\llbracket q \rrbracket_{\sigma[i:]}$. Bi-directional labels are interpreted based on the last configuration of σ , where i is the last index at which **cnd** evaluates to false. Therefore, $i \neq \|\sigma\|$ implies that **cnd** evaluates to true at the end of σ , and the label reduces to p . The label p is then evaluated under $\sigma^{[i+1]}$, ensuring that any potentially nested conditions are handled properly.

The interpretation of the labels above specifies when and where information can be released to: information with label p can be released to entities at or above level $\{p\}_\sigma$ at the end of σ . However, it is not suitable for information flow enforcement, since it does not allow direct comparison of dynamic labels at each program point. For example, label $\neg\text{release?S} \rightarrow_t \text{P}$ (declassification policy) and label S both evaluate to S when the security event **release** evaluates to false under σ . However, information flow from S to $\neg\text{release?S} \rightarrow_t \text{P}$ is obviously insecure as if the flow happens, the former is effectively declassified once **release** becomes true.

To address the challenge above, we observe that for a secure flow from label p to q under an event trace σ , we need to examine all possible *extensions* of σ so that p is always bounded by p' in the future (i.e., $\forall \sigma \preceq \sigma', \{p\}_{\sigma'} \sqsubseteq \{p'\}_{\sigma'}$). Consequently, we write $p \sqsubseteq_\sigma p'$ to denote that p' is at least as restrictive as p at σ : that is, for all future extensions of σ , wherever information with label p' can be released, information with label p can also be released:

DEFINITION 1 (PARTIAL ORDERING ON DYNAMIC RELEASE LABELS).

$$\forall p, p', \sigma. p \sqsubseteq_\sigma p' \iff \forall \sigma'. (\sigma \preceq \sigma' \Rightarrow \{p\}_{\sigma'} \sqsubseteq \{p'\}_{\sigma'})$$

Note that with dynamic release labels, the ordering on labels is parameterized on a security event trace as the relation changes during program execution. This contrasts with standard noninterference, where the security lattice remains fixed throughout program execution. For example, consider two security event traces σ_1 and σ_2 ($\sigma_1 \preceq \sigma_2$), where **release** is always false in σ_1 , but it turned true in σ_2 . It is easy to check that $(\neg\text{release?S} \rightarrow \text{P} \not\sqsubseteq_{\sigma_1} \text{P})$ and $(\neg\text{release?S} \rightarrow \text{P} \sqsubseteq_{\sigma_2} \text{P})$, meaning that the declassification label cannot flow to P under σ_1 but it can flow to P under σ_2 . On the other hand, $(\text{S} \not\sqsubseteq_{\sigma_1} \neg\text{release?S} \rightarrow \text{P})$ and $(\text{S} \not\sqsubseteq_{\sigma_2} \neg\text{release?S} \rightarrow \text{P})$, meaning that S can never flow to information with the declassification label.

Persistent vs. Transient Label. Our partial ordering definition on dynamic release labels (Definition 1) does not distinguish between persistent and transient labels. This might be surprising as intuition might suggest that a persistent policy is less restrictive than its counterpart of a transient policy (i.e., $s?p \rightarrow_t q$ is strictly more restrictive than $s?p \rightarrow_p q$). However, we found this intuition to be incorrect, as the difference between persistent and transient labels only shows up in the end-to-end security condition of dynamic release, rather than where information can be released to at each program point. We will elaborate on their differences in the type system (Section 4.4).

3.4 Dynamic Release Policy

Dynamic release [34] is the first end-to-end information flow policy that generalizes multiple kinds of dynamic information flow policies. Its precise definition is a bit involved and we only present the essentials in this section to provide sufficient background information to understand how to soundly enforce it, which is the main contribution of this paper. We follow the notations in the original paper, other than a few minor changes for readability.

Informally, a program c satisfies dynamic release policy iff for any output t produced by c , the *knowledge* gained about any variable x by observing t is bounded by what's *allowed* by the policy of x at the program point that outputs t .

To formalize this, we first define “indistinguishable” memory set w.r.t. some memory m and a set of variables X as the set of memories that are equivalent to m on the values of all variables in X .

DEFINITION 2 (MEMORY CLOSURE). *Given a memory m and a set of variables X , the memory closure of m on X is the set of memories that agree with m on all variables in X :*

$$\llbracket m \rrbracket_X \triangleq \{m' \mid \forall x \in X. m(x) = m'(x)\}$$

Given a memory m and a security label p , we write $\llbracket m \rrbracket_{\neq p}$ as the set of memories that agree with m on all variables except those whose security label is p . That is, $\llbracket m \rrbracket_{\neq p} \triangleq \llbracket m \rrbracket_{\{x \mid \Gamma(x) \neq p\}}$. Intuitively, only the values of variables with label p may differ between m and any $m' \in \llbracket m \rrbracket_{\neq p}$.

In addition to memory, we need to utilize a filter function on output traces, in order to precisely capture what an attacker can learn from a trace: the filter function returns false for irrelevant outputs on a trace. With any trace filter f , we define a trace projection as follows:

DEFINITION 3 (PROJECTION OF TRACE FILTER).

$$\lfloor \tau \rfloor_f \triangleq \langle \langle l, v, \sigma \rangle \in \tau \mid f(l, v, \sigma) \rangle$$

To formalize the attacker's observation, we define a trace projection up to level L , $\lfloor \tau \rfloor_L \triangleq \lfloor \tau \rfloor_{\lambda l, v, \sigma. l \leq L}$ that takes in the attacker's level L and returns a sub-trace of output events that are visible to the attacker.

Attacker's Knowledge from an Output Trace. Consider an attacker at security level L . The attacker is able to observe all output events $\langle L', v, \sigma \rangle$ such that $L' \subseteq L$. Hence, two traces τ_1 and τ_2 are *indistinguishable* to the attacker, written as $\tau_1 \sim_L \tau_2$, if they agree on all outputs at or below attacker's level L . More formally,

DEFINITION 4 (INDISTINGUISHABILITY).

$$\sim_L \triangleq \{(\tau_1, \tau_2) \mid \lfloor \tau_1 \rfloor_L \preceq \lfloor \tau_2 \rfloor_L\}$$

Therefore, the “knowledge” gained by observing an output trace τ produced by a program c is defined as all initial memory states that can produce an indistinguishable trace from τ :

DEFINITION 5 (KNOWLEDGE GAINED FROM OUTPUT TRACE).

$$k_1(c, \tau, L) \triangleq \{m \mid \langle m, c, \emptyset \rangle \hookrightarrow \tau' \wedge \tau \sim_L \tau'\}$$

Intuitively, the knowledge set consists of all initial memory states that remain possible given the portion of the execution trace observable to the attacker. Moreover, the smaller the knowledge set, the more information the attacker has learned from the program's execution.

By definition, the knowledge set monotonically decreases as additional events are produced along the execution trace, making it well suited for formalizing declassification policies [7]. In contrast, it is inadequate for erasure policies, which require reasoning about increases in uncertainty, or equivalently, expanding the knowledge set. Consider the credit-card example in Figure 1-ii. After the first output, the attacker learns the credit card number, so the knowledge set contains only those initial memories with the actual credit card number. After Line 7, however, the credit card number should no longer be inferable. Accordingly, the knowledge set should again include all possible credit card numbers, reflecting the restored uncertainty required by the erasure policy. To accommodate both kinds of dynamic policies within a unified framework, Dynamic Release reasons about the knowledge gained from each *individual* output event, rather than from the entire execution trace. We introduce this notion next.

Attacker's Knowledge from the Last Event. While the attacker's security level remains fixed throughout program execution, the meaning of a dynamic label may change over time. To precisely define what can be revealed at each program point, a major challenge solved by [34], we begin with one key observation: events produced by two program executions are not, in general, aligned according to their positions in the corresponding execution traces. The reason is that due to downgrading, the publicly observable outputs of different executions may have different lengths (Observation 2 in [34]). To address this issue, we introduce the *secret projection* of an execution trace, parameterized by an attacker at security level L and a secret labeled p . The secret projection

retains only the *effective outputs*, namely, those outputs produced while information labeled p is considered secret from an attacker at level L .

DEFINITION 6 (SECRET PROJECTION OF TRACE). *Given a policy p and an attacker at level L , the secret projection of a trace is the subtrace containing outputs visible at level L that occur while p remains secret from an attacker at level L :*

$$\lfloor \tau \rfloor_{p,L} \triangleq \lfloor \tau \rfloor_{\lambda l, v, \sigma. l \sqsubseteq L \wedge \llbracket p \rrbracket_{\sigma} \not\sqsubseteq L}$$

Using effective outputs, we introduce a notion of trace consistency. Two execution traces, τ_1 and τ_2 , are said to be *consistent* with respect to an attacker at security level L and a secret labeled p if their effective outputs (1) have the same length, and (2) end with the same output event.

DEFINITION 7 (CONSISTENCY). *Two output sequences τ_1 and τ_2 are consistent w.r.t. a policy p and an attack level L , written as $\tau_1 \equiv_{p,L} \tau_2$ if*

$$n = \|\lfloor \tau_1 \rfloor_{p,L}\| = \|\lfloor \tau_2 \rfloor_{p,L}\| \wedge \lfloor \tau_1 \rfloor_{p,L}^{[n]} = \lfloor \tau_2 \rfloor_{p,L}^{[n]}$$

With the above definitions, we can build the definition of the knowledge gained by the attacker at level L by observing the last event.

DEFINITION 8 (ATTACKER'S KNOWLEDGE GAINED FROM THE LAST EVENT). *For an attacker at level L , the attacker's knowledge w.r.t. information with policy p , after observing the last event of an output sequence traces program c , is the set of all initial memories that produce an output sequence that is indistinguishable to some consistent counterpart of the trace τ :*

$$k_2(c, \tau, L, p) = \bigcup_{\exists m', \sigma, j. \langle m', c, \sigma \rangle \hookrightarrow \tau' \wedge \tau'^{[j]} \equiv_{p,L} \tau} k_1(c, \tau'^{[j]}, L)$$

Finally, we define dynamic release security. For every output event produced during program execution, the attacker's knowledge gained from observing that output (Definition 8) must be bounded by the policy allowance at the corresponding program point. For a transient policy, the policy allowance is simply $\llbracket m \rrbracket_{\#p}$, where only the values of variables labeled p are treated as secret. For a persistent policy, the allowance is instead defined as $\llbracket m \rrbracket_{\#p} \cap k_1(c, \tau^{[i-1]}, L)$, thereby permitting the attacker to retain any knowledge acquired through previous outputs while still preventing any additional information beyond what the policy allows from being learned.

DEFINITION 9 (DYNAMIC RELEASE).

$$\forall m, L \subseteq \mathbb{L}, p \in \mathbb{B}, \sigma, \tau. \langle m, c, \sigma \rangle \hookrightarrow \tau \Rightarrow \forall 1 \leq i \leq \|\tau\|.$$

$$k_2(c, \tau^{[i]}, L, p) \supseteq \begin{cases} \llbracket m \rrbracket_{\#p}, & \text{transient} \\ \llbracket m \rrbracket_{\#p} \cap k_1(c, \tau^{[i-1]}, L), & \text{persistent} \end{cases}$$

4 Type System

To enforce dynamic release policy, we develop a static type system to control information flow at compile time. One challenge of doing so is to achieve a balance between soundness and permissiveness, given the fact that the meaning of each label can change throughout program execution. To address this challenge, we first introduce two static relations, each parameterized by S , a set of statically computed facts about security events that must hold at each program point. The relations approximate when information can securely flow between dynamic labels, and be released respectively at compile time. Built on them, we define a type system that enforces the end-to-end dynamic release policy on the whole program.

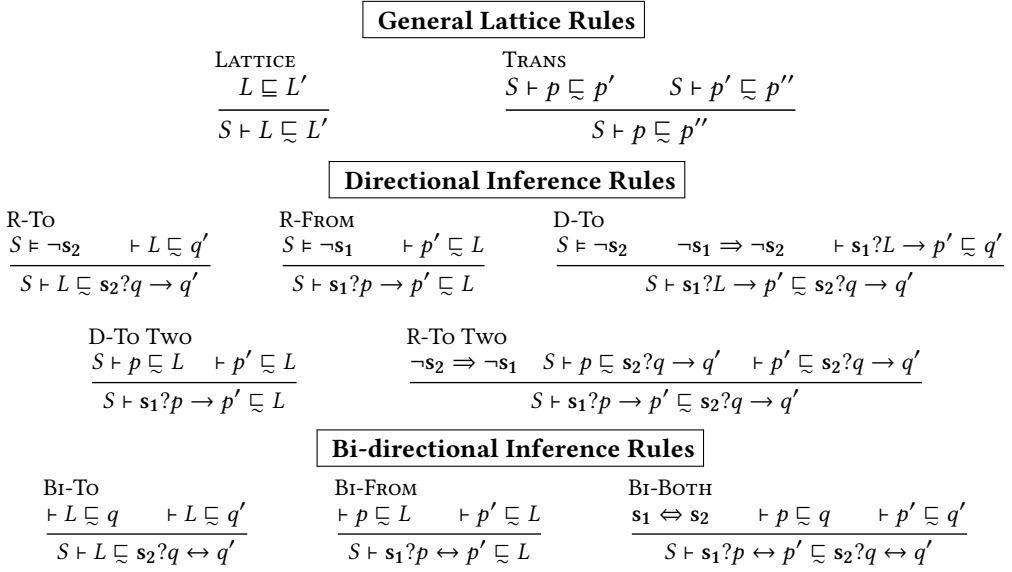


Fig. 5. Selected Static Rules for “Flows-to” Relation, which $S \models s_1$ denotes that s_1 must have been true given S .

4.1 Oracle on Security Events

Since the interpretation of dynamic release labels relies on a security event trace σ , the type system inevitably requires knowledge of the existence or absence of security events up to some program point. Moreover, for a persistent policy, dynamic release also depends on the outputs in the past (Definition 9). To decouple the reasoning of security events and outputs from information flow enforcement, we follow a simple design where the type system only uses the facts established by the using keyword in the output and relabel commands. For example, a set of facts $S = \{s_1, \neg s_1, \neg s_2, \widehat{s_3}, o_x@P\}$ states that up to the program point of interest, s_1 must have been both true and false, s_2 must have been false (and may be true), $\widehat{s_3}$ has never been true, and the value of x has been released to level L already. These facts are checked by commands such as `output( $P, x$ , using  $s_1, \neg s_1, \neg s_2, \widehat{s_3}, o_x@P$ )` when any condition evaluates to false, the command behaves like a nop command (Section 3.2). In general, we can also estimate S statically, such as using dataflow analysis [2], which we leave as future work. Hereafter, we use $S \models s_2$ to indicate that s_2 must have been true given S , and $S \models \neg s_2$ to indicate that s_2 must never have been true given S .

4.2 Static Rules for Flows-to Relation

We first define a *flows-to* relation on dynamic labels, parameterized by a set of security events S , such as $\{s_1, \neg s_2\}$. Intuitively, the flows-to relation $S \vdash p \sqsubseteq q$ holds if label p is less restrictive than, or equivalent to q for any event trace σ that satisfies S and its extensions. A selection of rules for the flows-to relation is summarized in Figure 5; all rules are included in [17].

Base Cases. Rules (LATTICE) and (TRANS) are general rules based on Denning-style security lattice and the rule of transitivity. These rules are the same as a typical static information flow system.

Directional Cases. Rule (R-To) compares a static level L against a dynamic label $s_2?q \rightarrow q'$ under the assumption that $\neg s_2$ evaluates to true (i.e., s_2 evaluates to false). Hence, we can simplify

the dynamic label to q' , as the transition to q' must have happened. Rule (R-FROM) is symmetric to Rule (R-TO). Note that in both rules, we intentionally omit the event facts S before $\vdash$ from the assumptions to ensure soundness. The reason is that, with nested conditions, events in S may be triggered before the top-level condition is evaluated. Consequently, propagating them to the more deeply nested sub-label may be unsound.

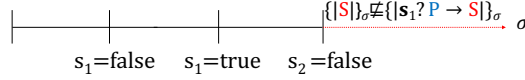
The general case when both ends of flows-to relation involve dynamic labels is a bit more complicated. To illustrate the subtlety, we consider a naive but *incorrect* extension to Rule (R-TO)

$$\frac{S \models \neg s_2 \quad \vdash s_1?p \rightarrow p' \sqsubseteq q'}{S \vdash s_1?p \rightarrow p' \sqsubseteq s_2?q \rightarrow q'}$$

At first glance, this rule is legit as it follows the same argument as (R-TO): it only reduces $s_2?q \rightarrow q'$ when s_2 turns false. However, this rule is unsound when q' has nested conditions. To see why, consider the following counterexample:

$$s_1?P \rightarrow S \sqsubseteq s_2?P \rightarrow (s_1?P \rightarrow S)$$

and a trace σ where s_1 is set to false and then to true in sequence. After that, s_2 turns false, as illustrated below.



At the end of σ , $\neg s_2$ holds. Hence, we can instantiate S with $\neg s_2$ in the proposed rule, and derive

$$\frac{\neg s_2 \models \neg s_2 \quad \vdash s_1?P \rightarrow S \sqsubseteq s_1?P \rightarrow S}{\neg s_2 \vdash s_1?P \rightarrow S \sqsubseteq s_2?P \rightarrow (s_1?P \rightarrow S)}$$

where the assumption that $\vdash s_1?P \rightarrow S \sqsubseteq s_1?P \rightarrow S$ holds trivially as both sides are identical.

However, we next show that the derived flow-to relation is unsound. By the semantics, $\{s_1?P \rightarrow S\}_\sigma$ reduces to S as s_1 turned to false on the trace. Moreover, $\{s_2?P \rightarrow (s_1?P \rightarrow S)\}_\sigma$ reduces to $\{s_1?P \rightarrow S\}_{\sigma[i]}$ where i is the first index when s_2 turned to false. Note that even though s_1 was false on the whole trace σ , the nested condition s_1 is evaluated on the sub-trace *after* the top-level condition s_2 is triggered (Figure 4). On that sub-trace $\sigma^{[i]}$, s_1 evaluates to true. Hence, $\{s_1?P \rightarrow S\}_{\sigma^{[i]}}$ further reduces to P . Hence, $\{s_1?P \rightarrow S\}_\sigma = S \not\sqsubseteq P = \{s_2?P \rightarrow (s_1?P \rightarrow S)\}_\sigma$, indicating that $s_1?P \rightarrow S$ is not always bounded by $s_2?P \rightarrow (s_1?P \rightarrow S)$ at σ (i.e., $s_1?P \rightarrow S \not\sqsubseteq_\sigma s_2?P \rightarrow (s_1?P \rightarrow S)$, Definition 1). A similar issue occurs when p contains nested conditions.

To ensure soundness while permitting permissiveness when possible, we observe that the soundness issue above can be avoided if p has no nested conditions (i.e., it is a level L), and s_1 is never false before s_2 turns false, which can be captured as $(\neg s_1 \Rightarrow \neg s_2)$, where $\Rightarrow$ is a logical implication. The general case is included in the sound derivation rule (D-TO) in Figure 5. The symmetric version Rule (D-FROM) is included in [17].

When the permissive rules (D-TO) and (D-FROM) are not applicable, rule (R-TO Two) conservatively checks that $s_1?p \rightarrow p'$ can flow to some level L . For the general case of $s_1?p \rightarrow p' \sqsubseteq s_2?q \rightarrow q'$, similar to Rule (D-TO), Rule (D-TO Two) assumes that $(\neg s_2 \Rightarrow \neg s_1)$ to be sound.

Bi-Directional Cases. Finally, we introduce the static rules for bi-directional policies, which align with the design of $\rightarrow$. Bi-directional policies only have the less permissive rules as the condition can change throughout trace extensions, and it is hard to define rules without binding to both ends. Rules (BI-TO) and (BI-FROM) correspond to the cases where one side is a plain static level L in the single-direction cases, checking bounds for both parts of the dynamic policy. Rule (BI-BOTH) assumes that the dynamic labels on both sides have equivalent conditions, and then checks the “flows-to” relation on the corresponding components.

LATTICE	TRANS	R-FROM-P	R-FROM-F	BI-FROM
$\frac{L \sqsubseteq L'}{S \vdash L \lesssim L'}$	$\frac{S \vdash L \lesssim L' \quad S \vdash L' \lesssim L''}{S \vdash L \lesssim L''}$	$\frac{S \models \neg s_1 \quad \vdash p' \lesssim L}{S \vdash s_1?p \rightarrow p' \lesssim L}$	$\frac{S \models \neg \widehat{s_1} \quad S \vdash p \lesssim L}{S \vdash s_1?p \rightarrow p' \lesssim L}$	$\frac{\vdash p \lesssim L \quad \vdash p' \lesssim L}{S \vdash s_1?p \leftrightarrow p' \lesssim L}$

Fig. 6. Static Rules for “Release-to” Relation.

Next, we establish the soundness of the static inference rules: for any labels p, q and a set of security events S , $S \vdash p \sqsubseteq q$ implies that q is more restrictive or equal to p on any event trace σ where S holds, as well as on any extension of σ .

THEOREM 1 (SOUNDNESS OF FLOWS-TO RELATION). *For all policies p and q , a trace σ , a set of security events $S = s_0 \dots s_k$, if $S \vdash p \sqsubseteq q$ and S hold on σ , then $p \sqsubseteq_\sigma q$.*

Proof sketch. We proceed by induction on the static inference rules. We present the proof for Cases (R-To) and (D-To) here; the complete proof is available in [17].

By Definition 1, it is sufficient to show that for any σ' such that $\sigma \preceq \sigma'$, we have $\{p\}_{\sigma'} \sqsubseteq \{q\}_{\sigma'}$.

- (R-To): By the assumption, we have $\vdash L \sqsubseteq q'$ and $S \models \neg s_2$. Since S holds on σ by assumption, $\sigma \preceq \sigma'$, and $S \models \neg s_2$, there must be some index i on σ' at which s_2 first turns false. According to the semantics of Figure 4, we know that $\{s_2?q \rightarrow q'\}_{\sigma'} = \{q'\}_{\sigma'[i]}$.

From the induction hypothesis and the assumption that $\vdash L \sqsubseteq q'$, we get $\{L\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]}$. Hence, $\{L\}_{\sigma'} = L = \{L\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]} = \{s_2?q \rightarrow q'\}_{\sigma'}$.

- (D-To): Similar to (R-To), we first derive $\{s_2?q \rightarrow q'\}_{\sigma'} = \{q'\}_{\sigma'[i]}$ from the assumptions, where i is the index when s_2 first turns false.

If s_1 has been false on σ' , let j be the index when it turns false the first time. Since $\neg s_1 \Rightarrow \neg s_2$, it is guaranteed that $j \geq i$. According to the semantics of Figure 4, we know that $\{s_1?L \rightarrow p'\}_{\sigma'} = \{p'\}_{\sigma'[j]} = \{s_1?L \rightarrow p'\}_{\sigma'[i]}$ where the second equation holds as s_1 must remain true between i and j . Moreover, from the induction hypothesis and the assumption $\vdash s_1?L \rightarrow p' \sqsubseteq q'$, we know that $\{s_1?L \rightarrow p'\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]}$. Hence $\{s_1?L \rightarrow p'\}_{\sigma'} = \{s_1?L \rightarrow p'\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]} = \{s_2?q \rightarrow q'\}_{\sigma'}$.

If s_1 has never been false on σ' , we know that $\{s_1?L \rightarrow p'\}_{\sigma'} = L = \{s_1?L \rightarrow p'\}_{\sigma'[i]}$. From the induction hypothesis and the assumption $\vdash s_1?L \rightarrow p' \sqsubseteq q'$, we know that $\{s_1?L \rightarrow p'\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]}$. Hence $\{s_1?L \rightarrow p'\}_{\sigma'} = \{s_1?L \rightarrow p'\}_{\sigma'[i]} \sqsubseteq \{q'\}_{\sigma'[i]} = \{s_2?q \rightarrow q'\}_{\sigma'}$. $\square$

4.3 Static Rules for Release-to Relation

Dynamic release policy controls information release at output commands. More formally, for each output command $\text{output}(L, e, \text{using } s_0 \dots s_k)$, information with label p can be released when $\{p\}_{\sigma} \sqsubseteq L$ (Figure 4). Accordingly, we define a release-to relation, written as $S \vdash p \lesssim L$, to statically check if we can safely release information with label p to output command at level L under the assumption that S holds when the output occurs. Compared with the flows-to relation $\sqsubseteq$, the release-to relation is simpler as (1) the target (i.e. the right-hand side) is always a security level by design, and (2) we do not need to consider any changes to p after the release.

Both rules (R-FROM-P) and (BI-FROM) are similar to their counterparts in the flows-to relation. The key difference is in rule (R-FROM-F): it reduces the dynamic label to p whenever $S \models \neg \widehat{s_1}$. This is sound for the release-to relation as by definition (Figure 4), how the interpretation of where information can be released to is only determined by the current event trace. But for the flows-to relation, we have to consider all extensions to the current trace, which may violate the assumption.

$\frac{\text{SKIP}}{pc, \Gamma \vdash \text{skip}}$	$\frac{\text{SEQUENCE} \quad pc, \Gamma \vdash c_1 \quad pc, \Gamma \vdash c_2}{pc, \Gamma \vdash c_1; c_2}$	$\frac{\text{IF} \quad \Gamma \vdash e : pc' \quad pc', \Gamma \vdash c_1 \quad pc', \Gamma \vdash c_2 \quad \vdash pc \sqsubseteq pc'}{pc, \Gamma \vdash \text{if } (e) \text{ then } c_1 \text{ else } c_2}$
$\frac{\text{ASSIGN} \quad \vdash pc \sqsubseteq \Gamma(x) \quad \Gamma \vdash e : \Gamma(x)}{pc, \Gamma \vdash x := e}$	$\frac{\text{WHILE} \quad \Gamma \vdash e : pc' \quad pc', \Gamma \vdash c \quad \vdash pc \sqsubseteq pc'}{pc, \Gamma \vdash \text{while } (e) \text{ do } c}$	
$\frac{\text{OUTPUT} \quad \Gamma \vdash e : p \quad s_0 \dots s_k \vdash p \lesssim L \quad \vdash pc \sqsubseteq L}{pc, \Gamma \vdash \text{output}(L, e, \text{using } s_0 \dots s_k)}$	$\frac{\text{OUTPUT-PER} \quad x \text{ immutable} \quad \Gamma(x) \text{ is persistent} \quad \vdash pc \sqsubseteq L}{pc, \Gamma \vdash \text{output}(L, x, \text{using } o_x @ L, s_0 \dots s_k)}$	
$\frac{\text{RELABEL} \quad \Gamma \vdash e : p_f \quad \vdash pc \sqsubseteq \Gamma(x) \quad s_0 \dots s_k \vdash L_t \sqsubseteq \Gamma(x) \quad s_0 \dots s_k \vdash p_f \sqsubseteq L_t}{pc, \Gamma \vdash x := \text{relabel}(e, p_f \text{ to } L_t \text{ using } s_0 \dots s_k)}$	$\frac{\text{EVENTON}(s) \quad \vdash pc \sqsubseteq \perp}{pc, \Gamma \vdash \text{EventOn}(s)}$	$\frac{\text{EVENTOFF}(s) \quad \vdash pc \sqsubseteq \perp}{pc, \Gamma \vdash \text{EventOff}(s)}$

Fig. 7. Typing Rules for Commands.

THEOREM 2 (SOUNDNESS OF RELEASE-TO RELATION). *For all policies p , a trace σ , a set of security events $S = s_0 \dots s_k$, if $S \vdash p \lesssim L$ and S holds on σ , then $\llbracket p \rrbracket_\sigma \sqsubseteq L$.*

The proof is similar to the proof of Theorem 1. The full proof is included in [17].

4.4 Typing Rules

Lastly, we present a type system that enforces the end-to-end dynamic release policy on the whole program. The overall structure of the typing rules is similar to other information flow type systems [19, 20, 43], where each command is checked under two conditions pc and Γ . Here, pc is the standard program counter label used to control implicit information flows [43], and Γ is a mapping that assigns each variable to its security label. The typing rules on expressions, in the form of $\Gamma \vdash e : p$, are completely standard, which is included in [17].

Several typing rules for commands are standard. Rule (skip) denotes that a nop command is always secure. Rule (SEQUENCE) enforces that both commands in a sequence are typed under the same pc label. Rule (IF) guards both branches of a conditional with a program counter pc' , which must be at least as restrictive as the condition's label, in order to control implicit flows. Rule (WHILE) is similar to Rule (IF). Rule (ASSIGN) requires that the policy on x is at least as restrictive as that on e to control explicit information flow from e to x .

Other rules are novel and carefully designed to check the dynamic release policy. Rule (RELABEL) is the first rule that utilizes annotated events $s_0 \dots s_k$ to check information flow from e to x in a more permissive way than Rule (ASSIGN). It allows information to be “relabelled” when possible.

Example 1: Bidding Game (Figure 1.i). We examine the relabel command on Line 9, which occurs after **release** is set to true. This line type-checks under rule (RELABEL) as follows (we omit less interesting parts of the proof for simplicity):

$$\frac{\Gamma \vdash \text{bid} : \neg\text{release?S} \rightarrow_t P \quad \frac{\text{release} \models \text{release} \quad \vdash P \sqsubseteq P}{\text{release} \vdash \neg\text{release?S} \rightarrow_t P \sqsubseteq P}}{P, \Gamma \vdash \text{wbid} := \text{relabel}(\text{bid}, \neg\text{release?S} \rightarrow_t P \text{ to } P \text{ using release})}$$

Recall that at runtime, `relabel` checks if all events after using `hold`; it turns the command into a `nop` if any assumed event is not correct (Figure 3). Hence, the typing rule can safely use event `release` to check if a flow from dynamic label $\neg\text{release?S} \rightarrow_t \text{P}$ to P is secure, via the permissive Rule (R-FROM) in Figure 5.

Rule (OUTPUT) is the second typing rule that utilizes annotated events $s_0 \dots s_k$ to check if the value of e can be released in a permissive way.

Example 2: Credit Card (Figure 1.ii). Both variables `copy` and `credit_card` are labeled with dynamic labels $\text{trans?M} \rightarrow_t \top$. The output in Line 6 can be type checked by the rule (OUTPUT), which allows the value of `copy` to be released as $\neg\text{trans}$ at the point:

$$\frac{\Gamma \vdash \text{copy} : \neg\text{trans?M} \rightarrow_t \top \quad \frac{\overline{\text{trans}} \models \overline{\text{trans}} \quad \vdash \text{M} \lesssim \text{M}}{\overline{\text{trans}} \vdash \neg\text{trans?M} \rightarrow_t \top \lesssim \text{M}}}{\text{M}, \Gamma \vdash \text{output}(\text{M}, \text{copy}, \text{using } \overline{\text{trans}})}$$

Recall that at runtime, the output command checks if all events after using are correct; it turns the command into a `nop` if any assumed event is not correct (Figure 3). Hence, the typing rule can use the fact that trans has never been true (i.e., $\overline{\text{trans}}$) to check if dynamic label $\neg\text{trans?M} \rightarrow_t \top$ can be released to M , via Rule (R-FROM-F) in Figure 6.

On the other hand, the same output command at Line 9, after the transaction is completed, is insecure. Although the command can still be type checked, we note that since the transaction is completed, the assumed event $\overline{\text{trans}}$ is no longer correct. Hence, nothing will be released at runtime (Rule (OS-OUTPUT-FAIL) in Figure 3).

Rule (OUTPUT-PER) is a more permissive version of Rule (OUTPUT), in the sense that it allows the value of x to be released *again* when x is immutable and $\Gamma(x)$ has a persistent policy. Here, the output event $\mathbf{o}_x$ assumes that x has already been released before the current output command.

Example 3: Library System (Figure 1.iii). Before Alice returns the book, the variable `notes` was released in Line 5, which can be type-checked in the same way as in Line 6 of the credit card example described above. According to the semantics of the output command, the output event $\mathbf{o}_{\text{notes}}$ is added to the execution trace at that point. After Alice returns the book on Line 6, the same value that was released on Line 5 can still be safely released again on Line 10 as the policy on `notes` and `book` is a persistent policy. This can be confirmed by the following proof tree:

$$\frac{\text{notes immutable} \quad \Gamma(\text{notes}) \text{ is persistent}}{\text{P}, \Gamma \vdash \text{output}(\text{P}, \text{notes}, \text{using } \mathbf{o}_{\text{notes}}@P)}$$

On the other hand, if we replace the same line with `output(P, book, using  $\mathbf{o}_{\text{book}}$ )`, the command is insecure as `book` was never released before it was returned. In this case, the insecure command is still type-checked. But since the assumed event $\mathbf{o}_{\text{book}}$ is no longer correct, nothing will be released at runtime (Rule (OS-OUTPUT-FAIL) in Figure 3).

Finally, Rules (EVENTON(s)) and (EVENTOFF(s)) handle distinguished commands that set/unset security events. For both of them, we require that the pc label be public (i.e., at the lowest level P in the lattice). The reason is to avoid label channels, where secret information interferes with the sensitivity level of variables [20].

5 Soundness Proof

The hallmark of information flow security is its ability to maintain end-to-end confidentiality and integrity across all system executions. For noninterference with a static policy, various proof techniques (e.g., [33, 40, 49]) have been developed to prove that static type systems can soundly enforce noninterference policy.

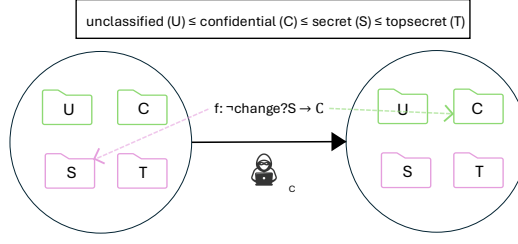


Fig. 8. An example in a multi-level military system, where the variable f belongs to levels before and after the security event **change** is set to true.

In those standard noninterference proofs, all variables' sensitivity/integrity are assumed to be fixed throughout program execution. Hence, for any attacker at a certain security level L , we can statically partition all variables into two sets: public set $\{x \mid \Gamma(x) \sqsubseteq L\}$ and secret set $\{x \mid \Gamma(x) \not\sqsubseteq L\}$. At a high level, standard noninterference proofs are built on an *indistinguishability relation* on memory states: two memory states m_1 and m_2 are said to be indistinguishable for an attacker at L when they agree on the public set of variables:

$$m_1 \approx m_2 \triangleq \forall x. \Gamma(x) \sqsubseteq L \Rightarrow m_1(x) = m_2(x)$$

Then, a standard proof proceeds by showing that the indistinguishability relation is an invariance on any two program executions with indistinguishable initial memory states.

To see the invariance more clearly, we consider a military-grade security system that includes four security levels: unclassified (U), confidential (C), secret (S), and top secret (T) in Figure 8. Here, an attacker with a confidential security clearance can access only unclassified and confidential files. Hence, two memory states are considered indistinguishable if they agree on $\{x \mid \Gamma(x) = U \vee \Gamma(x) = C\}$. Note that for a static policy, the set remains unchanged throughout program execution.

However, in the presence of dynamic labels, such a proof strategy no longer applies, as the set of secret variables can change throughout program execution. Suppose that there is a variable f associated with dynamic release label $\neg\text{change}?S \rightarrow_t C$. A naive extension of the indistinguishability relation to the dynamic setting is as follows:

$$m_1 \approx_\sigma m_2 \triangleq \forall x. \{\Gamma(x)\}_\sigma \sqsubseteq C \Rightarrow m_1(x) = m_2(x)$$

However, maintaining such an indistinguishability relation is impossible. Let σ and σ' be traces right before and after **change** is set to true in Figure 8. Given $m_1 \approx_\sigma m_2$, $m_1(f)$ and $m_2(f)$ might hold different values since $\{\Gamma(f)\}_\sigma \not\sqsubseteq C$. Hence, we are unable to prove $m_1 \approx_{\sigma'} m_2$ which requires that $m_1(f) = m_2(f)$ as $\{\Gamma(f)\}_{\sigma'} \sqsubseteq C$.

To address the challenge, in addition to the attacker level L , we also introduce a distinguished, arbitrary dynamic release label B and prove that the desired property of dynamic release (Definition 9) holds against any attacker's level L and source's label $B \in \mathbb{B}$. Intuitively, B is a distinguished *dynamic* label denoting the source label that is considered "sensitive" per security reasoning. With a fixed attacker level L and a fixed source label B , indistinguishability in memory is defined as

DEFINITION 10. Two memories m_1 and m_2 are indistinguishable under an event trace σ if they agree on all variables such that $B \not\sqsubseteq_\sigma \Gamma(x)$:

$$m_1 \approx_\sigma m_2 \iff \forall x. B \not\sqsubseteq_\sigma \Gamma(x) \Rightarrow m_1(x) = m_2(x)$$

Recall that by Definition 1, $p \sqsubseteq_\sigma p'$ denotes that $\{p\}_{\sigma'} \sqsubseteq \{p'\}_{\sigma'}$ for any extension σ' of σ .

A key insight here is that we can maintain the invariance since the size of $\{x \mid B \not\sqsubseteq_\sigma \Gamma(x)\}$ can never increase through program execution due to the following fact.

LEMMA 1 (TRACE PRESERVATION).

$$\forall p, p', \sigma. p \sqsubseteq_{\sigma} p' \Rightarrow (\forall \sigma'. \sigma \preceq \sigma' \Rightarrow p \sqsubseteq_{\sigma'} p')$$

The second challenge is that given an attacker's level L and the source label B , information leakage is allowed under some circumstances. Recall that under Definition 8, the consistency relation on traces ignores all outputs produced when $\{B\}_{\sigma} \sqsubseteq L$ (i.e., when information with B can be released to the attacker at L). Moreover, when B is a persistent policy, dynamic release allows information leaked in the past to be revealed again (Definition 9). Accordingly, in a few lemmas and theorems in the soundness proof, we added extra conditions under which intentional leakage is allowed. For persistent policy, note that the type system conservatively allows leakage when the value of x has been revealed previously (by checking $\mathbf{o}_x$ under using) and that x is immutable (Rule (OUTPUT-PER) in Figure 7). Hence, the relaxing condition is that the command c being executed can be type-checked by Rule (OUTPUT-PER).

The formal soundness proof follows a previous small-step noninterference proof [33], with notable changes as stated above. In the concrete proof, we introduce an augmented semantics where the “tainted” values are wrapped in brackets, written as $[n]$ instead of n . Moreover, instructions executed under “tainted” branch conditions (i.e., branch conditions that evaluate to bracketed values) are also wrapped in brackets, written as $[c]$ instead of c . A memory is said to be “well-formed” if all bracketed values are only stored in variables that are considered sensitive to an attacker at level L .

The full soundness proof, including definition of the augmented language, bracketed values and extended semantics, is provided in [17]. Next, we highlight important results and a proof sketch.

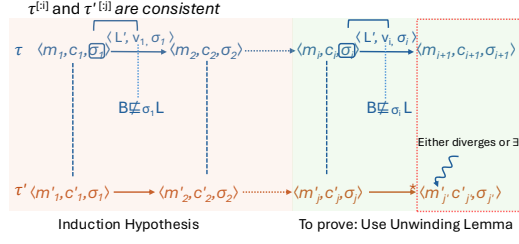
We consider any attacker's level L and source label B in the proof. The first important lemma states that indistinguishability in memory (Definition 10) is preserved in each step, as long as starting memories m_1 and m_2 are indistinguishable and each is well-formed ($\sigma \models m_1$ and $\sigma \models m_2$), c_1 and c_2 both type-check and are equivalent ($c_1 \approx c_2$), and the relaxing conditions are *not* met (i.e., $\{B\}_{\sigma} \not\sqsubseteq L$ and both c_1 and c_2 type-check without using the relax rule (OUTPUT-PER)). Note that the Lemma also allows c_2 to diverge, making the dynamic release assurance termination-insensitive [5, 7].

LEMMA 2 (UNWINDING).

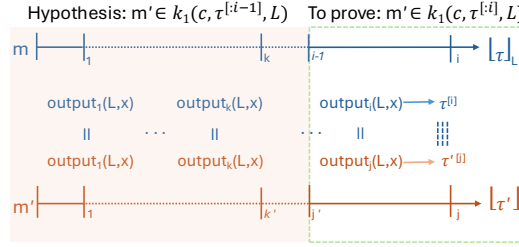
$$\begin{aligned} & \forall m_1, m_2, c_1, c_2, \tau_1, \tau_2, \sigma, x, pc, \Gamma. \sigma \models m_1 \wedge \sigma \models m_2 \wedge m_1 \approx_{\sigma} m_2 \wedge \\ & (c_1 \neq \text{output}(L', x, \text{using } \mathbf{o}_x, s_0 \dots s_k)) \wedge \{B\}_{\sigma} \not\sqsubseteq L \wedge c_1 \approx c_2 \wedge \langle m_1, c_1, \sigma \rangle \xrightarrow{\tau_1} \langle m'_1, c'_1, \sigma' \rangle \Rightarrow \\ & (\exists m'_2, c'_2, \tau_2. \langle m_2, c_2, \sigma \rangle \xrightarrow{\tau_2}^* \langle m'_2, c'_2, \sigma' \rangle \wedge m'_1 \approx_{\sigma'} m'_2 \wedge c'_1 \approx c'_2 \wedge \lfloor \tau_1 \rfloor_L = \lfloor \tau_2 \rfloor_L) \vee (\exists c. c_2 = [c] \text{ and } c \text{ diverges}) \end{aligned}$$

Proof sketch. We sketch the proof of the most interesting case with Rule (OUTPUT) here.

- (OUTPUT): Based on the assumption that $c_1 \neq \text{output}(L', x, \text{using } \mathbf{o}_x, s_0 \dots s_k)$, we know that only the typing rule (OUTPUT) is applicable in this case. As $c_1 \approx c_2$, it is trivial that $\text{skip} \approx \text{skip}$ and $m'_1 = m_1 \approx_{\sigma} m_2 = m'_2$. To prove $\lfloor \tau_1 \rfloor_L = \lfloor \tau_2 \rfloor_L$, there are two cases.
 - $L' \sqsubseteq L$: We show that $\langle m, e \rangle \Downarrow v$ for some n without brackets in this case. We proceed by assuming that $\langle m, e \rangle \Downarrow [n]$ and show a contradiction. From the typing rule, we know that $\Gamma \vdash e : p$ and $s_0 \dots s_k \vdash p \preceq L'$. From the assumption $\langle m, e \rangle \Downarrow [n]$ and Lemma 4 in [17], we have $B \sqsubseteq_{\sigma} p$, which implies $\{B\}_{\sigma} \sqsubseteq \{p\}_{\sigma}$ by definition. By Theorem 2 and $s_0 \dots s_k \vdash p \preceq L'$, we know $\{p\}_{\sigma} \sqsubseteq L'$ whenever the output semantics is not a nop. Hence, we can derive $\{B\}_{\sigma} \sqsubseteq L' \sqsubseteq L$, which contradicts the assumption that $\{B\}_{\sigma} \not\sqsubseteq L$ in the theorem statement.



(a) The Figure illustrates the proof concept on transient policy. The orange part is the induction hypothesis and the green part is the part to be proved with the Unwinding Lemma.



(b) This Figure shows that $m' \in k_1(c, \tau^{[i]}, L)$, which indicates the variable x has been released. This difference causes the proof not to be concluded with Unwinding Lemma.

Fig. 9. Illustrations on the proof of Theorem 3. Conceptual proof on (a) transient policy, (b) persistent policy.

Hence, it must be true that $\langle m_1, e \rangle \Downarrow n$ for some n . By Lemma 6 in [17], $\langle m_2, e \rangle \Downarrow n$ too.

Hence, both output traces under m_1 and m_2 produce the same output event $\langle L', n, \sigma \rangle$.

– $L' \not\subseteq L$: The projection traces on L are empty ($\lfloor \tau_1 \rfloor_L = \emptyset = \lfloor \tau_2 \rfloor_L$) as $L' \not\subseteq L$.

□

Based on the unwinding lemma, we next show the main soundness result.

THEOREM 3 (END-TO-END SOUNDNESS). *For all command c and typing environment Γ , if $pc, \Gamma \vdash c$ for some program counter pc , then the program c satisfies dynamic release policy, which means*

$$\forall m, \sigma, \tau. \langle m, c, \sigma \rangle \hookrightarrow \tau \Rightarrow \forall 1 \leq i \leq \|\tau\|. \quad k_2(c, \tau^{[i]}, L, B) \supseteq \begin{cases} \llbracket m \rrbracket_{\neq B}, & \text{transient} \\ \llbracket m \rrbracket_{\neq B} \cap k_1(c, \tau^{[i-1]}, L), & \text{persistent} \end{cases}$$

Proof sketch. We conduct a proof of induction on the trace length i . Note that unlike a traditional proof for static noninterference, we need to show that under two indistinguishable initial memories, they both produce *consistent*, rather than indistinguishable traces.

Transient. For a transient policy B , proving $k_2(c, \tau^{[i]}, L, B) \supseteq \llbracket m \rrbracket_{\neq B}$ boils down, after expanding the definitions of k_1 and k_2 , to show that for any m' indistinguishable from m and $\langle m', c, \sigma \rangle \hookrightarrow \tau'$, we can find some index j on τ' such that $\tau'^{[j]}$ and $\tau^{[i]}$ are consistent (i.e., $\tau'^{[j]} \equiv_{B, L} \tau^{[i]}$). The induction step is illustrated in Figure 9a, where the green part illustrates the new index j' that we need to find in order to match index $i + 1$ on trace τ . The orange part illustrates the induction hypothesis: traces $\tau^{[i]}$ and $\tau'^{[j]}$ are consistent.

There are two cases to consider at each induction step: $\{B\}_{\sigma_i} \not\subseteq L$ and $\{B\}_{\sigma_i} \subseteq L$.

- When $\{B\}_{\sigma_i} \not\sqsubseteq L$, we can directly apply Lemma 2 to show that $\lfloor \tau'^{[j:j']} \rfloor_{B,L} = \lfloor \tau^{[i:i+1]} \rfloor_{B,L}$. In addition to the induction hypothesis, we know that $\tau'^{[j:j']} \equiv_{B,L} \tau^{[i:i+1]}$ stands.
- When $\{B\}_{\sigma_i} \sqsubseteq L$, the event v produced by $\langle m_i, c_i, \sigma_i \rangle$ (in the green part of Figure 9a) must have an empty projection:

$$\lfloor \langle m_i, c_i, \sigma_i \rangle \xrightarrow{\langle L', v, \sigma_i \rangle} \langle m_{i+1}, c_{i+1}, \sigma_{i+1} \rangle \rfloor_{B,L} \triangleq \emptyset$$

Hence, we simply set $j' = j$ and have $\tau'^{[j:j']} \equiv_{B,L} \tau^{[i:i+1]}$ as $\tau'^{[j:j']} = \tau'^{[j:j]}$, $\tau^{[i]} = \tau'^{[i:i+1]}$ and, from the induction hypothesis, $\tau'^{[j:j]} \equiv_{B,L} \tau^{[i]}$.

Persistent. For persistent policy, the proof is mostly identical to the transient case, since, by definition, its security condition is a relaxation of its corresponding transient policy. The exception is for the case of

$$(c_i = \text{output}(L', x, \text{using } \mathbf{o}_x, \mathbf{s}_0 \dots \mathbf{s}_k)) \wedge (x \text{ is immutable})$$

because the unwinding lemma (Lemma 2) is not applicable. We need a new proof strategy for this case, which is illustrated by the green dotted box in Figure 9b. To proceed, we utilize the fact that $m' \in k_1(c, \tau^{[i-1]}, L)$ in the definition of persistent policy. By definition of k_1 , we further know that $\lfloor \tau^{[i-1]} \rfloor_L \preceq \lfloor \tau'^{[j:j']} \rfloor_L$ where j' is the index that corresponds to $i - 1$ from the induction hypothesis, as illustrated in the orange part in Figure 9b. Moreover, from the induction hypothesis that $\tau'^{[j:j']} \equiv_{B,L} \tau^{[i-1]}$, we know that all past outputs of x must produce the same value, say n . Note that since x is immutable, both $\tau^{[i+1]}$ and $\tau'^{[j'+1]}$ will output the same value n . Hence, we let $j = j' + 1$ and have $\tau'^{[j:j]} \equiv_{B,L} \tau^{[i+1]}$. $\square$

6 Implementation

Prior work such as Jif [38] provides fine-grained information flow control for a Java-like language, but its implementation requires modifications to the programming language and relies on non-standard compilation tools. To make our language and type system more accessible to end users, we follow the approach of recent systems such as Cocoon [32], Carapace [12], and Filament [18], which are implemented as Rust libraries without requiring language and compiler modifications.

In particular, our prototype implementation is built on top of Filament [18], which provides Jif-like fine-grained information flow control for Rust. Consequently, our implementation is fully compatible with the standard Rust language and compiler.

6.1 Dynamic Release on Rust

The recent work Filament [18] presents a static information flow library in Rust that enforces noninterference with no runtime checks. To attach a value with a label, a Filament program wraps the value in a generic struct $\text{Label}\langle T, L \rangle$, where T is the data type and L is a (static) security level drawn from a predefined lattice. Filament uses traits to define the ordering of security levels (i.e., the lattice structure) based on these labels. However, Filament only supports static security levels.

To support dynamic release labels, we expand the generic struct to support dynamic labels and leverage Rust's phantom types for compile-time checking. More specifically, we define our label and value wrapper as a generic struct $\text{DRLabel}\langle T, S, LFrom, LTo \rangle$, representing data of type T associated with a security label $S?LFrom \rightarrow LTo$ using the syntax in Figure 2, where $LFrom$ and LTo are security levels¹. Moreover, we maintain both a Boolean variable s and a unique type S to represent a security event. The difference is that s can be set by library functions `eventon` and `eventoff`, while S is a static type indicating s holds. The event trace σ is implemented as a vector that tracks `eventon`/`eventoff` operations at runtime. The library function `relabel` only statically

¹Our current implementation does not support first-order functions; we leave this feature for future work (Section 8.2).

<pre> 1 impl<T, S, P, Pp, L> FlowsTo<DRLabel<T, S, L, L>> for (DRLabel<T, S, P, Pp>, DynamicLabel<P, Pp>) 2 where Pp: LEQ<L>, SEvent<S>: Holds, {} (a) Implementing rule (R-FROM) with Rust's trait. </pre>	<pre> 1 impl<L, S, Q, Qq> FlowsTo<(DRLabel<(), S, Q, Qq>, DynamicLabel<Q, Qq>)> for DRLabel<(), S, L, L> 2 where L: LEQ<Qq>, SEvent<S>: Holds {} (b) Implementing rule (R-To) with Rust's trait. </pre>
---	---

Fig. 10. Rule (R-FROM) and (R-To) from the implementation.

checks information flow constraints with the assumption that S holds when s is set; otherwise, it does nothing (see (OS-RELABEL-SUCC) and (OS-RELABEL-FAIL) in Figure 3). Similarly to prior works ([12, 18, 32]), our implementations of flows-to and release-to relations use Rust traits for compile-time checking as shown in Figure 10, covering the cases from the typing rules in Figures 5 and 6. The trait implementation models the security lattice, and by encoding these rules directly in the security functions (relabel and Output), we are able to implement and type-check all secure examples throughout the paper with our prototype.

Following Filament, our implementation also supports procedure calls, where labeled parameters and return values are statically checked. Consider the following example:

<pre> 1 fn pwd_reminder(..., mut password: DRLabel< String, B1, S, P>,>->DRLabel<String, B1, P, P > { 2 //set B1 if the user owns the password 3 ...} </pre>	<pre> 1 let password=DRLabel::<String, B1, S, P> { 2 value: "secret123".to_string(), 3 cond: false,...}; 4 let reminder = pwd_reminder(..., password); </pre>
--	---

The parameter `password` is annotated with the label $B1?S \rightarrow P$, where the security event $B1$ is set when the user owns the password. Our implementation statically checks that the label of the actual argument `password` does not exceed the corresponding annotation in `pwd_reminder`. Moreover, Rust's type system automatically infers the label of `reminder` to match the function's declared return label.

For simplicity, our prototype currently lacks the following features (1) nested dynamic labels, (2) user-defined security events and (3) first-order functions. We plan to add them in future work.

6.2 Case Studies

To demonstrate the efficacy of dynamic release policy, we ported a conference management system written in Lifty [39] and Civitas [21, 30] (excluding its cryptography proof component) written in Jif in approximately 1,700 lines of Rust code on top of our prototype implementation. The evaluation demonstrates that, in contrast to prior systems, each built for a specific dynamic policy and its corresponding enforcement mechanism, our framework can accommodate a broader class of dynamic policies within the same type system. For example, Lifty supports only declassification policies and therefore cannot express erasure policies required by Civitas. The full implementation of the case studies is available in [16].

6.2.1 Conference Reviewing System. HotCRP [31] is a widely used conference reviewing system with known security flaws [51]. Previous work Lifty [39] provides several code snippets that model the essence of those flaws in their Haskell-like domain specific language. Under the hood, Lifty detects such flaws by combining Labeled IO (LIO) [46] with Liquid Types [42]. We port the Lifty-based implementation with all core features such as score release, review release, and password reminder. Below, we discuss two representative examples.

Password Disclosure. A representative security vulnerability in HotCRP involves password disclosure [51]. HotCRP has two features. First, when a user forgets the password, HotCRP sends a reminder email to the user. Second, HotCRP has an email preview mode that allows an administrator to display emails in the browser, including those reminder emails as shown in Figure 11b.

<pre> 1 eventoff(&mut password); 2 if is_requester_owner(current_user, password_owner) { 3 eventon(&mut password); 4 let relabel_result = relabel:::< _, TrueB1, S, P, P, P > (&password, &conditions); 5 relabel_result.assign_to(&mut passcode_out); 6 output_to(&output_label, &passcode_out, & conditions);} 7 else {reject, line 8-9 in pseudocode} </pre>	<pre> 1 //password: ¬checkuser?S → P, passcode:P 2 eventoff(checkuser); 3 if (isUser == true) { //check user 4 eventon(checkuser); 5 passcode=relabel(password, "¬checkuser?S → P" to "P" using checkuser); 6 output(P, passcode, using checkuser);} 7 else { // administrator 8 passcode = password; //reject 9 output(P, passcode, using checkuser);} //nop </pre>
(a) Implementation on the password example.	(b) Pseudocode of the password example.

Fig. 11. Password Disclosure Example from Conference Reviewing System.

Consequently, in the email preview mode, an administrator can trigger a reminder email, which directly reveals a user's password in the browser to the administrator [51].

To prevent the accidental leak of a password, we assign it the label $\neg\text{checkuser?S} \rightarrow_t \text{P}$. This label ensures the password remains at a high security level until a specific security event, **checkuser**, is activated (eventon from line 3 in Figure 11a). To achieve this, the implementation checks the functions `relabel` and `output_to` at compile time. This check is enforced by the flows-to relations: if it compiles, then the execution reaches lines 4-6 in Figure 11b; otherwise, it fails to compile according to the type system.

Releasing papers' score. The system checks other scenarios, including paper score release, which is a microbenchmark used in Lifty [39]. This scenario requires that the paper's score should be visible only when the review is finished. To address this, the scores of the papers are labeled with $\neg\text{done?S} \rightarrow_t \text{P}$. The typing rule `relabel` checks the status indicator **done**, which determines when the result can be viewed.

6.2.2 Secure Voting System. Civitas [21, 30] is a secure voting system that enforces declassification and erasure policies across four phases: setup, registration, voting, and tabulation. Prior case studies [20] highlight two erasure scenarios involving credential shares by authorities and voters, written in Jif [38]. We ported all Civitas components related to vote delivery and credential erasure.

For authorities, the credentials must be removed to ensure voters' anonymity after they are delivered to the voters. Hence, we label credentials with $\neg\text{delivered?M} \rightarrow \top$, so that the credential must be erased after delivery. The output rule uses the fact that **delivered** has never been true (i.e., **delivered**) to check that the dynamic label can be released through the output channel.

For voters, each share on the voter side is labeled with $\neg\text{postCombined?M} \rightarrow \top$. After receiving all the shares from the authorities and combining the shares, the voter should delete any record provided by the authorities. The proof tree for output is the same as the previous example. The code is available in [16].

6.3 Performance and Correctness

We implemented the semantics shown in Figure 3 for the novel constructs introduced in our language. Since the unique semantics require tracking a trace of security events and evaluating event conditions at runtime, our approach may introduce additional runtime overhead. To assess this overhead, we measured the median execution time over 100 runs for both the original and the verified Rust implementations across our case studies. The results are shown below.

The evaluation results suggest that the overhead is low. We attribute the low overhead to the fact that even though the event trace grows during program execution, the overall overhead remains

	Original	IFC
conference	0.029ms	0.033ms
Civitas	5.694ms	5.736ms

low as (1) event trace only grows with eventon/eventoff, and output commands, and (2) the trace is only checked with a relabel command. Since both kinds of policy-specific commands occur infrequently in practice, the overall overhead remains low.

Moreover, because relabel and output commands may alter program semantics when the corresponding using clauses are not satisfied, we verified that all case studies produce the same outputs as the original programs without information flow control.

7 Related Work

Dynamic Policies. To address the limitations of noninterference, prior work introduces various mechanisms [7, 8, 19–21] to allow information sensitivity/integrity to change over time. Additional studies include extensions to gradual release [10], temporally verified logic [9], and security-specific languages [28, 48], which support role-based information flow policies with dynamic updates. These works enable flexible label updates while accommodating both Denning-style and role-based lattices. Recent work [14] introduces nonmalleable progress leakage, adapting nonmalleable information flow to secure progress channels for downgrading policies. More recently, Ahmadian [1] introduced an attacker knowledge-based information flow policy. Zegarelli et al. [52] propose a modular enforcement mechanism that supports multiple interpretations of facets within a single framework. In contrast, our work uses the dynamic release policy [34], which provides a uniform formalization of various dynamic policies.

Enforcement of Dynamic Policies in Applications. Building on Myers’ influential work [36] in establishing a formal semantics for decentralized labels, various applications have been built on top of Jif [38], including some that involve dynamic policies such as declassification and erasure. Dynamic policies are also widely used in systems research. Guarnieri et al. [26, 27] explore the “forgetful attacker” model [4] to implement declassification via a security monitor. In addition, dynamic policies have been applied to applications such as conference reviewing systems [39, 46], Battleship Game [38, 47], and social media platforms [11].

In comparison, our dynamic release Rust library constitutes the first approach capable of expressing and verifying the full range of such policies. A detailed comparison with dynamic release policy and other variants of dynamic policies can be found in [34].

Soundness Proof. Most information flow policies enforce noninterference, and are typically enforced via static type systems, whose soundness has been proven using both big- and small-step semantics [15, 25, 33, 40, 41, 49]. As discussed in Section 5, these proof techniques are not directly applicable to dynamic policies.

Prior work in dynamic policies, notably [20], established the soundness of static enforcement for declassification and erasure. Unlike [20], which uses two separate sets of rules, our dynamic release policy is a superset of those supported policies and provides a single set of rules for various dynamic policies. Furthermore, differences in policy formalization mean that the soundness proof of [20] is not applicable to our dynamic release enforcement.

8 Discussion and Future Work

8.1 Annotation Burden

In this paper, we assume that using clauses are provided by programmers. From the programmer’s perspective, an appropriate annotation should specify, for each output and relabel command: (1)

the conditions under which information may be released safely, thereby enabling the command to satisfy the typing rules (Section 4.4); and (2) the conditions that hold at the corresponding program point, ensuring that the annotated program exhibits the same behavior as the original program whenever the original program is secure (Section 3.2). Across all of our case studies (Section 6), these conditions are sufficiently simple to identify and annotate manually. However, constructing such annotations may become more challenging in large code bases. Hence, an important direction for future work is the automatic inference of using clauses through static analysis. Our language design is well suited to this objective because security events are explicitly distinguished from ordinary program variables and can only be modified by the `eventon` and `eventoff` commands (Section 3.1). This separation enables a compositional inference strategy that we plan to explore: an intraprocedural dataflow analysis can propagate event-related facts along the control-flow graph to infer conditions that must hold at each program point, while the effect of each procedure on security events can be summarized independently by the set of events it may modify.

Moreover, our type system currently does not support label inference. Consequently, all label annotations, including the “from” and “to” labels in `relabel` commands, must be provided explicitly by programmers. A promising direction for future work is to extend the language with automatic label inference by formulating the flows-to relation as a system of constraints to be solved by a constraint solver. The primary challenge lies in the presence of dynamic release labels, which introduce dependencies among these constraints and substantially complicate the inference process. Addressing such dependencies is beyond the capabilities of our current Rust-based implementation.

8.2 Language Features

Our primary goal is to establish a theoretical foundation for enforcing dynamic release policies. In this context, the use of a minimal imperative language is deliberate and consistent with prior type systems that come with formal soundness proofs. Building on this foundation, an important direction for future work is to extend both the source language and the type system with more expressive language features, such as first-class functions and heap manipulation. In pursuing these extensions, we plan to draw on established designs such as Jif [38] and Cocoon [32].

For example, supporting first-class functions requires assigning security labels to those functions (including their parameters, return values, and side effects) just as for any other value in the program. These labels capture the security constraints governing the use of a function, such as the maximum sensitivity of its arguments and the minimum security level of the *pc* context in which it may be invoked securely. In the setting of dynamic release policies, an additional challenge arises because these constraints also depend on the state of security events at the call site. Consequently, the annotation may also require a `using` clause, specifying the necessary event states to ensure that function invocations remain secure.

9 Conclusion

In this paper, we present the first sound enforcement of the dynamic release policy introduced by Li and Zhang [34]. Compared to prior enforcements of dynamic policies, which are based on ad hoc, policy-specific enforcement mechanisms, our work offers simple, intuitive, and unified enforcement of various dynamic policies, such as declassification, endorsement, and erasure policies. We developed a metalanguage with carefully designed features, such as `relabel`, `eventon`, `eventoff` commands for writing code governed by embedded dynamic release policies. We further developed a type system that is formally proven to soundly enforce dynamic release policies, and implemented a prototype as an extension to Rust.

Data Availability Statement

An initial artifact for this paper consists of the prototype and the case studies from Section 6. A version of the source code is available at the link [16]. The full proof can be found in [17].

Acknowledgments

We would like to express our gratitude to the anonymous reviewers for their insightful feedback and suggestions. We also appreciate the insights from Peixuan Li and Quan Zhou. This research is sponsored by National Science Foundation grants CNS 2401496 and CNS 2401182.

References

- [1] Amir M Ahmadian and Musard Balliu. 2022. Dynamic policies revisited. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 448–466. doi:10.1109/EuroSP53844.2022.00035
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques & Tools* (2nd ed.). Pearson Education, Boston, MA, USA.
- [3] Owen Arden, Jinyang Liu, and Andrew C. Myers. 2015. Flow-Limited Authorization. In *2015 IEEE 28th Computer Security Foundations Symposium (CSF)*. IEEE, 569–583. doi:10.1109/CSF.2015.42
- [4] Aslan Askarov and Stephen Chong. 2012. Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 308–322. doi:10.1109/CSF.2012.31
- [5] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-insensitive noninterference leaks more than just a bit. In *Computer Security – ESORICS 2008: 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6–8, 2008. Proceedings*. Springer, 333–348. doi:10.1007/978-3-540-88313-5_22
- [6] Aslan Askarov, Scott Moore, Christos Dimoulas, and Stephen Chong. 2015. Cryptographic Enforcement of Language-based Information Erasure. In *2015 IEEE 28th Computer Security Foundations Symposium (CSF)*. IEEE, 334–348. doi:10.1109/CSF.2015.30
- [7] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *2007 IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 207–221. doi:10.1109/SP.2007.22
- [8] Aslan Askarov and Andrei Sabelfeld. 2009. Tight Enforcement of Information-Release Policies for Dynamic Languages. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 43–59. doi:10.1109/CSF.2009.22
- [9] Musard Balliu, Mads Dam, and Gurvan Le Guernic. 2011. Epistemic Temporal Logic for Information Flow Security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security (PLAS)*. ACM, 1–12. doi:10.1145/2166956.2166962
- [10] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. 2008. Expressive Declassification Policies and Modular Static Enforcement. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*. IEEE, 339–353. doi:10.1109/SP.2008.20
- [11] Thomas Bauereiss, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. 2018. CoSMed: A confidentiality-verified social media platform. *Journal of Automated Reasoning* 61 (2018), 113–139. doi:10.1007/s10817-017-9443-3
- [12] Vincent Beardsley, Chris Xiong, Ada Lambda, and Michael D Bond. 2025. Carapace: Static-Dynamic Information Flow Control in Rust. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 364–392. doi:10.1145/3720427
- [13] Niklas Broberg, Bart van Delft, and David Sands. 2015. The Anatomy and Facets of Dynamic Policies. In *2015 IEEE 28th Computer Security Foundations Symposium (CSF)*. IEEE, 122–136. doi:10.1109/CSF.2015.16
- [14] Ethan Cecchetti. 2025. Nonmalleable Progress Leakage. In *2025 IEEE 38th Computer Security Foundations Symposium (CSF)*. IEEE, 537–552. doi:10.1109/CSF64896.2025.00029
- [15] Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1249–1267. doi:10.1109/SP40001.2021.00084
- [16] Jeffrey Ching and Danfeng Zhang. 2026. Sound Enforcement of Dynamic Release Information Flow Policy Artifact. doi:10.5281/zenodo.21943585
- [17] Jeffrey C. Ching and Danfeng Zhang. 2026. Sound Enforcement of Dynamic Release Information Flow Policy – Full Version. *arXiv preprint arXiv:2608.09506* (2026). arXiv:2608.09506
- [18] Jeffrey C Ching, Quan Zhou, and Danfeng Zhang. 2026. Filament: Denning-Style Information Flow Control for Rust. *arXiv preprint arXiv:2604.14357* (2026). arXiv:2604.14357
- [19] Stephen Chong and Andrew C. Myers. 2005. Language-Based Information Erasure. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW)*. IEEE, 241–254. doi:10.1109/CSFW.2005.19
- [20] Stephen Chong and Andrew C. Myers. 2008. End-to-End Enforcement of Erasure and Declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF)*. IEEE, 98–111. doi:10.1109/CSF.2008.12
- [21] Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. 2008. Civitas: Toward a Secure Voting System. In *2008 IEEE Symposium on Security and Privacy (SP 2008)*. IEEE, 354–368. doi:10.1109/SP.2008.32

- [22] Dorothy E Denning. 1976. A lattice model of secure information flow. *Commun. ACM* 19, 5 (1976), 236–243. doi:10.1145/360051.360056
- [23] David Ferraioli, Janet Cugini, and D. Richard Kuhn. 1995. Role-based Access Control (RBAC): Features and Motivations. In *Proceedings of the 11th Annual Computer Security Applications Conference (ACSAC)*. IEEE Computer Society, 241–248.
- [24] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy (SP)*. IEEE, 11–20. doi:10.1109/SP.1982.10014
- [25] Simon Oddershede Gregersen, Johan Bay, Amin Timany, and Lars Birkedal. 2021. Mechanized logical relations for termination-insensitive noninterference. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. doi:10.1145/3434291
- [26] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. 2019. Information-Flow Control for Database-Backed Applications. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 79–94. doi:10.1109/EuroSP.2019.00016
- [27] Marco Guarnieri, Srdjan Marinovic, and David Basin. 2016. Strong and Provably Secure Database Access Control. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 163–178. doi:10.1109/EuroSP.2016.23
- [28] Michael Hicks, Stephen Tse, Boniface Hicks, and Steve Zdancewic. 2005. Dynamic updating of information-flow Policies. In *Proceedings of the International Workshop on Foundations of Computer Security (FCS)*. 7–18.
- [29] Catalin Hritcu, John Hughes, Benjamin C Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing noninterference, quickly. *ACM SIGPLAN Notices* 48, 9 (2013), 455–468. doi:10.1145/2500365.2500574
- [30] Ari Juels, Dario Catalano, and Markus Jakobsson. 2005. Coercion-Resistant Electronic Elections. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society (WPES)*. ACM, 61–70. doi:10.1145/1102199.1102213
- [31] Eddie Kohler. 2013. HotCRP. <https://hotcrp.com/>. Accessed: 19 September 2025.
- [32] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D Bond, and Zhiqiang Lin. 2024. Cocoon: Static Information Flow Control in Rust. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024), 166–193. doi:10.1145/3649817
- [33] Peixuan Li and Danfeng Zhang. 2017. Towards a Flow- and Path-Sensitive Information Flow Analysis: Technical Report. *arXiv preprint arXiv:1706.01407* (2017). arXiv:1706.01407
- [34] Peixuan Li and Danfeng Zhang. 2022. Towards a General-Purpose Dynamic Information Flow Policy. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. IEEE, 260–275. doi:10.1109/CSF54842.2022.9919639
- [35] Andrew C Myers and Barbara Liskov. 1997. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review* 31, 5 (1997), 129–142. doi:10.1145/269005.266669
- [36] Andrew C. Myers and Barbara Liskov. 1998. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy (SP)*. IEEE, 186–197. doi:10.1109/SECPRI.1998.674834
- [37] Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442. doi:10.1145/363516.363526
- [38] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java information flow. (July 2006). <http://www.cs.cornell.edu/jif>
- [39] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020), 1–30. doi:10.1145/3408987
- [40] François Pottier and Vincent Simonet. 2002. Information Flow Inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Portland, OR, USA, 319–330. doi:10.1145/503272.503302
- [41] Vineet Rajani and Deepak Garg. 2020. On the expressiveness and semantics of information flow types. *Journal of Computer Security* 28, 1 (2020), 129–156. doi:10.3233/JCS-191382
- [42] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 159–169. doi:10.1145/1375581.1375602
- [43] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications* 21, 1 (January 2003), 5–19. doi:10.1109/JSAC.2002.806121
- [44] Andrei Sabelfeld and David Sands. 2005. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 255–269. doi:10.1109/CSFW.2005.15
- [45] Andrei Sabelfeld and David Sands. 2009. Declassification: Dimensions and principles. *Journal of Computer Security* 17, 5 (2009), 517–548. doi:10.3233/JCS-2009-0352
- [46] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM Symposium on Haskell*. ACM, 95–106. doi:10.1145/2096148.2034688
- [47] Alley Stoughton, Andrew Johnson, Samuel Beller, Karishma Chadha, Dennis Chen, Kenneth Foner, and Michael Zhivich. 2014. You sank my battleship! A case study in secure programming. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. ACM, 2–14. doi:10.1145/2637113.2637115

- [48] Nikhil Swamy, Michael Hicks, Stephen Tse, and Steve Zdancewic. 2006. Managing policy updates in security-typed languages. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. IEEE, 13–pp. doi:10.1109/CSFW.2006.17
- [49] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *Journal of computer security* 4, 2-3 (1996), 167–187. doi:10.3233/JCS-1996-42-304
- [50] Lucas Waye, Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong. 2017. Cryptographically Secure Information Flow Control on Key-Value Stores. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 1893–1907. doi:10.1145/3133956.3134036
- [51] Alexander Yip, Xi Wang, Nikolai Zeldovich, and M. Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. ACM, 291–304. doi:10.1145/1629575.1629604
- [52] Antonio Zegarelli, Niki Vazou, and Marco Guarnieri. 2024. Modular Implementation and Formalization of Dynamic Policies: Work in Progress. In *Proceedings of the 2024 Workshop on Foundations of Computer Security (FCS)*. Informal Proceedings.

Received 2026-03-17; accepted 2026-08-06