Towards a General-Purpose Dynamic Information Flow Policy

Peixuan Li, Danfeng Zhang
Department of Computer Science and Engineering
Pennsylvania State University, University Park, PA United States
e-mail: {pzl129,zhang}@cse.psu.edu

***Abstract*—Noninterference offers a rigorous end-to-end guarantee for secure propagation of information. However, real-world systems almost always involve security requirements that change during program execution, making noninterference inapplicable. Prior works alleviate the limitation to some extent, but even for a veteran in information flow security, understanding the subtleties in the syntax and semantics of each policy is challenging, largely due to very different policy specification languages, and more fundamentally, semantic requirements of each policy.**

We take a top-down approach and present a novel information flow policy, called Dynamic Release, which allows information flow restrictions to downgrade and upgrade in arbitrary ways. Dynamic Release is formalized on a novel framework that, for the first time, allows us to compare and contrast various dynamic policies in the literature. We show that Dynamic Release generalizes declassification, erasure, delegation and revocation. Moreover, it is the only dynamic policy that is both applicable and correct on a benchmark of tests with dynamic policy.

I. Introduction

While noninterference [28] has become a cliché for end-to-end data confidentiality and integrity in information flow security, this well-accepted concept only describes the ideal security expectations in a *static* setting, i.e., when data sensitivity *does not* change throughout program execution. However, real-world applications almost always involve some dynamic security requirements, which motivates the development of various kinds of *dynamic* information flow policies:

- A *declassification policy* [7], [10], [22], [46], [26], [27], [37], [45] weakens noninterference by deliberately releasing (i.e., declassifying) sensitive information. For instance, a conference management system typically allows deliberate release of paper reviews and acceptance/rejection decisions after the notification time.
- An *erasure policy* [18], [19], [33], [23], [30], [5] strengthens noninterference by requiring some public information to become more sensitive, or be erased completely when certain condition holds. For example, a payment system should not retain any record of credit card details once the transaction is complete.
- An *delegation/revocation policy* [3], [32], [50], [38] updates dynamically the sensitivity roles in a security system to accommodate the mutable requirements of security, such as delegating/revoking the access rights of a new/leaving employee.

Moreover, there are a few case studies on the needed security properties in the light of one specific context or task [6], [31], [43], [49], and build systems that provably enforces some variants of declassification policy (e.g., CoCon [34], CosMeDis [12]) and erasure policy (e.g., Civitas [21]).

Although the advances make it possible to specify and verify *some variants* of dynamic policy, cherry-picking the *appropriate* policy is still a daunting task: different policies (even when they belong to the same kind) have very different syntax for specifying how a policy changes [47], very different nature of the security conditions (i.e., noninterference, bisimulation and epistemic [16]) and even completely inconsistent notion of security (i.e., policies might disagree on whether a program is secure or not [16]). So even for veteran researchers in information flow security, understanding the subtleties in the syntax and semantics of each policy is difficult, evidenced by highly-cited papers that synthesize existing knowledge on declassification policy [47] and dynamic policy [16]. Arguably, it is currently *impossible* for a system developer/user to navigate in the jungle of unconnected policies (even for the ones in the same category) when a dynamic policy is needed [16], [47].

In this paper, we take a top-down approach and propose Dynamic Release, the first information flow policy that enables declassification, erasure, delegation and revocation at the same time. One important insight that we developed during the process is that erasure and revocation both strengthen an information flow policy, despite their very different syntax in existing work. However, an erasure policy by definition disallows the same information leaked in the past (i.e., before erasure) to be released in the future, while most revocation policies allow so. This motivates the introduction of two kinds of policies, which we call *persistent* and *transient* policies. The distinction can be interpreted as a type of information flow which is permitted by some definitions but not by others, called facets [16].

Moreover, Dynamic Release is built on a novel formalization framework that is shown to subsume existing security conditions that are formalized in different ways (e.g., noninterference, bisimulation and epistemic [16]). More importantly, for the first time, the formalization framework allows us to make apple-to-apple comparison among existing policies, which are incompatible before (i.e., one cannot trivially convert one to another). Besides the distinction between *persistent* and *transient* policies mentioned earlier, we also notice that it is more challenging to define a *transient* policy (e.g., erasure), as it requires a definition of the *precise* knowledge gained from

A. Declassfication

```
1  // bid : S
2  submit := bid;
3  output(submit, S);
4  // bid : P
5  output(submit, P);
```

(i). Secure Program

```
1  // bid : S
2  submit := bid;
3  output(submit, P);
4  // bid : P
5  output(submit, P);
```

(ii). Insecure Program

B. Erasure

```
1  // credit_card : M
2  copy := credit_card;
3  output(copy, M);
4  // credit_card : ⊤
5  copy := 0;
6  output(copy, M);
```

(i). Secure Program

```
1  // credit_card : M
2  copy := credit_card
3  output(copy, M);
4  // credit_card : ⊤
5  // No Clear Up
6  output(copy, M);
```

(ii). Insecure Program

C. Delegate/Revoke

```
1  //book : bk, notes : Alice
2  //bk → Alice
3  notes := half(book);
4  output(notes, Alice);
5  //bk ↛ Alice
6  output(notes, Alice);
```

(i). Secure Program

```
1  //book : bk, notes : Alice
2  //bk → Alice
3  notes := half(book);
4  output(notes, Alice);
5  //bk ↛ Alice
6  output(book, Alice);
```

(ii). Insecure Program

Fig. 1: Examples of Dynamic Policies.

observing one output event, rather than the more standard *cumulative* knowledge that we see in existing *persistent* policies.

Finally, we built a new $\mathcal{A}$nnTrace benchmark for testing and understanding variants of dynamic policies in general. The benchmark consists of examples with dynamic policies from existing papers, as well as new subtle examples that we created in the process of understanding dynamic policies. We implemented our policy and existing policies, and found that Dynamic Release is the only one that is both applicable and correct on all examples.

To summarize, this paper makes the following contributions:

1) We present a language abstraction with concise yet expressive security specification (Section III) that allows us to specify various existing dynamic policies, including declassification, erasure, delegation and revocation.
2) We present a new policy Dynamic Release (Section IV). The new definition resolves a few subtle pitfalls that we found in existing definitions, and its security condition handles transient and persistent policies in a uniform way.
3) We generalize the novel formalization framework behind Dynamic Release and show that it, for the first time, allows us to compare and contrast various dynamic policies at the semantic level (Section V). The comparison leads to new insights that were not obvious in the past, such as whether an existing policy is transient or persistent.
4) We build a new benchmark for testing and understanding dynamic policies, and implemented our policy and existing ones (Section VI). Evaluation on the benchmark suggests that Dynamic Release is the only one that is both applicable and correct on all examples.

II. Background and Overview

A. Security Levels

As standard in information flow security, we assume the existence of a set of security levels $\mathbb{L}$, describing the intended confidentiality of information[1]. For generality, we *do not* assume that all levels form a Denning-style lattice. For instance, delegation and revocation typically use principals/roles (such as $\texttt{Alice}, \texttt{Bob}$) where the *acts-for* relation on principals can change at run time. For simplicity, we use the notation $\ell \in \mathcal{L}$ if all levels form a lattice $\mathcal{L}$, rather than $L \in \mathbb{L}$. Moreover, we use P (public), S (secret) to represent levels in a standard two-point lattice where $\mathrm{P} \sqsubset \mathrm{S}$ but $\mathrm{S} \not\sqsubseteq \mathrm{P}$.

[1] Since integrity is the dual of confidentiality, we will assume confidentiality hereafter.

B. Terminology

Some terms in dynamic policy are overloaded and used inconsistently in the literature. For instance, declassification is sometimes confused with dynamic policy [16]. To avoid confusion, we first define the basic terminology that we use throughout the paper.

Definition 1 (Dynamic (Information Flow) Policy): An information flow policy is dynamic if it allows the sensitivity of information to change during one execution of a program.

As standard, we say that a change of sensitivity is *downgrading* (resp. *upgrading*) if it makes information less sensitive (resp. more sensitive).

Next, we use the examples in Figure 1 to introduce the major kinds of dynamic policies in the literature. For readability, we use informal security specification in comments for most examples in the paper; a formal specification language is given in Section III.

a) Declassification: Given a Denning-style lattice $\mathcal{L}$, declassification occurs when a piece of information has its sensitivity level ℓ_1 downgraded to a lower sensitivity level ℓ_2 (i.e., $\ell_2 \sqsubset \ell_1$). Consider Figure 1-A which models an online bidding system. When bidders submit their bids to the system during the bidding phase, each bid is classified that no other bidders are allowed to learn the information. When the bidding ends, the bids are public to all bidders. In the secure program (i), the bid is only revealed to a public channel with level P (Line 5) when bidding ends. However, the insecure program (ii) leaks the bid during the bidding phase (Line 3).

b) Erasure: Given a Denning-style lattice $\mathcal{L}$, information erasure occurs when a piece of information has its sensitivity level ℓ_1 upgraded to a more restrictive sensitivity level, or an incomparable level ℓ_2 (i.e., $\ell_2 \not\sqsubseteq \ell_1$). Moreover, when information is erased to level $\top$, the sensitive information must be removed from the system as if it was never inputted into the system. Figure 1-B is from a payment system. The user of the system gives her credit card information to the merchandiser (at level `M`) as payment for her purchase. When the transaction is done, the merchandiser is not allowed to retain/use the credit card information for any other purpose (i.e., its level changes to $\top$). The secure program (i) only uses the credit card information during the transaction (Line 3), and any related information is erased after the transaction (Line 5). The insecure program (ii), however, fails to protect the credit card information after the transaction (Line 6).

c) Delegation and Revocation: Delegation and revocation are typically used together, in a principal/role-based system [1], [25], [41]. In this model, information is associated with principals/roles, and a dynamic policy is specified as changes (i.e., add or remove) to the "acts-for" relationship on principals/roles. Figure 1-C is from a book renting system, where its customers are allowed to read books during the renting period. In this example, `Alice` acts-for `bk` ($\mathtt{bk} \rightarrow \mathtt{Alice}$) before line 3. Hence, she is allowed to take notes from the book. When the renting is over, the book is no longer accessible to `Alice` ($\mathtt{bk} \not\rightarrow \mathtt{Alice}$), but the notes remain accessible to `Alice`. The secure program (i) allows the customer to get their notes (Line 6) learned during the renting period. The insecure program (ii) fails to protect the book (Line 6) after the renting is over.

C. Overview

We use Figure 1 to highlight two major obstacles of understanding/applying various kinds of dynamic policies.

First, we note that a delegation/revocation policy (Example C) and an erasure policy (Example B) use different formats to model sensitivity change. A delegation/revocation policy attaches *fixed* security levels to data throughout program execution; policy change is modeled as changing the acts-for relation on roles. On the other hand, an erasure policy uses a fixed lattice throughout program execution; policy change is modeled as *mutable* security levels on data. These two examples are similar from policy change perspective, as they are both upgrading policies. But due to the different specification formats, their relation becomes obscure.

Second, we note that Example B.ii and C.i are semantically very similar: both examples first read data when the policy allows so, and then try to access the data again when the policy on data forbids so. However, B.ii is considered *insecure* according to an erasure policy, while C.i is considered *secure* according to a revocation policy. Even when we only consider policies of the same kind (e.g., delegation/revocation), such inconsistency in the security notion also exists, which is called *facets of dynamic policies* [16].

Broberg et al. [16] have identified a few facets, but identifying other differences among existing policies is extremely difficult, as they are formalized in different nature (e.g., noninterference, bisimulation and epistemic). We can peek at the semantics-level differences based on a few examples, but an apple-to-apple comparison is still impossible at this point.

In this paper, we take a top-down approach that rethinks dynamic policy from scratch. Instead of developing four kinds of policies seen in prior work, we observe that there are only *two essential building blocks* of a dynamic policy: upgrading and downgrading. With an expressive specification language syntax (Section III), we show that in terms of upgrading and downgrading sensitivity, declassification (resp. erasure) is the same as delegation (resp. revocation). In terms of the formal security condition of dynamic policy, we adopt the epistemic model [7] and develop a formalization framework that can be informally understood as the following security statement:

> A program c is secure iff for any event t produced by c, the "knowledge" gained about secret by learning t is bounded by what's allowed by the policy at t.

We note that a key challenge of a proper security definition for the statement above is to properly define the "knowledge" of learning a *single* event t. During the process of developing the formal definition, we discovered a new facet of upgrading policies; the difference is that whether an upgrading policy automatically allows information leakage (after upgrading) when it has happened in the past. Consequently, we precisely define the "knowledge" of learning a single event and make semantics-level choices (called *transient* and *persistent* respectively) of the new facet explicit in Dynamic Release (Section IV).

To compare and contrast various dynamic policies (including Dynamic Release), we cast existing policies into the formalization framework behind Dynamic Release (Section V). We find that the semantics of erasure and revocation are drastically different: erasure policy is *transient* by definition, and most revocation policies are *persistent*. The semantics-level difference sheds light on why Example B.ii and C.i have inconsistent security under erasure and revocation policies, even though they are similar programs.

III. Dynamic Policy Specification

We first present the syntax of an imperative language with its security specification. Based on that, we show that the policy specification is powerful enough to describe declassification, erasure, delegation and revocation policies. Finally, we define a few notations to be used throughout the paper.

A. Language Syntax and Security Specification

In this paper, we use a simple imperative language with expressive security specification, as shown in Figure 2. The language provides standard features such as variables, assignments, sequential composition, branches and loops. Other features are introduced for security:

Variables (**Vars**)	x, y, z
Events ($\mathbb{S}$)	s
Expressions ($\mathbb{E}$)	$e ::= x \mid n \mid e \texttt{ op } e$
Commands	$c ::= \texttt{skip} \mid c_1; c_2 \mid x := e \mid \texttt{while } (e)\ c$ $\mid \texttt{if } (e) \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{output}(b, e)$ $\mid \texttt{EventOn}(s) \mid \texttt{EventOff}(s)$
Level Sets ($\mathbb{L}$)	L
Security Labels ($\mathbb{B}$)	$b ::= L \mid cnd?b_1 \circ b_2;$
Conditions	$cnd ::= s \mid e \mid cnd \wedge cnd$ $\mid cnd \vee cnd \mid \neg cnd$
Mutation Directions	$\circ ::= \rightarrow \mid \leftarrow \mid \leftrightarrows$
Policy Specification	$\Gamma : \textbf{Vars} \mapsto \mathbb{B}[\diamond]$
Policy Type	$\diamond ::= Tran \mid Per$

Fig. 2: Language Syntax with Security Specification.

- We explicitly model information release by a release command $\texttt{output}(b, e)$; it reveals the value of expression e to an information channel with security label b.[2]
- We introduce distinguished security events $\mathbb{S}$. An event $s \in \mathbb{S}$ is similar to a Boolean; we distinguish s and x in the language syntax to ensure that security events can only be set and unset using distinguished commands $\texttt{EventOn}(s)$ and $\texttt{EventOff}(s)$, which set s to `true` and `false` respectively. We assume that all security events are initialized with `false`.

1) Sensitivity Levels: For generality, we assume a predefined set $\mathbb{L}$ of all security levels, and use level set $L \subseteq \mathbb{L}$ to specify data sensitivity. Intuitively, a level set L consists of a set of levels where the associated information can flow to. Hence, L_1 is less restrictive as L_2, written as $L_1 \sqsubset L_2$ iff $L_2 \subset L_1$, and $L_1 \sqsubseteq L_2$ iff $L_2 \subseteq L_1$.

Although the use of level set is somewhat non-standard, we note that it provides better generality compared with existing specifications, such as a level from a Denning-style lattice [24] or a role in a role-based model [1], [25], [41].

- Denning-style lattice: let $\mathcal{L}$ be a security lattice. We can define $\mathbb{L}$ and the level set that represents $\ell \in \mathcal{L}$ as follows:

$$\mathbb{L} = \{\ell \mid \ell \in \mathcal{L}\};\ L_\ell \triangleq \{\ell' \in \mathcal{L} \mid \ell \sqsubseteq \ell'\} \qquad (1)$$

  Consider a two-point lattice $\{\mathsf{P}, \mathsf{S}\}$ with $\mathsf{P} \sqsubset \mathsf{S}$. It can be written as the follows in our syntax:

$$\mathbb{L} \triangleq \{\mathsf{P}, \mathsf{S}\};\quad L_{\mathsf{S}} \triangleq \{\mathsf{S}\};\quad L_{\mathsf{P}} \triangleq \{\mathsf{P}, \mathsf{S}\};$$

- Role-based model: let $\mathbb{P}$ be a set of principals/roles and `actsfor` be an acts-for relation on roles. We can define $\mathbb{L}$ and the level set that represents $P \in \mathbb{P}$ as follows:

$$\mathbb{L} = \mathcal{P}(\mathbb{P});\ L_P \triangleq \{P' \in \mathbb{P} \mid P' \texttt{ actsfor } P\} \qquad (2)$$

  Consider a model with two roles `Alice` and `Bob` with `Alice actsfor Bob` but not the other way around. It can be written as the follows in our syntax:

$$\mathbb{L} \triangleq \{\texttt{Alice}, \texttt{Bob}\};\ L_{\texttt{Alice}} \triangleq \{\texttt{Alice}\};\ L_{\texttt{Bob}} \triangleq \{\texttt{Alice}, \texttt{Bob}\};$$

2) Sensitivity Mutation: The core of specifying a dynamic policy is to define how data sensitivity changes at run time. This is specified by a security label b.

A label can simply be a level set L, which represents immutable sensitivity throughout program execution. In general, a label has the form of $cnd?b_1 \circ b_2$ where:

- A trigger condition cnd specifies *when* the sensitivity changes. There are two basic kinds of trigger conditions: a security event s and a (Boolean) program expression e. A more complicated condition can be constructed with logical operations on s and e. We assume that a type system checks that whenever cnd is an expression e, e is of the Boolean type.
- The mutation direction $\circ$ specifies *how* the information flow restriction changes. There are two one-time mutation directions: $cnd?b_1 \rightarrow b_2$ (resp. $cnd?b_1 \leftarrow b_2$) allows a *one-time* sensitivity change from b_1 to b_2 (resp. b_2 to b_1) the first time that cnd evaluates to `false` (resp. `true`). On the other hand, a two-way mutation $cnd?b_1 \leftrightarrows b_2$ allows arbitrary number of changes between b_1 and b_2 whenever the value of cnd flips.

3) Policy Specification: The information flow policy on a program is specified as a function from variables **Vars** to security labels $\mathbb{B}$ and a policy type $\diamond$. The policy type can either be transient, or persistent (formalized in Section IV).

B. Expressiveness

Despite the simplicity of our language syntax and security specification, we first show that all kinds of dynamic policies in Figure 1 can be concisely expressed. Then, we discuss how the specification covers the well-known *what, who, where* and *when* dimensions [47], [48] of dynamic policies.[3] Finally, we show that the specification language is powerful enough to encode Flow Locks [13] and its successor Paralocks [15], a well-known meta policy language for building expressive information flow policies.

1) Examples: We first encode the examples in Figure 1.

a) Declassification and Erasure: Both policies specify sensitivity changes as mutating security level of information from some level ℓ_1 to ℓ_2, where both ℓ_1 and ℓ_2 are drawn from a Denning-style lattice $\mathcal{L}$. Such a change can be specified as $L_{\ell_1} \rightarrow L_{\ell_2}$, where L_{ℓ_1} and L_{ℓ_2} are the level sets representing ℓ_1 and ℓ_2, as defined in Equation (1).

For example, the informal policy on *credit_card* in Figure 1-B can be precisely specified as $erase?\{\} \leftarrow \{\texttt{M}\}\ [Tran]$

[2] In the literature, it is also common to model information release as updates to a memory portion visible to an attacker. This can be modeled explicitly as requiring an assignment $x := e$ where x has label b to emit a release command $\texttt{output}(b, v)$.

[3] The original definitions focus on declassification policy, but the dimensions are applicable for dynamic policies as well.

(we will discuss why erasure is a transient policy in Section IV) with the security command $\texttt{EventOn}(erase)$ being inserted to Line 4 to trigger the mutation.

b) Delegation and revocation: Both policies specify sensitivity changes as modifying the acts-for relationship on principals, such as `Alice` and `Bob`. Such a change can be specified as the old and new sets of roles who acts-for the owner, say P, of information. That is, a change from from $\texttt{actsfor}_1$ to $\texttt{actsfor}_2$ can be specified as $L_1 \rightarrow L_2$, where $L_i \triangleq \{P' \in \mathbb{P} \mid P' \ \texttt{actsfor}_i \ P\}$.

For example, the policy on *book* in Figure 1-C can be specified as $revoke?\{\} \leftarrow \{\texttt{Alice}\}\ [Per]$ (we will discuss why revocation is a persistent policy in Section IV) with a security command $\texttt{EventOn}(revoke)$ being inserted to Line 5 to trigger the mutation. [4]

2) Dimensions of dynamic policy [47], [48]:

a) What: The *what* dimension regulates what information's sensitivity is changed. Since the policy specification is defined at variable level, our language does not fully support partial release, which only releases a part of a secret (e.g., the parity of a secret) to a public domain. However, we note that the language still has *some* support of partial release. Consider the example in Figure 1-C.i. The policy allows partial value $half(book)$ to be accessible by Alice after Line 5, while the whole value of *book* is not. As shown in Section III-B1, the partial release of $half(book)$ in this example can be precisely expressed in our language. We leave the full support of partial release as future work.

Moreover, we emphasize that the policy specification regulates the sensitivity on the *original value* of the variable. For example, consider $\Gamma(h) = \mathsf{S}, \Gamma(x) = s?\mathsf{S} \rightarrow \mathsf{P}$ for program:

$$x := h;\ \texttt{EventOn}(s);\ \texttt{output}(\mathsf{P}, x);$$

The policy on x states that its original value, rather than its value right before output (i.e., the value of h), is declassified to P. Hence, the program is insecure. Therefore, the specification language rules out laundering attacks [45], [47], which launders secrets not intended for declassification.

b) Where: The *where* dimension regulates *level locality* (where information may flow to) and *code locality* (where physically in the code that information's sensitivity changes). It is obvious that a label $cnd?b_1 \circ b_2$ declare where information may flow to after a policy change, and the security event s with the security commands $\texttt{EventOn}(s)$ and $\texttt{EventOff}(s)$ specify the code locations where sensitivity changes.

c) When: The *when* dimension is a temporal dimension, pertaining to when information's sensitivity changes. This is specified by the trigger condition *cnd*. For example, a policy $(\texttt{paid}?\mathsf{P} \leftarrow \mathsf{S})$ allows associated information (e.g., software key) to be released when payment has been received. This is an instance of "Relative" specification defined in [47].

d) Who: The *who* dimension specifies a *principal/role*, who controls the change of sensitivity; one example is the Decentralized Label Model (DLM) [40], which explicitly defines ownership in security labels. While our specification language does not explicitly define ownership, we show next that it is expressive enough to encode Flow Locks [13] and Paralocks [15], which in turn are expressive enough to encode DLM [15]. Hence, the specification language also covers the who dimension to some extent.

[4] We note that our encoding requires all changes to the acts-for relation to be *anticipated*, whereas a general delegation/revocation policy might also offer the flexibility of changing the acts-for relation dynamically.

`// x: {D,N}⇒a`	`// x:` $s_N?\{a\} \leftrightarrows \{\}$
`// y: {N}⇒a`	`// y:` $s_N?\{a\} \leftrightarrows \{\}$
`// z: {}⇒a`	`// z:` $\{a\}$
`open(D);`	$\texttt{EventOn}(s_D)$;
`y:=x;`	`y:=x;` $\texttt{output}(s_N?\{a\} \leftrightarrows \{\}, y)$
`close(D);`	$\texttt{EventOff}(s_D)$;
`open(N);`	$\texttt{EventOn}(s_N)$;
`z:=y;`	`z:=y;` $\texttt{output}(\{a\}, z)$

Fig. 3: An Example of Encoding Paralock for $A = \langle a, \{D\} \rangle$.

3) Encoding Flow Locks [13]: Both Flow Locks [13] and its successor Paralocks [15] introduce *locks*, denoted as σ, to construct dynamic policies. Let **Locks** be a set of locks, and $\mathbb{P}$ be a set of principals. A "flow lock" policy is specified with the following components:

- Flow locks in the form of $\Sigma \Rightarrow P$ where $\Sigma \subseteq \textbf{Locks}$ is the lock set for principal $P \in \mathbb{P}$.
- Distinguished commands $\texttt{open}(\sigma), \texttt{close}(\sigma)$ that open and close the lock $\sigma \in \textbf{Locks}$.

To simplify notation, we use $\Gamma(x, P) = \Sigma$ to denote the fact that $\{\Sigma \Rightarrow P\}$ is part of the "flow locks" of x. Paralocks security is formalized as *an extension* of Gradual Release [7]. In particular, paralock security is defined based on *sub-security condition* for each hypothetical attacker $A = (P_A, \Sigma_A)$ where $P_A \in \mathbb{P}$ and $\Sigma_A \subseteq \textbf{Locks}$:

- A variable is considered "public" for attacker A when $\Gamma(x, P_A) = \Sigma_x \subseteq \Sigma_A$; otherwise, it is considered "secret" for attacker A.
- A "release event", in gradual release sense, is defined as a period of program execution when the set of *opened* locks Σ_{open} satisfies $\Sigma_{open} \not\subseteq \Sigma_A$, as any such lock state might allow some "secret" to A to be released [15].

Consider the example in Figure 3 and a hypothetical attacker $A = (a, \{D\})$. Since $\Gamma(x, a) = \{D, N\} \not\subseteq \{D\}$, variable x is considered "secret" for A. Moreover, the first assignment $\texttt{y} := \texttt{x}$ is *not* under a release event since the set of opened locks, $\{D\}$, satisfies $\{D\} \subseteq \{D\} = \Sigma_A$. On the other hand, the last assignment $\texttt{z} := \texttt{y}$ is under a release event since $\Sigma_{open} = \{D, N\} \not\subseteq \{D\} = \Sigma_A$. Therefore, we need to ensure that the first assignment, which is not under a releasing event, does not reveal the value of x.

For each concrete attacker $A = (P_A, \Sigma_A)$, we can encode Paralocks security as follows:

- We define a security event s_σ for each lock $\sigma \in \Sigma$ and the lock command $\texttt{open}(\sigma)$ (resp. $\texttt{close}(\sigma)$) is converted to $\texttt{EventOn}(s_\sigma)$ (resp. $\texttt{EventOff}(s_\sigma)$).
- Let $\Gamma'(x) = \{P_A\}$ when $\Gamma(x, P_A) = \Sigma_x \subseteq \Sigma_A$; otherwise, $\Gamma'(x) = \{\}$ (i.e., secret for P_A).

- We define the dynamic policy of x as $cnd?\{P_A\} \leftrightarrows \Gamma'(x)$ where $cnd \triangleq \bigvee_{\sigma \notin \Sigma_A} s_\sigma$. Note that by definition, information x is under a release event (i.e., has level set $\{P_A\}$) whenever at least one lock not in Σ_A is currently open, which implies that $\Sigma_{open} \not\subseteq \Sigma_A$, as defined in Paralock.

Consider the transformed code in Figure 3 for an attacker $A = \langle a, \{D\}\rangle$. We note that under the encoding, the first assignment $y := x$ is *not* under declassification of x since s_N is closed and the effective level set of x is $\{\}$, which prohibits attacker A from learning its value. This is consistent with Paralock semantics, where the first assignment is *not* under a release event. On the other hand, the second assignment $z := y$ is under a release event since the effective level set of both x and y are $\{a\}$ (note that we have s_N as the security event is turned on), which allows attacker A to learn their values. This is also consistent with Paralock semantics, where the second assignment is under a release event.

Hence, we can encode Paralocks by explicitly checking that for every hypothetical attacker, the corresponding transformed program is secure.

C. Interpretation of Security Specification

Intuitively, the security specification in Figure 2 specifies at each program execution point, what is the sensitivity of the associated information. We formalize this as an interpretation function of the label, denoted as $[\![b]\!]_\tau$, which takes in a label b and a trace τ, and returns a level set L as information flow restrictions at the *end* of τ.

a) Execution trace: As standard, we model program state, called memory m, as a mapping from program variables and security events to their values. The small-step semantics of the source language is mostly standard (hence omitted), with exception of the output and security event commands:

$$\frac{\langle e, m\rangle \Downarrow v}{\langle \texttt{output}(b, e), m\rangle \xrightarrow{\langle b,v\rangle} \langle \texttt{skip}, m\rangle}\ \textsc{S-Output}$$

$$\langle \texttt{EventOn}(s), m\rangle \rightarrow \langle \texttt{skip}, m\{s \mapsto \texttt{true}\}\rangle\ \textsc{S-Set}$$

$$\langle \texttt{EventOff}(s), m\rangle \rightarrow \langle \texttt{skip}, m\{s \mapsto \texttt{false}\}\rangle\ \textsc{S-Unset}$$

The semantics records all output events, in the form of $\langle b, v\rangle$, during program execution, as these are the only information release events during program execution. Moreover, the distinguished security events s are treated as boolean variables, which can only be set/unset by the security event commands.

Based on the small-step semantics, executing a program c under initial memory m produces an execution trace τ with potentially empty output events:

$$\langle c, m\rangle \xrightarrow{b_1, v_1} \langle c_1, m_1\rangle \cdots \xrightarrow{b_n, v_n} \langle c_n, m_n\rangle.$$

We use $\tau^{[i]}$ to denote the configuration (i.e., a pair of program and memory) after the i-th evaluation step in the τ, and $\|\tau\|$ to denote the number of evaluation steps in the trace. For example, $\tau^{[0]}$ is always the initial state of the execution, $\tau^{[\|\tau\|]}$ is the ending state of a terminating trace τ. We use $\tau^{[:i]}$ (resp. $\tau^{[i:]}$) to denote a prefix (resp. postfix) subtrace of τ from the initial state up to (starting from) the i-th evaluation step. We use $\tau^{[i:j]}$ to denote the subtrace of τ between i-th and j-th (inclusive) evaluation steps. Finally, we write $\tau_1 \preccurlyeq \tau_2$ when τ_1 is a prefix of τ_2.

$$[\![L]\!]_\tau = L$$

$$[\![cnd?b_1 \rightarrow b_2]\!]_\tau = \begin{cases} [\![b_1]\!]_\tau, & \texttt{first}(cnd, \tau, \texttt{false}) = -1 \\ [\![b_2]\!]_{\tau^{[i:]}}, & i = \texttt{first}(cnd, \tau, \texttt{false}) \geq 0 \end{cases}$$

$$[\![cnd?b_1 \leftarrow b_2]\!]_\tau = \begin{cases} [\![b_2]\!]_\tau, & \texttt{first}(cnd, \tau, \texttt{true}) = -1 \\ [\![b_1]\!]_{\tau^{[i:]}}, & i = \texttt{first}(cnd, \tau, \texttt{true}) \geq 0 \end{cases}$$

$$[\![cnd?b_1 \leftrightarrows b_2]\!]_\tau = \begin{cases} [\![b_1]\!]_{\tau^{[i+1:]}}, & i = \texttt{last}(\texttt{cnd}, \tau, \texttt{false}) \neq \|\tau\| \\ [\![b_2]\!]_{\tau^{[i+1:]}}, & i = \texttt{last}(\texttt{cnd}, \tau, \texttt{true}) \neq \|\tau\| \end{cases}$$

where $\texttt{first}(cnd, \tau, bl)$ returns the first index of τ such that cnd evaluates to bl, or -1 if such an index does not exist; $\texttt{last}(cnd, \tau, bl)$ returns the last index of τ such that cnd evaluates to bl, or -1 if such an index does not exist.

Fig. 4: Interpretation of Security Labels

b) Interpretation of labels: We formalize the label semantics $[\![b]\!]_\tau$ in Figure 4. $[\![b]\!]_\tau$ returns a level set L that precisely specifies where the information with policy b can flow to at the end of trace τ. For a (static) level set L, its interpretation is simply L regardless of τ.

For more complicated labels, the semantics also considers the temporal aspect of label changes. For example, a one-time mutation label $cnd?b_1 \rightarrow b_2$ allows a one-time sensitivity change from b_1 to b_2 when the first time that cnd evaluates to `false`. Hence, let i be the *first* index of τ such that cnd evaluates to `false`. Then, $[\![cnd?b_1 \rightarrow b_2]\!]_\tau$ reduces to $[\![b_1]\!]_\tau$ when no such i exists (i.e., cnd always evaluates to `true` in τ), and it reduces to $[\![b_2]\!]_{\tau^{[i:]}}$ otherwise. Note that in the latter case, it reduces to $[\![b_2]\!]_{\tau^{[i:]}}$ rather than $[\![b_2]\!]_\tau$ to properly handle nested conditions: any nested condition in b_2 can only be evaluated *after* cnd becomes `false`. The dual with $\leftarrow$ is defined in a similar way. Note that $cnd?b_1 \rightarrow b_2$ and $\neg cnd?b_2 \leftarrow b_1$ are semantically the same; we introduce both for convenience.

Finally, the bi-directional label (with $\leftrightarrows$) is interpreted purely based on the last configuration of τ: let i be the *last* index in τ such that cnd evaluates to `false`. Then, $i \neq \|\tau\|$ implies that cnd evaluates to `true` at the end of τ; hence, the label reduces to b_1. Note that b_1 is evaluated under $\tau^{[i+1:]}$ in this case to properly handle (potentially) nested conditions in b_1: any nested condition in b_1 can only be evaluated *after* cnd becomes `true`.

Moreover, we can derive a *dynamic* specification for each execution point i, written as γ_i, such that

$$\forall x.\ \gamma_i(x) = [\![\Gamma(x)]\!]_{\tau_{[:i]}}$$

Additionally, we overload γ_i to track the dynamic interpretation of a label b for each execution point i:

$$\forall b.\ \gamma_i(b) = [\![b]\!]_{\tau_{[:i]}}$$

To simplify notation, we write

$$\langle c_0, m_0 \rangle \hookrightarrow \vec{t}$$

if the execution $\langle c_0, m_0 \rangle$ *terminates*[5] with an *extended* output sequence $\vec{t}$, which consists of extended output events $t \triangleq \langle b, v, \gamma \rangle$, where b, v are the output events on τ, and γ is the dynamic specification at the corresponding execution point. We use $t.b$, $t.v$ and $t.\gamma$ to refer to each component in the extended output event. We use the same index notation as in trace, where $\vec{t}^{[i]}$ returns the i-th output event, and $\vec{t}^{[:i]}$ returns the prefix output sequence up to (included) the i-th output. $\vec{t}^{[:0]}$ returns an empty sequence.

IV. Dynamic Release

In this section, we define Dynamic Release, an end-to-end information flow policy that allows information flow restrictions to downgrade and upgrade in arbitrary ways.

A. Semantics Notations

a) Memory Closure: For various reasons, we need to define a set of initial memories that are indistinguishable from some memory m. Given a set of variables X, we define the memory closure of m to be a set of memories who agree on the value of each variable $x \in X$:

Definition 2 (Memory Closure): Given a memory m and a set of variables X, the memory closure of m on X is:

$$[\![m]\!]_X \triangleq \{m' \mid \forall x \in X.\ m(x) = m'(x)\}$$

For simplicity, we use the following short-hands:

$$[\![m]\!]_{L,\gamma} \triangleq [\![m]\!]_{\{x \ \mid \gamma(x) \sqsubseteq L\}}$$
$$[\![m]\!]_{\neq b} \triangleq [\![m]\!]_{\{x \ \mid \Gamma(x) \neq b\}}$$

where $[\![m]\!]_{L,\gamma}$ is the memory closure on all variables whose sensitivity level is less or equally restrictive than level L according to γ, and $[\![m]\!]_{\neq b}$ is the memory closure on variables whose security policy is not b: a set of memories whose values only differ on variables with policy b.

b) Trace filter: For various reasons, we need a filter on output traces to focus on relevant subtraces (e.g., to filter out outputs that are not visible to an attacker). Each trace filter can be defined as a Boolean function on $\langle b, v, \gamma \rangle$. With a filter function f (that returns `false` for irrelevant outputs), we define the projection of outputs as follows:

Definition 3 (Projection of Trace):

$$\lfloor \vec{t} \rfloor_f \triangleq \langle \langle b, v, \gamma \rangle \in \vec{t} \ \mid \ f(b, v, \gamma) \rangle$$

We use the following short-hand for a commonly used filter, L-projection filter, where the resulting trace consists of outputs to channels that are observable to an attacker at level L :

$$\lfloor \vec{t} \rfloor_L \triangleq \lfloor \vec{t} \rfloor_{\lambda b, v, \gamma.\ \gamma(b) \sqsubseteq L}$$

[5] In this paper, we only consider output sequences $\vec{t}$ produced by $\langle c_0, m_0 \rangle \hookrightarrow \vec{t}$. Hence, only the terminating executions are considered in this paper, making our knowledge and security definitions in Section IV termination-insensitive. Termination sensitivity is an orthogonal issue to the scope of this paper: dynamic policy.

B. Key Factors of Formalizing a Dynamic Policy

Before formalizing Dynamic Release, we first introduce knowledge-based security (i.e., epistemic security) [7], which is widely used in the context of dynamic policy. Our formalization is built on the following informal security statement, which is motivated by [3]:

> A program c is secure iff for any event t produced by c, the "knowledge" gained about secret by observing t is bounded by what's allowed by the policy at t.

We first introduce a few building blocks to formalize "knowledge" and "allowance" (i.e., the allowed leakage).

1) Indistinguishability: A key component of information flow security is to define trace indistinguishability: whether two program execution traces are distinguishable to an attacker or not. Given an attacker at level set L, each release event $\langle b, v, \gamma \rangle$ is observable iff $\gamma(b) \sqsubseteq L$ by the attack model. Hence, as standard, we define an indistinguishability relation, written as $\sim_L$, on traces as

$$\sim_L \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L\}$$

Note that an attacker cannot rule out any execution whose prefix matches t_1. Hence, the prefix relation is used instead of identity.

2) Knowledge gained from observation: Following the original definition of knowledge in [7], we define the knowledge gained by an attacker at level set L via observing a trace $\vec{t}$ produced by a program c as:[6]

$$k_1(c, \vec{t}, L) \triangleq \{m \ \mid \ \langle c, m \rangle \hookrightarrow \vec{t'} \wedge \vec{t} \sim_L \vec{t'}\} \qquad (3)$$

Intuitively, it states that if one initial memory m produces a trace that is indistinguishable from $\vec{t}$, then the attacker cannot rule out m as one possible initial memory. Note that by definition, the smaller the knowledge set is, the more information (knowledge) is revealed to the attacker.

Recall that by definition, $\langle c, m \rangle \hookrightarrow \vec{t'}$ only considers *terminating* program executions. Hence, the knowledge definition above is the *termination-insensitive* version of knowledge defined in [7]. As a consequence, the security semantics that we define in this paper is also termination-insensitive.

3) Policy Allowance: To formalize security, we also need to define for each output event t on a trace, what is the allowed leakage to an attacker at a level set L. As knowledge, policy allowance, written as $\mathcal{A}(m, \vec{t}, b, L)$, is defined as a set of memories that (1) only differs from the actual initial memory m in variables whose label is b, and (2) should remain indistinguishable to m for an attacker at L who observes an output sequence $\vec{t}$ according to the dynamic policy.

Consider a dynamic label $b \in \mathbb{B}$, memory m and output sequence $\vec{t}$ of interest, as well as an attacker at level L, one possible policy allowance can be:

$$\mathcal{A}(m, \vec{t}, b, L) \triangleq [\![m]\!]_{\neq b}$$

[6] We slightly modified the original definition to exclude "initial knowledge", the attacker's knowledge before executing the program.

Intuitively, it specifies the initial knowledge of an attacker at level set L: the attacker cannot distinguish any value difference among variables with the dynamic label b. Thus, any variable with the label b is initially indistinguishable to the attacker. Eventually, Dynamic Release checks that for each label $b \in \mathbb{B}$, gained knowledge is bounded by the allowance with respect to b. Hence, the security of each variable is checked.

C. Challenges of Formalizing a General Dynamic Policy

We next show that it is a challenging task to formalize the security of a general-purpose dynamic policy that allows downgrading and upgrading to occur in arbitrary ways.

Challenge 1: Permitting both increasing and decreasing knowledge: Allowing both downgrading and upgrading in arbitrary ways means that our general policy must permit reasoning about both increasing knowledge (as in declassification) and decreasing knowledge (as in erasure). While Equation 3 and its variants are widely used to formalize declassification policy [7], [15], they *cannot* reason about increasing knowledge. The reason is that, it is easy to check that for any $c, \vec{t} \preccurlyeq \vec{t'}, L$, the knowledge set decreases:

$$\vec{t} \preccurlyeq \vec{t'} \Rightarrow k_1(c, \vec{t}, L) \supseteq k_1(c, \vec{t'}, L)$$

As other variants, the knowledge set k_1 is monotonically decreasing (hence, the knowledge that it represents is increasing by definition) as more events on the same execution are revealed to an attacker [7], [3], [51].

However, we need to reason about decreasing knowledge for an erasure policy. Consider the example in Figure 1-B, where the value of credit card is revealed by the first output at Line 3. Given any program execution $\langle c, m\rangle \hookrightarrow \vec{t}$, we have $k_1(c, \vec{t}^{[:i]}, M) = \{m\}$ for all $i \geq 1$. However, as the sensitivity of *credit_card* upgrades from M to $\top$ when $i = 2$ (i.e., the second output), the secure program (i) can be incorrectly rejected: $k_1(c, \vec{t}^{[:2]}, M) = \{m\}$ means that the value of *credit_card* is known to the attacker, which violates the erasure policy at that point.

Observation 1. Equation 3 is not suitable for an upgrading policy, since it fails to reason about decreasing knowledge. The issue is that knowledge gained from $\vec{t}$ is defined as the full knowledge gained from observing *all* outputs on $\vec{t}$. Return to the secure program in Figure 1-B.i. We note that the first and second outputs together reveal the value of *credit_card*, but the second event alone reveals no information, as it always outputs 0. Hence, we need to precisely define the *exact knowledge* gained from learning *each* output to permit both increasing and decreasing knowledge.

Challenge 2: Indistinguishability $\sim_L$ is inadequate for a general dynamic policy: As shown earlier, indistinguishability $\sim_L$ is an important component of a knowledge definition; intuitively, by observing an execution $\langle c, m\rangle \hookrightarrow \vec{t}$, an attacker at level set L can rule out any initial memory m' where $m \not\sim_L m'$ (i.e., $m' \notin k_1(c, \vec{t}, L)$). However, the naive definition of $\sim_L$ might be inadequate for declassified outputs. Consider the following secure program, where x is first downgraded to P and then upgraded to S.

```
1  // x : P
2  if (x>0) output(P,1);
3  output(P,1)
4  // x : S
5  output(P,2)
```

Note that the program is secure since the only output when x is secret reveals a constant value. Assume that the initial value of x is either 0 or 1. Then, there are two possible executions of the program with $\gamma_1(x) = \mathrm{P}$ and $\gamma_2(x) = \mathrm{S}$:

$$\langle c, m_1\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 2, \gamma_2\rangle$$
$$\langle c, m_2\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 2, \gamma_2\rangle$$

The issue is in the first execution. By observing the first output, an attacker at P cannot tell if the execution starts from m_1 or m_2, as both of them first output 1. However, the attacker can rule out m_2 by observing the second output with the value of 2. Note that the change of knowledge (from $\{m_1, m_2\}$ to $\{m_1\}$) violates the dynamic policy governing the second output: the policy on x is S, which prohibits the learning of the initial value of x.

Observation 2. The inadequacy of relation $\sim_L$ roots from the fact that, due to downgrading, the public outputs of different executions might have various lengths. Therefore, outputs at the same index but produced by different executions might be incomparable. To resolve the issue, we observe that any information release (of x) when x is P is *ineffective*, in the sense that the restriction on x is not in effect. In the example above, the outputs with value 1 are all ineffective, as x is public when the outputs at lines 2 and 3 are produced. This observation motivates the secret projection filter, which finds out the *effective* outputs for a given secret.

Definition 4 (Secret Projection of Trace): Given a policy b and an attacker at level L, a secret projection of trace is a subtrace where information with policy b cannot flow to L and the output channel is visible to L :

$$\lfloor \vec{t} \rceil_{b,L} \triangleq \lfloor \vec{t} \rceil_{\lambda b', n, \gamma.\ \gamma(b) \not\subseteq L\ \wedge\ \gamma(b') \subseteq L}$$

Return to the example above, the effective subtraces starting from m_1 and m_2 are both $\langle \mathrm{P}, 2, \gamma_2(x) = \mathrm{S}\rangle$, which remains indistinguishable to an attacker at level P.

Challenge 3: Effectiveness is also inadequate: With Observation 2, it might be attempting to define indistinguishability based on $\lfloor \vec{t} \rceil_{b,L}$, rather than $\lfloor \vec{t} \rceil_{L}$. However, doing so is problematic as shown by the following program.

```
// x : S
if (x>0) output(P, 1);
// x : P
if (x<=0) output(P, 1);
```

With two initial memories $m_1(x) = 0, m_2(x) = 1$, we have

$$\langle c, m_1\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1(x) = \mathrm{P}\rangle$$
$$\langle c, m_2\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_2(x) = \mathrm{S}\rangle$$

Note that only the value of x is revealed on the public channel. Hence, the program is secure as it always outputs 1. However, the effective subtrace starting from m_1 is $\emptyset$ and that starting from m_2 is $\langle \mathrm{P}, 1, \gamma_2(x) = \mathrm{S}\rangle$, suggesting that the program is insecure: the value of x is revealed by the first output from m_2, while the policy at that point (S) disallows so.

Observation 3. We note that both indistinguiability and effectiveness are important building blocks of a general-purpose dynamic policy. However, the challenge is how to combine them in a meaningful way. We will build our security definition on both concepts and justify why the new definition is meaningful in Section IV-D.

Challenge 4: Transient vs. Persistent Policy: So far, the policy allowance $\mathcal{A}(m, \vec{t}, b, L)$ ignores what information has been leaked in the past. However, in the persistent case such as Figure 1-C, the learned information (*note*) remains accessible even after the policy on *book* upgrades. In general, we define transient and persistent policy as:

Definition 5 (Transient and Persistent Policy): A dynamic security policy is persistent if it always allows to reveal information that has been revealed in the past. Otherwise, the policy is transient.

Observation 4. Both transient and persistent policy have real-world application scenarios. Hence, a general-purpose dynamic policy should support both kinds of policies, in a unified way.

D. Dynamic Release

We have introduced all ingredients to formalize Dynamic Release, a novel end-to-end, general-purpose dynamic policy.

To tackle the challenges above, we first formalize the attacker's knowledge gained by observing the *last* event t' on a trace $\vec{t} \cdot t'$. Note that simply computing the knowledge difference between observing $\vec{t} \cdot t'$ and observing $\vec{t}$ does not work. Consider the example in Figure 1-B.ii. Given any program execution $\langle c, m\rangle \hookrightarrow \vec{t}$, we have $k_1(c, \vec{t}^{[:i]}, M) = \{m\}$ for all $i \geq 1$. Hence, the difference between the knowledge gained with or without the output at Line 6 is $\emptyset$, suggesting that no knowledge is gained by observing the output at Line 6 alone, which is incorrect as it reveals the credit card number.

Instead, we take inspiration from probabilities to formalize the attacker's knowledge gained by observing a single event on a trace. Consider a program c that produces the following sequences of numbers give the corresponding inputs:

$$\text{input 1: } s_1 = (1 \cdot 1 \cdot 3)$$
$$\text{input 2: } s_2 = (2 \cdot 2 \cdot 3)$$
$$\text{input 3: } s_3 = (1 \cdot 1 \cdot 3)$$
$$\text{input 4: } s_4 = (2 \cdot 2 \cdot 2)$$

Consider the following question: what is the probability that the program generates a sequence where the last number is identical to the last number of s_1? Obviously, besides s_1, we also need to consider sequences s_2 and s_3 since albeit a different sequence, s_2 is *consistent* with s_1 in the sense that the last output is 3, and s_3 is *indistinguishable* (i.e., identical) to s_1. More precisely, we can compute the probability as follows:

$$\Sigma_{s \in \texttt{consist}(s_1)} P(s)$$

where the consistent set $\texttt{consist}(s_1)$ is the set of sequences that produce the same last number as s_1, i.e., $\{(1 \cdot 1 \cdot 3), (2 \cdot 2 \cdot 3)\}$. Assuming a uniform distribution on program inputs, we have that the probability is $P(1 \cdot 1 \cdot 3) + P(2 \cdot 2 \cdot 3) = (0.25 + 0.25) + 0.25 = 0.75$. Note that the indistinguishable sequences s_1 and s_3 are implicitly accounted for in $P(1 \cdot 1 \cdot 3)$.

To compute the knowledge associated with the *last* event on a trace $\vec{t}$, we first use effectiveness to identify *consistent* traces whose last event on the effective subset is the same:

Definition 6 (Consistency Relation): Two output sequences $\vec{t_1}$ and $\vec{t_2}$ are consistent w.r.t. a policy b and an attack level L, written as $\vec{t_1} \equiv_{b,L} \vec{t_2}$ if

$$n = \|\lfloor \vec{t_1} \rfloor_{b,L}\| = \|\lfloor \vec{t_2} \rfloor_{b,L}\| \wedge \lfloor \vec{t_1} \rfloor_{b,L}^{[n]} = \lfloor \vec{t_2} \rfloor_{b,L}^{[n]}$$

Note that despite the extra complicity due to trace projection, the consistency relation is similar to the consistent set $\texttt{consist}(s_1)$ in the probability computation example. Next, we define the precise knowledge gained from the last event of $\vec{t}$ based on both the consistency relation and knowledge. Note that since knowledge is a set of memories, rather than a number, the summation in the probability case is replaced by a set union. Similar to the probability of observing each sequence, the knowledge k_1 also implicitly accounts for all indistinguishable traces (Equation 3).

Definition 7 (Attacker's Knowledge Gained from the Last Event): For an attacker at level set L, the attacker's knowledge w.r.t. information with policy b, after observing the last event of an output sequence $\vec{t}$ of program c, is the set of all initial memories that produce an output sequence that is indistinguishable to some consistent counterpart of $\vec{t}$:

$$k_2(c, \vec{t}, L, b) = \bigcup_{\exists m', j.\ \langle c, m'\rangle \hookrightarrow \vec{t'} \wedge \vec{t'}^{[:j]} \equiv_{b,L} \vec{t}} k_1(c, \vec{t'}^{[:j]}, L)$$

To see how Definition 7 tackles Challenges 2 and 3, we revisit the code example under each challenge.

- Challenge 2: Recall that with $m_1(x) = 0$, $m_2(x) = 1$, $\gamma_1(x) = \mathrm{P}$ and $\gamma_2(x) = \mathrm{S}$, there are two execution traces

$$\langle c, m_1\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 2, \gamma_2\rangle$$
$$\langle c, m_2\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 1, \gamma_1\rangle \cdot \langle \mathrm{P}, 2, \gamma_2\rangle$$

  It is easy to check that the two output sequences are consistent w.r.t. the label of x and an attacker at P according to Definition 6 since all outputs are ineffective. Hence, in both traces, the knowledge gained from the last output is $\{m_0, m_1\}$, due to the big union in k_2. Hence, we correctly conclude that no information is leaked by the last output in both traces.
- Challenge 3: Recall that with $m_1(x) = 0$, $m_2(x) = 1$, there are two execution traces

$$\langle c, m_1\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_1(x) = \mathrm{P}\rangle$$
$$\langle c, m_2\rangle \hookrightarrow \langle \mathrm{P}, 1, \gamma_2(x) = \mathrm{S}\rangle$$

  While the two traces are not consistent with each other, we know that $k_1(c, \langle \mathrm{P}, 1, \gamma_2(x) = \mathrm{S}\rangle, \mathrm{P}) = \{\langle \mathrm{P}, 1, \gamma_1(x) = \mathrm{P}\rangle, \langle \mathrm{P}, 1, \gamma_2(x) = \mathrm{S}\rangle\}$ since the two traces satisfy $\sim_{\mathrm{P}}$. Hence, the knowledge gained from the last event is $\{m_0, m_1\}$, and we correctly conclude that no information is leaked by the last output.

To tackle Challenge 4, we observe that a persistent policy allows information leaked in the past to be released again, while a transient policy disallows so. This is made precise by the following refinement of policy allowance:

$$\mathcal{A}(m, \vec{t}, b, L) \triangleq \begin{cases} [\![m]\!]_{\neq b}, & b \text{ is transient} \\ [\![m]\!]_{\neq b} \cap k_1(c, \vec{t}^{[:\|\vec{t}\|-1]}, L), & b \text{ is persistent} \end{cases} \quad (4)$$

where $k_1(c, \vec{t}^{[:\|\vec{t}\|-1]}, L)$ is the knowledge from every output event in $\vec{t}$ except the last one. Note that since the knowledge here represents the cumulative knowledge gained from observing all events, we use the standard knowledge k_1 instead of the knowledge gained from the last event k_2 here.

Putting everything together, we have Dynamic Release security, where for any output of the program, the attacker's knowledge gained from observing the output is always bounded by the policy allowance at that output point.

Definition 8 (Dynamic Release):

$$\forall m, L \subseteq \mathbb{L}, b \in \mathbb{B}, \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t} \implies \forall 1 \leq i \leq \|\vec{t}\|.$$

$$k_2(c, \vec{t}^{[:i]}, L, b) \supseteq \begin{cases} [\![m]\!]_{\neq b}, & \text{transient} \\ [\![m]\!]_{\neq b} \cap k_1(c, \vec{t}^{[:i-1]}, L), & \text{persistent} \end{cases}$$

V. Semantics Framework For Dynamic Policy

While various forms of formal policy semantics exist in the literature, different policies have very different nature of the security conditions (i.e., noninterference, bisimulation and epistemic [16]). In this section, we generalize the formalization of Dynamic Release (Definition 8) by abstracting away its key building blocks. Then we convert various existing dynamic policies into the formalization framework and provide the first apple-to-apple comparison between those policies.

A. Formalization Framework for Dynamic Policies

We first abstract way a few building blocks of Definition 8. To define them more concretely, we consider an output sequence $\vec{t}$ produced by $\langle c, m\rangle$, i.e., $\langle c, m\rangle \hookrightarrow \vec{t}$, as the context.

As already discussed in Section IV, the building blocks are:

- *Output Indistinguishability, written as* $\sim$: two output sequences $\vec{t_1}$ and $\vec{t_2}$ satisfies $\vec{t_1} \sim \vec{t_2}$ when they are considered indistinguishable to the attacker.
- *Policy Allowance, written as* $\mathcal{A}$: a set of initial memory that should be indistinguishable to m according to the dynamic policy.
- *Consistency Relation, written as* $\equiv$: when trying to precisely define the knowledge gained from each output event, two sequences are considered "consistent", even if they are not identical (Definition 6).

With the abstracted parameters, we first generalize the knowledge definition of k_1 (Equation 3) on an arbitrary relation $\sim$ on output sequences:

Definition 9 (Generalized Knowledge):

$$\mathcal{K}(c, \vec{t}, \sim) \triangleq \{m \mid \langle c, m\rangle \hookrightarrow \vec{t'} \wedge \vec{t} \sim \vec{t'}\ \} \quad (5)$$

Therefore, with abstract $\sim$, $\mathcal{A}$ and $\equiv$, we can generalize Definition 8 as the following framework:

Definition 10 (Formalization Framework): Given trace indistinguishability relation $\sim$, consistency relation $\equiv$ and policy allowance $\mathcal{A}$, a command c satisfies a dynamic policy iff the knowledge gained from observing any output does not exceed its corresponding policy allowance:

$$\forall m, L \subseteq \mathbb{L}, b \in \mathbb{B}, \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t} \implies \forall 1 \leq i \leq \|\vec{t}\|.$$

$$\bigcup_{\exists m', j.\ \langle c, m'\rangle \hookrightarrow \vec{t'} \wedge \vec{t'}^{[:j]} \equiv \vec{t}^{[:i]}} \mathcal{K}(c, \vec{t'}^{[:j]}, \sim) \supseteq \mathcal{A}(m, \vec{t}^{[:i]}, b, L)$$

Let $\sim_{\mathtt{DR}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L\}$, $\mathcal{A}_{\mathtt{DR}}$ be as defined in Equation (4), and $\equiv_{\mathtt{DR}}$ be as defined in Definition 7, it is easy to check that Definition 10 is instantiated to Definition 8.

Moreover, when $\equiv$ is instantiated with an equality relation $=$, a case that we have seen in all existing dynamic policies, the general framework can be simplified to the following form:

$$\forall c, m, L \subseteq \mathbb{L}, b \in \mathbb{B}, \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t} \implies \forall 1 \leq i \leq \|\vec{t}\|.$$
$$\mathcal{K}(c, \vec{t}^{[:i]}, \sim) \supseteq \mathcal{A}(m, \vec{t}^{[:i]}, b, L)$$

We use this simpler form for existing dynamic policies where consistency is simply defined as equivalence.

B. Existing works in the formalization framework

We incorporate existing definitions into the formalization framework; the results are summarized in Table I. We first highlight a few insights from Table I. Then, for each work (except for Paralock due to space constraint), we sketch how to convert it into the formalization framework. The conversion of Paralock and the correctness proofs of all conversions are available in the Appendix.

1) Insights from Table I: To the best of our knowledge, this is the first work that enables apple-to-apple comparison between various dynamic policies. We highlight a few insights.

First, an erasure policy (e.g., According to Policy and Cryptographic Erasure) defines indistinguishability $\sim$ in a substantially more complicated way compared with others. The complexity suggests that formalizing an erasure policy is more involved compared with other dynamic policies.

Second, besides Dynamic Release, Gradual Release, Paralock and Forgetful Attacker also have $\mathcal{K}(c, \vec{t}^{[:i-1]}, \sim)$ as part of policy allowance. Recall that $\mathcal{K}(c, \vec{t}^{[:i-1]}, \sim)$ represents the *past knowledge* excluding the last output on $\vec{t}$. Hence, these policies are *persistent* policies. On the other hand, all other dynamic policies are transient policies.

Third, since an erasure policy by definition is transient, persistent policies such as Gradual Release and Paralock cannot be used as an erasure policy, such as the example in Figure 1-B.

2) Gradual Release: Gradual Release assumes a mapping Γ from variables to levels in a Denning-style lattice. A *release event* is generated by a special command $x :=$ $\mathtt{declassify}(e)$. Informally, a program is secure when illegal

	$\sim(\vec{t_1}, \vec{t_2})$	$\mathcal{A}(m, \vec{t}, b, L)$, $i = \|\vec{t}\|$	$\equiv(\vec{t_1}, \vec{t_2})$
Gradual Release	$\lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L$	$[\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\texttt{GR}})$	$=$
Tight Gradual Release	$\lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L$	$[\![m]\!]_{L, \vec{t}^{[i]}.\gamma}$	$=$
NI According to Policy	$\exists R.\ \forall (i,j) \in R.\ \ \lfloor \vec{t_1}^{[i]} \rfloor_{b,L} \cong \lfloor \vec{t_2}^{[j]} \rfloor_{b,L}$	$[\![m]\!]_{\neq b}$	$=$
Cryptographic Erasure	$\lfloor \vec{t_1} \rfloor_L = \lfloor \vec{t_2} \rfloor_L^{[i:j]}$	$\bigcap_{t \in \vec{t}} [\![m]\!]_{L, t.\gamma}$	$=$
Forgetful Attacker	$\exists \vec{t'} \preccurlyeq \vec{t_2}.\ \texttt{atk}(\lfloor \vec{t_1} \rfloor_L) = \texttt{atk}(\lfloor \vec{t'} \rfloor_L)$	$[\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\texttt{FA}})$	$=$
Paralock	$\lfloor \vec{t_1} \rfloor_A \preccurlyeq \lfloor \vec{t_2} \rfloor_A$	$\begin{cases} [\![m]\!]_A \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\texttt{PL}}), & \vec{t}_{[i]}.\Delta \subseteq \Sigma_A \\ [\![m]\!]_{\emptyset}, & \text{otherwise} \end{cases}$	$=$
Dynamic Release	$\lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L$	$\begin{cases} [\![m]\!]_{\neq b}, & b \text{ is transient} \\ [\![m]\!]_{\neq b} \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim), & b \text{ is persistent} \end{cases}$	$\lfloor \vec{t_1} \rfloor_{b,L}^{[n]} = \lfloor \vec{t_2} \rfloor_{b,L}^{[n]}$

TABLE I: Existing End-to-End Security Policies and Dynamic Release Written in the Formalization Framework.

flow w.r.t. Γ only occurs along with release events. Hence, at policy syntax level, we encode a release event as

$$\texttt{EventOn}(r); x := e; \texttt{output}(\Gamma(x), e); \texttt{EventOff}(r);$$

where r is a distinguished event for release, and we set $\forall x.\ \Gamma'(x) = r?\mathbb{L} \leftrightarrows \Gamma(x)$ to state that any leakage of any variable is allowed when this is a release event. But otherwise, the information flow restriction of Γ is obeyed.

At semantics level, Gradual Release is formalized on the insight that "knowledge must remain constant between releases":

Definition 11 (Gradual Release [7]): A program c satisfies gradual release w.r.t. Γ if[7]

$$\forall c, m, L, i, \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t} \implies$$
$$\forall \text{i not release event. } k(c, m, \vec{t}^{[:i]}, L, \Gamma) = k(c, m, \vec{t}^{[:i-1]}, L, \Gamma)$$

where $k(c, m, \vec{t}, L, \Gamma) \triangleq$

$$\{m' \mid m' \in [\![m]\!]_{L,\Gamma} \wedge\ \langle c, m\rangle \hookrightarrow \vec{t'} \wedge \vec{t} \preccurlyeq \vec{t'}\} \tag{6}$$

While the original definition does not immediately fit our framework, we prove that they are equivalent by:

$$\sim_{\texttt{GR}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L\} \qquad \equiv_{\texttt{GR}} \triangleq =$$
$$\mathcal{A}_{\texttt{GR}} \triangleq [\![m]\!]_{L,\ \vec{t}^{\|\vec{t}\|}.\gamma} \cap \mathcal{K}(c, \vec{t}^{[:\|\vec{t}\|-1]}, \sim_{\texttt{GR}})$$

Note that in our encoding, a release event sets security event r which sets all dynamic labels in the form of $r?\mathbb{L} \leftrightarrows \Gamma(x)$ to the least restrictive level set $\mathbb{L}$. Hence, when there is a release event, the allowance check $\mathcal{K}(\dots) \supseteq \mathcal{A}$ trivially true, resembling Definition 11.

Lemma 1: With $\sim \triangleq \sim_{\texttt{GR}}$, $\equiv \triangleq \equiv_{\texttt{GR}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{GR}}$, Definition 10 is equivalent to Definition 11.

3) Tight Gradual Release: Tight Gradual Release [2], [8] is an extension of Gradual Release. Similar to Gradual Release, it assumes a base policy Γ and uses $x := \texttt{declassify}(e)$ to declassify the value of e. However, the encoding of declassification command is different for two reasons. First, we can only encode a subset of Tight Gradual Release where declassification command contains $\texttt{declassify}(x)$, since our language does not fully support partial release (Section III-B2). Second, declassification in Tight Gradual Release is both precise (i.e., only variable x in $\texttt{declassify}(x)$ is downgraded) and permanent (i.e., the sensitivity of x cannot upgrade after x is declassified). Hence, we encode $x' := \texttt{declassify}(x)$ as

$$\texttt{EventOn}(r_x); x' := x; \texttt{output}(\Gamma(x'), x);$$

where r_x is a distinguished security event for releasing just x, and we set $\Gamma'(x) = r_x?\mathbb{L} \leftarrow \Gamma(x)$ to state that x is declassified once r_x is set.

Tight Gradual Release uses the same knowledge definition from Gradual Release, except that its execution traces also dynamically track the set of declassified variables X:

$$\langle c, m, \emptyset\rangle \to^* \langle c', m', X\rangle$$

Definition 12 (Tight Gradual Release): A program c satisfies tight gradual release if for any trace $\vec{t}$, initial memory m and attacker at level L, we have

$$\forall i.\ 1 \le i \le \|\vec{t}\|.\ ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{X_i}) \subseteq k(c, m, \vec{t}^{[:i]}, L, \Gamma)$$

where X_i is the set of declassified variables associated with the i-th output.

Due to the encoding of declassification commands, we know that for each output at index i in $\vec{t}$ we have:

$$[\![m]\!]_{L, \vec{t}^{\|\vec{t}\|}.\gamma} = ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{X_i})$$

Hence, we can rephrase Tight Gradual Release as follows:

$$\sim_{\texttt{TGR}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_L \preccurlyeq \lfloor \vec{t_2} \rfloor_L\}$$
$$\equiv_{\texttt{TGR}} \triangleq = \qquad \mathcal{A}_{\texttt{TGR}} \triangleq [\![m]\!]_{L, \vec{t}^{\|\vec{t}\|}.\gamma}$$

Lemma 2: With $\sim \triangleq \sim_{\texttt{TGR}}$, $\equiv \triangleq \equiv_{\texttt{TGR}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{TGR}}$, Definition 10 is equivalent to Definition 12.

Observation: Tight Gradual Release is more precise than Gradual Release since the policy precisely downgrades the sensitivity of x but not any other variables, while as Gradual Release downgrades all variables under a release event.

Compared to Dynamic Release, the most important difference is that the consistency relation $\equiv$ is defined in com-

[7]Note that $\langle c, m\rangle \hookrightarrow \vec{t}$ only considers terminating program executions by definition. So we used the termination-insensitive version of Gradual Release.

pletely different ways. As discussed in Section IV-C, it is important to define it properly for general dynamic policies. The other major difference is that the security semantics of Tight Gradual Release cannot model erasure policies. Consider the example in Figure 1-B.i with $m_1(credit\_card) = 0$, $m_2(credit\_card) = 1$ and attacker level M. Given a program execution $\langle c, m_1\rangle \hookrightarrow \vec{t}$, we have $\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\mathtt{TGR}}) = \{m_1\}$ for all $i \geq 1$. However, $credit\_card$ is upgraded from M to $\top$ when $i = 2$ (i.e., the second output), the secure program (i) is incorrectly rejected by Tight Gradual Release since $\mathcal{K}(c, \vec{t}^{[:2]}, \sim_{\mathtt{TGR}}) = \{m_1\} \not\supseteq \{m_1, m_2\} = [\![m_1]\!]_{M, \vec{t}^{[2}.\gamma}$.

4) NI According to Policy: Chong and Myers propose noninterference according to policy [18], [19] to integrate erasure and declassification policies. We use the formalization in the more recent paper [19] as the security definition.

This work uses *compound labels*, a similar security specification as ours: a label is either a simple level ℓ drawn from a Denning-style lattice, or in the form of $q_1 \xrightarrow{e} q_2$, where q_1 and q_2 are themselves compound labels. Hence, converting the specification to ours is straightforward.

Noninterference according to policy is defined for each variable in a two-run style. In particular, it requires that for any two program executions where the initial memories differ *only* in the value of the variable of interest, their traces are indistinguishable regarding a *correspondence* R:

Definition 13 (Noninterference According To Policy [19]): A program c is noninterference according to policy if for any variable x (with policy b) we have:[8]

$$\forall m_1, m_2, \ell, \vec{t_1}, \vec{t_2}.\ \forall y \neq x.\ m_1(y) = m_2(y) \\ \wedge \langle c, m_1\rangle \hookrightarrow \vec{t_1} \wedge \langle c, m_2\rangle \hookrightarrow \vec{t_2} \implies \\ \exists R.\ \Big(\forall (i,j) \in R, \ell.\ \ell \notin [\![b]\!]_{\tau_{1[:i]}} \wedge \ell \notin [\![b]\!]_{\tau_{2[:j]}} \Rightarrow \tau_{[i]} \approx_\ell \tau'_{[j]}\Big)$$

where a *correspondence* R between traces τ_1 and τ_2 is a subset of $\mathbb{N} \times \mathbb{N}$ such that:

1) (Completeness) either $\{i \mid (i,j) \in R\} = \{i \in \mathbb{N} \mid i < |\tau_1|\}$ or $\{j \mid (i,j) \in R\} = \{j \in \mathbb{N} \mid j < |\tau_2|\}$, and
2) (Initial configurations) if $\|R\| > 0$ then $(0,0) \in R$, and
3) (Monotonicity) for all $(i,j) \in R$ and $(i',j') \in R$, if $i < i'$ then $j \leq j'$ and symmetrically, if $j < j'$ then $i \leq i'$.

To transform Definition 13 to our framework, we make a few important observations:

- The definition relates two memories that differ in *exactly one variable* (i.e., $\forall y \neq x.\ m_1(y) = m_2(y)$), which is different from the usual low-equivalence requirement in other definitions. However, it is easy to prove that (shown shortly) it is equivalent to a per-policy definition $[\![m]\!]_{\neq b}$ in our framework, that considers memories that differ *only* for variables with a particular policy b.

[8]The original definition uses a specialized *label semantics*, denoted as $[\![b]\!]_{\langle c,m\rangle}$, and requires $(\langle c_i, m_i\rangle, \ell) \notin [\![b]\!]_{\langle c,m\rangle}$ which means that if by the time $\langle c, m\rangle$ reaches state $\langle c_i, m_i\rangle$, confidentiality level ℓ' may not observe the information. It is easy to convert that to $\ell \notin [\![b]\!]_{\tau_{[:i]}}$ in our notation.

- The component of $\ell \notin [\![q]\!]_{\vec{t_1}^{[:i]}} \wedge \ell \notin [\![q]\!]_{\vec{t_2}^{[:j]}}$ filters out non-interesting outputs, which functions the same as the filtering function $\lfloor \vec{t} \rfloor_{b,L}$.
- We define $\cong$ on two output sequence as below:

$$\vec{t_1} \cong \vec{t_2} \iff \neg(\|\vec{t_1}\| = \|\vec{t_2}\| \wedge \exists i.\ \vec{t_1}^{[i]} \neq \vec{t_2}^{[i]})$$

Based on the observations, we convert Definition 13 into our framework as follows:

$$\sim_{\mathtt{AP}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \exists R.\ \forall (i,j) \in R.\ \lfloor \vec{t_1}^{[i]} \rfloor_{b,L} \cong \lfloor \vec{t_2}^{[j]} \rfloor_{b,L}\}$$
$$\equiv_{\mathtt{AP}} \triangleq = \qquad \mathcal{A}_{\mathtt{AP}} \triangleq [\![m]\!]_{\neq b}$$

Lemma 3: With $\sim \triangleq \sim_{\mathtt{AP}}$, $\mathcal{A} \triangleq \mathcal{A}_{\mathtt{AP}}$, and outside equivalence $\equiv \triangleq \equiv'_{\mathtt{AP}}$, Definition 10 is equivalent to Definition 13.

Observation: Compared with Gradual Release and Tight Gradual Release, the most interesting component of According to Policy is in its unique indistinguishability definition, which uses the correspondent relationship R. Intuitively, According to Policy relaxes the indistinguishability definition in the way that two executions are indistinguishable as long as a correspondence R exists to allow decreasing knowledge. However, as shown later in the evaluation, the relaxation with R could be too loose: it falsely accepts some insecure programs.

5) Cryptographic Erasure: Cryptographic erasure [5] also uses compound labels to specify erasure policy and knowledge is defined as:

$$k_{\mathtt{CE}}(c, L, \vec{t}) = \{m \mid \langle c, m\rangle \xrightarrow{\vec{t_1}}{}^{*} \langle c_1, m_1\rangle \xrightarrow{\vec{t_2}}{}^{*} \langle c', m'\rangle \\ \wedge \lfloor \vec{t_2} \rfloor_L = \lfloor \vec{t} \rfloor_L\}$$

Unlike other policies, the definition specifies knowledge based on the *subtrace* relation, rather than the standard *prefix* relation. The reason is that it has a different attack model: it assumes an attacker who might *not* be able to observe program execution from the beginning.

Definition 14 (Cryptographic Erasure Security [5]): A program c is secure if any execution starting with memory m, the following holds:

$$\forall c_0, m_0, c_i, m_i, c_n, m_n, \vec{t_1}, \vec{t_2}, L, i, n. \\ \langle c_0, m_0\rangle \xrightarrow{\vec{t_1}}{}^{*} \langle c_i, m_i\rangle \xrightarrow{\vec{t_2}}{}^{*} \langle c_n, m_n\rangle \\ \Rightarrow k_{\mathtt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{t \in \vec{t_2}} [\![m]\!]_{L, t.\gamma}$$

To model subtraces, we adjust the $\forall 1 \leq i \leq \|\vec{t}\|$ quantifier in the framework with $\forall 1 \leq i < j \leq \|\vec{t}\|$, and write $\vec{t}[i:j]$ for the subtrace between i and j. Then, converting Definition 14 into our framework is relatively straightforward:

$$\sim_{\mathtt{CE}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_L \text{ subtrace of } \lfloor \vec{t_2} \rfloor_L\}$$
$$\equiv_{\mathtt{CE}} \triangleq = \qquad \mathcal{A}_{\mathtt{CE}} \triangleq \bigcap_{t \in \vec{t}} [\![m]\!]_{L, t.\gamma}$$

Lemma 4: With $\sim \triangleq \sim_{\mathtt{CE}}$, $\equiv \triangleq \equiv_{\mathtt{CE}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\mathtt{CE}}$, Definition 10 with adjusted attack model is equivalent to Definition 14.

Observation: Compare with other dynamic policies, the most interesting part of cryptographic erasure is that its indistinguishability and policy allowance are both defined on subtraces; moreover, the latter uses the weakest policy on the subtrace. Intuitively, we can interpret Cryptographic Erasure security as: the subtrace-based knowledge gained from observing a subtrace should be bounded by the smallest allowance (i.e, the weakest policy) on the trace.

6) Forgetful Attacker: Forgetful Attacker [3], [51] is an expressive policy where an attacker can "forget" some learned knowledge. To do so, an attacker is formalized as an automaton $\mathtt{Atk}\langle Q_A, q_{init}, \delta_A\rangle$, where Q_A is a set of attacker's states, $q_{init} \in Q_A$ is the initial state, and δ_A is the transition function. The attacker observes a set of events produced by a program execution, and updates its state accordingly:

$$\begin{aligned}\mathtt{Atk}(\epsilon) =& q_{init}\\ \mathtt{Atk}(\vec{t}_{\preccurlyeq i}) =& \delta(\mathtt{Atk}_A(\vec{t}_{\preccurlyeq i-1}), t^{[i]})\end{aligned}$$

Given a program c, an automaton $\mathtt{Atk}$ and attacker's level L, knowledge is defined as the set of initial memory that could have resulted in the same state in the automaton:

$$\begin{aligned}k_{\mathtt{FA}}(c, L, \mathtt{Atk}, \vec{t}) = \{m \mid \langle c, m\rangle \xrightarrow{\vec{t_1}} {}^*\langle c', m'\rangle \xrightarrow{\vec{t_2}} {}^*m''\\ \wedge\ \mathtt{Atk}(\lfloor\vec{t_1}\rfloor_L) = \mathtt{Atk}(\lfloor\vec{t}\rfloor_L)\}\end{aligned}$$

Definition 15 (Security for Forgetful Attacker [3]): A program c is secure against an attacker $\mathtt{Atk}\langle Q_A, q_{init}, \delta_A\rangle\rangle$ with level L if:

$$\begin{aligned}&\forall c, c', m, m', \vec{t}, t', L.\ \langle c, m_1\rangle \hookrightarrow \vec{t}\cdot t' \Rightarrow\\ &\quad k_{\mathtt{FA}}(c, L, \mathtt{Atk}, \vec{t}\cdot t')\ \supseteq\ k_{\mathtt{FA}}(c, L, \mathtt{Atk}, \vec{t})\ \cap\ [\![m]\!]_{L,\gamma'}\end{aligned}$$

The conversion of Definition 15 to our framework is straightforward:

$$\begin{aligned}\sim_{\mathtt{FA}}\triangleq& \{(\vec{t_1}, \vec{t_2}) \mid \exists\vec{t'} \preccurlyeq \vec{t_2}.\ \mathtt{Atk}(\vec{t_1}) = \mathtt{Atk}(\vec{t'})\}\\ \equiv_{\mathtt{FA}}\triangleq=\qquad& \mathcal{A}_{\mathtt{FA}} \triangleq \mathcal{K}(c, \vec{t}^{[:\|\vec{t}\|-1]}, \sim_{\mathtt{FA}})\ \cap\ [\![m]\!]_{L,\vec{t}^{[\|\vec{t}\|]}.\gamma}\end{aligned}$$

Lemma 5: With $\sim\triangleq\sim_{\mathtt{FA}}$, $\mathcal{A} \triangleq \mathcal{A}_{\mathtt{FA}}$, and outside equivalence $\equiv\triangleq\equiv_{\mathtt{FA}}$, Definition 10 is equivalent to Definition 15.

Observation: We note that Forgetful Attacker was originally formalized in a similar format as our framework, making the conversion straightforward. However, there are various differences compared with Dynamic Release. Most importantly, Forgetful Attacker security is parameterized by an automaton $\mathtt{Atk}$; in other words, a program might be both "secure" and "insecure" depending on the given automaton. Consider the program in Figure 1-B(i). The program satisfies Forgetful Attacker security with any automation that forgets about the credit card information. Nevertheless, characterizing such "willfully stupid" attackers is an open question [3]. Second, the definition of the consistency relation $\equiv$ is completely different. As discussed in Section IV-C, it is important to define it properly to allow information flow restrictions to downgrade and upgrade in arbitrary ways.

```
lat=Lattice()
lat.add_sub(Label("M"), Label("Top"))
Program(
    secure=True,
    source_code="""
        // credit_card: M
        copy := credit_Card
        output(copy, M);
        // credit_card: Top
        copy := 0;
        output(copy, M)""",
    persistent=False,
    traces=[
        Trace(init_memory=dict(cc=0), outputs=[
            Out('M', 0, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})]),
        Trace(init_memory=dict(cc=1), outputs=[
            Out('M', 1, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})]),
        Trace(init_memory=dict(cc=2), outputs=[
            Out('M', 2, {'cc': 'M'}),
            Out('M', 0, {'cc': 'Top'})])
    ],
    lattice=lat )
```

Fig. 5: Annotated Program for Fig. 1-B(i)

VI. Evaluation

In this section, we introduce $\mathcal{A}$nnTrace benchmark and implement the dynamic policies as the form shown in Table I. The benchmark and implementations are available on github[9].

A. $\mathcal{A}$nnTrace Benchmark

To facilite testing and understanding of dynamic policies, we created the $\mathcal{A}$nnTrace benchmark. It consists of a set of programs annotated with *trace-level* security specifications. Among 58 programs in the benchmark, 35 of them are collected from existing works [7], [3], [5], [45], [19], [14]. References to the original examples are annotated in the benchmark programs. The benchmark also includes 23 programs that we created, such as the programs in Figure 1, and the counterexamples in Figure 6.

The benchmark is written in Python. Fig. 5 shows an example of annotated program for the source code in Fig. 1-B(i). As shown in the example, each program consists of:

- *secure*, a boolean value indicating whether this program is a secure program; the ground truth of our evaluation.
- *source code*, written in the syntax shown in Fig 2;
- *persistent*, a boolean value indicating whether the intended policy in this program is persistent (or transient);
- *lattice*, $\mathcal{L}$, the security lattice used by the program[10];
- *traces*, executions of the program. Each trace τ has:
  - *initial memory*, m , mapping from variables to integers
  - *outputs*, $\vec{t}$, a list of output events, each t in type `Out`:
    - *output level*, ℓ , a level from the lattice $\mathcal{L}$

[9] https://github.com/psuplus/AnnTrace

[10] We use lattice instead of level set for conciseness in the implementation.

	Examples in Fig 1						Existing(35)			New (23)		
	A(i)	A(ii)	B(i)	B(ii)	C(i)	C(ii)	✓	×	-	✓	×	-
Gradual Release	✓	✓	-	-	✓	✓	28	2	5	14	1	8
Tight Gradual Release	✓	✓	-	-	✓	✓	18	0	17	8	0	15
According to Policy p	✓	✓	✓	×	-	-	17	6	12	12	4	7
Cryptographic Erasure	-	-	✓	✓	-	-	21	0	14	7	1	15
Forgetful Attacker-Single	✓	✓	✓	×	✓	✓	31	4	0	19	4	0
Dynamic Release	✓	✓	✓	✓	✓	✓	35	0	0	23	0	0

'✓' means the policy checks the program as intended (same as ground truth); '×' means the policy fails to check the program as intended. '-' means the program is not in the scope of the policy (not applicable).

TABLE II: Evaluation Results.

* *output value*, v, an integer value
* *policy state*, γ, mapping from variables to levels

Given a program in existing work, we (1) use the claimed security of code as the ground truth, (2) convert the program into our specification language and to a security lattice, (3) mark persistent (or transient) according to if the correponding paper presents a persistent (or transient) policy, and (4) manually write down a finite number of traces that are sufficient for checking the dynamic policy involved in the example.

B. Implementation

We implemented all dynamic policies in Table I in Python, according to the formalization presented in the table. With exception of Forgetful Attacker and Paralocks, all implemented policies can directly work on the trace annotation provided by the $\mathcal{A}$nnTrace benchmark. Forgetful Attack policy requires an automaton as input. So we use a single memory automaton that only remembers the last output and forgets all previous outputs. Paralocks security requires "locks" in a test program but most tests do not have locks. So we are unable to directly evaluate it on the $\mathcal{A}$nnTrace benchmark.[11]

Existing policies are not generally applicable to all tests. Recall that each test has a persistent/transient field. Moreover, for each test, we automatically generate the following two features from the *traces* field:

A. there is no policy upgrading in the trace;
B. there is no policy downgrading in the trace;

These tags are used to determine if a concrete policy is appliable to the test. For example, Cryptographic Erasure is a transient policy that only allows upgrading. Hence, it is applicable to the tests with tag *transient* and B.

C. Results

The evaluation results are summerized in Table II. For the examples shown in Figure 1 (classical examples for declassification, erasure and delegation/revocation), we note that Dynamic Release is the only one that is both applicable and correct in all cases.

Among the 35 programs collected from prior papers and the 23 new programs, Dynamic Release is still both applicable and correct to all programs. In contrast, the existing works fall short in one way or another: with limited applicability or incorrect judgement on secure/insecure programs. Interestingly, According to Policy, Cryptographic Erasure and Gradual Release all make wrong judgment on some corner cases. Here, we discuss a few representative ones.

[11] Although we are unable to evaluation Paralocks directly, we believe its results should resemble those of Gradual Release, as its security condition is a generalisation of the gradual release definition [15].

(A)	(B)

`// x : L`
`output(L, 0);`
`// x : H`
`if (x == 0)`
`  output(L, 0);`
	`// h, h1: {D}⇒a`
	`// l, l2: {}⇒a`
	`open(D);`
	`if (h) { l2:=h1;}`
	`close(D);`
	`l:=0;`

Fig. 6: Counterexamples for Crypto-Erasure and Paralocks.

For According to Policy, the problematic part is the R relation. The policy states that as long as a qualified R can be found to satisfy the equation, a program is secure. We found that the restriction on R is too weak in many cases: a qualified R exists for a few insecure programs.

For Crypto-Erasure policy, the failed examples is shown in Figure 6-(A). It is an insecure program as the attacker learns that $x = 0$ if two outputs are observed. However, Crypto-Erasure accepts this program as secure for the reason that their policy ignores the location of an output. In this example, for the output 0, the security definition of Crypto-Erasure assumes that two executions are indistinguishable to the attacker if there exists a 0 output anywhere in the execution. Therefore, an execution with a single 0 output appears indistinguishably to the execution with two 0 outputs (both exists a 0 output). Thus, the policy fails to reject this program.

For Gradual Release, it fails on the following secure program, where $h, h1$: S and $l, l2$: P.

$$\texttt{if } (h) \texttt{ then } l2 := \texttt{declassify}(h1);$$
$$l := 0;$$

This example might seem insecure on the surface, as the branch condition h was not part of the `declassify` expression. But in the formal semantics (Section V-B2), a release event declassifies all information in the program (i.e., Gradual Release does not provide a precise bound on the released information as pointed out in [7], [2]). The program is secure since $h1$ is assigned to $l2$ when both h and $h1$ are declassified by the release event.

To check if a similar issue also exists in Paralocks, whose security condition is a generalization of the Gradual Release, we created a Paralock version of the same code, as shown in Figure 6-(B). Thanks to the cleaner syntax of Paralocks, it is more obvious that the program is secure: h and $h1$ have the same lock set $\{D\}$. Lock D is opened before the if statement, allowing value of both h and $h1$ to flow to $l, l2$. So the assignment in the if branch is secure. After that, only a constant 0 is assigned to l when the lock D is closed. However, the Paralock implementation rejects this program as insecure.

To understand why, Paralocks requires the knowledge of an attacker remains the same if the current lock state is a subset of the lock set that the attacker have. We are interested in attacker $A_1 = (a, \emptyset)$, who has an empty lock set. When lock D is open, since $\{D\} \not\subseteq \emptyset$, there is no restriction for the assignment $l2 := h1$. However, for the assignment $l := 0$, the current lock set is $\emptyset$, which is a subset of A's lock set ($\emptyset$). That is, for all the executions, the attacker A's knowledge should not change by observing the output event from assignment $l := 0$. However, this does not hold for the execution starting with $h = 0$. The initial knowledge of attacker A knows nothing about h or $h1$ since they are protected by lock D. With $h = 0$, the assignment in the branch is not executed. The attacker only observes the output from $l := 0$. By observing that output, the attacker immediately learns that $h = 0$. Therefore, Paralock rejected this program as insecure.

VII. Related Work

The most related works are those present high-level discussions on what/how end-to-end secure confidentiality should look like for some dynamic security policy. The major ones are already discussed and compared in the paper.

To precisely describe a dynamic policy, RIF [36], [35] uses reclassification relation to associate label changes with proram outputs. While this approach is highly expressive, writing down the correct relation with regards to numerous possible outputs is arguably a time-consuming and error-prone task. Similarly, flow-based declassification [44] uses a graph to pin down the exact paths leading to a declassification. However, the policy specification is tied up to the literal implementation of a program, which might limit its use in practice.

Bastys et al. [11] present six informal design principles for security definitions and enforcements. They summarize and categorize existing works to build a road map for the state-of-art. Then, from the top-down view, they provide guidance on how to approach a new enforcement or definition. In contrast, the framework and the benchmark proposed in this paper are post-checks after one definition is formalized.

Recent work [20] presents a unified framework for expressing and understanding for downgrading policies. Similar to Section IV, the goal of the framework is to make obvious the meaning of existing work. Based on that, they move further to sketch safety semantics for enforcement mechanism. However, they do not provide a define a formalization framework that allows us to compare various policies at their semantics level.

Many existing work [39], [29], [17] reuses or extends the representative policies we discussed in this paper. They adopt the major definition for their specialized interest, which are irrelevant to our interest. Hunt and Sands [33] present an interesting insight on erasure, but their label and final security definition are attached to scopes, which is not directly comparable with the end-to-end definitions discussed in this work. Contextual noninterference [42] and facets [9] use dynamic labels to keep track of information flows in different branches. The purpose of those labels is to boost flow- or path-sensitivity, not intended for dynamic policies.

VIII. Conclusion and Future Work

We present the first formalization framework that allows apple-to-apple compassion between various dynamic policies. The comparison sheds light on new insights on existing definitions, such as the distinguishing between transient and persistent policies, as well as motivates Dynamic Release, a new general dynamic policy proposed in this work. Moreover, we built a new benchmark for testing and understanding dynamic policies in general.

For future work, we plan to investigate semantic security condition of dynamic information flow methods, especially those use dynamic security labels. Despite the similarity that security levels are mutable, issues such as label channels might be challenging to be incorporate in our formalization framework. Moreover, Dynamic Release offers a semantic definition for information-flow security, but checking it on real programs is infeasible unless only small number of traces are produced. We plan to develop a static type system to check Dynamic Release in a sound and scalable manner.

Another future direction is to fully support partial release with expression-level specification. However, doing so is tricky since the expressions might have conflicting specifications. For example, consider a specification x, y : S and $x + y, x - y$: P. It states that the values of x and y are secrets, but the values of $x + y$ and $x - y$ are public. Mathematically, learning the values of $x + y$ and $x - y$ can also reveal the concrete values of x and y. Thus, it becomes tricky to define security in the presense of expression-level specification.

IX. Acknowledgement

We would like to thank the anonymous CSF reviewers for their constructive feedback. This research was supported by NSF grants CNS-1942851 and CNS-1816282.

References

[1] O. Arden, J. Liu, and A. C. Myers, "Flow-limited authorization," in *2015 IEEE 28th Computer Security Foundations Symposium*, July 2015, pp. 569–583.

[2] A. Askarov and A. Sabelfeld, "Tight enforcement of information-release policies for dynamic languages," in *2009 22nd IEEE Computer Security Foundations Symposium*, July 2009, pp. 43–59.

[3] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *Proc. IEEE Symp. on Computer Security Foundations*. IEEE, 2012, pp. 308–322.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *European symposium on research in computer security*. Springer, 2008, pp. 333–348.

[5] A. Askarov, S. Moore, C. Dimoulas, and S. Chong, "Cryptographic enforcement of language-based information erasure," in *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 2015, pp. 334–348.

[6] A. Askarov and A. Sabelfeld, "Security-typed languages for implementation of cryptographic protocols: A case study," in *European Symposium on Research in Computer Security*. Springer, 2005, pp. 197–221.

[7] ——, "Gradual release: Unifying declassification, encryption and key release policies," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2007, pp. 207–221.

[8] ——, "Localized delimited release: combining the what and where dimensions of information release," in *Proceedings of the 2007 workshop on Programming languages and analysis for security*, 2007, pp. 53–60.

[9] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 165–178.

[10] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," in *Proc. IEEE Symp. on Security and Privacy (S&P)*. IEEE, 2008, pp. 339–353.

[11] I. Bastys, F. Piessens, and A. Sabelfeld, "Prudent design principles for information flow control," in *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*, ser. PLAS '18. New York, NY, USA: ACM, 2018, pp. 17–23.

[12] T. Bauereiß, A. P. Gritti, A. Popescu, and F. Raimondi, "Cosmedis: a distributed social media platform with formally verified confidentiality guarantees," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 729–748.

[13] N. Broberg and D. Sands, "Flow locks: Towards a core calculus for dynamic flow policies," in *European Symposium on Programming*. Springer, 2006, pp. 180–196.

[14] ——, "Flow-sensitive semantics for dynamic information flow policies," in *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*. ACM, 2009, pp. 101–112.

[15] ——, "Paralocks: role-based information flow control and beyond," in *ACM Symposium on Principles of Programming Languages*, vol. 45, no. 1. ACM, 2010, pp. 431–444.

[16] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *Proc. IEEE Symp. on Computer Security Foundations*. IEEE, 2015, pp. 122–136.

[17] P. Buiras and B. van Delft, "Dynamic enforcement of dynamic policies," in *Proceedings of the 10th ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS'15. New York, NY, USA: ACM, 2015, pp. 28–41.

[18] S. Chong and A. C. Myers, "Language-based information erasure," in *Proc. IEEE Computer Security Foundations Workshop*. IEEE, 2005, pp. 241–254.

[19] ——, "End-to-end enforcement of erasure and declassification," in *IEEE Symp. on Computer Security Foundations*, 2008, pp. 98–111.

[20] A. Chudnov and D. A. Naumann, "Assuming you know: Epistemic semantics of relational annotations for expressive flow policies," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 189–203.

[21] M. R. Clarkson, S. Chong, and A. C. Myers, "Civitas: Toward a secure voting system," in *Proc. IEEE Symp. on Security and Privacy (S&P)*, 2008, pp. 354–368.

[22] E. S. Cohen, "Information transmission in sequential programs," *Foundations of Secure Computation*, pp. 297–335, 1978.

[23] F. Del Tedesco, S. Hunt, and D. Sands, "A semantic hierarchy for erasure policies," in *International Conference on Information Systems Security*. Springer, 2011, pp. 352–369.

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

[25] D. Ferraiolo, J. Cugini, and D. R. Kuhn, "Role-based access control (rbac): Features and motivations," in *Proceedings of 11th annual computer security application conference*, 1995, pp. 241–48.

[26] R. Giacobazzi and I. Mastroeni, "Abstract non-interference: Parameterizing non-interference by abstract interpretation," in *ACM SIGPLAN Notices*, vol. 39, no. 1, 2004, pp. 186–197.

[27] ——, "Adjoining declassification and attack models by abstract interpretation," in *European Symposium on Programming*. Springer, 2005, pp. 295–310.

[28] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symp. on Security and Privacy (S&P)*, Apr. 1982, pp. 11–20.

[29] A. Gollamudi and S. Chong, "Automatic enforcement of expressive security policies using enclaves," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016, vol. 51, no. 10. ACM, 2016, pp. 494–513.

[30] R. R. Hansen and C. W. Probst, "Non-interference and erasure policies for java card bytecode," in *6th International Workshop on Issues in the Theory of Security (WITS'06)*, 2006.

[31] B. Hicks, K. Ahmadizadeh, and P. McDaniel, "From languages to systems: Understanding practical application development in security-typed languages," in *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 2006, pp. 153–164.

[32] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *Proc. of Foundations of Computer Security Workshop*, 2005, pp. 7–18.

[33] S. Hunt and D. Sands, "Just forget it–the semantics and enforcement of information erasure," in *European Symposium on Programming*. Springer, 2008, pp. 239–253.

[34] S. Kanav, P. Lammich, and A. Popescu, "A conference management system with verified document confidentiality," in *International Conference on Computer Aided Verification*. Springer, 2014, pp. 167–183.

[35] E. Kozyri, O. Arden, A. C. Myers, and F. B. Schneider, "Jrif: reactive information flow control for java," in *Foundations of Security, Protocols, and Equational Reasoning*. Springer, 2019, pp. 70–88.

[36] E. Kozyri and F. B. Schneider, "Rif: Reactive information flow labels," *Journal of Computer Security*, no. Preprint, pp. 1–38, 2020.

[37] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference," in *ACM SIGPLAN Notices*, vol. 40, no. 1. ACM, 2005, pp. 158–170.

[38] A. A. Matos and G. Boudol, "On declassification and the non-disclosure policy," in *Proc. IEEE Computer Security Foundations Workshop (CSFW)*. IEEE, 2005, pp. 226–240.

[39] M. McCall, H. Zhang, and L. Jia, "Knowledge-based security of dynamic secrets for reactive programs," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, July 2018, pp. 175–188.

[40] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Symp. on Operating Systems Principles (SOSP)*, 1997, pp. 129–142.

[41] ——, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

[42] N. Polikarpova, J. Yang, S. Itzhaky, T. Hance, and A. Solar-Lezama, "Enforcing information flow policies with type-targeted program synthesis," *arXiv preprint arXiv:1607.03445*, 2016.

[43] S. Preibusch, "Information flow control for static enforcement of user-defined privacy policies," in *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*. IEEE, 2011, pp. 133–136.

[44] B. P. Rocha, S. Bandhakavi, J. den Hartog, W. H. Winsborough, and S. Etalle, "Towards static flow-based declassification for legacy and untrusted programs," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 93–108.

[45] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *International Symposium on Software Security*. Springer, 2003, pp. 174–191.

[46] A. Sabelfeld and D. Sands, "A per model of secure information flow in sequential programs," *Higher-order and symbolic computation*, vol. 14, no. 1, pp. 59–91, 2001.

[47] ——, "Dimensions and principles of declassification," in *IEEE Computer Security Foundations Workshop*. IEEE, 2005, pp. 255–269.

[48] ——, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[49] A. Stoughton, A. Johnson, S. Beller, K. Chadha, D. Chen, K. Foner, and M. Zhivich, "You sank my battleship," *A case study in secure programming*, 2014.

[50] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *Proc. IEEE Computer Security Foundations Workshop (CSFW)*. IEEE, 2006.

[51] B. van Delft, S. Hunt, and D. Sands, "Very Static Enforcement of Dynamic Policies," in *Principles of Security and Trust*, R. Focardi and A. Myers, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 32–52.

APPENDIX

A. Paralock in Table I

Paralock [13], [14], [15] uses locks to formalize the sensitivity of security objects. Paralock uses fine-grained model to encode role-based access control systems. Its covers both declassification and revocation of information to a principal in the system. As described in Section III-B3, security specification is written as $\{\Sigma \Rightarrow a; ...\}$, where Σ is a *lock set* and a is an *actor*. An actor a is the base sensitivity entity of the model, which is used to model a lattice level L in two-point lattice $\{H, L\}$ in [14], and a principal p in role-base access control system in [15].

To formalize Paralock security, an attacker $A = (a, \Sigma)$ is modeled as an actor a with a (static) set of open locks Σ. To simplify notation, we use $\Gamma(x, a) = \Sigma$ to denote the fact that $\{\Sigma \Rightarrow a; \}$ is part of the security policy of x, otherwise $\Gamma(x, a) = \top$. With respect to an attacker $A = (a, \Sigma)$, a variable x is observable to A iff $\Gamma(x, a) \subseteq \Sigma$, meaning that the attacker possesses more opened locks than what's required in the policy.

To simplified the notations in this work, we extend the output event t to also record the current open locks. So, for a trace fragment $\langle c, m\rangle \xrightarrow{b,v,\gamma} \langle c', m'\rangle$, it generates the output event $t = \langle b, v, \gamma, \Delta\rangle$, where $\Delta = \texttt{unlock}(\langle c, m\rangle)$.

Let $\|A\|$ be the set of variables that are visible to A, and $\lfloor \vec{t} \rfloor_A$ be the outputs that are visible to $A = (a_A, \Sigma_A)$:

$$\begin{aligned}\|A\| &\triangleq \{x \mid \forall x \in \textbf{Vars}.\ \Gamma(x, a_A) \subseteq \Sigma_A\} \\ \lfloor \vec{t} \rfloor_A &\triangleq \lfloor \vec{t} \rfloor_{\lambda b,n,\gamma,\Delta.\Gamma(b,a_A)\subseteq\Sigma_A}\end{aligned}$$

Paralock security defines attacker's knowledge[12] as follows:

$$\begin{aligned}k_{\texttt{PL}}(c, m, \vec{t}, A) = \{m' \mid\ & m' \approx_{\|A\|} m \\ & \wedge \langle c, m'\rangle \xrightarrow{\vec{t_1}}{}^{*} \langle c', m''\rangle \xrightarrow{\vec{t_2}}{}^{*} m''' \wedge \lfloor \vec{t_1} \rfloor_A = \lfloor \vec{t} \rfloor_A\}\end{aligned}$$

Paralock security semantics extends that of gradual release, by treating "unlock" events as releasing events:

Definition 16 (Paralock Security): A program c is Paralock secure if for any attacker $A = \langle a, \Sigma\rangle$, the attacker's knowledge remains unchanged whenever $\texttt{unlock}(\tau_{[i]}) \subseteq \Sigma_A$:

$$\begin{aligned}&\forall c, m, m', \vec{t}, t', a, \Sigma_A, A, i. \\ &\quad \langle c, m\rangle \xrightarrow{\vec{t}} \langle c', m'\rangle \xrightarrow{t} \langle c'', m''\rangle \wedge A = \langle a, \Sigma_A\rangle \wedge \\ &\qquad \texttt{unlock}(\langle c'', m''\rangle) \subseteq \Sigma_A \\ &\qquad\quad \Rightarrow k_{\texttt{PL}}(c, m, \vec{t} \cdot t', A) = k_{\texttt{PL}}(c, m, \vec{t}, A)\end{aligned}$$

We use the memory closure on A for memory that looks the same to attacker A:

$$\begin{aligned}[\![m]\!]_A \triangleq \{m' \mid\ & \forall m'.\ \forall x \in \textbf{Vars}. \\ & \Gamma(x, a_A) \subseteq \Sigma_A \Rightarrow m(x) = m'(x)\}\end{aligned}$$

The conversion of Definition 16 to our framework is straightforward.

$$\sim_{\texttt{PL}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \lfloor \vec{t_1} \rfloor_A \text{ prefix of } \lfloor \vec{t_2} \rfloor_A\} \qquad \equiv_{\texttt{PL}} \triangleq =$$

$$\mathcal{A}_{\texttt{PL}} \triangleq \begin{cases} \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\texttt{PL}}) \ \cap\ [\![m]\!]_A, & \vec{t}^{[\|\vec{t}\|]}.\Delta \subseteq \Sigma_A \\ [\![m]\!]_\emptyset, & \text{otherwise}\end{cases}$$

Lemma 6: With $\sim \triangleq \sim_{\texttt{PL}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{PL}}$, Definition 10 is equivalent to Definition 16.

B. Equivalence Proof for Table I

We first introduce a useful lemma which allows us to rewrite cast the orginal knowledge defintion in [7] to the knowledge defitnion $\mathcal{K}$ in this paper.

Lemma 7: Let k be defined as in Equation 6 and $\mathcal{K}$ be defined as in Equation 5, then we have

$$\begin{aligned}&\forall c, m, L, \Gamma, \vec{t}, M. \\ &\quad M \subseteq [\![m]\!]_{L,\Gamma} \wedge k(c, m, \vec{t}, L, \Gamma) \subseteq M \implies \\ &\qquad k(c, m, \vec{t}, L, \Gamma) = M \iff \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \supseteq M\end{aligned}$$

Proof. By definition, we know

$$k(c, m, \vec{t}, L, \Gamma) = \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \cap [\![m]\!]_{L,\Gamma}$$

- case $\implies$: we know

$$\begin{aligned}M = k(c, m, \vec{t}, L, \Gamma) &= \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \cap [\![m]\!]_{L,\Gamma} \\ \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) &\supseteq \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \cap [\![m]\!]_{L,\Gamma}\end{aligned}$$

  Thus, we have $\mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \supseteq M$.
- case $\impliedby$: we know

$$\begin{aligned}k(c, m, \vec{t}, \Gamma) &= \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \ \cap\ [\![m]\!]_{L,\Gamma} \\ M &\subseteq \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \\ M &\subseteq [\![m]\!]_{L,\Gamma} \\ M \cap M &\subseteq \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \ \cap\ [\![m]\!]_{L,\Gamma}\end{aligned}$$

  Thus, we know $k(c, m, \vec{t}, L, \Gamma) \supseteq M$. From assumption $k(c, m, \vec{t}, \Gamma) \subseteq M$, we know $k(c, m, \vec{t}, L, \Gamma) = M$.

So, we have $k(c, m, \vec{t}, L, \Gamma) = M \iff \mathcal{K}(c, \vec{t}, \sim_{\texttt{NI}}) \supseteq M$. □

1) Gradual Release: ***Lemma 1.*** With $\sim \triangleq \sim_{\texttt{GR}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{GR}}$, Definition 10 is equivalent to Definition 11:

$$\begin{aligned}&\forall c, m, L, i, \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t} \implies \\ &\text{i not release event} \Rightarrow k(c, m, \vec{t}_{[:i]}, L, \Gamma) = k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) \\ &\quad \iff \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\texttt{GR}}) \supseteq [\![m]\!]_{L,\ \vec{t}^{[i]}.\gamma} \cap \mathcal{K}(c, \vec{t}_{[:i-1]}, \sim_{\texttt{GR}})\end{aligned}$$

Proof. From the encoding of Gradual Release, we know:

$$\vec{t}^{[i]}.\gamma = \begin{cases}\gamma_\perp, & \text{i is a release event} \\ \Gamma, & \text{i not a release event}\end{cases}$$

[12] We revised the definition for termination insensitivity. We note that Paralock also presents a different termination insensitive policy following ideas from[4]. However, here we follow Gradual Release to define terminated insensitive knowledge by taking a intersection with the initial memory of traces that terminates.

- case when i is a release event: $\vec{t}^{[i]}.\gamma = \gamma_\perp$. From the definition, we know $[\![m]\!]_{L,\ \gamma_\perp}$ returns the singleton set $\{m\}$.
  From $\langle c, m\rangle \hookrightarrow \vec{t}$ and the definition of $\mathcal{K}$, we know $\forall j.\ m \in \mathcal{K}(c, \vec{t}^{[:j]}, \sim_{\mathtt{GR}})$:
  $$\begin{aligned} m &\in \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\mathtt{GR}}) \\ m &\in \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\mathtt{GR}}) \\ \{m\} &= [\![m]\!]_{L,\ \vec{t}^{[i]}.\gamma} \end{aligned}$$
  Thus, both Definition 10 and 11 are trivially true.
- case when i is not a release event: $\vec{t}^{[i]}.\gamma = \Gamma$. From the definitions, we know $\sim_{\mathtt{GR}} = \sim_{\mathtt{NI}}$. We know from the monotonicity of the knowledge that:
  $$\begin{aligned} k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) &\subseteq [\![m]\!]_{L,\Gamma} \\ k(c, m, \vec{t}^{[:i]}, L, \Gamma) &\subseteq k(c, m, \vec{t}_{[:i-1]}, L, \Gamma) \end{aligned}$$
  So, we can instantiate Lemma 7 with:
  $$M := k(c, m, \vec{t}^{[:i-1]}, L, \Gamma), \qquad \vec{t} := \vec{t}^{[:i]}$$
  and we get:
  $$\begin{aligned} & k(c, m, \vec{t}^{[:i}, L, \Gamma) = k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) \\ & \quad \iff \quad \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\mathtt{GR}}) \supseteq k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) \end{aligned}$$
  By definition, we know
  $$k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) = [\![m]\!]_{L,\Gamma} \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\mathtt{GR}})$$
  Thus, when i is not a release event, we have:
  $$\begin{aligned} & k(c, m, \vec{t}^{[:i]}, L, \Gamma) = k(c, m, \vec{t}^{[:i-1]}, L, \Gamma) \\ & \iff \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\mathtt{GR}}) \supseteq [\![m]\!]_{L,\ \vec{t}^{[i]}.\gamma} \cap \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\mathtt{GR}}) \end{aligned}$$

Therefore, Definition 10 is equivalent to Definition 11. □

2) Tight Gradual Release: ***Lemma 2.*** With $\sim \triangleq \sim_{\mathtt{TGR}}$, $\equiv \triangleq \equiv_{\mathtt{TGR}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\mathtt{TGR}}$, Definition 10 is equivalent to Definition 12.

$$\begin{aligned} & \forall i.\ 1 \le i \le \|\vec{t}\|. \\ & \quad ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{E_i}) \subseteq k(c, m, \vec{t}^{[:i]}, L, \Gamma) \\ & \quad\quad \iff \\ & \quad\quad\quad \mathcal{K}(c, \vec{t'}^{[:i]}, \sim_{\mathtt{TGR}}) \supseteq [\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \end{aligned}$$

Proof. The encoding limited E_i to a variable set X_i, thus, we assumes $E_i = X_i$. From the definition, we know that

$$k(c, m, \vec{t}, L, \Gamma) = \mathcal{K}(c, \vec{t'}^{[:i]}, \sim_{\mathtt{TGR}}) \cap [\![m]\!]_{L,\Gamma} \tag{7}$$

From encoding, we know that

$$[\![m]\!]_{L, \vec{t}^{[i]}.\gamma} = ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{E_i}) \tag{8}$$

- case $\Longrightarrow$: From Equation 8 and the assumption, we know
  $$[\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \subseteq k(c, m, \vec{t}^{[:i]}, L, \Gamma)$$
  From Equation 7, we know
  $$k(c, m, \vec{t}^{[:i]}, L, \Gamma) \subseteq \mathcal{K}(c, \vec{t'}^{[:i]}, \sim_{\mathtt{TGR}})$$
  Therefore, we have $[\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \subseteq \mathcal{K}(c, \vec{t'}^{[:i]}, \sim_{\mathtt{TGR}})$.
- case $\Longleftarrow$: By taking an intersection with $[\![m]\!]_{L,\Gamma}$ on both side of the assumption, we have:
  $$\mathcal{K}(c, \vec{t'}^{[:i]}, \sim_{\mathtt{TGR}}) \cap [\![m]\!]_{L,\Gamma} \supseteq [\![m]\!]_{L, \vec{t}^{[i]}.\gamma} \cap [\![m]\!]_{L,\Gamma}$$
  Apply Equation 7 to the left and Equation 8 to the right:
  $$\begin{aligned} k(c, m, \vec{t}^{[:i]}, L, \Gamma) \supseteq\ & ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{E_i}) \cap [\![m]\!]_{L,\Gamma} \\ =\ & [\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{E_i} \end{aligned}$$
  Thus, we have $k(c, m, \vec{t}^{[:i]}, L, \Gamma) \supseteq ([\![m]\!]_{L,\Gamma} \cap [\![m]\!]_{E_i})$.

Therefore, Definition 10 is equivalent to Definition 12. □

3) According to Policy p: ***Lemma 3.*** With $\sim \triangleq \sim_{\mathtt{AP}}$, $\mathcal{A} \triangleq \mathcal{A}_{\mathtt{AP}}$, and outside equivalence $\equiv \triangleq \equiv'_{\mathtt{AP}}$, Definition 10 is equivalent to Definition 13.

Proof. First, we convert a security levels L from the Denning's style to our attacker levels l as described in Section III, and outputs any intermediate memory of the trace to its variable's level. That is,

$$\begin{aligned} & \forall c, m_0, m', c_i, m_i, i, e, \Gamma. \\ & \tau = \langle c, m_0\rangle_{\gamma_0} \to^* \langle c_i, m_i\rangle_{\gamma_i} \to^* m' \wedge \tau_{[i]} = \langle c_i, m_i\rangle_{\gamma_i} \\ & \iff \\ & \langle c, m_0\rangle \hookrightarrow \vec{t} \\ & \wedge\ \vec{t}^{[i]} = \{\langle ch, n, \gamma\rangle \mid ch = e \wedge n = m_i(e) \wedge \gamma = \gamma_i\} \end{aligned}$$

We note that our normal $\vec{t}^{[i]}$ returns a single output event, say some $t = \langle ch, n, \gamma\rangle$. But here we overload $\vec{t}^{[i]}$ to return a set of output events that output all values on memory $\tau_{[i]}$. Thus, with all values on memory outputted, we have:

$$\begin{aligned} & \forall c, m_1, m_2, m_1', m_2', \tau, \tau', \vec{t_1}, \vec{t_2}. \\ & \quad \wedge\ \tau = \langle c, m_1\rangle \to m_1' \wedge \tau' = \langle c, m_2\rangle \to m_2' \\ & \quad \wedge\ \langle c, m_1\rangle \hookrightarrow \vec{t_1} \wedge \langle c, m_2\rangle \hookrightarrow \vec{t_2} \implies \\ & \quad\quad \tau_{[i]} \approx_l \tau'_{[j]} \iff \lfloor \vec{t}^{[i]} \rfloor_l = \lfloor \vec{t}^{[j]} \rfloor_l. \end{aligned}$$

Thus, we rewrite Definition 13 in following two-run style:

$$\begin{aligned} & \forall c, m_1, m_2, l, p, \vec{t_1}, \vec{t_2}. \\ & \quad m_2 \in [\![m_1]\!]_p \wedge \langle c, m_1\rangle \hookrightarrow \vec{t_1} \wedge \langle c, m_2\rangle \hookrightarrow \vec{t_2} \implies \\ & \quad \exists R. \Big(\forall (i,j) \in R.\ \vec{t_1}_{[i]} \in \lfloor \vec{t_1} \rfloor^{p,l} \wedge \vec{t_2}_{[j]} \in \lfloor \vec{t_2} \rfloor^{p,l} \\ & \quad\quad \Rightarrow \lfloor \vec{t_1} \rfloor_l = \lfloor \vec{t_2} \rfloor_l\Big) \end{aligned}$$

We combine the two filters and assume R' as R after filtering:

$$\begin{aligned} & \forall c, m_1, m_2, l, p, \vec{t_1}, \vec{t_2}. \\ & \quad m_2 \in [\![m_1]\!]_p \wedge \langle c, m_1\rangle \hookrightarrow \vec{t_1} \wedge \langle c, m_2\rangle \hookrightarrow \vec{t_2} \implies \\ & \quad\quad \exists R'. \Big(\forall (i,j) \in R'. (\lfloor \vec{t_1} \rfloor_{p,l})_{[i]} = (\lfloor \vec{t_2} \rfloor_{p,l})_{[j]}\Big) \end{aligned}$$

With $\mathcal{K}(c, \vec{t}, \sim_{\texttt{AP}})$ unfolded as below:

$$\begin{aligned}\mathcal{K}(c, \vec{t}, \sim_{\texttt{AP}}) = \{m_2 \mid \forall m_2, \vec{t_2}.\ \langle c, m_2\rangle \hookrightarrow \vec{t_2} \\ \wedge \exists R'.\forall (i,j) \in R'.\ (\lfloor \vec{t} \rfloor_{p,l})_{[i]} = (\lfloor \vec{t_2} \rfloor_{p,l})_{[j]}\}\end{aligned}$$

We can further rewrite the definition as follow:

$$\begin{aligned}&\forall c, m_1, m_2, l, p, \vec{t_1}.\\ &\quad m_2 \in [\![m_1]\!]_p \wedge \langle c, m_1\rangle \hookrightarrow \vec{t_1} \implies m_2 \in \mathcal{K}(c, \vec{t_1}, \sim_{\texttt{AP}})\end{aligned}$$

That is,

$$\forall c, m_1, l, p, \vec{t_1}.\langle c, m_1\rangle \hookrightarrow \vec{t_1} \wedge [\![m_1]\!]_p \subseteq \mathcal{K}(c, \vec{t_1}, \sim_{\texttt{AP}})$$

We note that only the equivalence relation in $\sim_{\texttt{AP}}$ is $\equiv_{\texttt{AP}}$. The equivalence relation in $\vec{t'} \equiv \vec{t}$ in Definition 10 in this case is not $\equiv_{\texttt{AP}}$, but $\equiv'_{\texttt{AP}} \triangleq \{(\vec{t_1}, \vec{t_2}) \mid \vec{t_1} = \vec{t_2}\}$. □

4) Cryptographic Erasure: ***Lemma 4.*** With $\sim \triangleq \sim_{\texttt{CE}}$, $\equiv \triangleq \equiv_{\texttt{CE}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{CE}}$, Definition 10 with adjusted attack model is equivalent to Definition 14.

$$\begin{aligned}&\forall c, m, \gamma_0, c_i, m_i, \gamma_i, c_n, m_n, \gamma_n, m', \vec{t_1}, \vec{t_2}, l, i, j, n.\\ &\quad \langle c, m\rangle_{\gamma_0} \xrightarrow{\vec{t_1}} \langle c_i, m_i\rangle_{\gamma_i} \xrightarrow{\vec{t_2}} \langle c_n, m_n\rangle_{\gamma_n} \to^* m' \implies\\ &\qquad k_{\texttt{CE}}(c, L, \sim_{\texttt{CE}}) \supseteq \bigcap_{i\le j\le n} [\![m]\!]_{L,\gamma_j} \iff\\ &\qquad\qquad \mathcal{K}(c, \vec{t_2}, \sim_{\texttt{CE}}) \supseteq \bigcap_{t_n \in \vec{t_2}} [\![m]\!]_{L,t_n.\gamma}\end{aligned}$$

Proof. We note that in Definition 14, γ_j are state policies attached to the configurations, not from the output event $\vec{t}$. According to the definition, $\vec{t}$ does not contain empty events. In Definition 14, it takes $n-j$ steps to generate output sequence $\vec{t_2}$, we know $n - j \ge \|\vec{t_2}\|$. We first show that the right hand side allowance defined using γ_j is the same as using state policy from the output sequence $\vec{t}$:

$$\begin{aligned}&k_{\texttt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{i\le j\le n} [\![m]\!]_{L,\gamma_j}\\ &\qquad \iff k_{\texttt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{t_n \in \vec{t_2}} [\![m]\!]_{L,t_n.\gamma}\end{aligned} \tag{9}$$

- case $\implies$: Crypto [5] supports only erasure policy (and static policy). That is, the sensitivity of any security entity is monotonically increasing:

  $$\forall j \in [i, n].\gamma_j \preccurlyeq \gamma_{j+1}$$

  From the definition of memory closure, we know:

  $$\forall \gamma_1, \gamma_2.\ \gamma_1 \preccurlyeq \gamma_2 \implies [\![m]\!]_{L,\gamma_1} \subseteq [\![m]\!]_{L,\gamma_2}$$

  Thus, we know:

  $$\begin{aligned}\bigcap_{i\le j\le n} [\![m]\!]_{L,\gamma_j} &= [\![m]\!]_{L,\gamma_i}\\ \bigcap_{t_n \in \vec{t_2}} [\![m]\!]_{L,t_n.\gamma} &= [\![m]\!]_{L,\vec{t_2}_{[0]}.\gamma}\end{aligned}$$

  From the definitions, we know $\vec{t_2}_{[0]}.\gamma = \gamma_i$ if $\langle c_i, m_i\rangle_{\gamma_i}$ does not immediately generates an empty output event. Otherwise, if the first non-empty event is generated at configuration $\langle c_{i'}, m_{i'}\rangle_{\gamma_{i'}}$, $(i < i' < n)$, we know:

  $$[\![m]\!]_{L,\gamma_i} \subseteq [\![m]\!]_{L,\gamma_{i'}} = [\![m]\!]_{L,\vec{t_2}_{[0]}.\gamma}$$

  We can instantiate Definition 14 for $i := i'$, and we get:

  $$k_{\texttt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{i'\le j\le n} [\![m]\!]_{L,\gamma_j} = [\![m]\!]_{L,\gamma_{i'}} = [\![m]\!]_{L,\vec{t_2}_{[0]}.\gamma}$$

  Thus, we have $k_{\texttt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{t_n \in \vec{t_2}} [\![m]\!]_{L,t_n.\gamma}$.
- case $\impliedby$: from $n - j \ge \|\vec{t_2}\|$, we know:

  $$\begin{aligned}\forall t_n \in \vec{t_2}.\ \exists j' \in [i, n].\ \gamma_{j'} &= t_n.\gamma\\ \{t_n.\gamma \mid t_n \in \vec{t_2}\} &\subseteq \{\gamma_j \mid i \le j \le n\}\\ \bigcap_{t_n \in \vec{t_2}} [\![m]\!]_{L,t_n.\gamma} &\supseteq \bigcap_{i\le j\le n} [\![m]\!]_{L,\gamma_j}\end{aligned}$$

  Thus, we have $k_{\texttt{CE}}(c, L, \vec{t_2}) \supseteq \bigcap_{i\le j\le n} [\![m]\!]_{L,\gamma_j}$.

Therefore, we know Equation 9 is true.
Now we convert a security level L from Denning's style to our attacker level l as described in SectionIII. Let $[\![c]\!] \triangleq \{m \mid \exists \vec{t}.\ \langle c, m\rangle \hookrightarrow \vec{t}\}$ denote the set of memory that terminates. From definition we know:

$$\mathcal{K}(c, \vec{t}, \sim_{\texttt{CE}}) = k_{\texttt{CE}}(c, L, \vec{t}) \cap [\![c]\!]$$

For the interest of a termination-insensitive policy, we can ignore the difference made by the terminated set $[\![c]\!]$. Thus, we assumes $\mathcal{K}(c, \vec{t}, \sim_{\texttt{CE}}) = k_{\texttt{CE}}(c, L, \vec{t})$. □

5) Forgetful Attacker: ***Lemma 5.*** With $\sim \triangleq \sim_{\texttt{FA}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\texttt{FA}}$, Definition 10 is equivalent to Definition 15.

Proof. In the forgetful attacker[3], the sensitivity level is changed by `setPolicy` command. Recall from our encoding, `setPolicy` is encoded using security commands and generates a security event, but no output event. So, there is no sensitivity change between the two states that generates an output. That is, for the output event t' in the trace:

$$\langle c, m\rangle \xrightarrow{\vec{t}\cdot t'} {}^*\langle c', m'\rangle \xrightarrow{\langle b,v,\gamma'\rangle\cdot\ldots} {}^* \Rightarrow$$

We know $t'.\gamma = \gamma'$ and therefore, we have

$$[\![m]\!]_{L,\gamma'} = [\![m]\!]_{L,\ t'.\gamma}$$

Definition 15 is rephrased as:

$$\begin{aligned}&\forall c, m, L, i, \vec{t}.\langle c, m\rangle \hookrightarrow \vec{t} \implies\\ &k_{\texttt{FA}}(c, L, \texttt{Atk}, \vec{t}^{[:i]}) \subseteq k_{\texttt{FA}}(c, L, \texttt{Atk}, \vec{t}^{[:i-1]}) \cap [\![m]\!]_{L,\ \vec{t}^{[i]}.\gamma}\end{aligned}$$

By definition, we know:

$$k_{\texttt{FA}}(c, L, \texttt{Atk}, \vec{t}) = \mathcal{K}(c, \vec{t}, \sim_{\texttt{FA}})$$

Thus, we know Definition 15 is equivalent to Definition 10. □

6) Paralock: ***Lemma 6.*** With $\sim \triangleq \sim_{\text{PL}}$ and $\mathcal{A} \triangleq \mathcal{A}_{\text{PL}}$, Definition 10 is equivalent to Definition 16.

$$\forall c, m, \vec{t}, \vec{t'}, i, A.\ \ \langle c, m\rangle \hookrightarrow \vec{t} \wedge \vec{t'} = \vec{t}^{[:i]} \implies$$
$$\vec{t}^{[i]}.\Delta \subseteq \Sigma_A \Rightarrow$$
$$k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A) = k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A)$$
$$\iff$$
$$\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \supseteq \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$$

Proof. We omit the case when $\vec{t}^{[i]}.\Delta \not\subseteq \Sigma_A$ since both definitions are trivially true. By Definition, we know:

$$\forall j.\ k_{\text{PL}}(c, m, \vec{t}^{[:j]}, A) = \mathcal{K}(c, \vec{t}^{[:j]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$$

- case $\implies$: we know:

$$\begin{aligned} k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) &= k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A) \\ k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A) &= \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \cap [\![m]\!]_A \\ \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) &\supseteq \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \cap [\![m]\!]_A \end{aligned}$$

  Thus, we have

$$\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \supseteq k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A)$$

  With $k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) = \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$, we get $\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \supseteq \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$.

- case $\impliedby$: we know:

$$\begin{aligned} \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) &\supseteq \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\text{PL}}) \cap [\![m]\!]_A \\ k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) &= \mathcal{K}(c, \vec{t}^{[:i-1]}, \sim_{\text{PL}}) \cap [\![m]\!]_A \end{aligned}$$

  Thus, we have

$$\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \supseteq k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \tag{10}$$

  We know $[\![m]\!]_A$ is the initial knowledge of A before observing any output event. From the monotonicity of Paralock knowledge, we know:

$$[\![m]\!]_A \supseteq k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \tag{11}$$

  By taking an intersection on both side of Equation (10) and Equation (11), we have:

$$\begin{aligned} &(\mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \cap [\![m]\!]_A) \\ \supseteq\ &(k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \cap k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A)) \\ =\ &k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \end{aligned}$$

  Thus, we have

$$k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \subseteq \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$$

  By Definitions, we have:

$$k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A) = \mathcal{K}(c, \vec{t}^{[:i]}, \sim_{\text{PL}}) \cap [\![m]\!]_A$$

  Thus, we know:

$$k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \subseteq k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A)$$

  From the monotonicity of the Paralock knowledge, we know

$$k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) \supseteq k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A)$$

  Thus, we have:

$$k_{\text{PL}}(c, m, \vec{t}^{[:i-1]}, A) = k_{\text{PL}}(c, m, \vec{t}^{[:i]}, A)$$

Therefore, Definition 10 is equivalent to Definition 16. □