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 release (i.e., declassify) 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 endorsing/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
observing one output event, rather than the more standard cumulative knowledge that we see in existing persistent policies.

Finally, we built a new AmTrace 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 Alice, Bob) where the acts-for relation on principals can change at run time. For simplicity, we use the notation \( \ell \in \mathbb{L} \) if all levels form a lattice \( \mathbb{L} \), rather than \( \ell \in \mathbb{L} \). Moreover, we use \( P \) (public), \( S \) (secret) to represent levels in a standard two-point lattice where \( P \subseteq S \) but \( S \not\subseteq P \).

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 \( \mathbb{L} \), declassification occurs when a piece of information has its sensitivity level \( \ell_1 \) downgraded to a lower sensitivity level \( \ell_2 \) (i.e., \( \ell_2 \subseteq \ell_1 \)). 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 \( \mathbb{L} \), information erasure occurs when a piece of information has its sensitivity

---

![Fig. 1: Examples of Dynamic Policies.](image)

---

\(^1\)Since integrity is the dual of confidentiality, we will assume confidentiality hereafter.
level \( \ell_1 \) upgraded to a more restrictive sensitivity level, or an incomparable level \( \ell_2 \) (i.e., \( \ell_2 \not\subseteq \ell_1 \)). Moreover, once erased, 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 merchantiser (at level \( \mathbb{M} \)) as payment for her purchase. When the transaction is done, the merchantiser 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 use 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 their customers get to read books during the renting period. In this example, Alice acts-for \( bk \) (\( bk \rightarrow Alice \)) 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 (\( bk \not\rightarrow Alice \)), but the notes remain accessible to Alice. The secure program (i) allows the customer to get their notes (Line 5) learned during the renting period. The insecure program (ii) fails to protect the book (Line 5) 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 an delegation/revocation policy (Example C) and an erasure policy (Example B) use different formats to model sensitivity change. An 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 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 is to properly define the “knowledge” of learning a single event \( t \). Moreover, 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. 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 semantically.

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 (\(S\)) \( s \)
Expressions (\(E\)) \( e ::= x \mid n \mid e \ op e \)
Commands \( c ::= \text{skip} \mid c_1; c_2 \mid x ::= e \mid \text{while} (e) \ c \)
\( \mid \text{if} (e) \ \text{then} \ c_1 \ \text{else} \ c_2 \mid \text{output}(b, e) \)
Level Sets \( \{ e \mid L \} \)
Security Labels (\(B\)) \( b ::= L \mid \text{cnd} ? b \mid b_2 \)
Conditions \( \text{cnd} ::= s \mid e \mid \text{cnd} \land \text{cnd} \mid \text{cnd} \lor \text{cnd} \mid \neg \text{cnd} \)

\( \circ ::= \rightarrow \mid \leftarrow \mid \doteq \)
Policy Specification \( \Gamma : \text{Vars} \rightarrow B[\circ] \)
Policy Type \( \circ ::= \text{Tran} \mid \text{Per} \)

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:
\[
L \triangleq \{ \text{Alice,Bob} \}; \ L_{\text{Alice}} \triangleq \{ \text{Alice} \}; \ L_{\text{Bob}} \triangleq \{ \text{Alice,Bob} \};
\]

2) Sensitivity Mutations: 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 \( \text{cnd} ? b \mid b_2 \) where:

- A trigger condition \( \text{cnd} \) 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 \( \text{cnd} \) is an expression, \( e \) is of the Boolean type.
- The mutation direction \( \circ \) specifies how the information flow restriction changes. There are two one-time mutation directions: \( \text{cnd} ? b_1 \rightarrow b_2 \) (resp. \( \text{cnd} ? b_1 \leftarrow b_2 \)) allows a one-time sensitivity change from \( b_1 \) to \( b_2 \) (resp. \( b_2 \) to \( b_1 \)) the first time that \( \text{cnd} \) evaluates to \text{false} (resp. \text{true}). On the other hand, a two-way mutation \( \text{cnd} ? b_1 \leftrightarrow b_2 \) allows arbitrary mutation between \( b_1 \) and \( b_2 \) whenever the value of \( \text{cnd} \) changes.

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

B. Expressiveness

To show the expressiveness of our specification, we first show that all kinds of dynamic policies in Figure 1 can be concisely specified using our label. Then, we discuss how the specification covers the well-known \textit{what}, \textit{who}, \textit{where} and \textit{when} dimensions [47], [48]. 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 encode the examples in Figure 1.

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

For example, the policy on \textit{credit_card} in Figure 1-B can be specified as \( \text{erase}\{\} \leftarrow \{\} \) [Tran] (we will discuss why erasure is a transient policy in Section IV) with the security

\[ L = P(\mathbb{P}); \ L_P \triangleq \{P' \mid P' \ \text{actsfor} \ P \} \]
command 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 roles, such as Alice, Bob. Such a change can be specified as the old and new sets of roles who acts-for the owner, say R, of information. That is, a change from from actsFor1 to actsFor2 can be specified as L1 \rightarrow L2, where L_i = \{R' \in \mathbb{R} | R' actsFor R\}.

For example, the policy on book in Figure 1-C can be specified as revoke?{} \leftarrow \{Alice\} \{Per\} (we will discuss why revocation is a persistent policy in Section IV) with a security command EventOn(revoke) being inserted to Line 5 to trigger the mutation.\footnote{We note that our encoding requires all changes to the acts-for relation to be anticipated, whereas a general delegation and revocation policy offers the flexibility of changing the acts-for relation dynamically.}

2) Dimensions of dynamic policy [47], [48]: 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 encoded to 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 \(\Gamma(h) = S\), \(\Gamma(x) = s?S\rightarrow P\) for program:

\[
\begin{align*}
&x := h; \text{EventOn}(s); \text{output}(P, x);
\end{align*}
\]

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.

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 \(\text{cmd?b}_1 \circ b_2\) declare where information may flow to after policy change, and the security event \(s\) with the security commands EventOn(s) and EventOff(s) are design to specify the code location where sensitivity changes.

The when dimension is a temporal dimension, pertaining to when information’s sensitivity changes. This is specified by the trigger condition \(\text{cmd}\). For example, a policy (\text{paid}\:P \leftarrow 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].

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 in Section III-B3 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.

3) Encoding Flow Locks [13]: These policies introduce locks, denoted as \(\sigma\) to construct dynamic policy. Let \(\Sigma\) be a set of locks, and \(P\) be a set of principals. A “lock” policy is specified with the following components:

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

To simplify notation, we use \(\Gamma(x, P) = \Sigma\) to denote the fact that \(\{\Sigma \Rightarrow P\}\) is part of the security policy of \(x\), otherwise \(\Gamma(x, P) = \top\). Paralocks security is formalized as an extension of gradual release. In particular, paralocks security is defined based on sub-security condition for each hypothetical attacker \(A = (P_A, \Sigma_A)\), where \(P_A \in \mathbb{P}\) and \(\Sigma_A \subseteq \text{Locks}\).

Hence, for each concrete \(A = (P_A, \Sigma_A)\), we can encode Paralocks security as follows:

- We define a security event \(s_\sigma\) for each lock \(\sigma \in \Sigma\) and the lock command \(\text{open}(\sigma)\) (resp. \(\text{close}(\sigma)\)) is converted to EventOn(s_\sigma) (resp. EventOff(s_\sigma)).
- Let \(\Gamma(x) = \{P_A\}\) when \(\Gamma(x, P_A) = \Sigma_x \subseteq \Sigma_A\); otherwise, \(\Gamma(x) = \emptyset\) (i.e., secret for \(P_A\)).
- Following the encoding of gradual release, we define \(\Gamma(x) = \text{cmd}\:\Gamma(x) : \{P_A\}\) where \(\text{cmd} \triangleq \bigwedge_{\sigma \not\in \Sigma_A} s_\sigma\), i.e., all locks not in \(\Sigma_A\) must be currently closed, which implies an output event (not a release event); \(\Sigma_{open} \subseteq \Sigma_A\); otherwise, for a release event, \(x\) is public to \(P_A\).

As a concrete example, we show the original Paralocks code and its transform code in Figure 3 for \(A = (a, \{D\})\). We note that under the encoding, the first assignment \(y := x\) is under a release event since only lock \(D\) is open, which is a subset of \(\Sigma_A = \{D\}\); both the output channel and the value can be read by \(a\). On the other hand, the second assignment \(z := y\) is not under a release event, as an opened lock \(N\) is not possessed by attacker \(A\). This is also reflected by the encoding: while the

\[
\begin{align*}
\text{Fig. 3: An Example of Encoding Paralock for } A = (a, \{D\}).
\end{align*}
\]
output channel is observable to a unconditionally, the value of
y has policy \{\}\ at that point, as \( s_N = \text{true} \).

Hence, we can encode Paralocks by explicitly checking the
security of each transformed program for each A, and accept
the program iff all transformed programs are 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 \( \tau \), and returns a level set \( L \) as information flow
restrictions at the end of \( \tau \).

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:

\[
\begin{align*}
\langle e, m \rangle \downarrow v & \quad \text{S-OUTPUT} \\
\langle \text{output}(b, e), m \rangle \xrightarrow{(b, v)} \langle \text{skip}, m \rangle & \quad \text{S-OUTPUT}
\end{align*}
\]

The semantics records all output events, in the form of \( (b, v) \),
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 \( \tau \) with
potentially empty output events:

\[
\langle e, m \rangle \xrightarrow{b_1} \cdots \xrightarrow{b_n} \langle c, m_n \rangle
\]

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

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 \( \tau \). For a (static) level set \( L \),
its interpretation is simply \( L \) regardless of \( \tau \).

For more complicated labels, the semantics also considers the
temporal aspect of label changes. For example, a one-
time mutation label \( \text{cnd?} b_1 \rightarrow b_2 \) allows a one-time sensitivity
change from \( b_1 \) to \( b_2 \) when the first time that \( \text{cnd} \) evaluates
to \text{false}. Hence, let \( i \) be the first index of \( \tau \) such that \( \text{cnd}
\]

\[
[L]_\tau = L \\
[\text{cnd?} b_1 \rightarrow b_2]_\tau = \begin{cases} [b_1]_\tau, & \text{first}(\text{cnd,} \tau, \text{false}) = -1 \\
[b_2]_\tau, & \text{first}(\text{cnd,} \tau, \text{false}) = 0 \end{cases}
\]

\[
[\text{cnd?} b_1 \leftarrow b_2]_\tau = \begin{cases} [b_2]_\tau, & \text{first}(\text{cnd,} \tau, \text{true}) = -1 \\
[b_1]_\tau, & \text{first}(\text{cnd,} \tau, \text{true}) = 0 \end{cases}
\]

where \( \text{first}(\text{cnd,} \tau, \text{false}) \) returns the first index of \( \tau \) such that \( \text{cnd}
evaluates to \text{false} \). Then, \([\text{cnd?} b_1 \rightarrow b_2]_\tau\) reduces to \( [b_1]_\tau \)
when no such \( i \) exists (i.e., \( \text{cnd} \) always evaluates to \text{true}
in \( \tau \)), and it reduces to \( [b_2]_\tau \) otherwise. Note that in the
latter case, it reduces to \( [b_2]_\tau \) rather than \( [b_2]_\tau \) to properly
handle nested conditions: any nested condition in \( b_2 \) can only
be evaluated after \( \text{cnd} \) becomes \text{false}. The dual with \( \leftarrow \)
is defined in a similar way. Note that \( \text{cnd?} b_1 \rightarrow b_2 \) and
\( \neg\text{cnd?} b_2 \leftarrow b_1 \) 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 \( \tau \): let \( i \) be the last
index in \( \tau \) such that \( \text{cnd} \) evaluates to \text{false}. Then, \( i \neq ||\tau|| \)
implies that \( \text{cnd} \) evaluates to \text{true} at the end of \( \tau \); 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 \( \text{cnd} \)
becomes \text{true}.

Moreover, we can derive a dynamic specification for each
execution point \( i \), written as \( \gamma_i \), such that

\[
\forall x. \gamma_i(x) = [\Gamma(x)]_{\tau^{[i]}}
\]

Additionally, we overload \( \gamma_i \) 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

\[
(c_0, m_0) \mapsto \vec{t}
\]

if the execution \( \langle c_0, m_0 \rangle \) terminates\(^5\) with an extended output
sequence \( \vec{t} \), which consists of extended output events
\( t \triangleq (b, v, \gamma) \), where \( b, v \) are the output events on \( \tau \), and \( \gamma \)
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 \( \vec{t}^{[\cdot]} \)

\(^5\) In this paper, we only consider output sequences \( \vec{t} \) produced by
(\( c_0, m_0 \)) \mapsto \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.
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 memory who agrees 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:

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

For simplicity, we use the following short-hands:

$$\llbracket m \rrbracket_{L,\gamma} \triangleq \{ m \mid \gamma(x) \leq L \}$$

$$\llbracket m \rrbracket_{\neq b} \triangleq \{ m \mid \gamma(x) \neq b \}$$

where $\llbracket m \rrbracket_{L,\gamma}$ is the memory closure on all variables whose sensitivity level is less or equally restrictive than a level $L$ according to $\gamma$, and $\llbracket m \rrbracket_{\neq b}$ is the memory closure on variables whose security policy is not $b$; a set of memories whose value 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 $\text{false}$ for irrelevant outputs), we define the projection of outputs as follows:

Definition 3 (Projection of Trace):

$$[\vec{t}]_f \triangleq \{ \langle b, v, \gamma \rangle \in \vec{t} \mid f(b, v, \gamma) \}$$

We define the following short-hand for commonly used filter, $L$-projection filter, where the resulting trace consists of outputs currently observable at level $L$:

$$[\vec{t}]_L \triangleq [\vec{t}]_{\neq b, L, \gamma(b) \leq L}$$

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 visible iff $\gamma(b) \leq L$ by the attack model. Hence, as standard, we define an indistinguishability relation, written as $\sim_L$, on traces as:

$$\sim_L \triangleq \{ \langle \vec{t}_1, \vec{t}_2 \rangle \mid [\vec{t}_1]_L \equiv [\vec{t}_2]_L \}$$

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:

$$k_1(c, \vec{t}, L) \triangleq \{ m \mid (c, m) \rightarrow \vec{t} \land \vec{t} \sim_L \vec{t}' \}$$

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, $(c, m) \rightarrow \vec{t}$ 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 $A(m, \vec{t}, b, L)$, is defined as a set of memories that should remain indistinguishable to the actual initial memory $m$ at the end of output sequence $\vec{t}$.

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

$$A(m, \vec{t}, b, L) \triangleq \llbracket m \rrbracket_{\neq b}$$

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.

\footnote{We slightly modified the original definition to exclude “initial knowledge”, the attacker’s knowledge before executing the program.}
**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. For example, it is easy to check that for any \( c, t, p, L \), we have

\[
\vec{t} \not\subseteq \vec{u} \Rightarrow k_1(c, \vec{t}, L) \supseteq k_1(c, \vec{u}, L)
\]

according to Equation 3. 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 \( (c, m) \rightarrow \vec{t} \), 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 \( T \) 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-Bi. 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 can 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 \( (c, m) \rightarrow \vec{t} \), an attacker at level set \( L \) can rule out any initial memory \( m' \) where \( m \not\sim_L m' \) (i.e., \( m' \not\in k_1(c, \vec{t}, L) \)). 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 \( S \).

```plaintext
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. Hence, there are two possible executions of the program with \( \gamma_1(x) = P \) and \( \gamma_2(x) = S \):

\[
\langle c, m_1 \rangle \rightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle
\]

\[
\langle c, m_2 \rangle \rightarrow \langle P, 1, \gamma_1 \rangle \cdot \langle P, 1, \gamma_1 \rangle \cdot \langle P, 2, \gamma_2 \rangle
\]

The issue is in the first execution. By observing the 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 \( \approx_L \) 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 \( L \):

\[
\langle \vec{I} \rangle b, L \triangleq \langle \vec{I} \rangle b, L, \gamma(b), L \wedge \gamma(b) \not\subseteq L
\]

Return to the example above, the effective subtraces starting from \( m_1 \) and \( m_2 \) are both \( \langle P, 2, \gamma_2(x) = S \rangle \), 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 \( \langle \vec{I} \rangle b, L \), rather than \( \langle \vec{I} \rangle L \). However, doing so is problematic as shown by the following program.

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

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

\[
\langle c, m_1 \rangle \rightarrow \langle P, 1, \gamma_1(x) = P \rangle
\]

\[
\langle c, m_2 \rangle \rightarrow \langle P, 1, \gamma_2(x) = S \rangle
\]

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 \( \langle P, 1, \gamma_2(x) = S \rangle \), 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 indistinguability 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 \( \mathcal{A}(m) \) 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 \( t \cdot t' \). Note that simply computing the knowledge difference between observing \( t \cdot t' \) and observing \( t \) does not work. Consider the example in Figure 1-B.ii. Given any program execution \( \langle c, m \rangle \rightarrow t \), we have \( k_1(c, t^{[m]}, M) = \{m\} \) 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:

- input 1: \( s_1 = (1 \cdot 1 \cdot 3) \)
- input 2: \( s_2 = (2 \cdot 2 \cdot 3) \)
- input 3: \( s_3 = (1 \cdot 1 \cdot 3) \)
- 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:

\[
\sum_{s \in \text{consist}(s_1)} P(s)
\]

where the consistent set \( \text{consist}(s_1) \) 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

\[
n = \| \vec{t}_1^{[b]} \| \setminus L \setminus \| \vec{t}_2^{[b]} \| = \| \vec{t}_1^{[n]} \| \setminus L \setminus \| \vec{t}_2^{[n]} \|
\]

Note that despite the extra complicity due to trace projection, the consistency relation is similar to the consistent set \( \text{consist}(s_1) \) 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 \( \vec{t} \):

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

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

- Challenge 2: Recall that with \( m_1(x) = 0, m_2(x) = 1, \gamma_1(x) = \text{P} \) and \( \gamma_2(x) = \text{S} \), there are two execution traces \( \langle c, m_1 \rangle \rightarrow \langle \text{P}, 1, \gamma_1 \rangle \cdot \langle \text{P}, 2, \gamma_2 \rangle \)
  \( \langle c, m_2 \rangle \rightarrow \langle \text{P}, 1, \gamma_1 \rangle \cdot \langle \text{P}, 2, \gamma_2 \rangle \)

  It is easy to check that the two output sequences are consistent according to Definition 6. 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 \( m_1(x) = 0, m_2(x) = 1, \gamma_1(x) = \text{P} \) and \( \gamma_2(x) = \text{S} \), there are two execution traces \( \langle c, m_1 \rangle \rightarrow \langle \text{P}, 1, \gamma_1 \rangle \)
  \( \langle c, m_2 \rangle \rightarrow \langle \text{P}, 1, \gamma_1 \rangle \)

  While the two traces are not consistent with each other, we know that \( k_1(c, \langle \text{P}, 1, \gamma_2(x) = \text{S} \rangle, \text{P} \) = \( \{\langle \text{P}, 1, \gamma_1(x) = \text{P}\rangle, \langle \text{P}, 1, \gamma_2(x) = \text{S}\}\) since the two traces satisfy \( \text{\sim}_p \). 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:

\[
A(m, \vec{t}, b, L) \triangleq \begin{cases} 
\|m\|_b, & b = \text{transient} \\
\|m\|_b \cap k_1(c, \vec{t}^{[\|\vec{t}^{-1}\| - 1]}, L), & b = \text{persistent}
\end{cases}
\]
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):**

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

$$k_2(c, \vec{t}^{|i|}, L, b) \supseteq \left\{ \begin{array}{ll} \left\lfloor m \right\rfloor \neq b, & \text{transient} \\ \left\lfloor m \right\rfloor \neq b \cap k_1(c, \vec{t}^{|i-1|}, L), & \text{persistent} \end{array} \right.$$

**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 $(c, m)$, i.e., $(c, m) \rightarrow \vec{t}$, 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 $A(m, \vec{t}, b, L)$: a set of initial memory that should be indistinguishable to attacker at $L$ at the end of sequence $\vec{t}$.
- **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):**

$$K(c, \vec{t}, \sim) \triangleq \{ m \mid \langle c, m \rangle \rightarrow \vec{t} \land \vec{t} \sim \vec{t} \} \quad (5)$$

Therefore, with abstract $\sim$, $A(m, \vec{t}, b, L)$ 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 $A$, a command $c$ satisfies a dynamic policy iff the knowledge gained from observing any output does not exceed its corresponding policy allowance:

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

$$\bigcup \exists m^\prime, j. \langle c, m^\prime \rangle \rightarrow \vec{t}^{|j|} \equiv \vec{t}^{|i|} \quad \forall 1 \leq j \leq ||\vec{t}||.$$  

Let $\sim_{BR} \triangleq \{ \langle \vec{t}_1, \vec{t}_2 \rangle \mid ||\vec{t}_1||_L \preceq \text{oump}_L(\vec{t}_2) \}$, $A_{BR}$ be as defined in Equation (4), and $\equiv_{BR}$ 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:

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

$$K(c, \vec{t}^{|i|}, \sim) \supseteq A(m, \vec{t}^{|i|}, b, L)$$

We use this simpler form for any dynamic policy where consistency is simply defined as equivalence.

**B. Existing works in the formalization framework**

Next, 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 (with potentially different security specification language and semantic formalization) into the specification language in Figure 2 and Definition 10 respectively. The conversion of Paralock and the correctness proofs of all conversions are available in the Supplementary Material.

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 $K(c, \vec{t}^{|i-1|}, \sim)$ as part of policy allowance. Recall that $K(c, \vec{t}^{|i-1|}, \sim)$ 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 check erasure policy, such as the example in Figure 1-B: leaking credit card after erasure violates the erasure policy.

2) **Gradual Release:** Gradual Release assumes a mapping $\Gamma$ from variables to levels in a Denning-style lattice. A release event is generated by a special command $x := \text{declassify}(e)$. Informally, a program is secure when illegal flow w.r.t $\Gamma$ only occurs along with release events. Hence, we encode a release event as

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

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

Gradual Release is formalized on the insight that “knowledge must remain constant between releases”: 

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

\[
\begin{array}{|c|c|c|}
\hline
\text{Gradual Release} & |\begin{array}{c}
\forall i, j, k, m, c, \bar{t}, \langle c, m \rangle \rightarrow \bar{t} \implies \forall i. (m', m) \in [m]_L, I \land \langle c, m \rangle \rightarrow \bar{t}' \land \bar{t} \subseteq \bar{t}'
\end{array}| & \text{No secure} \\
\text{Tight Gradual Release} & |\begin{array}{c}
\forall i, j, k, m, c, \bar{t}, \langle c, m \rangle \rightarrow \bar{t} \implies \forall i. (m', m) \in [m]_L, I \land \langle c, m \rangle \rightarrow \bar{t}' \land \bar{t} \subseteq \bar{t}'
\end{array}| & \text{Secure}
\hline
\text{According to Policy} & [m]_{\partial b} \land K(c, t[i-1], \sim_{GR}) = & [m]_{\partial b} \land K(c, t[i-1], \sim_{FA}) = \\
\text{Cryptographic Erasure} & [m]_{\partial b} \land K(c, t[i-1], \sim_{PL}) = & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = \\
\text{Forgetful Attacker} & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = \\
\text{Paralock} & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = \\
\text{Dynamic Release} & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = & [m]_{\partial b} \land K(c, t[i-1], \sim_{TA}) = \\
\hline
\end{array}
\]

\textbf{Definition 11 (Gradual Release [7]):} A command \( c \) satisfies gradual release w.r.t. \( \Gamma \) if\(^7\)

\[\forall c, m, L, i, \bar{t}, \langle c, m \rangle \rightarrow \bar{t} \implies \forall i. (m', m) \in [m]_L, I \land \langle c, m \rangle \rightarrow \bar{t}' \land \bar{t} \subseteq \bar{t}'\]

where \( k(c, m, \bar{t}, L, \Gamma) \) is

\[\{m' | m' \in [m]_L, I \land \langle c, m \rangle \rightarrow \bar{t}' \land \bar{t}' \subseteq \bar{t}'\}\] (6)

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

\[\sim_{GR} \triangleq \{ (\bar{t}_1, \bar{t}_2) | [\bar{t}_1]_L \subseteq [\bar{t}_2]_L \} \equiv_{GR} [\bar{t}_1]_L \land K(c, t[i-1], \sim_{GR}) = [\bar{t}_2]_L \]

Recall that in our encoding, a release event emits an output event \( \langle k(x), e, \gamma \rangle \), where \( \gamma \) maps all variable to public. This essentially makes the allowance check \( K(\ldots) \subseteq A \) trivially true, resembling Definition 11.

\textbf{Lemma 1:} With \( \sim_{GR} \triangleq \equiv_{GR} \) and \( A \triangleq A_{GR} \), Definition 10 is equivalent to Definition 11.

\textbf{Observation:} From Table I, it is obvious that Gradual Release uses indistinguishability \( \sim_{L} \). Its policy allowance is defined by the last dynamic specification \( t[i i-1] \gamma \), as well as the knowledge gained from previous outputs.

3) \textbf{Tight Gradual Release:} Tight Gradual Release [2], [8] is an extension of Gradual Release. Similar to Gradual Release, it assumes a base policy \( \Gamma \) and uses a \( x := \text{declassify}(c) \) command 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 \( \text{declassify}(c) \), since our language does not support partial release (Section III-B2). Second, declassification in Tight Gradual Release is both precise (i.e., only variable \( x \) in \( \text{declassify}(c) \) is downgraded) and permanent (i.e., the sensitivity of \( x \) cannot upgrade after \( x \) is declassified). Hence, we encode \( x' := \text{declassify}(c) \) as

\[\text{Event}(r_x) : x' := x; \text{output}(\Gamma(x'), x);\]

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

Tight Gradual Release uses the same knowledge definition from Gradual Release, but its execution traces also dynamically track the set of declassified variables \( X \):

\[\langle c, m, \emptyset \rangle \rightarrow^* \langle c', m', X \rangle\]

\textbf{Definition 12 (Tight Gradual Release):} A program \( c \) is secure if for any trace \( \bar{t} \), initial memory \( m \) and attacker at level \( L \), we have

\[\forall i, 1 \leq i \leq ||\bar{t}||. (||m||_L, I \land ||m||_X_i \subseteq k(c, m, t[i], L, \Gamma) \]

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 \( \bar{t} \) we have:

\[||m||_L, I \land ||m||_X_i \subseteq ||m||_L \land ||m||_X_i \]

Hence, we can rephrase Tight Gradual Release as follows:

\[\sim_{TGR} \triangleq \{ (\bar{t}_1, \bar{t}_2) | [\bar{t}_1]_L \subseteq [\bar{t}_2]_L \} \equiv_{TGR} [\bar{t}_1]_L \land K(c, t[i-1], \sim_{TGR}) = [\bar{t}_2]_L \]

\textbf{Lemma 2:} With \( \sim_{GR} \triangleq \equiv_{GR} \) and \( A \triangleq A_{GR} \), Definition 10 is equivalent to Definition 12.

\textbf{Observation:} Tight Gradual Release is more precise than Gradual Release since the encoding of \( \text{declassify}(c) \) precisely downgrades the sensitivity of \( x \) but not any other variables, while the encoding for Gradual Release downgrades all variables.

Compared to Dynamic Release, the most important difference is that the consistency relation \( \equiv \) is defined in completely 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 I-B.1 with \( m_1(\text{credit card}) = 0 \), \( m_2(\text{credit card}) = 1 \) and attacker level \( M \). Given a program execution \( \langle c, m_1 \rangle \rightarrow \bar{t}, \) we have \( K(c, t[i], \sim_{GR}) = \{ m_1 \} \) for

\[\text{Event}(r_x) : x' := x; \text{output}(\Gamma(x'), x);\]
all $i \geq 1$. However, $credit\_card$ is upgraded from $M$ to $T$
when $i = 2$ (i.e., the second output), the secure program
(i) is incorrectly rejected since $K(c, t), \sim_{GR} = \{m_1\} \not\sim \{m_1, m_2\} = [m_1]_R \triangleq \gamma$.

4) According to Policy: Chong and Myers propose noninterference according to policy in [18], [19] to integrate erasure
and declasification 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 is either a simple level $l$ drawn from a Denning-style lattice, or in the form of $q_1 \triangleq q_2$, 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:

$$∀m_1, m_2, ℓ, \vec{t}_1, \vec{t}_2, ∀y \neq x. m_1(y) = m_2(y) \land (c, m_1) → \vec{t}_1 \land (c, m_2) → \vec{t}_2 \implies R. \left(∀(i, j) ∈ R, ℓ \not\in [b]_{τ_1 \downarrow i} \land ℓ \not\in [b]_{τ_2 \downarrow i} \implies τ_1 \equiv τ_2 \right)$$

where a correspondence $R$ between traces $τ_1$ and $τ_2$ is a subset of $N × N$ such that:

1) (Completeness) either $(i, j) ∈ R = \{i ∈ N | i < |τ_1|\} or \{j | j, i ∈ N | j < |τ_2|\}$, and
2) (Initial configurations) if $|R| > 0$ then $(0, 0) ∈ R$, and
3) (Monotonicity) for all $(i, j) ∈ R$ and $(i’, j’) ∈ R$, if $i < i’$ then $j ≤ j’$ and symmetrically, if $j < j’$ then $i ≤ i’$.

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., $∀y \neq x. m_1(y) = m_2(y)$), 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 $∥m∥_b$ in our framework, that considers memories that differ only for variables with a particular policy $b$.
- The component of $ℓ \not\in [q]_{τ_1 \downarrow i} \land ℓ \not\in [q]_{τ_2 \downarrow i} filters out non-interesting outputs, which functions the same as the filtering function $[ℓ]_{i,b,L}$.
- We define $≡$ on two output sequence as below:

$$[ℓ]_{i,b,L} \equiv [ℓ]_{i,b,L} \iff −→ (∥[ℓ]∥ \land ∃i. [ℓ]_{i} \not\equiv [ℓ]_{i})$$

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

$$∀m_1, m_2, ℓ, \vec{t}_1, \vec{t}_2 \implies \neg(∥[ℓ]∥ \land ∃i. [ℓ]_{i} \not\equiv [ℓ]_{i})$$

$≡_{AP} ≜ A_{AP} ≜ ∥m∥_b$.

**Lemma 3:** With $∼_{AP} ≜ A_{AP}$, and outside equivalence $≡_{AP} ≜ A_{AP}$, 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 falsly accepts insecure programs.

5) **Cryptographic Erasure:** Cryptographic erase [5] uses the same compound labels to describe eraseus policy and knowledge is defined as:

$CE(c, L, ℓ) = \{m | \langle c, m \rangle \rightarrow^* \langle c_1, m_1 \rangle \rightarrow^* \langle c', m' \rangle \land \vec{t}_2_L = \vec{t}_1_L\}$

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 nor 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:

$$∀c_0, m_0, c_1, c_i, c, c_n, m_n, \vec{t}_1, \vec{t}_2, L, i, n.
⟨c_0, m_0⟩ \rightarrow^* ⟨c_i, m_i⟩ \rightarrow^* ⟨c, m_n⟩
\implies CE(c, L, ℓ) ⊇ \bigcup_{i ∈ \vec{t}_2} ∥m∥_{L, i, γ}$$

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

$$∼_{CE} ≜ \{(1, 2) | \vec{t}_1 \downarrow _L \text{ subtrace of } \vec{t}_2 \downarrow _L\}$$

**Lemma 4:** With $∼_{CE} ≜ ∪_{i ∈ [ℓ]} ∥m∥_{L, i, γ}$ and $A_{CE} ≜ A_{CE}$, Definition 10 with adjusted attack model is equivalent to Definition 14.

**Observation:** Compare with other works, 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 \( \text{Atk}(Q_A, q_{init}, \delta_A) \), where \( Q_A \) is a set of attacker’s states, \( q_{init} \in Q_A \) is the initial state, and \( \delta_A \) is the transition function. The attacker observes a set of events produced by a program execution, and updates its state accordingly:

\[
\text{Atk}(e) = q_{init} \\
\text{Atk}(q_{\delta_i}) = \delta(\text{Atk}(q_{\delta_{i-1}}), t[i])
\]

Given a program \( c \), an automaton \( \text{Atk} \) 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:

\[
k_{FA}(c, L, \text{Atk}, \vec{t}) = \{ m \mid \langle c, m \rangle \xrightarrow{t_1} \langle c', m' \rangle \xrightarrow{t_2} \ast m'' \wedge \text{Atk}(\langle \vec{t}_1 \rangle_L) = \text{Atk}(\langle \vec{t}_2 \rangle_L) \}
\]

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

\[
\forall c, c', m, m', \vec{t}, \vec{t}', L. \langle c, m_1 \rangle \rightarrow \vec{t}, \vec{t}' \Rightarrow \\
k_{FA}(c, L, \text{Atk}, \vec{t}, \vec{t}') \supseteq k_{FA}(c, L, \text{Atk}, \vec{t}) \cap \llbracket m \rrbracket_{L, \gamma'}
\]

The conversion of Definition 15 to our framework is straightforward:

\[
\sim_{FA} \triangleq \{(\vec{t}_1, \vec{t}_2) \mid \exists \vec{t} \leq \vec{t}_2, \text{Atk}(\vec{t}_1) = \text{Atk}(\vec{t})\}
\]

\[
\equiv_{FA} \triangleq \forall A_{FA} \in A_{FA}, \sim_{FA}, \text{outside equivalence } \equiv_{FA} \triangleq A_{FA}, \text{Definition 10 is equivalent to Definition 15.}
\]

**Observation**: We note that Forgetful Attacker (Definition 15) was originally formalized in the same format as Dynamic Release (the persistent case). However, there are various differences in the modeling, as can be observed from Table I. Most importantly, Forgetful Attacker security is parameterized by an automaton \( \text{Atk} \); 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 automaton 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.

**VI. EVALUATION**

In this section, we introduce \( \text{AnnTrace} \) benchmark and implement the dynamic policies as the form shown in Table I. The benchmark and implementations are available on github\(^9\).

**A. \( \text{AnnTrace} \) Benchmark**

To facilitate testing and understanding of dynamic policies, we created the \( \text{AnnTrace} \) benchmark. It consists of a set of programs annotated with \textit{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:

- \textit{secure}, a boolean value indicating whether this program is a secure program; the ground truth of our evaluation.
- \textit{source code}, written in the syntax shown in Fig 2;
- \textit{persistent}, a boolean value indicating whether the intended policy in this program is persistent (or transient);
- \textit{lattice}, \( \mathcal{L} \), the security lattice used by the program\(^10\);
- \textit{traces}, executions of the program. Each trace \( \tau \) has:
  - \textit{initial memory}, \textit{m}, mapping from variables to integers
  - \textit{outputs}, \textit{t}, a list of output events, each \( t \) in type \textit{Out}:
    - \textit{output level}, \textit{\ell}, a level from the lattice \( \mathcal{L} \)
    - \textit{output value}, \textit{v}, an integer value
    - \textit{policy state}, \( \gamma \), 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 corresponding 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.

\(^9\)https://github.com/psuplus/AnnTrace

\(^10\)We use lattice instead of level set for conciseness in the implementation.
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 AnTrace 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 AnTrace 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 applicable 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 summarized 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.

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.

![Figure 6: Counterexamples for Crypto-Erasure and Paralocks.](image)

11 Although we are unable to evaluate 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].
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 \( A_1 = (a, 0) \), who has an empty lock set. When lock \( D \) is open, since \( \{D\} \not\subseteq \emptyset \), there is no restriction for the assignment \( I2 := h1 \). 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 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 presence 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

APPENDIX

A. Paralock in Table I

Paralock [13], [14], [15] uses locks to formalize the sensitivity of security objects. Paralock uses fine-grained modal logic 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 \( \{ \Sigma \Rightarrow a; \ldots \} \), where \( \Sigma \) 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 trusted location \( L \) 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 \( \Sigma \). To simplify notation, we use \( \Gamma(x, a) = \Sigma \) to denote the fact that \( \{ \Sigma \Rightarrow a; \ldots \} \) 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 \( \langle c, m \rangle \xrightarrow{b:a,\gamma,\Delta} \langle c', m' \rangle \), it generates the output event \( t = \langle b, v, \gamma, \Delta \rangle \), where \( \Delta = \text{unlock}(\langle c, m \rangle) \).

Let \( \|A\| \) be the set of variables that are visible to \( A \), and \( [f]_A \) be the outputs that are visible to \( A = (a, \Sigma, A) \):

\[
\|A\| \triangleq \{ x \mid \forall x \in \text{Vars. } \Gamma(x, a) \subseteq \Sigma \}
\]

\[
[f]_A \triangleq \{ f \} \text{Ab,n,\gamma,\Delta} \cdot \Gamma(b,a) \subseteq \Sigma A
\]

Paralock security defines attacker’s knowledge\(^{12}\) as follows:

\[
k_{PL}(c, m, \vec{t}, A) = \{ m' \mid \forall x \in \text{Vars. } \Gamma(x, a) \subseteq \Sigma \}
\]

\[
\wedge (c, m') \xrightarrow{\vec{t}} \ast(c', m'') \xrightarrow{\vec{t}} \ast m''' \wedge \langle f \rangle_A = \langle f \rangle_A
\]

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 \( A = (a, \Sigma) \), the attacker’s knowledge remains unchanged whenever unlock(\( \tau_i[a] \)) \( \subseteq \Sigma A \):

\[
\forall c, m, m', \vec{t}, t', a, \Sigma A, \text{A, i.} \quad \langle c, m \rangle \xrightarrow{\vec{t}} \langle c', m' \rangle \xrightarrow{\vec{t}} \langle c'', m'' \rangle \wedge A = \langle a, \Sigma A \rangle \wedge \text{unlock}(c'', m'') \subseteq \Sigma A
\]

\[
\Rightarrow k_{PL}(c, m, \vec{t}, t', A) = k_{PL}(c, m, \vec{t}, A)
\]

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

\[
\|m\|_A \triangleq \{ m' \mid \forall x \in \text{Vars. } \Gamma(x, a) \subseteq \Sigma \Rightarrow m(x) = m'(x) \}
\]

The conversion of Definition 16 to our framework is straightforward.

\[
\sim_{PL} \triangleq \{ \langle \vec{t}_1, \vec{t}_2 \rangle \mid \| \vec{t}_1 \|_A \text{ prefix of } \| \vec{t}_2 \|_A \} \equiv_{PL} \Rightarrow
\]

\[
\mathcal{A}_{PL} \triangleq \{ K(c, \vec{t}^{[i]-1}, \sim_{PL}) \cap \| m \|_A, \vec{t}^{[i]} \} \subseteq \| m \|_0, \text{ otherwise}
\]

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

B. Equivalence Proof for Table I

We first introduce a useful lemma which allows us to rewrite the orginal knowledge definition in [7] to the knowledge definition \( K \) in this paper.

**Lemma 7:** Let \( k \) be defined as in Equation 6 and \( K \) be defined as in Equation 5, then we have

\[
\forall c, m, L, \Gamma, \vec{t}, M,
\]

\[
M \subseteq \| m \| L, \Gamma \wedge k(c, m, \vec{t}, L, \Gamma) \subseteq M \Rightarrow k(c, m, \vec{t}, L, \Gamma) = M \iff K(c, \vec{t}, \sim_{GR}) \supseteq M
\]

**Proof.** By definition, we know

\[
k(c, m, \vec{t}, L, \Gamma) = K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

\[
\text{case } \Rightarrow: \text{ we know}
\]

\[
M = k(c, m, \vec{t}, L, \Gamma) = K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

\[
K(c, \vec{t}, \sim_{GR}) \supseteq K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

Thus, we have \( K(c, \vec{t}, \sim_{GR}) \supseteq M \).

\text{case } \Leftarrow: \text{ we know}

\[
k(c, m, \vec{t}, L, \Gamma) = K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

\[
M \subseteq K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

\[
M \cap M \subseteq K(c, \vec{t}, \sim_{GR}) \cap \| m \| L, \Gamma
\]

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

So, we have \( k(c, m, \vec{t}, L, \Gamma) = M \iff K(c, \vec{t}, \sim_{GR}) \supseteq M \).

\[\square\]

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

\[
\forall c, m, L, \vec{t}, L, \vec{t} \quad \langle c, m \rangle \xrightarrow{\vec{t}} \Rightarrow
\]

\[\text{i not release event} \Rightarrow k(c, m, \vec{t}^{[i]} \cdot L, \Gamma) = k(c, m, \vec{t}^{[i]-1}, L, \Gamma)
\]

\[
\iff K(c, \vec{t}^{[i]} \cdot \gamma, \sim_{GR}) \supseteq \| m \| L, \vec{t}^{[i]} \cdot \gamma \cap K(c, \vec{t}^{[i]-1}, \sim_{GR})
\]

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

\[
\vec{t}^{[i]} \cdot \gamma = \{ \gamma \sqcup, i \text{ is a release event}
\]

\[
\{ \Gamma, \text{ i not a release event}
\]

\[\square\]
• case when \( i \) is a release event: \( \tilde{t}^{[i]}, \gamma = \gamma_L \). From the definition, we know \( \llbracket m \rrbracket_L, \gamma_L \) returns the singleton set \( \{ m \} \).
  From \( (c, m) \leftrightarrow \tilde{t} \) and the definition of \( \mathcal{K} \), we know \( \forall j. m \in \mathcal{K}(c, \tilde{t}^{[i]}, \gamma_L) \):
  \[
  m \in \mathcal{K}(c, \tilde{t}^{[i]}, \gamma_L)
  \]
  \[
  \{ m \} = \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma
  \]
  Thus, both Definition 10 and 11 are trivially true.

• case when \( i \) is not a release event: \( \tilde{t}^{[i]}, \gamma = \Gamma \). From the definitions, we know \( \sim_{GR} = \sim_{XI} \). We know from the monotonicity of the knowledge that:
  \[
  k(c, m, \tilde{t}^{[i-1]}, L, \Gamma) \subseteq \llbracket m \rrbracket_L, \Gamma
  \]
  \[
  k(c, m, \tilde{t}^{[i]}, L, \Gamma) \subseteq k(c, m, \tilde{t}^{[i-1]}, L, \Gamma)
  \]
  So, we can instantiate Lemma 7 with:
  \[
  M := k(c, m, \tilde{t}^{[i-1]}, L, \Gamma), \quad \tilde{t} := \tilde{t}^{[i]}
  \]
  and we get:
  \[
  k(c, m, \tilde{t}^{[i]}, L, \Gamma) = k(c, m, \tilde{t}^{[i-1]}, L, \Gamma)
  \]
  \[
  \iff \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{GR}) \supseteq k(c, m, \tilde{t}^{[i-1]}, L, \Gamma)
  \]
  By definition, we know
  \[
  k(c, m, \tilde{t}^{[i-1]}, L, \Gamma) = \llbracket m \rrbracket_L, \Gamma \cap \mathcal{K}(c, \tilde{t}^{[i-1]}, \sim_{GR})
  \]
  Thus, when \( i \) is not a release event, we have:
  \[
  k(c, m, \tilde{t}^{[i]}, L, \Gamma) = k(c, m, \tilde{t}^{[i-1]}, L, \Gamma)
  \]
  \[
  \iff \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{GR}) \supseteq \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma \cap \mathcal{K}(c, \tilde{t}^{[i-1]}, \sim_{GR})
  \]
  Therefore, Definition 10 is equivalent to Definition 11. \( \square \)

2) Tight Gradual Release: Lemma 2. With \( \sim_{GR} = \sim_{TGR} \) and \( A = A_{TGR} \), Definition 10 is equivalent to Definition 12.

\[ \forall i. 1 \leq i \leq \| \tilde{t} \|. \quad (\llbracket m \rrbracket_L, \Gamma \cap \llbracket m \rrbracket_{E_i}) \subseteq k(c, m, \tilde{t}^{[i]}, L, \Gamma) \iff \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{TGR}) \supseteq \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma \]

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:
  \[
  k(c, m, \tilde{t}, L, \Gamma) = \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{TGR}) \cap \llbracket m \rrbracket_L, \Gamma
  \]
  From encoding, we know that
  \[
  \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma = (\llbracket m \rrbracket_L, \Gamma \cap \llbracket m \rrbracket_{E_i})
  \]

• case \( \iff \): From Equation 8 and the assumption, we know
  \[
  \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma \subseteq k(c, m, \tilde{t}^{[i]}, L, \Gamma)
  \]

From Equation 7, we know
  \[
  k(c, m, \tilde{t}^{[i]}, L, \Gamma) \subseteq \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{TGR})
  \]
  Therefore, we have \( \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma \subseteq \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{TGR}) \).

• case \( \iff \): By taking an intersection with \( \llbracket m \rrbracket_L, \Gamma \) on both side of the assumption, we have:
  \[
  \mathcal{K}(c, \tilde{t}^{[i]}, \sim_{TGR}) \cap \llbracket m \rrbracket_L, \Gamma \supseteq \llbracket m \rrbracket_L, \tilde{t}^{[i]}, \gamma \cap \llbracket m \rrbracket_L, \Gamma
  \]
  Apply Equation 7 to the left and Equation 8 to the right:
  \[
  k(c, m, \tilde{t}^{[i]}, L, \Gamma) \supseteq (\llbracket m \rrbracket_L, \Gamma \cap \llbracket m \rrbracket_{E_i}) \cap \llbracket m \rrbracket_L, \Gamma
  \]
  \[
  = \llbracket m \rrbracket_L, \Gamma \cap \llbracket m \rrbracket_{E_i}
  \]
  Thus, we have \( k(c, m, \tilde{t}^{[i]}, L, \Gamma) \supseteq (\llbracket m \rrbracket_L, \Gamma \cap \llbracket m \rrbracket_{E_i}) \).
  Therefore, Definition 10 is equivalent to Definition 12. \( \square \)

3) According to Policy p: Lemma 3. With \( \sim_{AP} \equiv \sim_{AP} \) and \( A = A_{AP} \), outside equivalence \( \equiv \equiv_{AP} \), 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,
  \[
  \forall c, m_0, m', c_i, m_i, i, e, \Gamma.
  \]
  \[
  \tau = (c, m_0)_{\gamma_0} \rightarrow (c, m_i)_{\gamma_i} \rightarrow m' \quad \land \quad \tau'[i] = (c, m_i)_{\gamma_i}
  \]
  \[
  \iff (c, m_0) \rightarrow \tilde{t}^{[i]}
  \]
  \[
  \land \tilde{t}^{[i]} = \{ (c, h, n, \gamma) \mid ch = e \land n = m(c) \wedge \gamma = \gamma_i \}
  \]
  We note that our normal \( \tilde{t}^{[i]} \) returns a single output event, say some \( l = (c, h, n, \gamma) \). But here we overload \( \tilde{t}^{[i]} \) to return a set of output events that output all values on memory \( \gamma[i] \). Thus, with all values on memory outputted, we have:
  \[
  \forall c, m_1, m_2, m'_1, m'_2, \tau, \tau', \tilde{t}_1, \tilde{t}_2.
  \]
  \[
  \land \tau = (c, m_1) \rightarrow m'_1 \quad \land \tau' = (c, m_2) \rightarrow m'_2
  \]
  \[
  \land (c, m_1) \rightarrow \tilde{t}_1 \quad \land (c, m_2) \rightarrow \tilde{t}_2
  \]
  \[
  \iff \tau[i] \cong \tau'[i] \iff [\tilde{t}^{[i]}]_l = [\tilde{t}^{[i]}]_{l'}.
  \]
  Thus, we rewrite Definition 13 in following two-run style:
  \[
  \forall c, m_1, m_2, l, p, \tilde{t}_1, \tilde{t}_2.
  \]
  \[
  m_2 \in \llbracket m \rrbracket_L \land (c, m_1) \rightarrow \tilde{t}_1 \quad \land (c, m_2) \rightarrow \tilde{t}_2 \iff \exists R' \left( \forall (i, j) \in R. \tilde{t}_1[i] \in \llbracket \tilde{t}_1 \rrbracket_{p,l} \land \tilde{t}_2[i] \in \llbracket \tilde{t}_2 \rrbracket_{p,l} \quad \iff \| \tilde{t}_1 \|_l = \| \tilde{t}_2 \|_{l'} \right)
  \]
  We combine the two filters and assume \( R' \) as \( R \) after filtering:
  \[
  \forall c, m_1, m_2, l, p, \tilde{t}_1, \tilde{t}_2.
  \]
  \[
  m_2 \in \llbracket m \rrbracket_L \land (c, m_1) \rightarrow \tilde{t}_1 \quad \land (c, m_2) \rightarrow \tilde{t}_2 \iff \exists R' \left( \forall (i, j) \in R. (\| \tilde{t}_1 \|_{p,l})[i] = (\| \tilde{t}_2 \|_{p,l})_{[i]} \right)
  \]
With $\mathcal{K}(c, \vec{t}, \sim_{AP})$ unfolded as below:

$$\mathcal{K}(c, \vec{t}, \sim_{AP}) = \{m_2 | \forall m_2, \vec{t}'_2. \langle c, m_2 \rangle \rightarrow \vec{t}'_2$$
\[
\land \exists R'. \forall (i, j) \in R'. \langle (\vec{t}'_2)_p, l \rangle_{[i]} = \langle (\vec{t}'_2)_p, l \rangle_{[j]} \} \]

We can further rewrite the definition as follow:

$$\forall c, m_1, l, p, \vec{t}'_1.

m_2 \in \|m_1\|_p \land \langle c, m_1 \rangle \rightarrow \vec{t}'_1 \implies m_2 \in \mathcal{K}(c, \vec{t}'_1, \sim_{AP})$$

That is,

$$\forall c, m_1, l, p, \vec{t}'_1, \langle c, m_1 \rangle \rightarrow \vec{t}'_1 \land \|m_1\|_p \subseteq \mathcal{K}(c, \vec{t}'_1, \sim_{AP})$$

We note that only the equivalence relation in $\sim_{AP}$ is $\equiv_{AP}$. The equivalence relation in $\vec{t}' \equiv \vec{t}$ in Definition 10 in this case is not $\equiv_{AP}$, but $\equiv_{AP}' \equiv \mathcal{K}_{CE}$.\[\square\]

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

$$\forall c, m, m_0, c_i, m_i, m_n, \gamma_n, m', \vec{t}'_1, \vec{t}_2, l, i, j, n.

\langle c, m_0 \rangle \rightarrow_{\vec{t}_1} \langle c_i, m_i \rangle \rightarrow_{\vec{t}_2} \langle c_n, m_n \rangle \rightarrow{*} m' \implies

\mathcal{K}(c, \vec{t}_1, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} \iff

\mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$$

**Proof.** We note that in Definition 14, $\gamma_j$ 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 $n-j \geq \|\vec{t}_2\|$. We first show that the right hand side allowance defined using $\gamma_j$ is the same as using state policy from the output sequence $\vec{t}$:

$$\mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} \iff \mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$$

**Proof.** We note that in Definition 14, $\gamma_j$ 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 $n-j \geq \|\vec{t}_2\|$. We first show that the right hand side allowance defined using $\gamma_j$ is the same as using state policy from the output sequence $\vec{t}$:

$$\mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} \iff \mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$$

**Proof.** We note that in Definition 14, $\gamma_j$ 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 $n-j \geq \|\vec{t}_2\|$. We first show that the right hand side allowance defined using $\gamma_j$ is the same as using state policy from the output sequence $\vec{t}$:

$$\mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} \iff \mathcal{K}(c, \vec{t}_2, \sim_{CE}) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$$

From the definitions, we know $\vec{t}_{2[0]} \gamma = \gamma_i$ if $\langle c_i, m_i \rangle$ does not immediately generates an empty output event. Otherwise, if the first non-empty event is generated at configuration $\langle c_i, m_i \rangle$ of $i < i' < n$, 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:

$$k_{CE}(c, L, \vec{t}_2) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} = \|m\|_{L, \vec{t}_{2[0]} \gamma}$$

Thus, we have $k_{CE}(c, L, \vec{t}_2) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$.

- case $\iff$: from $n-j \geq \|\vec{t}_2\|$, we know:

$$\forall t_n \in \vec{t}_2. \exists j' \in [i, n]. \gamma_{j'} = t_n \gamma$$

$$\{t_{n-1} \subseteq \gamma_{j} : i \leq j \leq n \}

\cap \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j} \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_j}$$

Thus, we have $k_{CE}(c, L, \vec{t}_2) \supseteq \bigcap_{i \leq j \leq n} \|m\|_{L, \gamma_{j'}}$.

Therefore, we know Equation 9 is true.

Now we convert a security level $L$ from Denning’s style to our attacker level $\vec{t}$ as described in SectionIII. Let $[c] \equiv \{m | \exists \vec{t}. \langle c, m \rangle \rightarrow \vec{t} \}$ denote the set of memory that terminates. From definition we know:

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

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

5) Forgetful Attacker: Lemma 5. With $\sim_{AP}$ and $A \equiv A_{AP}$. 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:

$$\langle c, m \rangle \rightarrow \vec{t'} \rightarrow \langle c', m' \rangle \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet$$

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

$$\|m\|_{L, \gamma'} = \|m\|_{L, \gamma'}$$

Definition 15 is rephrased as:

$$\forall c, m, L, i, \vec{t'}, \langle c, m \rangle \rightarrow \vec{t'} \implies

k_{FA}(c, L, \text{Atk}, \vec{t'}[i]) \subseteq k_{FA}(c, L, \text{Atk}, \vec{t'}[i-1]) \cap \|m\|_{L, \vec{t'}[0]}$$

By definition, we know:

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

Thus, we know Definition 15 is equivalent to Definition 10.\[\square\]
6) Paralock: Lemma 6. With \( \sim_{\text{PL}} \) and \( A \triangleq A_{\text{PL}} \), Definition 10 is equivalent to Definition 16.

\[
\forall c, m, \vec{t}, \vec{t'}, i, A. \ (c, m) \iff \vec{t} \land \vec{t'} = \vec{t}^{[i]} \implies
k_{\text{PL}}(c, m, \vec{t}^{[i]}, A) = k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A)
\]

Thus, we have:

\[
k_{\text{PL}}(c, m, \vec{t}^{[i-1]}, A) \supseteq k_{\text{PL}}(c, m, \vec{t'}^{[i]}, A)
\]

Therefore, Definition 10 is equivalent to Definition 16. \( \square \)

**Proof.** We omit the case when \( \vec{t}^{[i]}, \Delta \nsubseteq \Sigma_A \) since both definitions are trivially true. By Definition, we know:

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

- case \( \implies \): we know:

\[
k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A) = k_{\text{PL}}(c, m, \vec{t}^{[i]}, A)
\]

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

\[
K(c, \vec{t}^{[i]}, \sim_{\text{PL}}) \supseteq K(c, \vec{t'}^{[i-1]}, \sim_{\text{PL}}) \cap \|m\|_A
\]

Thus, we have

\[
K(c, \vec{t}^{[i]}, \sim_{\text{PL}}) \supseteq k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A)
\]

With \( k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A) = K(c, \vec{t'}^{[i-1]}, \sim_{\text{PL}}) \cap \|m\|_A \), we get \( K(c, \vec{t}^{[i]}, \sim_{\text{PL}}) \supseteq K(c, \vec{t'}^{[i-1]}, \sim_{\text{PL}}) \cap \|m\|_A \).

- case \( \iff \): we know:

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

Thus, we have

\[
k_{\text{PL}}(c, m, \vec{t}^{[i]}, A) \supseteq k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A)
\]

(10)

We know \( \|m\|_A \) is the initial knowledge of \( A \) before observing any output event. From the monotonicity of Paralock knowledge, we know:

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

(11)

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

\[
(K(c, \vec{t}^{[i]}, \sim_{\text{PL}}) \cap \|m\|_A)
\]

\[
\supseteq (k_{\text{PL}}(c, m, \vec{t}^{[i-1]}, A) \cap k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A))
\]

\[
= k_{\text{PL}}(c, m, \vec{t'}^{[i-1]}, A)
\]

Thus, we have

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

By Definitions, we have:

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

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