Dynamic Policies Revisited

Amir M. Ahmadian
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

***Abstract*—Information flow control and dynamic policies is a difficult relationship yet to be fully understood. While dynamic policies are a natural choice in many real-world applications that downgrade and upgrade the sensitivity of information, understanding the meaning of security in this setting is challenging. In this paper we revisit the knowledge-based security conditions to reinstate a simple and intuitive security condition for dynamic policies: A program is secure if at any point during the execution the attacker's knowledge is in accordance with the active security policy at that execution point. Our key observation is the new notion of *policy consistency* to prevent policy changes whenever an attacker is already in possession of the information that the new policy intends to protect. We use this notion to study a range of realistic attackers including the perfect recall attacker, bounded attackers, and forgetful attackers, and their relationship. Importantly, our new security condition provides a clean connection between the dynamic policy and the underlying attacker model independently of the specific use case. We illustrate this by considering the different facets of dynamic policies in our framework.**

On the verification side, we design and implement DynCoVer, a tool for checking dynamic information-flow policies for Java programs via symbolic execution and SMT solving. Our verification operates by first extracting a graph of program dependencies and then visiting the graph to check dynamic policies for a range of attackers. We evaluate the effectiveness and efficiency of DynCoVer on a benchmark of use cases from the literature and designed by ourselves, as well as the case study of a social network. The results show that DynCoVer can analyze small but intricate programs indicating that it can help verify security-critical parts of Java applications. We release DynCoVer publicly to support open science and encourage researchers to explore the topic further.

1. Introduction

Information flow control provides an appealing security framework for reasoning about dependencies between information sources and information sinks, and for ensuring that these dependencies adhere to desirable security policies. In a language-based setting, this security framework has the following ingredients: (1) an execution model which is given by the execution semantics of a program; (2) an attacker model specifying the observation power of an attacker over the attacker-visible sources and sinks; (3) a security policy specifying, for each execution point, the permitted information flows from sources to sinks, disallowing information flows from secret/high sources to public/low sinks; (4) a security condition (or security property) capturing a program's security with respect to an execution model, an attacker model, and a security policy. A classical security condition is noninterference requiring that any two executions starting with equal values on public sources yield equal values on public sinks [1].

A common trait in much recent work on information flow control has been the appeal to attacker-centric security conditions based on the concept of knowledge as a fundamental mechanism to bring out what security property is being sought and compare it with the knowledge permitted by the security policy [2], [3], [4]. Arguably, this appeal to knowledge, usually as equivalence relations on initial states, has produced clear and intuitive security conditions able to accommodate various notions of information downgrading (declassification) on which soundness arguments for enforcement mechanisms, e.g., security type systems, can be based [5], [6], [7]. In a nutshell, knowledge-based security conditions capture an intuitive requirement: A program is secure with respect to a security policy if at any point during the program's execution, the attacker's knowledge is not greater than the knowledge permitted by the active security policy at that execution point.

While this simple and elegant condition is well-understood for static multilevel security policies that only downgrade the sensitivity of information, it does not appropriately capture the security requirements of systems that change their security policies dynamically, thus both downgrading and upgrading the sensitivity of information in response to security-sensitive events. This is not satisfactory because dynamic policies are a natural choice for many real-world applications, e.g., healthcare systems, social networks, database systems, where access to information may be granted or revoked to different principals in accordance with their specific role at a given moment [8], [9], [10], [11]. Existing works address the challenge of dynamic policies by proposing security conditions that capture specific *facets* of a targeted use case [12], [13], [2], [14], [15], [16], [8], [9]. Broberg et al. [11] systematize existing research on dynamic policies illuminating the different facets exhibited by existing security conditions.

In this paper we revisit the state-of-the-art of security conditions for dynamic policies to reinstate attacker-centric knowledge-based conditions. Our starting point is the work of Askarov and Chong [8] which proposes a general framework for weakening of the attackers' observation power to accommodate dynamic policies. We further revise and develop this framework targeting three types of realistic attackers: (1) *perfect recall* attackers recalling all observations on public sinks; (2) *bounded* memory attackers recalling a bounded number of observations on public

sinks; and (3) *forgetful* attackers recalling observations on public sinks up to a security policy change. While the first two attackers are standard, the third attacker, as we will see, is useful in settings where the release of knowledge is transient and it is limited to the event of a security policy change. For example, the event of changing the database policy to revoke access on a table to user A may reflect the security requirement that user A should no longer read data from that table, independently of whether or not user A accessed the table before the policy change.

Our key observation is the notion of *policy consistency* to reflect the observation power of an attacker and thus prevent a policy change whenever the attacker is already in possession of the information that the new policy intends to protect. For example, under the model of a perfect recall attacker, a policy change that revokes access to a resource that the attacker has already observed (possibly at a past time when access to that resource was granted to the attacker) should result in an inconsistent policy, since it violates the assumption on the perfect recall attacker. Unfortunately, existing works assume that policy changes are always consistent, which has often resulted in ad hoc and unintuitive security conditions. This simple but fundamental insight allows us to reestablish clear and intuitive attacker-centric knowledge-based conditions for dynamic policies. More importantly, the new security conditions are in line with the above-mentioned ingredients required by a security framework and they provide a clean separation between policy concerns and enforcement concerns. A policy designer can instantiate our framework in accordance with the security requirements for the use case at hand, by specifying the most suitable attacker model. We validate our framework by revisiting the facets of dynamic policies by Broberg et al. [11]. Moreover, in contrast to Askarov and Chong [8], we prove that, in absence of inconsistent policy changes, a perfect recall attacker is indeed stronger than a bounded memory attacker and a forgetful attacker. Finally, we discuss policy updates whenever inconsistencies are detected.

Our second contribution is the design and implementation of algorithms for verifying dynamic information-flow policies via symbolic execution and SMT solving. Our verification method operates by first extracting program dependencies and then using these dependencies to check dynamic policies for a range of attackers including perfect recall, bounded memory, and forgetful attackers. The verification algorithms adapt and extend existing approaches for checking noninterference via automated theorem proving [17], [18] and self composition [19] to the setting of dynamic policies, including the detection and repair of inconsistent policy changes. We implement [20] an open-source prototype for Java programs and evaluate the effectiveness and efficiency on a collection of benchmark from the literature and designed by ourselves, as well as the case study of a social network. The results show that DYNCOVER can analyze small but intricate programs indicating that it can help verify security-critical parts of Java applications.

In summary the paper offers these contributions:

- We revisit the state-of-the-art security conditions for dynamic policies and reinstate clear and intuitive knowledge-based conditions based on the observation power of the attacker (Section 2 and 4).
- We show how our new framework can be used to capture the different facets of dynamic policies proposed in the literature (Section 5).
- We design verification algorithms based on symbolic execution and SMT solving to check the security of Java programs for a range of attacker models, as well as to detect inconsistent policies (Section 6).
- We implement DYNCOVER [20] and evaluate the efficiency and effectiveness on a collection of benchmarks and the case study of a social network (Section 7).

2. Problem Setting and Solution Overview

This section gives an informal overview of dynamic policies discussing the challenges, pointing out limitations of existing solutions, and arguing for revised knowledge-based security conditions. The key question is: *What is a suitable security condition for dynamic policies?*

To provide a common ground for comparing the different approaches, we borrow the notation and examples from Askarov and Chong [8] and Broberg et al. [11]. We write $A \rightarrow B$ ($A \not\rightarrow B$) to denote a security policy allowing (disallowing) information flows from security level A to security level B. We assume that no information flows between different security levels are allowed initially. For simplicity, the name of a program variable (e.g., `movie`) will match the security level of the variable (e.g., $Movie$).

We write k_i and p_i to denote the *attacker's knowledge* and the *active security policy* at program location i, respectively. By default, we assume that attackers are perfect recall, remembering any information they observe during a program's execution. A popular security condition [3], [4], [10], which is used in systems that handle only declassification of information, is given by equation (1) requiring that at any location i the attacker's knowledge k_i is smaller[1] than the policy knowledge (i.e., the knowledge allowed by the security policy) p_i.

$$p_i \subseteq k_i \tag{1}$$

Consider the scenario in Program 1 where user $Alice$ purchases a time-limited subscription on a streaming service to watch a `movie`. After the subscription ends, the security policy changes, however Alice still attempts to watch `movie`.

One could argue that this program should be considered insecure because $Alice$ watches `movie` when she no longer has a subscription. Here, the release of knowledge is considered transient and it should satisfy the active security policy at every program location. Hence, despite being perfect recall, $Alice$ should not be able to watch `movie` at a time this is disallowed by the policy (line 4). In fact, equation (1) holds in line 2 since $Alice$ watches the movie and the policy allows her to watch `movie`, i.e., hence $p_2 \subseteq k_2$ since $\{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. However, in line 4 we have that $p_4 \not\subseteq k_4$ since $All \not\subseteq \{\texttt{movie}\}$, where All denotes the set of all possible movies. Hence, the program is correctly rejected by the security condition (1).

[1] In this notation, knowledge corresponds to uncertainty, hence the bigger the set, the smaller the knowledge (see Section 4).

```
1 Movie → Alice
2 Alice.watch(movie)
3 Movie ↛ Alice
4 Alice.watch(movie)
```

Program 1

```
1 Movie → Alice
2 Alice.watch(movie)
3 Movie ↛ Alice
4 Alice.watch("NoSubscription!")
```

Program 2

```
1 Alice → Eve
2 Bob → Eve
3 output_Eve((Alice.salary + Bob.salary) / 2)
4 Alice ↛ Eve
5 output_Eve(Bob.salary)
```

Program 3

Consider now Program 2, a variation of Program 1, displaying the message `NoSubscription!` after the second policy change. This program is also rejected by condition (1) since $p_4 \not\subseteq k_4$, i.e., $All \not\subseteq \{\texttt{movie}\}$, even though the `NoSubscription!` message does not leak anything.

To address this case, Askarov and Chong [8] identify the power of perfect recall attacker as a key issue and present a security condition that accounts for weaker attackers. Their security condition requires that the attacker's change in knowledge should be allowed by the active policy, thus at any program location $i+1$ the attacker's knowledge k_{i+1} should be smaller than the attacker's prior knowledge and the policy knowledge at location i:

$$p_i \cap k_i \subseteq k_{i+1} \tag{2}$$

Condition (2) now accepts Program 2 as secure since $p_3 \cap k_3 \subseteq k_4$, i.e., $All \cap \{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. However, surprisingly condition (2) also accepts Program 1 since $p_3 \cap k_3 \subseteq k_4$, i.e., $All \cap \{\texttt{movie}\} \subseteq \{\texttt{movie}\}$. The root of the issue here is that perfect recall attacker is too powerful and can remember any observations made in the past, e.g., in line 2. To overcome this issue, Askarov and Chong only consider condition (2) for weaker attackers with bounded memory. For example, a bounded memory attacker that remembers only the last observation would now reject Program 1 since the second observation of `movie` in line 4 reveals new information to a bounded memory attacker. In fact, now $p_3 \cap k_3 \subseteq k_4$ since $All \cap All \not\subseteq \{\texttt{movie}\}$. Similarly, Program 2 is secure since a bounded attacker learns nothing about `movie` by observing the message `NoSubscription!`, namely $p_3 \cap k_3 \subseteq k_4$ since $All \cap All \subseteq All$. Condition (1) treats these programs similarly for a bounded memory attacker.

A key question arises at this point: *How does a policy designer choose the right attacker model, and hence security condition, for their setting*? While in principle there may always exist a bounded memory attacker that accommodates specific use cases as above, it is unclear what such attacker model should be. Askarov and Chong answer this question by requiring that condition (2) holds *for all attackers*, including perfect recall and bounded memory attackers. This is important because it enables compositional reasoning and facilitates enforcement by a security type system, however, security for all attackers can be too restrictive in settings where, e.g, only the perfect recall or a bounded memory attacker is realistic. In fact, Broberg et al. [11] discuss use cases where the same program can be considered either secure or insecure under different attacker models.

More importantly, condition (2) permits any policy changes although these changes may contradict the assumptions about the attacker. Consider Program 3 handling information about users' salaries. Initially, both $Alice$ and Bob allow Eve to learn their salaries, however, the program displays only the average salary to Eve. Then $Alice$ decides that her salary should no longer be visible to Eve and the program displays Bob's salary.

Let As and Bs be the salary of $Alice$ and Bob, respectively. Under a perfect recall attacker, Program 3 satisfies condition (2). Indeed Eve can combine the average salary (line 2) and Bob's salary to learn $Alice$'s salary, hence $k_5 = \{(As, Bs)\}$. Therefore, $p_4 \cap k_4 \subseteq k_5$ since $\{All \times Bs\} \cap \{(a,b) \mid (a+b)/2 = (As+Bs)/2\} = \{(As, Bs)\} \subseteq \{(As, Bs)\}$. The program is also accepted for a bounded memory attacker that remembers only the last output.

Under the perfect recall attacker, we argue that Program 3 should not be accepted. The mere definition of perfect recall assumes that the attacker remembers any observations and can use these observations to infer information about the salaries of $Alice$ and Bob. The real problem lies in the change of the policy in line 4. Because Eve's knowledge in line 3 reveals some information about $Alice$'s salary (and Eve has perfect recall) the policy change in line 4 is *inconsistent*, trying to revoke access to a resource, i.e., $Alice$'s salary, that Eve has already some information about. Hence, the policy change in line 4 should be disallowed in the case of a perfect recall attacker. A similar argument applies to Program 1 and Program 2 under the perfect recall attacker. The reader may find this surprising, especially for Program 2, but, again, the attacker has perfect recall, hence they remember the observation of `movie` in line 2. Therefore, restricting access to the attacker to some information they already have is meaningless and should be prevented by the security condition. In fact, by considering the intersection of the knowledge and policy ($p_i \cap k_i$), condition (2) effectively enforces condition (1) under a different policy $p = p_i \cap k_i$. This leads us to proposing a new (class of) conditions which is parameterized by an attacker A:

$$p_i^A \subseteq k_i^A \tag{3}$$

In contrast to condition 1, condition 3 makes the role of the attacker explicit in the definitions of knowledge and policy, as well as considers the consistency of a security policy. We instantiate the security condition (3) to characterize three attacker models: *perfect recall*, *bounded memory*, and *forgetful*. For the perfect recall attacker, our security condition corresponds to equation (1), ensuring that policy changes are consistent at any program location i. For a bounded memory attacker, the condition captures a weaker attacker which remembers observations up to a predefined bound m, extending the weaker attackers of Askarov and Chong [8] with policy consistency checks. Finally, the forgetful attacker captures scenarios in which the release of knowledge is transient and limited to the event of a policy change, thus ensuring that the attacker forgets (or resets) their knowledge whenever there is a

policy change. This attacker model allows: (1) Reject Program 1 since *Alice* attempts to (re-)watch the movies at a time this is prevented by the active policy; (2) Accept Program 2 (Program 3) since *Alice* (*Eve*) does not learn any information about `movie` (*Alice*'s `salary`) at any time this is prevented by the active policy.

3. Language Design

We present a simple imperative language with extended commands for policy change and outputs. We assume that outputs are performed on channels associated with security labels $X, Y, \ell \in \mathcal{L}$.

Syntax Figure 1 presents the syntax of our language. Expression e consists of program variables x, values v, and binary operations $\oplus$. For simplicity, we restrict values to only integers n. Most of the commands are standard with the exception of `output` and `setPolicy`. Output command $\texttt{output}_\ell(e)$ evaluates expression e to some value v and then outputs v on channel ℓ. Command $\texttt{setPolicy}(p)$ sets the current security policy to p.

Semantics Figure 2 presents the operational semantics of the language. A configuration is a tuple $\langle c, \sigma, p\rangle$ consisting of a command c, a store σ mapping variables to values (i.e., $\sigma = Vars \to Val$), and a policy p that represents the current active security policy. We use judgments of the form $\langle c, \sigma, p\rangle \xrightarrow{\alpha} \langle c', \sigma', p'\rangle$ to denote that configuration $\langle c, \sigma, p\rangle$ can take a single step to configuration $\langle c', \sigma', p'\rangle$ and optionally emit an event $\alpha \in Ev$.

Events Ev include $o(v, \ell)$ to denote the output of value v on channel ℓ, $np(p')$ to denote the activation of new policy p', or ϵ to indicate that no event was emitted.

We write $\sigma(e) = v$ to indicate that expression e evaluates to value v in store σ. We write $\sigma[x \mapsto v]$ to denote a new store that maps variable x to value v and otherwise behaves the same as σ. Most of the semantic rules are standard. Command $\texttt{setPolicy}(p')$ modifies a configuration to activate policy p' and emits the new policy event $np(p')$. Output command $\texttt{output}_\ell(e)$ evaluates e to value v, and emits the event $o(v, \ell)$.

A trace $t \in Ev^*$ is a (possibly empty) sequence of events. We write $\mid t \mid$ for the length of trace t and $t_1.t_2$ for concatenation of traces t_1 and t_2. We define projection of an event α to channel ℓ, written $\alpha\lfloor_\ell$, as: $\alpha\lfloor_\ell = \alpha$ if $\alpha = o(v, \ell)$, otherwise $\alpha\lfloor_\ell = \epsilon$. We lift projection to traces as: $(\alpha.t')\lfloor_\ell = \alpha\lfloor_\ell \,.t'\lfloor_\ell$ if $t = \alpha.t'$, otherwise $t\lfloor_\ell = \epsilon$.

We write $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow} \langle c', \sigma', p'\rangle$ if $\langle c, \sigma, p\rangle$ takes *one or more* steps to reach configuration $\langle c', \sigma', p'\rangle$ while producing the trace t. We write $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow}_n \langle c', \sigma', p'\rangle$ to denote n execution steps and omit the final configuration whenever it is irrelevant, as in $\langle c, \sigma, p\rangle \overset{t}{\Rightarrow}$. An execution point i denotes a configuration (c_i, σ_i, p_i) such that $\langle c_0, \sigma_0, p_0\rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i\rangle$.

4. Security Framework

In this section, we present knowledge-based and attacker-centric security conditions for a range of relevant attackers and explore their differences. As discussed informally in Section 2, the security condition has the form $p_i^A \subseteq k_i^A$, meaning that in every step of the program's execution, the active security policy should be the upper bound of the attacker's knowledge. We instantiate this condition for different attacker models to consider security-relevant events such as policy changes and attacker-visible program outputs.

4.1. Security Polices

We consider a multi-user setting where a program c handles data on behalf of different users which are identified by security labels $\ell \in \mathcal{L}$. For simplicity, we assume that the users' data is read upfront and resides in the initial store of the computation. Section 4.4 presents an extension to programs with arbitrary inputs.

A security policy p is a list of flows of the form $\ell_1 \to \ell_2$ and is used to define what an observer at a specific security label ℓ_2 is allowed to learn about the initial values with label ℓ_1. We assign security labels to the variables and use function Γ which is a mapping from variables to security labels (i.e., $Vars \mapsto \ell$). For simplicity, the name of a program variable (e.g., `x`) will match its security label (e.g., X), and we write $X \to A$ to denote that an observer at channel with label A can learn values with label X (i.e., X can flow to A). Throughout this paper, we use p_{init} as a predefined initial policy from which all programs start their execution. It is a simple reflexive policy that only allows a security label to flow to a corresponding channel with the same label (i.e., $X \to X$). We use $X \to A$ to add new non-reflexive flows to the list of allowed policies, and $X \not\to A$ to revert (disallow) such a flow.

A security policy induces an equivalence relation over stores. Intuitively, for the flow $X \to A$, two stores are related to each other by an equivalence relation for an observer on channel A if they have identical values for variables with security label X. Formally:

$$\sigma \equiv_A^p \sigma' \quad \text{iff} \quad \forall x \in Var:\ \Gamma(x) = X \ and\ X \to A \in p.\ \sigma(x) = \sigma'(x)$$

We write $[\sigma]_A^p$ for the set of stores in the same equivalence class as σ with respect to a policy p and an observer A. If $\sigma \equiv_A \sigma'$ (i.e., $\sigma' \in [\sigma]_A^p$), then an observer on channel A cannot distinguish between stores σ and σ'. Henceforth, we call such an observer the attacker and fix its label to A. Observe that more fine-grained policies can be defined in the expected manner by refining the definition of $[\sigma]_A^p$, e.g., by mapping each label to an equivalence relation on program stores as defined by the policy p [10], [21]. We discuss fine-grained policies in Section 6.

In line with existing work on dynamic policies [8], [10], [21], we assume that policy changes are not observable externally, e.g., to an attacker A. In our multi-user setting, policy changes result from internal events of the underlying system itself, e.g., restricting access to a service, and these operations are typically carried out by the system administrator. This assumption applies to real-world scenarios where a user does not directly control their access rights, while the policies governing these access rights are introduced to the system by an administrator. Nevertheless, our framework can be easily extended to accommodate observable policy changes by considering such events similar to program outputs.

$$
\begin{array}{llll}
\textit{Values} & v & ::= & n \\
\textit{Expressions} & e & ::= & v \mid x \mid e_1 \oplus e_2 \\
\textit{Commands} & c & ::= & \texttt{skip} \mid x := e \mid c_1; c_2 \mid \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2 \\
 & & & \mid \texttt{while } e \texttt{ do } c \mid \texttt{output}_{\ell}(e) \mid \texttt{setPolicy}(p)
\end{array}
$$

Figure 1: Language Syntax

```
1 setPolicy(X → A);
2 output_A(1);
3 setPolicy(X ↛ A);
4 output_A(x);
```

Program 4

$$
\textsc{Skip}\ \frac{}{\langle \texttt{skip}; c, \sigma, p\rangle \xrightarrow{\epsilon} \langle c, \sigma, p\rangle} \qquad
\textsc{Assign}\ \frac{\sigma(e) = v}{\langle x := e, \sigma, p\rangle \xrightarrow{\epsilon} \langle \texttt{skip}, \sigma[x \mapsto v], p\rangle} \qquad
\textsc{Seq}\ \frac{\langle c_1, \sigma, p\rangle \xrightarrow{\alpha} \langle c_1', \sigma', p'\rangle}{\langle c_1; c_2, \sigma, p\rangle \xrightarrow{\alpha} \langle c_1'; c_2, \sigma', p'\rangle}
$$

$$
\textsc{If-Else-T}\ \frac{\sigma(e) \neq 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma, p\rangle \xrightarrow{\epsilon} \langle c_1, \sigma, p\rangle} \qquad
\textsc{If-Else-F}\ \frac{\sigma(e) = 0}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \sigma, p\rangle \xrightarrow{\epsilon} \langle c_2, \sigma, p\rangle}
$$

$$
\textsc{While}\ \frac{}{\langle \texttt{while } e \texttt{ do } c, \sigma, p\rangle \xrightarrow{\epsilon} \langle \texttt{if } e \texttt{ then } (\texttt{while } e \texttt{ do } c) \texttt{ else skip}, \sigma, p\rangle}
$$

$$
\textsc{Output}\ \frac{\sigma(e) = v}{\langle \texttt{output}_{\ell}(e), \sigma, p\rangle \xrightarrow{o(v,\ell)} \langle \texttt{skip}, \sigma, p\rangle} \qquad
\textsc{Set-Policy}\ \frac{}{\langle \texttt{setPolicy}(p'), \sigma, p\rangle \xrightarrow{np(p')} \langle \texttt{skip}, \sigma, p'\rangle}
$$

Figure 2: Semantics

4.2. Security Conditions and Attacker Models

Our main focus is on the confidentiality of data, hence we consider a (logically omniscient) passive attacker that knows the program's source code and wants to deduce sensitive information about the initial store values. Our goal is to identify security conditions for dynamic policies by investigating the relationship between the attacker's knowledge and the policy knowledge for a range of attackers.

We present our knowledge-based security conditions for perfect recall and forgetful attackers. For space reasons, we refer the reader to Appendix D for similar results on bounded memory attackers.

4.2.1. Perfect Recall Attacker. We model this attacker's knowledge of the initial store σ as a set k, which includes all of the possible stores that can produce the same observable trace. We assume the attacker with security level A is passively observing all outputs on channel with label A. When a command such as $\texttt{output}_A(v)$ executes, the attacker sees this output and learns the value v. We define attacker's knowledge as:

Definition 1 (Perfect Recall Attacker Knowledge at Point i)**.** *Given program c with initial store σ, and initial policy p_{init}, which produces trace t after i execution steps, i.e., $\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i$, the knowledge of a perfect recall attacker that observes program outputs on channel A is defined as:*

$$
k_i(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle \overset{t'}{\Rightarrow}_j \ \wedge\ t\downarrow_A = t'\downarrow_A\}
$$

Intuitively, $k_i(c, \sigma, p_{init}, A)$ is the set of initial stores that the attacker at channel A believes are possible when observing the trace $t\downarrow_A$ at execution point i. Thus, the larger the knowledge set, the less certain the attacker is of the actual values in σ.

The *Perfect Recall* attacker has unlimited memory and can remember all outputs on channel A. This attacker is the most powerful attacker, because once they observe an output value they will never forget it, hence, a policy can no longer restrict the knowledge resulting from the attacker's past observations. Arguably, in presence of such an attacker, programs like 3 should be rejected, because, as mentioned earlier, we cannot make the attacker forget what they already know. Therefore, it is not reasonable to issue a new policy that prevents this attacker from learning information which they already know.

With this intuition in mind, we need to ensure that any policy update is consistent with the current knowledge of the perfect recall attacker:

Definition 2 (Policy Consistency)**.** *For all execution points i, and security policies p_i such that*

$$
\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_{i-1} \langle \texttt{setPolicy}(p_i); c_i, \sigma_{i-1}, p_{i-1}\rangle \xrightarrow{np(p_i)} \langle c_i, \sigma_{i-1}, p_i\rangle
$$

we say policy p_i is consistent with the current attacker knowledge $k_{i-1}(c, \sigma, p_{init}, A)$ if $[\sigma]_A^{p_i} \subseteq k_{i-1}(c, \sigma, p_{init}, A)$.

A security policy induces an equivalence relation over all possible stores, and $[\sigma]_A^{p_i}$ is a set of stores in the same equivalence class as initial store σ. Since by Definition 1 attacker knowledge $k_{i-1}(c, \sigma, p_{init}, A)$ is also a set of initial stores, it is straightforward to check $[\sigma]_A^{p_i} \subseteq k_{i-1}(c, \sigma, p_{init}, A)$.

For example in Program 4, the new policy $X \not\rightarrow A$ in line 3 is consistent, because the attacker does not learn anything about x by observing the output in line 2, hence the new policy that disallows learning x is consistent. However, under this new policy, Program 4 should be rejected, because the output of x in line 4 happens at a time when the policy does not allow it. We follow this intuition to define security.

Definition 3 (Observation Security)**.** *For all execution points i such that*

$$
\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_{i-1} \langle \texttt{output}_A(e); c_i, \sigma_{i-1}, p_{i-1}\rangle \xrightarrow{\alpha} \langle c_i, \sigma_i, p_i\rangle
$$

program c is secure if $[\sigma]_A^{p_i} \subseteq k_i(c, \sigma, p_{init}, A)$.

This definition ensures that, whenever an output on channel A happens, the attacker's knowledge at that point

is allowed by the current policy. In other words, the policy places an upper bound on the attacker's knowledge[2], and if the knowledge does not exceeds that limit, the program is secure.

Definition 4 (Security Condition for Perfect Recall)**.** *A program c is secure under the perfect recall attacker if Definitions 2 and 3 hold.*

We deliberately separate Definitions 2 and 3 to distinguish between policy consistency checks and security checks. A failed policy consistency check means that there is a mismatch between the attacker's power and the new policy. Therefore, a policy inconsistency can be repaired with a new policy that takes into account the attacker's knowledge. On the other hand, a failure of observation security (Definition 3) cannot be repaired and it implies that the program is insecure.

4.2.2. Forgetful Attacker. We now consider an attacker that resets its knowledge after a policy change. This is specially useful for real-world applications where the release of information is not permanent and should be consistent with the active security policy at the time of the release. Program 1 in Section 2 is an example of the usefulness of this attacker model. Intuitively, a policy change at an execution point i means that, from the point i onward, the new policy should govern the release of information and any past knowledge should be ignored. The term "forgetful attacker" may not exactly reflect a real world attacker that suddenly forgets everything after a policy change. It is an artifact of modeling scenarios in which a policy change enforces a new condition on information release, independently of what an attacker may already know as result of past observations. For example, an employee may have accessed a company's information (and even stored it externally), however, no access to the same information should be allowed when they leave the company. This setting requires ignoring the attacker's knowledge prior to the policy change, essentially resulting in a forgetful attacker. We first discuss some examples illuminating the subtleties of forgetful attackers and then present our security condition.

Consider Program 5 as an example. When the execution reaches the `setPolicy` command in line 5, there are two possible traces that the attacker could have observed: $t_1 = y.1$ and $t_2 = y$, both leaking the value of `y`. One may think that trace t_1 leaks the sign of x but this is not the case. Because the attacker's knowledge is derived through observations and the policy changes are not observable, the attacker cannot tell which `if` statement has produced the output 1 of trace t_1. Therefore, these traces reveal nothing about `x`, and at the time of policy change, the attacker only knows `y` as stipulated by the policy in line 1. The new policy now prevents the attacker from learning `y` again, hence no outputs after the policy change should leak `y`. In fact, all executions after the policy change will output the values 1 and 2. Note that even though the output 1 on line 7 happens after the policy change, it still cannot leak the sign of `x`, therefore, Program 5 is secure.

Program 6 is similar except that it outputs the value of `y` at lines 4 and 7. In this case, if the execution did not take the first `if` statement, the attacker observes the trace $t = 1$ which leaks nothing about `x` or `y`, thus at the time of policy change the attacker forgets nothing. However, when `y` is outputted in line 7, the attacker learns this value and because this is not allowed by the policy, the program is insecure.

Programs can leak through the progress of computation e.g., when the number of outputs depends on information that is disallowed by the policy [22]. It is our intuition that forgetful attackers should not forget these progress leaks. Once the length of a trace, i.e., the number of outputs, leaks some information, any extension of that trace will leak the same information again, thus it is not reasonable for a forgetful attacker to forget progress leaks. This can be captured by making forgetful attackers remember the number of observed outputs, including the ones that happened before a policy change. This approach captures progress leaks even when they manifest after a policy change. This is similar to the idea of counting attackers presented in van Deft et al. [10].

Program 7 illustrates the effect of progress leaks on the attacker's knowledge. When the execution reaches the policy change at line 7, the attacker could have observed trace $t_1 = 1$ or $t_2 = 1.1$. Since the policy change event is not observable by the attacker and any program execution can yield at least 2 outputs (e.g., trace 1.1), the attacker learns nothing about `x`. Later, after the new policy at line 7 becomes active, the output at line 8 occurs. One of the traces that the attacker could have observed at this point is $t_2 = 1.1.1$. This trace leaks that the first `if` statement must have been executed and `x > 0`, which violates the active policy. Therefore, the number of outputs leaks the sign of x, which happens after the policy change, hence the program should be flagged as insecure.

With these intuitions in mind, we proceed to define the knowledge of forgetful attacker. We call the trace between two policy changes an epoch and use the policy events ($np(p)$) to partition the trace into multiple epochs. At each step, the observable events of the last epoch, as well as the number of events in previous epochs can affect the forgetful attacker's knowledge. To be able to separate the last epoch from the whole trace, we define the following auxiliary functions: *splitPolicy(t)* takes a trace t and returns a tuple containing all events before and after the last new policy ($np(p)$) event; *split(t, n)* takes a trace t and a number n, and returns a tuple containing the first n events and the reminder of events in the trace.

Definition 5. *Given a trace t such that $t = \alpha_1 ... \alpha_i ... \alpha_k$, splitPolicy(t) =*

$$\begin{cases} (\epsilon, t) & \text{if } \alpha_r \neq np(p)\ r = 1...k \\ (\alpha_1 ... \alpha_i\ ,\ \alpha_{i+1} ... \alpha_k) & \text{if } \alpha_i = np(p) \wedge \\ & \alpha_r \neq np(p)\ r = i+1...k \\ (t, \epsilon) & \text{if } \alpha_k = np(p) \end{cases}$$

Definition 6. *Given a trace t such that $t = \alpha_1 ... \alpha_i . \alpha_{i+1} ... \alpha_k$,*

$$\textit{split(t, n)} = \begin{cases} (t, \epsilon) & \text{if } k \leq n \\ (\alpha_1 ... \alpha_i\ ,\ \alpha_{i+1} ... \alpha_k) & \text{if } i = n \end{cases}$$

Using these auxiliary functions, we can proceed to define the forgetful attacker's knowledge as:

[2]Observe that the attacker's knowledge, in contrast to the policy knowledge, is precise, and it is not an upper bound.

```
1 setPolicy(X → A, Y → A);
2 output_A(y);
3 if (x > 0) then
4     output_A(1);
5 setPolicy(X ↛ A, Y ↛ A);
6 if (x <= 0) then
7     output_A(1);
8 output_A( 2 );
```

Program 5

```
1 setPolicy(X → A, Y → A);
2 output_A(1);
3 if (x > 0) then
4     output_A(y);
5 setPolicy(X ↛ A, Y ↛ A);
6 if (x <= 0) then
7     output_A(y);
8 output_A(2);
```

Program 6

```
1 setPolicy(X → A);
2 if (x > 0) then
3     output_A(1);
4     output_A(1);
5 else
6     output_A(1);
7 setPolicy(X ↛ A);
8 output_A(1);
```

Program 7

Definition 7 (Forgetful Attacker Knowledge at Point i)**.** *Program c with initial store σ and initial policy p_{init} produces trace t after i execution steps, i.e., $\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i$. Let $(t_1, t_2) = splitPolicy(t)$, we define for the knowledge of a forgetful attacker that observes the program outputs on channel A as:*

$$k_i^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle \overset{t''}{\Rightarrow}_j \\ \wedge\ (t_1'', t_2'') = split(t''\lfloor_A, \mid t_1\lfloor_A\mid) \\ \wedge\ t_2'' = t_2\lfloor_A\}$$

Intuitively, for each execution point i, we identify the traces before (t_1) and after (t_2) the last policy change. The goal is to forget the knowledge induced by attacker's trace $t_1\lfloor_A$ and compute the knowledge induced by $t_2\lfloor_A$. We achieve this by considering any initial states that produce the same *number* of outputs as $\mid t_1\lfloor_A\mid$ and the same outputs as $t_2\lfloor_A$. Note that this condition (as all our conditions) is progress sensitive and accounts for progress leaks. We remark that progress leaks are never forgotten once they are revealed at some execution point.

We can now use the definition of knowledge from Definition 7 to obtain the security condition for forgetful attackers.

Definition 8 (Security condition for Forgetful Attacker)**.** *For all execution points i such that*

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_{i-1} \langle \mathbf{output}_A(e); c_i, \sigma_{i-1}, p_{i-1}\rangle \\ \overset{\alpha}{\rightarrow} \langle c_i, \sigma_i, p_i\rangle$$

program c is secure if $[\sigma]_A^{p_i} \subseteq k_i^{frg}(c, \sigma, p_{init}, A)$.

Appendix B exercises the definition for Programs 5–7 to investigate their security.

We can now show that if a program is secure against the perfect recall attacker, it is also secure against less powerful attackers such as bounded memory attackers and forgetful attackers. Here, we present a theorem and prove this claim for the forgetful attackers.

Theorem 1. *Given a program c, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against forgetful attacker A_{frg}. Formally:*

$$[\sigma]_A^{p_i} \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies \\ [\sigma]_A^{p_i} \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

Appendix D contains the proof of Theorem 1 and the corresponding theorem for bounded memory attackers.

4.3. Repairing Inconsistent Policies

An inconsistent policy means that the policy is incompatible with the current knowledge of the attacker. One approach is to always reject the programs with inconsistent polices, because $p_i \not\sqsubseteq k_i$. Alternatively, we can suggest the user a new policy that is consistent with the attacker's current knowledge. Generally, a policy change is inconsistent if it restricts access to information that has already been learned by the attacker. Our goal is to relax these restrictions and add what has been learned by the attacker to the new policy to achieve a consistent policy. The intersection of the new policy and the attacker's knowledge ($p_i \cap k_i$) is a good candidate because it includes all of the new flows introduced by the new policy, and uses the knowledge of the attacker to relax the restrictions of the new policy. Intuitively, the consistent policy $p_i \cap k_i$ corresponds to the most adequate policy that meets the intention of the policy change p_i while being in line with the attacker's current knowledge k_i.

Definition 9 (Consistent Policy Repair)**.** *For all execution points i such that*

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_{i-1} \langle setPolicy(p_i); c_i, \sigma_{i-1}, p_{i-1}\rangle \\ \xrightarrow{np(p_i')} \langle c_i, \sigma_{i-1}, p_i'\rangle$$

the repaired policy p_i' is induced by $[\sigma]_A^{p_i} \cap k_{i-1}(c, \sigma, p_{init}, A)$.

While previous approaches [8], [10], [21] use intersection as part of the security conditions, here we emphasize that it corresponds to a new consistent policy, thus making it explicit for the user that security of the program is checked against a different (repaired) policy.

4.4. Generalization to Programs with Inputs

In a framework of dynamic policies, it is natural to model new information arriving into the system via input channels. We show how our framework can accommodate programs with inputs with minimal changes. We extend the syntax of the language with an input command $\texttt{input}_\ell(\texttt{x})$ which reads a value from the input channel with label ℓ and assigns it to variable x. Clark and Hunt [23] have shown that for deterministic interactive systems, streams are sufficient to model arbitrary interactive input strategies. An input stream is an infinite sequence of values representing the pending inputs on a channel. We assume there is one input channel for each security level ℓ and an input environment ω mapping labels to input streams. We extend the configurations $\langle c, \sigma, p, \omega\rangle$ with the

input environment ω and the evaluation steps as expected. We write $v : vs$ for a input stream with the first element v and remaining elements vs, and $\omega[\ell \mapsto vs]$ for the input environment that maps input stream with label ℓ to vs and otherwise behaves the same as ω. The semantics of input command is defined as:

$$\textsc{Input}\ \frac{\omega(\ell) = v : vs \quad \omega' = \omega[\ell \mapsto vs]}{\langle \texttt{input}_\ell(\texttt{x}), \sigma, p, \omega\rangle \xrightarrow{i(v,\ell)} \langle \texttt{skip}, \sigma[x \mapsto v], p, \omega'\rangle}$$

This command updates the store σ with value v for variable x, and continues with vs as the reminder of the input stream of label ℓ, while emitting the input event $i(v, \ell)$. Events and traces are extended with input events in the expected manner.

We can now define security policies as equivalence relations over input environments. Two input environments are equivalent for a policy p and an attacker on channel A, i.e., $\omega \equiv_p^A \omega' \quad$ iff $\forall \ell \to A \in p\ .\ \omega(\ell) = \omega'(\ell)$. We write $[\omega]_A^p$ for the equivalence class of ω with respect to the policy p and attacker A. With these definitions at hand, we can easily redefine the attacker knowledge over input environments and use the same security conditions adapted with the new definitions of attacker knowledge and security policies. This extension are straightforward and we omit them here in the interest of space.

5. Facets of Dynamic Policies

In this section we revisit the facets of dynamic policies, introduced by Broberg et al. [11], and discuss them in our framework from an attacker-centric perspective. Our goal is to show how these facets can be accommodated in our framework, illuminating on the different types of flows. We have modified and adapted the use cases to fit our language model with explicit outputs.

Time-transitive flows A flow is time-transitive if it moves information from level X to level Z via a third level Y, while a direct flow from X to Z is never allowed by the policy. Program 8 illustrates such a flow. It reveals information about $Patient$ to $DrPhil$ who joined the hospital after $Patient$ had left.

According to Broberg et al. [11] time-transitive flows should be considered insecure in scenarios where a data flow such as $Patient \to Hospital$ is only allowed temporary for as long as $Patient$ is under treatment in the hospital. Our framework can identify the insecurity of such scenarios; when $\texttt{output}_{DrPhil}\texttt{(drPhil)}$ happens it indirectly reveals the value of `patientData` which is not allowed by the active policy. A similar argument applies to bounded memory and forgetful attackers. The main reason for rejecting these flows is that the observer $DrPhil$ did not see the data at the time he was allowed to and the actual flow has happened at a time when patient had already left the hospital.

Broberg et al. [11] also interpret time-transitive flows as secure by considering the assignment in line 2 as a *permanent declassification*. Using permanent declassification means changing the label of `patientData` to $Hospital$ permanently, however, this means that the policy $Patient \not\to Hospital$ becomes irrelevant, since `patientData` no longer has the label of $Patient$ and hence it not affected by its policy. In our framework, this interpretation amounts to a program that allows flows from $Hospital$ to $DrPhil$ and subsequently outputs the data to $DrPhil$.

Replaying flows model scenarios in which when a piece of information is released, it can be released again, regardless of the active policy. Program 9 illustrates such a flow. When `creditcard` is written to a log file, it should be available until the log is cleared.

Replaying flows should be considered secure when the release of information is *permanent* [11]. Permanent release of information means that an observer can access any information they had learned before, irregardless of the active policy. For example in Program 9, if the output in line 2 permanently releases the `creditcard` information to Log, the observer at level Log can always access it later. This definition can be captured by our framework, by adapting the Definition 9 for inconsistent policies. This means that the new policy will be the intersection of the knowledge of the attacker k and new the policy p, and since the output in line 2 adds `creditcard` to the knowledge of the attacker, $k \cap p$ will always include that knowledge, hence permanently releasing it.

However, considering information as permanently released is not always the natural choice in every situation; for example in Program 1, Alice should not have access to the `movie` after her subscription ended. Forgetful attackers in our framework are good candidates for dealing with such scenarios where we want to ignore the effects of the earlier release and accept or reject programs only based on the current active policy. This intuition is similar to the *insecure* time-limited subscription example of Broberg et al. [11].

Direct Release means that information is considered released as soon as the current policy permits the flow. Program 10 illustrates such a flow where `salary` is not printed to the screen when the flow is allowed, but it is printed when the flow is no longer permitted.

Broberg et al. [11] argue that these flows are insecure when the attacker can only observe information that is actively provided (through for example an output channel). In Program 10 nothing about `salary` has been printed to the screen, hence it makes sense to assume that an observer does not know this information. Our framework follows this intuition and rejects this flow under all attacker types and policy checks; because an attacker learns nothing from output of line 2 and the output on line 4 always violates the active policy.

However, this type of flow can be considered secure if we model attackers as constantly observing, directly in the memory, all the information which they are allowed to learn. We can model such a behavior by outputting all the variables with label $Salary$ as soon as the policy $Salary \to Screen$ is activated. However, doing so effectively changes the nature of this flow to a replaying flow, and as it was discussed earlier, replaying flows can be secure only if we consider permanent release of information.

Whitelisting flows A flow is allowed whenever there is some part of the policy that allows for it. Program 3 in Section 2 is an example of whitelisting flows, where the observer Eve can use her knowledge of `(Alice.salary + Bob.salary) / 2` and `Bob.salary` to learn the salary of Alice. For perfect recall attackers,

```
1 setPolicy(Patient → Hospital, Hospital ↛ DrPhil);
2 hospital := patientData;
3 setPolicy(Patient ↛ Hospital, Hospital → DrPhil);
4 drPhil := hospital;
5 output_DrPhil(drPhil);
```

Program 8

```
1 setPolicy(Creditcard → Log);
2 output_Log(creditcard);
3 setPolicy(Creditcard ↛ Log);
4 output_Log(creditcard);
```

Program 9

```
1 setPolicy(Salary → Screen);
2 output_Screen(0);
3 setPolicy(Salary ↛ Screen);
4 output_Screen(salary);
```

Program 10

```
1 setPolicy(Secret → Public, Key → Public);
2 output_Public(secret XOR key);
3 setPolicy(Secret ↛ Public, Key → Public);
4 output_Public(key);
```

Program 11

our framework rejects this class of programs because they have inconsistent policy changes. This is inline with the insecure example presented by Broberg et al. [11] which argue that information belonging to two entities $Alice$ and Bob (in this case the average of their salaries) should be available only when both of them allow it.

Broberg et al. [11] presents Program 11 as a secure example for whitelisting flows, where first the encrypted value (`secret XOR key`) is released and then later the `key`. Broberg et al. [11] argue for the security of this example on the grounds that `key` is an encryption key, and "with the release of this key an observer learns the secret information that was earlier released encrypted under that key, even though part of the policy does not allow the secret to be released". This intuition is inline with inconsistent policy repair of Definition 9, since we know that the encrypted values have already been outputted and publishing the key releases them as well, we should either explicitly add $Secret \rightarrow Public$ to the policy, or use Definition 9 to update the policy.

6. Verification of Dynamic Policies

This section discusses the precise verification of dynamic policies by symbolic execution and automated theorem proving. Our verification approach operates in two phases: (1) it extracts the dependencies of a source program by means of symbolic execution and (2) it verifies the security conditions for dynamic policies under different attacker models by relying on an SMT solver. We impose some restrictions on the source program to make the analysis in (1) feasible. First, we assume a bounded model of runtime behavior, hence programs always terminate. Second, we assume all inputs from external environments can be read at the beginning. Hence, to support programs with inputs, one can assign fresh variables to each of the elements of a (finite) input stream. The output of phase (1) is a graph capturing dependencies between program inputs and outputs. We refer to existing works for details on symbolic execution [24].

Specifically, we analyze source programs symbolically to extract precise dependencies between program inputs and outputs. Observe that this information is sufficient to reason about security because security policies refer to program inputs and attacker observations are made though program outputs. For each program output, our symbolic analysis stores a path condition $Pc \in PC$ and an output expressions $e \in Exp$ which are defined over the program inputs. The path condition is a predicate that represents the set of initial concrete values that trigger the execution of an output expression e. In particular, any *satisfying assignment*[3] δ of path condition Pc determines a concrete program output as computed by $\delta(e)$.

We represent these dependencies in the form of symbolic output trees (SOT) consisting of: (a) a set of nodes B labeled with output expressions $e \in Exp$; (b) a set of control flow edges $E \subseteq B \times B$; (c) a set of path conditions PC; (d) a mapping from nodes to output expressions $O : B \mapsto Exp$; (e) a mapping from edges to path conditions $L : E \mapsto PC$; and (f) a root node $Start$. We also extend the SOT with a special node End, in order to make terminal states explicit in the construction. Figure 3 illustrates the SOT of Program 12. Node $n3$ indicates that for all initial values of variable y such that $y \leq 0$, the second output of the program is the expression 2.

We define security policies with respect to an attacker at security level A. For a program variable x such that $X \rightarrow A$, we denote its initial value by l (for low) to reflect that variable x can be observed by A, otherwise we denote it by h (for high). We lift this notation to tuples of input variables $\overrightarrow{l}$ and $\overrightarrow{h}$ in the expected manner. Moreover, we support fine-grained policies modeled by predicates ϕ over initial values of variables. We define the policy P over $\overrightarrow{l}$, $\overrightarrow{h}$, written as $P(\overrightarrow{l}, \overrightarrow{h})$ by the predicate:

$$\overrightarrow{l} = \overrightarrow{l}' \wedge \phi \tag{4}$$

where $\overrightarrow{l}'$ stands for the renames of low variables, and predicate ϕ represents the leaked (i.e., declassified) expressions which is defined over low and high identifiers, and their renames. The policy predicate $P(\overrightarrow{l}, \overrightarrow{h})$ induces an equivalence relation $[\sigma]_A^P$ over initial stores σ, which corresponds to the policy knowledge (cf. Section 4.1). The relation can be constructed as follows: Let σ and σ' be two program stores over program variables and their renames, respectively, $unprime(S)$ be an operator "undoing" the variable renames over a set S, and $q(\sigma)(\sigma')$ be the evaluation of a predicate q over σ and σ'. Then the equivalence relation $[\sigma]_A^P = unprime(\{\sigma' | P(\overrightarrow{l}, \overrightarrow{h})(\sigma)(\sigma')\})$ defines the policy knowledge induced by the predicate $P(\overrightarrow{l}, \overrightarrow{h})$.

For example, the policy in line 1 of Program 12 means that variable x is low, variable y is high, and expression $y > 0$ is leaked. Following equation (4), we write the policy $P(x, y)$ as $x = x' \wedge (y > 0 = y' > 0)$.

[3] A satisfying assignment is a mapping from the free variables of Pc to values, which makes the predicate Pc evaluate to true.

```
1  setPolicy(X → A, (Y > 0) → A);
2  output_A(x);
3  if (y > 0) then
4      output_A(1);
5  else
6      output_A(2);
7  output_A(3);
```

Program 12

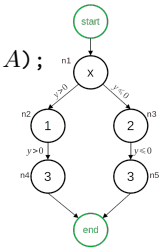

Figure 3: SOT of Program 12

```
1  setPolicy(X → A, Y ↛ A);
2  output_A(x);
3  output_A(1);
4  output_A(1);
5  if (y > 0) then
6      output_A(2);
7  setPolicy(X ↛ A, Y ↛ A);
8  if (y <= 0) then
9      output_A(2);
10 output_A(3);
```

Program 13

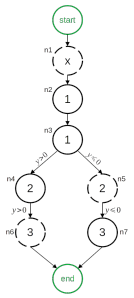

Figure 4: SOT of Program 13

We remark that the policy $P(\overrightarrow{l}, \overrightarrow{h})$ corresponds to a global static policy encoding of a standard declassification policy ϕ [25]. In fact, prior work by Balliu et al. [18] shows how a static policy $P(\overrightarrow{l}, \overrightarrow{h})$ can be verified against an SOT by means of an SMT solver. Definition 10 presents the process of generating such a formula.

Definition 10. *An SOT S is secure wrt. a security policy $P(\overrightarrow{l}, \overrightarrow{h})$ iff the following formula is unsatisfiable:*

$$P(\overrightarrow{l}, \overrightarrow{h}) \wedge \bigvee_{n \in N(S)} \Big(Pc_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Big(\bigwedge_{n' \in N(S)} \neg \Big(Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}') \wedge \overrightarrow{O}_n(\overrightarrow{l}, \overrightarrow{h}) = \overrightarrow{O}_{n'}(\overrightarrow{l}, \overrightarrow{h}') \Big) \Big) \Big)$$

where $\overrightarrow{O}_n$ is the tuple of output expressions along the SOT path to node n, $\overrightarrow{O}_n = \overrightarrow{O}_{n'}$ denotes the component-wise equality of two tuples, and $N(S)$ is the nodes of S.

Definition 10 presents a logical encoding of our security condition $p \subseteq k_i$ of Section 4 for the perfect recall attacker and a static policy p. Specifically, the condition is true if all initial stores that satisfy policy p are contained in the attacker's knowledge set k_i at each program point i. We focus only on program outputs, since non-observable commands do not affect knowledge. The non-satisfiability of the formula above implies that for any initial state $(\overrightarrow{l}, \overrightarrow{h})$ that satisfies the policy $P(\overrightarrow{l}, \overrightarrow{h})$ and reaches some output node n (i.e., $n \in N(S)$ and $Pc_n(\overrightarrow{l}, \overrightarrow{h})$) yielding an output sequence $\overrightarrow{O}_n$, it is impossible to find another state $(\overrightarrow{l}, \overrightarrow{h}')$ that satisfies the policy and reaches some output node n' (i.e., $n \in N(S)$ and $Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}')$) yielding a different output sequence $\overrightarrow{O}_{n'}$. Consequently, for any initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, we have that $(\overrightarrow{l}, \overrightarrow{h}) \in k_i$, thus verifying the security condition. By contrast, if the formula is satisfiable, there exist two stores that satisfy the policy $P(\overrightarrow{l}, \overrightarrow{h})$, but either one store does not reach the output node or the two stores produce different output sequences. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i$, thus violating the security condition.

For example, the SOT of Figure 3 is secure wrt. the above-mentioned policy $P(x, y)$, as witnessed by the following unsatisfiable formula:

$$x = x' \wedge (y > 0 = y' > 0) \wedge \bigvee_{n \in N(S)} \Big(Pc_n(x, y) \wedge \Big(\bigwedge_{n' \in N(S)} \neg \Big(Pc_{n'}(x, y') \wedge \overrightarrow{O}_n(x, y) = \overrightarrow{O}_{n'}(x, y') \Big) \Big) \Big)$$

We revise this condition to verify deterministic programs with dynamic policies for our attacker models. In a dynamic setting, the active policy at each node of S might be different from its parent or children. Therefore, instead of generating a single formula for the whole SOT, we need to generate a formula for every node n corresponding to its policy $P_n(\overrightarrow{l}, \overrightarrow{h})$. To this end, we modify the SOT generation algorithm and enrich each node with an additional attribute to store the active policy at the time of its creation.

6.1. Perfect Recall Attacker

We use Definition 11 to check the security of a program wrt. the perfect recall attacker.

Definition 11. *An SOT S secure iff for all $n \in N(S)$ with active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$, the following formula is unsatisfiable:*

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Big(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \wedge \Big(\bigwedge_{n' \in N(S)} \neg \Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \Big) \Big) \Big)$$

In contrast to Definition 11 here the active policy can be different in each node (as denoted by $Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n})$). For a node n the formula is unsatisfiable only if there is no other node with a satisfiable path condition $Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}')$ that can produce a different output. Unsatisfiability of the formula for a node n means that the program is secure wrt. the active policy at that node. To ensure security for the SOT we repeat this process for all nodes, regenerate the formula at each node and check its satisfiability. If none of the formulas are satisfiable, we can conclude that the SOT S is secure.

We use a similar approach to verify the policy consistency. However, because we do not have specific nodes for policy changes, we mark all of the nodes that appear right after a policy change and only check the policy consistency on those nodes using the following definition:

Definition 12. *Given an SOT S, active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$, and $parent(n)$ which returns the parent of node n, a policy change is consistent iff for all $n \in N(S)$ such that n comes right after a policy change, the following formula is unsatisfiable:*

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n}) \wedge \Big(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}')\Big)\Big)\Bigg)$$

If a node was marked as an output after a policy change, before checking its security using Definition 11, we first use Definition 12 to check the policy consistency. The process is similar to Definition 11, except that here instead of using the path condition and output of node n, we use the path condition and output of its parent ($Pc_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$ and $\overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$, respectively).

Following Definition 2 in Section 4, we check that the attacker knowledge is allowed by the new policy. If node n is marked by the policy change it means that a policy change has happened between n and $parent(n)$, so we use the output of the node before the policy change (parent node) $\overrightarrow{O}_{parent(n)}(\overrightarrow{l_n}, \overrightarrow{h_n})$ and the new policy (policy of the current node) $P_n(\overrightarrow{l}, \overrightarrow{h})$ to generate the formula, and check that the new policy is in line with the observations up to n's parent. The unsatisfiability of this formula means that the policy change between nodes $parent(n)$ and n is consistent.

Figure 4 illustrates the SOT of Program 13. As in the previous example, a node with expression x represents outputting the initial value of x (with $Pc = true$), while $y > 0$ and $y \leq 0$ are path conditions. The nodes following a policy change are shown with dashed lines (nodes $n1$ and $n5$). This program is rejected by the Definition 12, because the policy change between nodes $n3$ and $n5$ is inconsistent. The generated formula for node $n5$ is:

$$\Bigg(Pc_{n3}(\emptyset, \{x, y\}) \wedge \Big(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\emptyset, \{x', y'\}) \wedge \overrightarrow{O}_{n3}(\emptyset, \{x, y\}) = \overrightarrow{O}_{n'}(\emptyset, \{x', y'\})\Big)\Big)\Bigg)$$

The path condition of node $n3$ is $true$ and its output sequence is $O_{n3} = (x, 1, 1)$. The formula is satisfiable if there exists a value for y or x where $Pc_{n3}(y)$ holds and *for all nodes* falsifies either the path conditions or the equality between outputs. The only node on the same level as $n3$ is $n3$ itself, which clearly means that the path condition is also $true$. However, since the output sequences are $(x, 1, 1)$ and $(x', 1, 1)$, it is sufficient to pick any value for x and x' such that $x' \neq x$ to satisfy the following formula. This implies that the policy change at node $n5$ is inconsistent.

$$\Big(true \wedge \neg\Big(true \wedge (x, 1, 1) = (x', 1, 1)\Big)\Big)$$

The following theorems show soundness of Definition 11 and Definition 12 wrt. the security conditions of Definition 3 and Definition 2, respectively.

Theorem 2. *Given a SOT S, if the formula in Definition 11 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition 3.*

Theorem 3. *Given a SOT S, if the formula in Definition 12 is unsatisfiable for all nodes $n \in S$ such that n follows a policy change, then S satisfies Definition 2.*

6.2. Forgetful Attacker

In line with the definitions of forgetful attackers in Section 4, we ignore the actual values of the output expressions occurring before the last policy change. Therefore the output tuple $\overrightarrow{O}_n^{frg}$ for the forgetful attacker replaces all of the outputs that occurred before the last policy change with the constant value `1`. Additionally, while generating the inner conjunction of the formula, we only consider the nodes that have the same number of policy changes as n, using the auxiliary function $sameNP(S, n)$. Definition 13 adapts Definition 11 for forgetful attackers:

Definition 13. *Given an SOT S, policy $P_n(\overrightarrow{l}, \overrightarrow{h})$ at node n, S is secure wrt. forgetful attacker iff for all $n \in N(S)$, the following formula is unsatisfiable:*

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \wedge \Big(\bigwedge_{n' \in sameNP(S,n)} \neg\Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n^{frg}(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^{frg}(\overrightarrow{l_n}, \overrightarrow{h_n}')\Big)\Big)\Bigg)$$

It uses $\overrightarrow{O}_n^{frg}$ to compute output sequences, ignoring the values leaked before the policy change. Additionally, it only generates the formula for the nodes with the same number of policy changes. This is because the only relevant nodes for a forgetful attacker are the ones that are on the same epoch as the current node. Progress leaks are captured by the number of constant values in $\overrightarrow{O}_n^{frg}$ and the actual values leaked in the other epochs are ignored.

To illustrate this process, we revisit Program 7 and its SOT in Figure 5, and check the security for the forgetful attacker using Definition 13. For example, at node $n5$ the generated formula is:

$$\Bigg(Pc_{n5}(\emptyset, x) \wedge \Big(\bigwedge_{n' \in sameNP(S,n)} \neg\Big(Pc_{n'}(\emptyset, x') \wedge \overrightarrow{O}_{n5}^{frg}(\emptyset, x) = \overrightarrow{O}_{n'}^{frg}(\emptyset, x')\Big)\Big)\Bigg)$$

The path condition of node $n5$ is $x > 0$ and its output sequence is $\overrightarrow{O}_{n5}^{frg} = (true, true, 1)$. The formula is satisfiable if there exists a value for x where $Pc_{n5}(x)$ holds, and *for all nodes* it falsifies either the path condition or the equality between outputs. The only node with the same number of policy changes and at the same level as $n5$ is $n5$ itself. This clearly means that the output sequences are equal. Therefore, to satisfy the formula we need a value x' that falsifies the path condition $x' > 0$, which can be any non-positive value. Thus, the formula is satisfiable and the program is insecure.

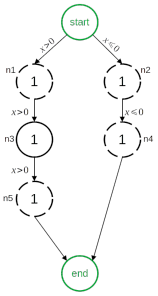

Figure 5: SOT of program 7

Theorem 4 shows the soundness of Definition 13 wrt. the security condition for the forgetful attacker.

Theorem 4. *Given a SOT S, if the formula in Definition 13 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition 3 for the forgetful attacker.*

We refer to Appendix C for proof sketches of the theorems, and Appendix A for the verification algorithm of bounded memory attackers.

6.3. Policy Repair

This section we discuss an approach for generating repair policies. As discussed in Section 4.3, a repair policy should ideally be the intersection between attacker knowledge and the new inconsistent policy. The approach presented here leverages the information provided by the SOT to calculate the attacker's knowledge at node n as a combination of the direct (outputs) and indirect (Pc values) observations made by the attacker. However, this approach does not always result in the intersection as defined by Definition 9 as it sometimes over-approximates the attacker's knowledge. Developing a precise approach for generating consistent policies is left for future work.

When the policy consistency check fails at some node n, we can calculate the knowledge by traversing the SOT from n up to node $start$ and collecting the attacker's explicit observations (i.e., the output expressions) and implicit observation (i.e., Pcs). It is important to note that not all output expressions and Pcs leak information, therefore we should only collect the ones that are *leaking* some information. Here we achieve this by applying a preprocessing step to the SOT, which uses a bounded memory attacker with memory capacity of $m = 1$ to determine the leaked expressions at each level of SOT. The addition of these leaked expressions to the new policy gives a candidate repaired policy.

The intuition behind this approach is that a new policy can only be inconsistent if it is more restrictive than the current policy. Thus a repaired policy should ease some of those restrictions. To this end, we extract already leaked expressions and add them to the inconsistent policy, which gives us the most restrictive version of policy that is consistent with the knowledge. After generating the repaired policy, we should also update the policy field of n's children up to the next policy change. This is because all of the nodes between n and the ones with a new policy are affected by the inconsistent policy.

7. Implementation and Evaluation

We implemented the algorithms presented in Section 6 by extending ENCoVER [26] and creating a prototype dubbed DynCoVer [20]. Like ENCoVER, DynCoVer relies on Symbolic PathFinder (SPF) [27], an extension of Java PathFinder (JPF) [28], to concolically execute programs and extract the symbolic output trees from Java bytecode.

DynCoVer analyzes the program by means of concolic testing and does the following in a loop to explore all execution paths of the program and generate the SOT: it starts with concrete and symbolic values for input variables and executes the program concolically to collect each step's path condition. These conditions are then passed to a constraint solver to generate new inputs that explore different paths. Upon reaching an output statement, the output expression is evaluated in the symbolic state and a new node representing the result of that evaluation is added to the SOT. The path condition that directed the program to this output statement is also saved in the node.

After generating the SOT, DynCoVer traverses the tree using a depth-first search (DFS) strategy, and for each node, depending on the attacker, it generates the formulas described in Section 6. Then, DynCoVer feeds the generated formula to a satisfiability modulo theory (SMT) solver (Z3 in the current implementation). If the SMT solver answers that the formula is satisfiable, then the analyzed program is deemed insecure. DynCoVer repeats this process for all of the nodes in SOT and if the SMT solver's answer was unsatisfiable for all nodes, the program is accepted as secure.

For the perfect recall and bounded memory attackers, DynCoVer also checks the policy consistency for all nodes that are marked as "nodes following a policy change". DynCoVer uses Definition 12 for generating the policy consistency check formula, feeds it to the SMT solver, and if the result was unsatisfiable, it deems the policy change as consistent, and moves on to checking the security on that node.

DynCoVer also supports policy repair by relying on the heuristic of Section 6.3. If configured in repair mode, DynCoVer performs preprocessing on the SOT and identifies leaking expressions. Upon reaching an inconsistent policy, it shows a warning message, proceeds to generate the repaired policy, and prints it to the user.

DynCoVer uses ENCoVER [26] as a basis and extends it with support for dynamic polices, policy consistency checks, and policy repair. DynCoVer is approximately 6 KLOC as computed by CLOC and includes nearly 86 classes/interfaces. Like ENCoVER, the class of programs that DynCoVer can handle is indirectly limited by the class of programs SPF (JPF core and its symbolic extension) can handle and the class of constraints Z3 can solve.

7.1. Case Studies

To evaluate the effectiveness and efficiency of DynCoVer, we carried out two different experiments. First, we created a micro benchmark suite to facilitate checking and understanding dynamic policy scenarios and different

types of attackers. Second, we implemented and verified the core of a social network to demonstrate the effectiveness of our security conditions in a real-world scenario.

These case studies target three objectives: (1) to validate that the results of DYNCOVER are in line with the conditions in Section 4; (2) to ensure that the policy repair heuristic works as expected; (3) to evaluate the performance of DYNCOVER.

7.1.1. Benchmark. Our benchmark consists of 25 programs, including programs from the paper, to demonstrate different aspects of dynamic policies. It includes programs with various constructs, loops, and implicit leaks. This benchmark is implemented in Java, and the only use of non-standard command is the `setPolicy` method, indicating a policy change. Each program has a configuration file which defines the attacker type, its memory capacity, and the method used to deal with inconsistent policies. These `.jpf` configuration files are used by JPF's virtual machine and DYNCOVER to verify the program.

Table 1 shows an excerpt of programs from the benchmark, Table 3 in the Appendix contains more programs. As we can see in column "DYNCOVER Result", DYNCOVER rejects insecure programs and accepts secure ones, in line with the definitions in Section 4. In addition to the results for security and policy consistency, Table 1 also reports some information about the efficiency and performance of DYNCOVER. For simple programs with small number of outputs, the SOT size and the evaluation time of DYNCOVER is low. But when testing more complex programs with multiple loops and outputs, the performance decreases. Memory usage of DYNCOVER is around 235 MB for simple programs, and starts to increase when the number of loops and instructions increases.

7.1.2. Social Network. In this case study, we implemented a social network that simulates the interactions between users, and contains some of the main functionalities of a social network, such as following, unfollowing, blocking other users, sending DMs, and creating groups and events. Users also have privacy settings and change their setting to hide their sensitive information such as phone number. The high level of interactivity between entities in a social networks makes it a good candidate for dynamic policy analysis.

A social network is naturally an interactive program, whose behavior is determined by the actions of different users of the system. To model these behaviors and make them amenable to extract the SOT of the program with DYNCOVER, we implemented an additional program which simulates the behavior of different users involved in the execution of the interactive program.

The Java implementation of the social network has 5 classes and 659 LOCs. There is one class for each of the entities: server, user, group, event, and post. To evaluate this case study, 6 different scenarios have been implemented and examined. The results of these experiments are reported in Table 2.

In the `postForFollowers` scenario, users interact by following each other and creating posts. The goal here is to ensure that a post is only visible to the followers of a certain user. This scenario is secure for a forgetful attacker, and inconsistent for a perfect recall attacker. This is because the ex-follower has already seen some of the unfollowed user's posts. The second scenario is similar, but this time a user can block their followers. Similarly, this program is secure for a forgetful attacker and inconsistent for a perfect recall attacker. The policy repair mode does not make sense for these scenarios, because after unfollowing or getting blocked, the observer should no longer be able to see the user's old posts.

The next scenario simulates forwarding a user's DM to another user. This scenario is insecure for all types of attackers, because a third user should not be able to see other users' DMs. The `phoneNumberPrivacy` scenario checks the privacy settings of a user. Initially, the user's information such as the phone number is private. However, a user can changes their privacy setting to make the phone number public. The goal here is to make sure that users cannot see other users' private information. This scenario is secure for both perfect recall and forgetful attackers.

Next, we consider a scenario in which a user's membership in a group should be kept secret from all users that are not in that group. A user cannot see the group members until they are added to that group. This scenario is secure for all three types of attackers. Now, if we change this scenario in such a way that a users leaves the group after learning the names of its members, then the program is inconsistent for perfect recall and secure for forgetful attackers. In the last scenario we consider events. Here a user should not be able to see an event's information such as its title or date unless they are invited to it. The program is secure for this scenario.

8. Related Work

This section discusses closely related works targeting dynamic policies and information flow control. We refer to Broberg et al. [11] for a survey on dynamic policies.

Our security framework is inspired by the work of Askarov and Chong [8] on knowledge-based security conditions for dynamic policies. They propose a general framework for capturing the semantics of dynamic policies for all attackers, showing that secure program under perfect recall attacker can be insecure under a weaker attacker. We revisit and extend their framework to accommodate three realistic attacker models and point out the challenges with policy consistency. Because the notions of perfect recall attacker and bounded memory attackers have well-defined interpretations in applications and epistemic logics [29], security conditions should be specific about the attacker model and uncover inconsistent policies. This allows us to show that the security of a program under a stronger attacker implies security under weaker attackers. Moreover, we propose a security condition for forgetful attackers to capture transient release of sensitive information, instantiating the framework of Askarov and Chong. On the enforcement side, DYNCOVER uses automated theorem proving while Askarov and Chong design a security type system, each targeting well-known trade-offs between precision and scalability. Van Deft et al. [10] improve the framework of Askarov and Chong with regards to progress-insensitive security, showing how a type system enforces security against all attackers. By contrast, our conditions are progress sensitive, while progress insensitivity can be accommodated

TABLE 1: Benchmark evaluation results

	DynCoVer Result	Attacker Type	Inconsistent Policy Mode	JPF Inst	Time (ms) OA	ME	FG	FS	PR	SOT Nodes
Program 3	↯	Perfect	Reject	2940	219.5	8.7	1.5	31.6	–	2
Program 3	✓	Perfect	Repair	2940	231.6	8.8	0.6	31.5	20.4	2
Program 3	✓	Forgetful	–	2940	217.4	8.7	1.6	31.4	–	2
Program 7	×	Perfect	Reject	2964	316.4	38.4	2.6	84.1	–	5
Program 7	×	Bounded	Reject	2964	301.0	38.6	2.5	84.3	–	5
Program 7	×	Forgetful	–	2964	298.2	38.5	2.3	69.0	–	5
Program 13	↯	Perfect	Reject	2982	292.7	49.5	1.9	63.3	–	7
Program 13	✓	Perfect	Repair	2982	367.1	49.0	1.9	91.5	47.7	7
Program 13	✓	Bounded	Reject	2982	341.6	47.0	3.6	111.5	–	7
Program 13	✓	Forgetful	–	2982	300.0	48.6	2.9	68.2	–	7

DynCoVer Results: ✓ the program is secure; × the program is insecure; ↯ the program has an inconsistent policy change.
Inconsistent Policy: What to do when facing an inconsistent policy: *Reject* the inconsistent policies; *Repair* the policy.
JPF Inst: total number of instructions executed by JPF
Time: *OA*: overall; *ME*: model extraction ; *FG*: interference formula generation; *FS*: interference formula satisfiability checking; *PR*: policy repair (only if applicable)
SOT Nodes: Number of nodes in the generated Symbolic Output Tree

TABLE 2: Social network evaluation results

	DynCoVer Result	Attacker Type	Inconsistent Policy Mode	JPF Inst	Time (ms) OA	ME	FG	FS	PR	SOT Nodes
postForFollowers	↯	Perfect	Reject	13953	659.4	270.5	11.5	168.0	–	24
postForFollowers	✓	Forgetful	–	13953	795.6	293.3	12.4	160.3	–	24
blockingUser	↯	Perfect	Reject	14133	656.3	274.5	10.5	163.5	–	25
blockingUser	✓	Forgetful	–	14133	599.4	226.6	11.5	160.1	–	25
forwardingDM	×	Perfect	Reject	13037	519.0	169.5	4.2	154.1	–	12
forwardingDM	×	Bounded	Reject	13037	499.5	166.8	4.4	145.0	–	12
forwardingDM	×	Forgetful	–	13037	495.9	164.4	4.1	130.0	–	12
phoneNumberPrivacy	✓	Perfect	Reject	13135	656.6	181.0	14.0	261.5	–	21
phoneNumberPrivacy	✓	Forgetful	–	13135	680.7	212.7	9.8	242.9	–	21
leakMembership	✓	Perfect	Reject	18569	747.0	286.8	10.1	232.0	–	19
leakMembership_leave	↯	Perfect	Reject	18850	802.2	314.9	8.7	283.8	–	21
leakMembership_leave	✓	Forgetful	–	18850	769.0	301.8	13.0	231.7	–	21
leakEventInfo	✓	Perfect	Reject	10676	427.5	159.5	2.0	73.9	–	4

following Van Deft et al. [10]. Broberg et al. [11] illuminate the different facet of dynamic policies proposed in the literature [30], [12], [13], [14], [16], [9], [21]. We revisit and discuss this facets in our framework by developing the corresponding security conditions or pointing out mismatches. Other works addressing the challenge of flexible policies include Chudnov and Naumann [31] framework for downgrading policies in reactive programs, Lu and Zhang's [32] framework for non-transitive policies, and Kozyri and Schneider's [33] reactive labels.

Recently, Li and Zhang [34] propose a general-purpose framework for dynamic policies. Their approach categorizes dynamic policies into persistent and transient policies, and uses the notion of effective traces to define a unified knowledge-based security condition. By contrast, our work departs from existing works on dynamic policies, by focusing on an attacker-centric approach and tries to address the issues of dynamic policies through policy consistency and the relation between attacker power and policy.

Our enforcement techniques build on the line of work on verifying static information flow policies by automated theorem proving [17], [19], [18], [35], [36]. To our best knowledge, none of these works addresses either dynamic policies or the issues of policy consistency and policy repair. A precursor of our approach is the work of Balliu et al. [18] which also uses Java Pathfinder to extract program dependencies and verify static noninterference policies by symbolic execution. Our verification conditions are similar to Balliu et al. [18] for static policies, and develop them further to accommodate dynamic policies. Paragon [37] extends Java with support for dynamic policies using a security type system [16], [38] and it enforces security for the perfect recall attacker. By contrast, DynCoVer additionally supports bounded memory and forgetful attacker models with policy consistency and repair. Other languages and tools supporting information flow control for perfect recall attackers include JIF [39], LIO [40], Jeeves [41], and JOANA [42].

9. Conclusion

We have revised knowledge-based security conditions for dynamic policies and proposed attacker-centric conditions for perfect recall, bounded memory, and forgetful attackers. Drawing on the notion of policy consistency, we studied the relationship with the different facets of dynamic policies as well as policy repair. To verify and repair dynamic policies under different attacker models, we designed, implemented, and evaluated DynCoVer, an open source tool based on symbolic execution and SMT solving. An interesting avenue for future work is to investigate the interplay between integrity and confidentiality for dynamic policies [43], [44], [45], [46], [47].

Acknowledgments We thank Roberto Guanciale and anonymous reviewers for the feedback. This work is partially supported by the JointForce project funded by Swedish Research Council (VR), Swedish Foundation for Strategic Research (SSF), Facebook, and Digital Futures.

References

[1] J. A. Goguen and J. Meseguer, "Security policies and security models," in *S&P*, 1982, pp. 11–20.

[2] A. Askarov and A. Sabelfeld, "Gradual release: Unifying declassification, encryption and key release policies," in *S&P*, 2007.

[3] ——, "Tight enforcement of information-release policies for dynamic languages," in *CSF*. IEEE, 2009, pp. 43–59.

[4] M. Balliu, M. Dam, and G. Le Guernic, "Epistemic Temporal Logic for Information Flow Security," in *PLAS*, 2011.

[5] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *J. Comput. Secur.*, vol. 4, pp. 167–187, 1996.

[6] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *JSAC*, 2003.

[7] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *JCS*, 2009.

[8] A. Askarov and S. Chong, "Learning is change in knowledge: Knowledge-based security for dynamic policies," in *CSF*. IEEE, 2012, pp. 308–322.

[9] M. Balliu, "A logic for information flow analysis of distributed programs," in *NordSec*, 2013.

[10] B. van Delft, S. Hunt, and D. Sands, "Very static enforcement of dynamic policies," in *POST*. Springer, 2015, pp. 32–52.

[11] N. Broberg, B. van Delft, and D. Sands, "The anatomy and facets of dynamic policies," in *CSF*. IEEE, 2015, pp. 122–136.

[12] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic, "Dynamic updating of information-flow policies," in *FCS*, 2005.

[13] N. Swamy, M. Hicks, S. Tse, and S. Zdancewic, "Managing policy updates in security-typed languages," in *CSF Workshop*. IEEE, 2006, pp. 13–pp.

[14] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive Declassification Policies and Modular Static Enforcement," in *S&P*, 2008, pp. 339–353.

[15] A. A. Matos and G. Boudol, "On declassification and the non-disclosure policy," *J. Comput. Secur.*, vol. 17, no. 5, pp. 549–597, 2009.

[16] N. Broberg and D. Sands, "Flow-sensitive semantics for dynamic information flow policies," in *PLAS*, 2009, pp. 101–112.

[17] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *SPC*, 2005, pp. 193–209.

[18] M. Balliu, M. Dam, and G. Le Guernic, "Encover: Symbolic exploration for information flow security," in *CSF*. IEEE, 2012, pp. 30–44.

[19] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *MSCS*, 2011.

[20] A. M. Ahmadian and M. Balliu, "DynCoVer," March 2022, software release. [Online]. Available: https://github.com/amir-ahmadian/jpf-dyncover

[21] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld, "Information-flow control for database-backed applications," in *EuroS&P*. IEEE, 2019, pp. 79–94.

[22] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, "Termination-insensitive noninterference leaks more than just a bit," in *ESORICS*. Springer, 2008, pp. 333–348.

[23] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *FAST*. Springer, 2008, pp. 50–66.

[24] C. S. Păsăreanu and W. Visser, "Verification of java programs using symbolic execution and invariant generation," in *Model Checking Software*, S. Graf and L. Mounier, Eds., 2004, pp. 164–181.

[25] A. Sabelfeld and A. C. Myers, "A model for delimited information release," in *ISSS*, vol. 3233. Springer, 2003, pp. 174–191.

[26] M. Balliu and G. Le Guernic, "ENCoVer," 2012, software release. [Online]. Available: http://www.nada.kth.se/~musard/encover

[27] C. S. Pasareanu, W. Visser, D. H. Bushnell, J. Geldenhuys, P. C. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 391–425, 2013.

[28] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.

[29] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about knowledge*. Cambridge, Mass.: MIT Press, 1995.

[30] H. Mantel, "Information flow control and applications - bridging a gap," in *FME*. Springer, 2001, pp. 153–172.

[31] A. Chudnov and D. A. Naumann, "Assuming you know: Epistemic semantics of relational annotations for expressive flow policies," in *CSF*. IEEE, 2018, pp. 189–203.

[32] Y. Lu and C. Zhang, "Nontransitive security types for coarse-grained information flow control," in *CSF*. IEEE, 2020, pp. 199–213.

[33] E. Kozyri and F. B. Schneider, "Rif: Reactive information flow labels," *J. Comput. Secur.*, vol. 28, pp. 191–228, 2020.

[34] P. Li and D. Zhang, "Towards a general-purpose dynamic information flow policy," *arXiv preprint*, 2021.

[35] D. Milushev, W. Beck, and D. Clarke, "Noninterference via symbolic execution," in *Formal Techniques for Distributed Systems*. Springer, 2012, pp. 152–168.

[36] Q. H. Do, R. Bubel, and R. Hähnle, "Automatic detection and demonstrator generation for information flow leaks in object-oriented programs," *computers & security*, vol. 67, pp. 335–349, 2017.

[37] N. Broberg, B. van Delft, and D. Sands, "Paragon - practical programming with information flow control," *J. Comput. Secur.*, vol. 25, pp. 323–365, 2017.

[38] N. Broberg and D. Sands, "Paralocks: role-based information flow control and beyond," in *POPL*, 2010, pp. 431–444.

[39] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *POPL*, 1999, p. 228–241.

[40] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières, "Flexible dynamic information flow control in haskell," in *SIGPLAN*, 2011, pp. 95–106.

[41] J. Yang, K. Yessenov, and A. Solar-Lezama, "A language for automatically enforcing privacy policies," in *POPL*, 2012, p. 85–96.

[42] G. Snelting, D. Giffhorn, J. Graf, C. Hammer, M. Hecker, M. Mohr, and D. Wasserrab, "Checking probabilistic noninterference using joana," *it Inf. Technol.*, vol. 56, pp. 280–287, 2014.

[43] A. Myers and A. Askarov, "Attacker control and impact for confidentiality and integrity," *Logical Methods in Computer Science*, vol. 7, 2011.

[44] A. C. Myers, A. Sabelfeld, and S. Zdancewic, "Enforcing robust declassification," in *CSF Workshop*. IEEE, 2004, pp. 172–186.

[45] M. Balliu and I. Mastroeni, "A weakest precondition approach to active attacks analysis," in *PLAS*, 2009, pp. 59–71.

[46] ——, "A weakest precondition approach to robustness," *Trans. Comput. Sci.*, vol. 10, pp. 261–297, 2010.

[47] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *CCS*, 2017, pp. 1875–1891.

Appendix A. Bounded Memory Attacker

The unlimited memory of perfect recall attacker means that it can remember everything it once observed. A bounded memory attacker is a variant of perfect recall attacker with a limited memory (called m hereafter). After observing m outputs, its memory will become full and in order to capture any new outputs the oldest observation should be removed from it (in a FIFO manner).

A.1. Security Policies

The security condition used for bounded memory attacker is similar to Definition 4, except that when computing the knowledge of a bounded memory attacker, we have to consider its memory capacity (m) as well as the number of outputs it has observed. If the number of outputs is less than m, the attacker is going to behave just like prefect recall, and if it is more than m, the attacker is going to only remember the last m observations.

With this intuition, we can define the auxiliary function *suffix(t,m)* which takes a trace t and returns the last m events of the trace:

Definition 14. *Given trace t as $t = \alpha_1.\alpha_2...\alpha_k$, suffix(t,m) is defined as:*

$$\textit{suffix(t,m)} = \begin{cases} t & \text{if } k \leq m \\ \alpha_{k-m}...\alpha_k & \text{if } k > m \end{cases}$$

Now, we can define the knowledge of a bounded memory attacker at execution point i.

Definition 15. *Program c with initial store σ, and initial policy p_{init} produces trace t after i execution steps, i.e., $\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i$. We write $k_i^{bnd}(c, \sigma, p_{init}, A, m)$ for the knowledge of an attacker that observes the outputs of this program on channel A and has a bounded memory capacity of length m, and define it as follows:*

$$k_i^{bnd}(c, \sigma, p_{init}, A, m) = \{\sigma' \mid \langle c, \sigma, p_{init}\rangle \overset{t'}{\Rightarrow}_j \\ \wedge\ \textit{suffix}(t\lfloor_A, m) = \textit{suffix}(t'\lfloor_A, m)\}$$

By adapting Definitions 2 and 3 to $k_i^{bnd}(c, \sigma, p_{init}, A, m)$ for attacker knowledge, we can use Definition 4 to check security for bounded memory attackers as well.

A.2. Verification of Dynamic Policies

Since a bounded memory attacker is a special type of perfect recall with limited observations, the verification approach used for checking a program's security against a bounded attacker is also similar to the verification for a perfect recall attacker. We just have to modify Definition 11 to account for the limited memory of the attacker.

Definition 16. *Given an SOT S, active policy $P_n(\overrightarrow{l}, \overrightarrow{h})$ at node n, S is secure wrt. bounded memory attacker with memory capacity m iff for all nodes $n \in N(S)$, the following formula is unsatisfiable:*

$$P_n(\overrightarrow{l}, \overrightarrow{h}) \wedge \Bigg(Pc_n(\overrightarrow{l_n}, \overrightarrow{h_n}) \ \wedge \Big(\bigwedge_{n' \in N(S)} \\ \neg\Big(Pc_{n'}(\overrightarrow{l_n}, \overrightarrow{h_n}') \wedge \overrightarrow{O}_n^m(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^m(\overrightarrow{l_n}, \overrightarrow{h_n}')\Big)\Big)\Bigg)$$

*where $\overrightarrow{O}_n^m$ is a tuple of the **last** m output expressions encountered on a path in SOT from node Start to node n, $\overrightarrow{O}_n^m = \overrightarrow{O}_{n'}^m$ denotes the component-wise equality between two tuples, and $N(S)$ is the nodes of SOT S.*

In this definition we limit the length of the attacker's observations by m, which is the capacity of his memory. In other words, $\overrightarrow{O}_n^m$ contains the last m outputs observable by the attacker. Therefore by checking $\overrightarrow{O}_n^m(\overrightarrow{l_n}, \overrightarrow{h_n}) = \overrightarrow{O}_{n'}^m(\overrightarrow{l_n}, \overrightarrow{h_n}')$ for all $n' \in N(S)$ we are looking for nodes that can – from the attacker's perspective – produce the same trace of outputs.

The process of checking a program's policy consistency for a bounded memory attacker is similar to that of a perfect recall attacker (Definition 12); the only difference is that we should use $\overrightarrow{O}_n^m$ instead of $\overrightarrow{O}_n$ during the generation of the formula for checking consistency.

To illustrate this, we revisit Program 13 and check its policy consistency under a bounded memory attacker with memory capacity of $m = 2$. For example, the generated formula for node $n5$ is:

$$\Bigg(Pc_{n3}(\emptyset, \{x, y\}) \ \wedge \Big(\bigwedge_{n' \in N(S)} \neg\Big(Pc_{n'}(\emptyset, \{x', y'\}) \\ \wedge\ \overrightarrow{O}_{n3}^2(\emptyset, \{x, y\}) = \overrightarrow{O}_{n'}^2(\emptyset, \{x', y'\})\Big)\Big)\Bigg)$$

The output sequence of node $n3$ is $O_{n3}^2 = (1, 1)$. The formula is satisfiable if there exists a value for y or x where $Pc_{n3}(y)$ holds, and *for all nodes* it falsifies either the path conditions or the equality between outputs. For this attacker, it is possible for the nodes which are not on the same level to have equal outputs, so we have to consider all $n' \in N(S)$. However, in this example, there are two nodes that can possibly produce an output sequence equal to $(1, 1)$. $n2$ with output sequence $(x, 1)$ and $n3$ with $(1, 1)$, for both of which the path condition is also $true$. We consider $n3$ first, because in this case both of the output sequences are $(1, 1)$, which means that the inner formula is true, hence the result of conjunction is false. This means that even without considering $n2$ we can be conclude that the whole formula is unsatisfiable and the policy change at node $n5$ is consistent. Similarly, we can apply Definition 16 to all of the nodes in S and show that Program 13 is in fact secure.

The following theorem shows soundness of Definition 16 wrt. the security condition for a bounded memory attacker.

Theorem 5. *Given a SOT S, if the formula in Definition 16 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition 3 for the bounded memory attacker.*

Proof. Similar to the proof of Theorem 2. □

Appendix B. Examples for the Forgetful Attacker

In this section, we revisit some of the examples presented in Section 4 to demonstrate how a forgetful attacker's knowledge is calculated.

For Program 5 we calculate the attacker's knowledge after the output of line 8. Without the loss of generality, let us assume x = 5 and y = 7. At this point, the attacker has observed the trace $t = 7.1.2$. There is also an unobservable new policy event between 1 and 2, thus the *splitPolicy(t)* function gives us the sub-traces $(7.1, 2)$. The length of the

sub-trace $t_1 = 7.2$ is 2, so the knowledge of attacker will be all of the stores that can produce a trace ending with 2 that has exactly two other observable events (of any value) before that:

$$k_8^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle \overset{t''}{\Longrightarrow}_j \\ \wedge\ (t_1'', t_2'') = \text{split}(t''\lfloor_A, 2) \\ \wedge\ t_2'' = 2\}$$

This corresponds to all the stores with any value for x, and since the active policy p at execution point $i = 8$ is also the set of all the stores, security condition 3 holds. If we repeat this process for all execution points, we can see that the program is accepted by Definition 3.

Similarly, the knowledge of the attacker at line 8 of Program 6 will be all of the stores that can produce a trace that ends with sub-trace 7.2 and have exactly one other observable event before that, which will be the stores with any value for x, but *only* value 7 for y. Since the active security policy at this point is the set of all stores with any value for both x and y, the security condition $p_8 \subseteq k_8^{frg}(c, \sigma, p_{init}, A)$ *does not* hold and Program 6 is rejected as insecure.

In Program 7, for a positive x, the attacker observes the trace $t = 1.1.1$ after the output on line 8. The *splitPolicy(t)* function gives us the tuple $(1.1, 1)$ and since $\mid 1.1 \mid$ is 2, the attacker knowledge will be all of the stores that can produce a trace that ends with 1, and have exactly two other observable event before that:

$$k_8^{frg}(c, \sigma, p_{init}, A) = \{\sigma' \mid \langle c, \sigma', p_{init}\rangle \overset{t''}{\Longrightarrow}_j \\ \wedge\ (t_1'', t_2'') = \text{split}(t''\lfloor_A, 2) \\ \wedge\ t_2'' = 1\}$$

The only stores that can produce such a trace are the ones with `x > 0`, which implies that the security condition *does not* hold and the program is rejected as insecure.

Appendix C. Proof of Verification Soundness

Here we present sketches for the proofs of Theorems 2, 3, and 4.

Theorem 2. *Given a SOT S, if the formula in Definition 11 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition 3.*

Proof. By Definition 3, a program is insecure if there is a point i during the execution in which the policy is not contained in the knowledge (i.e., $p_i \not\subseteq k_i$). This means that there is a value such that it is in the policy but not in the knowledge. Similarly, in the SOT, if the formula corresponding to a node n is satisfiable, it means that there exist two stores that satisfy the policy $P(\overrightarrow{l}, \overrightarrow{h})$, but either one store does not reach the output node or the two stores produce different output sequences. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i$, thus violating the security condition.

On the other hand if the formula at node n is unsatisfiable, it implies that for any initial state $(\overrightarrow{l}, \overrightarrow{h})$ that satisfies the policy $P(\overrightarrow{l}, \overrightarrow{h})$ and reaches node n producing the output sequence $\overrightarrow{O}_n$, it is impossible to find another state $(\overrightarrow{l}, \overrightarrow{h}')$ that satisfies the policy and reaches some output node n' (i.e., $n \in N(S)$ and $Pc_{n'}(\overrightarrow{l}, \overrightarrow{h}')$) yielding a different output sequence $\overrightarrow{O}_{n'}$. If we repeat this process for all nodes $n \in S$, and none of their formulas are satisfiable, it means that there are no outputs in SOT S such that $p_i \not\subseteq k_i$. □

Theorem 3. *Given a SOT S, if the formula in Definition 12 is unsatisfiable for all nodes $n \in S$ such that n follows a policy change, then S satisfies Definition 2.*

Proof. The proof of this theorem is similar to Theorem 2. However, because in the SOT we do not have any nodes for the policy change, we have to capture the policy changes on the next output nodes.

In the policy consistency check formula (Definition 12) the policy part of the formula ($P(\overrightarrow{l}, \overrightarrow{h})$) is generated at node n because we want it to reflect the new policy, however, the rest of the formula is generated for n's parent node (i.e., $parent(n)$). The satisfiability of this formula means that there exist two stores that satisfy the new policy $P(\overrightarrow{l}, \overrightarrow{h})$ at node n, but either one store does not reach an output or the two stores produce different output sequences up to $parent(n)$. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P_n(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_{i-1}$, thus violating the policy consistency condition.

If we repeat this process for all nodes $n \in S$ with new policy, and none of their formulas are satisfiable, it means that there are no policy changes in SOT S such that $p_i \not\subseteq k_{i-1}$. Here we assume that there is always an output after a policy change. For programs that have a policy change as their last command, we can use the node End for policy consistency check and apply Definition 12. □

Theorem 4. *Given a SOT S, if the formula in Definition 13 is unsatisfiable for all nodes $n \in S$, then S satisfies Definition 3 for the forgetful attacker.*

Proof. The difference between the the security condition of the forgetful attacker and the perfect recall is that the latter uses Definition 7 to calculate the attacker's knowledge.

This definition limits the observations of the attacker to the number of outputs before the policy change and the values of outputs after a policy change. The output function $\overrightarrow{O}_n^{frg}$ used in the forgetful attacker's formula (Definition 13) captures this behavior by ignoring the value of outputs before the last policy change (replacing them with a constant value), and only keeping the actual value of the outputs that happened after the policy change.

The rest is similar to the proof of Theorem 2. The satisfiability of the formula of Definition 13 means that there exist two stores that satisfy the new policy $P(\overrightarrow{l}, \overrightarrow{h})$ at node n, but if they reach an output, the output sequences up to node n wrt. $\overrightarrow{O}_n^{frg}$ will be different. This implies that there is an initial store $(\overrightarrow{l}, \overrightarrow{h}) \in P_n(\overrightarrow{l}, \overrightarrow{h})$, such that $(\overrightarrow{l}, \overrightarrow{h}) \notin k_i^{frg}$, thus violating the security condition condition. □

Appendix D.
Proof of Attacker Power

In this section, we present Theorem 6 to prove the claim that in the absence of inconsistent policy changes, a program which is secure against a perfect recall attacker is also secure against a bounded memory attacker.

Theorem 6. *Given a program c with no inconsistent policy changes, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against bounded memory attacker A^m_{bnd} with memory capacity $m \in N$. Formally:*

$$p_i \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m) \quad \forall m \in N$$

Proof. We should consider all execution points i such that

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i\rangle$$

and continue with structural induction on command c_i.

All of the commands presented in Figure 1 should be considered here. However, not all of them have an effect on the knowledge, therefore we only investigate command $\texttt{output}_\ell(e)$. Without the loss of generality let us assume that e is visible to the attacker A^m_{bnd}.

Since knowledge is monotone, the more observations an attacker has, the smaller his knowledge set will be. We can use this fact to limit the number of cases we have to investigate for different values of m. Thus, we only consider two scenarios:

- if $\mid (t.\alpha)\downarrow_{A^m_{bnd}} \mid \leq m$ then the bounded memory attacker with memory capacity m is going to know everything that the perfect recall attacker knows. Hence

  $$k_i(c, \sigma, p_{init}, A_{per}) = k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  and since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude:

  $$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

- if $\mid (t.\alpha)\downarrow_{A^m_{bnd}} \mid > m$ then the bounded memory attacker A^m_{bnd} had less observations than the perfect recall attacker. Hence

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  and since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude that

  $$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

  also holds.

As a result, the security condition:

$$p_i \subseteq k_i^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m) \quad \forall m \in N$$

holds for all values of $m \in N$.

Additionally, if c_i is $\texttt{setPolicy}(p')$ we can use the assumption that program c does not have any inconsistent policies to conclude that Definition 2 holds for perfect recall attacker A_{per} for all execution points i. Since we already established that bounded memory attacker A^m_{bnd}'s knowledge is less than or equal to A_{per} at each execution point, it is straightforward to show that:

$$p_i \subseteq k_{i-1}^{bnd}(c, \sigma, p_{init}, A^m_{bnd}, m)$$

which means that the policy changes are also consistent for bounded memory attacker A^m_{bnd}. □

Theorem 1 proves a similar claim for the forgetful attackers.

Theorem 1. *Given a program c, initial store σ, and initial policy p_{init}, if for all execution points i, c is secure against perfect recall attacker A_{per}, it is also secure against forgetful attacker A_{frg}. Formally:*

$$[\sigma]^{p_i}_A \subseteq k_i(c, \sigma, p_{init}, A_{per}) \implies [\sigma]^{p_i}_A \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

Proof. The proof of this theorem is similar to Theorem 6. We consider all execution points i such that:

$$\langle c, \sigma, p_{init}\rangle \overset{t}{\Rightarrow}_i \langle c_i, \sigma_i, p_i\rangle$$

and only investigate the case where command c_i is $\texttt{output}_\ell(e)$, and assume that e is visible to the attacker A_{frg}. Let us consider three scenarios:

- If *splitPolicy(t)* is (ϵ, t), then the observations of forgetful attacker are the same as the observations of the perfect recall attacker, hence:

  $$k_i(c, \sigma, p_{init}, A_{per}) = k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

  Since by assumption we know that $p_i \subseteq k_i(c, \sigma, p_{init}, A_{per})$, we can conclude:

  $$p_i \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

- If *splitPolicy(t)* is (t, ϵ), then the forgetful attacker makes no observations after the policy change and only knows $\mid t \mid$. Thus,

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

  Hence, we can conclude that:

  $$p_i \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

- If *splitPolicy(t)* is (t_1, t_2). Since knowledge is monotone and t_2 is a sub-trace of t, the knowledge set of an observer that sees t_2 is bigger that the knowledge set of the observer of t. Additionally, for all events before t_2, the attacker A_{per} observed the actual value of the event while the attacker A_{frg} only knows that an event has occurred. Therefore, it is straightforward to show that the knowledge of A_{frg} is less than the knowledge of A_{per}:

  $$k_i(c, \sigma, p_{init}, A_{per}) \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

  Therefore we can conclude that:

  $$p_i \subseteq k_i^{frg}(c, \sigma, p_{init}, A_{frg})$$

□

TABLE 3: Benchmark evaluation results (Extended Table)

	DynCoVer Result	Attacker Type	Inconsistent Policy Mode	JPF Inst	Time (ms) OA	ME	FG	FS	PR	SOT Nodes
Program 1	↯	Perfect	Reject	2937	279.1	8.7	1.4	69.3	–	2
Program 1	✓	Perfect	Repair	2937	242.5	8.5	0.3	39.7	19.6	2
Program 1	×	Forgetful	–	2937	222.0	8.5	1.4	29.5	–	2
Program 2	↯	Perfect	Reject	2937	211.3	8.6	1.4	27.6	–	2
Program 2	✓	Perfect	Repair	2937	249.5	8.6	0.3	52.0	19.9	2
Program 2	✓	Forgetful	–	2937	210.8	8.5	1.4	25.7	–	2
Program 3	↯	Perfect	Reject	2940	219.5	8.7	1.5	31.6	–	2
Program 3	✓	Perfect	Repair	2940	231.6	8.8	0.6	31.5	20.4	2
Program 3	✓	Forgetful	–	2940	217.4	8.7	1.6	31.4	–	2
Program 4	×	Perfect	Reject	2937	229.2	6.8	1.6	41.3	–	2
Program 4	×	Forgetful	–	2937	216.5	7.2	1.5	31.5	–	2
Program 5	↯	Perfect	Reject	2972	252.4	46.8	1.5	32.0	–	5
Program 5	✓	Perfect	Repair	2972	342.9	48.7	1.3	74.3	47.2	5
Program 5	✓	Forgetful	–	2972	279.8	48.1	2.3	51.7	–	5
Program 6	×	Perfect	Reject	2972	294.3	48.9	2.1	48.4	–	5
Program 6	×	Bounded	Reject	2972	278.3	51.0	1.8	42.3	–	5
Program 6	×	Forgetful	–	2972	250.4	45.9	1.6	31.7	–	5
Program 7	×	Perfect	Reject	2964	316.4	38.4	2.6	84.1	–	5
Program 7	×	Bounded	Reject	2964	301.0	38.6	2.5	84.3	–	5
Program 7	×	Forgetful	–	2964	298.2	38.5	2.3	69.0	–	5
Program 8	×	Perfect	Reject	2938	214.2	6.7	1.3	14.8	–	1
Program 8	×	Perfect	Repair	2938	213.8	6.7	0.1	13.6	27.2	1
Program 8	×	Forgetful	–	2938	198.5	6.5	1.3	13.3	–	1
Program 9	↯	Perfect	Reject	2942	226.9	6.9	1.7	39.3	–	3
Program 9	✓	Perfect	Repair	2942	250.0	6.7	0.5	51.3	20.6	3
Program 9	×	Forgetful	–	2942	221.4	6.7	1.6	36.4	–	3
Program 10	×	Perfect	Reject	2937	224.2	6.4	1.6	39.0	–	2
Program 10	×	Forgetful	–	2937	211.6	7.0	1.5	28.9	–	2
Program 11	↯	Perfect	Reject	2940	218.1	8.6	2.4	23.9	–	2
Program 11	✓	Perfect	Repair	2940	330.8	9.4	0.4	39.2	29.1	2
Program 12	✓	Perfect	Reject	2954	265.5	37.8	2.0	52.3	–	4
Program 12	✓	Bounded	Reject	2954	269.3	37.8	2.0	49.3	–	4
Program 12	✓	Forgetful	–	2954	276.8	38.5	2.1	43.4	–	4
Program 13	↯	Perfect	Reject	2982	292.7	49.5	1.9	63.3	–	7
Program 13	✓	Perfect	Repair	2982	367.1	49.0	1.9	91.5	47.7	7
Program 13	✓	Bounded	Reject	2982	341.6	47.0	3.6	111.5	–	7
Program 13	✓	Forgetful	–	2982	300.0	48.6	2.9	68.2	–	7
WhileLoop_5	✓	Perfect	Reject	3393	705.6	96.2	26.7	385.9	–	30
WhileLoop_10	✓	Perfect	Reject	4093	1743.7	172.8	100.4	1226.4	–	85
WhileLoop_50	✓	Perfect	Reject	20493	198030	1797	68486	126451	–	1425

DynCoVer Results: ✓ the program is secure; × the program is insecure; ↯ the program has an inconsistent policy change.
Inconsistent Policy: What to do when facing an inconsistent policy: *Reject* the inconsistent policies; *Repair* the policy.
JPF Inst: total number of instructions executed by JPF
Time: *OA*: overall; *ME*: model extraction ; *FG*: interference formula generation; *FS*: interference formula satisfiability checking; *PR*: policy repair (only if applicable)
SOT Nodes: Number of nodes in the generated Symbolic Output Tree