2015 IEEE 28th Computer Security Foundations Symposium

Understanding and Enforcing Opacity

Daniel Schoepe and Andrei Sabelfeld

Chalmers University of Technology

***Abstract*—This paper puts a spotlight on the specification and enforcement of *opacity*, a security policy for protecting sensitive properties of system behavior. We illustrate the fine granularity of the opacity policy by *location privacy* and *privacy-preserving aggregation* scenarios. We present a framework for opacity and explore its key differences and formal connections with such well-known information-flow models as *noninterference*, *knowledge-based security*, and *declassification*. Our results are machine-checked and parameterized in the observational power of the attacker, including *progress-insensitive*, *progress-sensitive*, and *timing-sensitive* attackers. We present two approaches to enforcing opacity: a *whitebox monitor* and a *blackbox sampling-based enforcement*. We report on experiments with prototypes that utilize state-of-the-art *Satisfiability Modulo Theories (SMT) solvers* and the *random testing* tool *QuickCheck* to establish opacity for the location and aggregation-based scenarios.**

I. Introduction

This paper puts a spotlight on the specification and enforcement of *opacity* [11, 46, 39, 14], a security policy for protecting sensitive properties of system behavior. Intuitively, a predicate on system behaviors is opaque if for any behavior that satisfies the predicate, there is another behavior that is indistinguishable by the attacker but where the predicate no longer holds.

Scenarios for opacity Opacity is a natural policy in many scenarios. As of 2011, Flickr's geoprivacy settings [25] include *geofences* for geotagging policies in sensitive areas, e.g., as depicted in Figure 1, no one can see the location of the photos that a user tags in a geofenced area (e.g., the user's home). Scenarios like Flickr's aim at improving privacy of *location-based services (LBS)*, an important area with much recent attention and progress [42, 62]. We elaborate on variations of the scenario where an LBS is required to protect users' location, but only when they are within sensitive areas. Note that there are many scenarios beyond location privacy, e.g., regulating how the result of a health test can be released depending on its outcome, in a similar fashion as location release that depends on the location.

Suppose hX and hY store the user's potentially sensitive (or *high*) Cartesian location coordinates. We assume the coordinates are static, as tracking users over time would require additional protection. Say, a clinic, representing a sensitive location, occupies a rectangular area with the corners $(200, 50)$ and $(400, 150)$. Consider the scenario of an LBS that outputs on a public (or *low*) channel, with the goal of privacy-preserving output.

Figure 1. Flickr's geofences

```
          /* Program 1 */
 /* Location privacy with fixed output */
clinicXmin := 200; clinicXmax := 400;
clinicYmin := 50; clinicYmax := 150;
if (hX >= clinicXmin && hX <= clinicXmax &&
    hY >= clinicYmin && hY <= clinicYmax) {
    out L (100, 200);
} else {
    out L (hX, hY);
}
```

In Program 1, when the user is inside the clinic, the program reports a default location $(100, 200)$ and otherwise the user's real location. Can the common security definitions directly support the scenario? *Noninterference* [29] and its *knowledge-based* analogues [36, 22] allow no flow from high to low whatsoever and thus wrongfully reject the intuitively secure program. *Declassification* [58] and *knowledge-based release* [3] models for intentional information release also have difficulties with the scenario. Fundamentally, declassification is often about *what can be released* [19, 41, 57, 54, 28, 43], while opacity is, conversely, about *what must be kept secret*. Declassification policies can be categorized along the dimensions of *what* information is released, *when* and *where* in the system the release takes place, and by *whom* the release is controlled [58]. When focusing on protecting secret inputs, closest to opacity are the *what* policies since they are concerned with separating secrets from non-secrets in the provided input. These *partial release* [19, 41, 57, 54, 28, 43] policies specify what is released by splitting the domain of secrets into subdomains and only protecting secret variation within the subdomains. When it comes to specifying partial

DOI 10.1109/CSF.2015.41

release, the subdomains can be expressed as *kernels* for *escape hatch expressions* [54], but since in our scenarios the released information depends on the location, escape hatch expressions would need to be nearly as complex as the program itself! We will come back to the relation to declassification in Section VIII.

On the other hand, opacity is a natural policy for this scenario. The property to protect is whether the user is inside the clinic. If the user is indeed there, the program will return the default location $(100, 200)$. Rightfully, there is another run of the program that originates outside the clinic (in $(100, 200)$) and produces the same observation as the original run.

Although the scope of this paper limits opacity to protecting information about input environments, opacity in general allows arbitrary behavioral properties of a system to be protected, and not only the values of secret inputs.

```
            /* Program 2 */
   /* Location privacy with random output */
clinicXmin := 200; clinicXmax := 400;
clinicYmin := 50; clinicYmax := 150;
if (hX >= clinicXmin && hX <= clinicXmax &&
    hY >= clinicYmin && hY <= clinicYmax) {
    randomize(x);
    while (x >= clinicXmin && x <= clinicXmax) {
          randomize(x);
    }
    randomize(y);
    while (y >= clinicYmin && y <= clinicYmay) {
          randomize(y);
    }
    out L (x, y);
} else {
    out L (hX, hY);
}
```

Program 2 is similar to Program 1 except a (pseudo)random outside location is returned whenever the user is inside the clinic. It makes it more difficult for the attacker to learn anything about the user's location. When observing the output, the attacker is unable to deduce whether it is the user's real location outside the clinic or a randomly generated location while the user is actually inside. As we elaborate later, this program satisfies *symmetric opacity* in the sense that both the sensitive predicate (whether the user is inside the clinic) and its negation are protected.

```
            /* Program 3 */
/* Location privacy with suppressed output */
privXmin := 78; privXmax := 159;
privYmin := 120; privYmax := 234;
if (!(hX >= privXmin && hX <= privXmax &&
      hY >= privYmin && hY <= privYmax)) {
    out L (hX, hY);
} else { skip; }
```

Program 3 directly models Flickr's geoprivacy policy. The user's location is simply suppressed if within the sensitive area and output normally otherwise. When observing the output, the attacker learns that the user is outside the sensitive area, which is safe. As elaborated later, this scenario connects to progress-insensitive security [2, 4, 10].

```
       /* Program 4  - Statistics aggregation */
int[10] hHasDisease; /* Declare hHasDisease as array */
i := 0;
result := 0;
while (i < 10) {
      result := result + ((hHasDisease[i] > 0) ? 1 : 0);
      i := i + 1;
}
out L result;
```

In a different scenario, Program 4 iterates over an array $hHasDisease$ (whose size is public) checking for patients diagnosed with a disease and aggregate the total of positive cases. The sensitive predicate to protect is whether a given patient is diagnosed positively or negatively. Common security definitions require a special treatment for such corner cases as when there is only one patient and when the count of the positively diagnosed patients is zero or the same as the total count. On the other hand, opacity covers corner cases by design because it directly focuses on the key property (predicate) to be hidden. In similar vein, opacity is a good fit for the *electronic voting* scenario, allowing to reveal the total count of votes for a given candidate without revealing the individual votes of any given voter.

Research questions With the above building the intuition for opacity in a variety of scenarios, the paper seeks to answer fundamental questions on understanding and enforcing opacity. Given the rich literature on security definitions and specifications [44, 26, 52, 55], the question is whether such common concepts as *noninterference* [29], *knowledge-based security* [36, 22, 3], and *declassification* [58] relate to opacity. If so, what is the exact relation? How does the relation depend on the power of the attacker?

An important unexplored problem is how to assure opacity. This leads to several questions. How can opacity be enforced? Is whitebox or blackbox enforcement more appropriate or perhaps both can be useful? If so, how well can such techniques help with the outlined scenarios?

Contributions To answer the questions above, we present a framework for opacity, spell out the differences, and formalize and machine-check the relation to noninterference and knowledge-based security in a fashion parametric in the attacker power, including *progress-insensitive*, *progress-sensitive*, and *timing-sensitive* attackers (Section II). We instantiate the formal connection results to batch-job (Section III) and interactive (Section IV) settings. We relate opacity to the *what* dimension of declassification, as captured by the model of delimited release [54] (Section V). We present two approaches to enforcement, a *whitebox dynamic monitor* and a *blackbox sampling-based mechanism*, both established sound (Section VI). We build prototypes that

utilize state-of-the-art *Satisfiability Modulo Theories (SMT) solvers Z3* [20] and *CVC4* [7] as well as the *random testing* tool *QuickCheck* [16] and report on our experiments for the scenarios of location privacy and statistics aggregation (Section VII). Finally, the paper offers a discussion of related work (Section VIII) and concluding remarks (Section IX).

II. Framework

We characterize the connection between opacity and common security definitions in a general, language-independent way, to be instantiated to a batch-job setting in Section III and an I/O setting in Section IV. Our results are parameterized in the attacker's power. Further, we relate opacity to knowledge-based security.

Assume $\mathbb{E}$ to be the set of *environments*. Denote the set of possible *results* of a program by $\mathbb{R}$. A *configuration* $\langle c, e\rangle$ is a pair consisting of a command $c \in \mathbb{C}$ and an environment $e \in \mathbb{E}$. Assume that there is an evaluation relation $\langle c, e\rangle \Downarrow r$ relating a configuration $\langle c, e\rangle$ with a result $r \in \mathbb{R}$.

Note that this allows for *nondeterministic* semantics. However, we assume that the secrets that should be kept confidential are about initial environments, not about nondeterministic components in outputs. Focusing on the protection of initial environments enables us to formally connect opacity with the common security definitions. As foreshadowed earlier, hiding general system properties is one of the advantages of opacity, opening promising avenues for future work, such as reasoning about program obfuscation.

A. Opacity

Drawing on the original work on opacity [11, 46, 39, 14], we present a general framework for opacity in a semantics-based setting. For a predicate φ to be opaque for a command c, any environment satisfying φ must correspond to another, observably equivalent, environment that does not satisfy φ while producing observably equivalent outputs. Intuitively this states that an attacker can never be certain that φ holds given public input and output of a program.

The notion is parametric in equivalence $\overset{i}{\sim}$ on environments and relation $\overset{o}{\sim}$ on results. The intuition is that $\overset{i}{\sim}$ expresses what information the attacker can observe about environments while $\overset{o}{\sim}$ captures what parts of the result are visible to the attacker.

Definition 1 ($Op(c, \overset{i}{\sim}, \overset{o}{\sim})$)**.** *$\varphi$ is* opaque *for command c, equivalence relation $\overset{i}{\sim}$ and relation $\overset{o}{\sim}$ (written $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$) if and only if whenever $\langle c, e_1\rangle \Downarrow r_1$ and $e_1 \in \varphi$, then there exist e_2, r_2 such that $\langle c, e_2\rangle \Downarrow r_2 \wedge e_1 \overset{i}{\sim} e_2 \wedge r_1 \overset{o}{\sim} r_2 \wedge e_2 \notin \varphi$.*

This definition allows for the attacker to learn that φ does not hold, based on observing public behavior of a program. In some scenarios, it should also be kept secret that the predicate does *not* hold, i.e. an attacker should neither be able to infer that φ is not satisfied.

Note that the implication trivially holds if $e_1 \notin \varphi$, e.g the empty set is trivially opaque for any program c and any relations $\overset{i}{\sim}$ and $\overset{o}{\sim}$.

Recall the scenario of keeping secret whether the user is located in a clinic. Program 1 resolves this by outputting the same outside coordinate when the user is actually inside. Consider an initial memory m_1 where $m_1(hX) = 5$ and $m_1(hY) = 5$. In this case, the attacker observes the output $(5, 5)$. Based on the program the attacker can then infer that the user must be actually located at position $(5, 5)$: if the user had been inside the sensitive area, the observable output would have been $(100, 200)$. The following definition captures scenarios where leakage of this form is not acceptable:

Definition 2 ($SOp(c, \overset{i}{\sim}, \overset{o}{\sim})$)**.** *$\varphi$ is* symmetrically opaque *for command c and relation $\overset{o}{\sim}$ (denoted by $\varphi \in SOp(c, \overset{i}{\sim}, \overset{o}{\sim})$) if and only if $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ and $\overline{\varphi} \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$, where $\overline{\varphi}$ denotes the complement of φ.*

As demonstrated above, this condition is violated by Program 1. To achieve symmetric opacity, one solution is to output *random* coordinates outside of the sensitive area as in Program 2, discussed in detail in Section VII.

In order to reason about opacity of a predicate for a single run of a program, we also consider a single-run version of opacity:

Definition 3. *A predicate $\varphi \subseteq \mathbb{M}$ is opaque for command c, environment e_1, and result r_1 (written $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim}, e_1, r_1)$) if and only if*

$$e_1 \in \varphi \wedge \langle c, e_1\rangle \Downarrow r_1 \Rightarrow$$
$$\exists e_2, r_2. \langle c, e_2\rangle \Downarrow r_2 \wedge e_1 \overset{i}{\sim} e_2 \wedge r_1 \overset{o}{\sim} r_2 \wedge e_2 \notin \varphi$$

This corresponds to the intuition of opacity: When an attacker sees one particular run, he cannot infer whether a predicate holds, since there is another run not satisfying the predicate producing the same observations. Moreover, the following lemma substantiates this intuition:

Lemma 1. *$\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ if and only if $\forall e_1, r_1. \varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim}, e_1, r_1)$.*

All proofs have been formalized in Isabelle/HOL and can be found online[1], along with an extended version containing pen-and-paper proofs.

Additionally, the following properties connect opacity to various logical operations:

Lemma 2. *If $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ or $\psi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$, then $\varphi \cap \psi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$.*

Lemma 3. *If $\varphi \cup \psi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$, then $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ and $\psi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$.*

[1] http://www.cse.chalmers.se/~schoepe/opacity

The reverse implications do not hold in general: Consider $\psi = \overline{\varphi}$. Since $\varphi \cup \overline{\varphi}$ cannot be opaque, the converse of Lemma 3 does not hold if φ and $\overline{\varphi}$ are opaque. Similarly, a counter-example for the converse of Lemma 2 can be constructed by choosing φ and ψ such that $\varphi \cap \psi = \emptyset$ with φ or ψ opaque.

Moreover, Lemma 3 can be used to extend enforcement mechanisms to several predicates at once, albeit at the loss of precision.

B. Noninterference

We now present a definition of noninterference [29], also parametric in equivalence $\overset{i}{\sim}$ on environments and relation $\overset{o}{\sim}$ on results.

Definition 4 ($NI(\overset{i}{\sim}, \overset{o}{\sim})$)**.** *$c$ is* noninterferent *for equivalence $\overset{i}{\sim}$ and relation $\overset{o}{\sim}$ (written $c \in NI(\overset{i}{\sim}, \overset{o}{\sim})$) if and only if whenever $\langle c, e_1\rangle \Downarrow r_1$ and $e_1 \overset{i}{\sim} e_2$ then there exists a result r_2 such that $\langle c, e_2\rangle \Downarrow r_2$ and $r_1 \overset{o}{\sim} r_2$.*

This definition of noninterference is sufficiently general to capture different flavors of noninterference commonly considered in literature. By varying the relation $\overset{o}{\sim}$ we can obtain *termination-insensitive* [63] and *termination-sensitive* [64] notions of noninterference in for batch-job programs (Section III) as well as *progress-insensitive* [2, 4, 10], *progress-sensitive* [4], and *timing-sensitive* [55] noninterference for I/O programs (Section IV).

By varying $\overset{i}{\sim}$, we can express both baseline noninterference and delimited release [54], a notion of declassification (Section V). The results for declassification apply to both batch-job programs and I/O programs.

Recall that noninterference prohibits the example programs (Program 1, Program 2, Program 3, and Program 4), although they are intuitively secure.

C. Main Lemma

We connect noninterference and opacity by identifying a set of predicates that need to be opaque for a program to satisfy noninterference.

Intuitively, noninterference holds when no secret is leaked through the attacker's observations of a program run. Conversely, opacity for one predicate allows leaks of any kind *except* leaking whether the predicate in question is satisfied. This leads to the following connection: A program satisfies noninterference if and only if it is opaque for "almost all" predicates.

We need to restrict the set of predicates under consideration since a predicate referring only to the actual observations, instead of secrets, can never be opaque: any execution resulting in the same observations will result in either both or none of the runs satisfying the predicate.

As an example, consider the predicate on memories $\varphi_l = \{m | m(l) = 1\}$. If a memory m_1 satisfies φ_l, then $m_1(l) = 1$. Hence any other memory m_2 for which $m_1 \overset{i}{\sim} m_2$ holds (i.e. is indistinguishable to the attacker from m_1), will also satisfy $m_2(l) = 1$ and therefore $m_2 \in \varphi$. Hence this predicate can never be opaque, not even for a program producing no attacker-visible results (e.g. a program consisting solely of the command **skip**). Predicates of this form must be ruled out to obtain a connection between noninterference and opacity.

We can express this by requiring that for every class of equivalent environments, the predicate must be violated for at least some memory in that class. Otherwise it cannot possibly state information about secrets, since, within such an equivalence class, all the secrets can vary arbitrarily.

Definition 5 ($\ker(\overset{i}{\sim})$)**.** *The* kernel *of an equivalence relation $\overset{i}{\sim}$ is the set of equivalence classes of environments with respect to the relation: $\ker(\overset{i}{\sim}) = \mathbb{E} \;/\; \overset{i}{\sim}$. The equivalence class of an environment e is denoted by $[e]_{\overset{i}{\sim}}$.*

We say a predicate is informative if it refers to secrets; formally, a predicate φ is informative wrt. some equivalence class for observations on environments E if and only if $\varphi \subsetneq E$.

We can now provide a precise connection between noninterference and opacity, building on the intuition sketched above.

Lemma 4. *For every equivalence $\overset{i}{\sim}$ and any reflexive relation $\overset{o}{\sim}$, the following holds:*

$$c \in NI(\overset{i}{\sim}, \overset{o}{\sim}) \Leftrightarrow \forall E \in \ker(\overset{i}{\sim}).\forall \varphi \subsetneq E.\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$$

Alternatively, the connection can be phrased in terms of predicates that span several equivalence classes, but do not fully subsume any equivalence class:

Lemma 5. *For every equivalence $\overset{i}{\sim}$ and any reflexive relation $\overset{o}{\sim}$, it holds that $c \in NI(\overset{i}{\sim}, \overset{o}{\sim})$ if and only if*

$$\forall \varphi.(\forall E \in \ker(\overset{i}{\sim}).\varphi \cap E \subsetneq E) \Rightarrow \varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$$

Intuitively, each predicate required to be opaque in the lemma can be seen as one possible leak of information via the public behavior of the program. If all such predicates are opaque, no such leak is present in the program and hence the program is noninterferent. To illustrate the lemma further, consider the program **if** $(h > 0)$ { **out L** 1 } **else** { **out L** 2 }, leaking one bit of information about a secret variable h using an *if*-statement.

This program is clearly *not* noninterferent, since the output produced depends on whether $h > 0$ holds. Moreover, the predicate $\{m | m(h) > 0\}$ is *not* opaque: For an initial memory m_1 where $m_1(h) > 0$, there is no corresponding memory $m_2 \notin \varphi$ producing the trace $\langle 1\rangle$.

Moreover, the following lemma establishes that opacity for a smaller set already implies opacity for the set of predicates in Lemma 4:

Lemma 6. $\forall E \in \ker(\overset{i}{\sim}).\forall e \in E.(E \setminus \{e\}) \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ *if and only if* $\forall E \in \ker(\overset{i}{\sim}).\forall \varphi \subsetneq E.\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$.

Intuitively, for every informative predicate $\varphi \subsetneq E$ for equivalence class E, there must exist some environment e that does not satisfy φ. If the set $E \setminus \{e\}$ is then opaque, this must yield an equivalent execution starting in environment e, since it is the only environment in this equivalence class not satisfying $E \setminus \{e\}$.

D. Knowledge-based Characterization

To give further insight into the connection between noninterference and opacity, we also relate opacity and knowledge-based formulations of the attacker's uncertainty [36, 22, 3].

The basic intuition of the knowledge-based approach is to characterize what the attacker can infer about initial memories in terms of the sets of memories that are possible after seeing a certain result (and the initial values of low variables). We define the set of possible initial memories given some environment e_0 as well as the set of possible initial memories after observing some result.

Definition 6. *We define the set of knowledge given an initial environment* e_0 *and an equivalence* $\overset{i}{\sim}$ *on environments as* $k(\overset{i}{\sim}, e_0) = \{e | e \overset{i}{\sim} e_0\}$.

The knowledge set after observing a run of $\langle c, e_0\rangle$ *producing result* r_0 *is defined by* $k(c, \overset{i}{\sim}, \overset{o}{\sim}, e_0, r_0) = \{e | \exists r.e \overset{i}{\sim} e_0 \wedge \langle c, e\rangle \Downarrow r \wedge r \overset{o}{\sim} r_0\}$.

In terms of knowledge, opacity then holds if there is always a possible initial memory not satisfying φ.

Lemma 7. $\varphi \in Op(c, \overset{i}{\sim}, \overset{o}{\sim})$ *iff* $\forall m_0.\forall r_0 \in \mathbb{R}(c, e_0).k(c, \overset{i}{\sim}, \overset{o}{\sim}, e_0, r_0) \cap \overline{\varphi} \neq \emptyset$.

where $\mathbb{R}(c, e_0) = \{r_0 | \langle c, e_0\rangle \Downarrow r_0\}$.

Intuitively this states that for a predicate to be opaque, the attacker can gain more knowledge as long as he cannot rule out that the predicate is still not satisfied.

III. Batch-job Programs

To demonstrate the applicability of our definitions to settings commonly considered in the literature, we first show how we can capture batch-job programs. To do so we define the set of possible results of a program as either a final memory state or $\bot$ to indicate divergence.

As is common, assume a *lattice* of security levels $(\mathcal{L}, \sqsubseteq, \sqcup)$. An attacker at level ℓ can see information at all levels ℓ' as long as $\ell' \sqsubseteq \ell$.

Instantiation 1. *Let* $\mathbb{E} = \mathbb{M}$ *where* $\mathbb{M} = Var \to Val$ *for some set of variables* Var *and set of possible values* Val. *We assume that there is a security level* $\Gamma(x) \in \mathcal{L}$ *associated with each variable* $x \in Var$. *Let* $\mathbb{R} = \mathbb{M} \cup \{\bot\}$.

This characterization then allows defining both termination-insensitive and termination-sensitive variants of noninterference by defining appropriate relations on $\mathbb{R}$ and $\mathbb{E}$.

We first define equivalence between initial memory states. Two memories are equivalent for the attacker if they coincide on all variables visible to the attacker:

Definition 7. $m_1 =_\ell m_2$ *iff* $\forall x.\Gamma(x) \sqsubseteq \ell \Rightarrow m_1(x) = m_2(x)$.

In the *termination-insensitive* [63] setting, termination is not observable to the attacker. Therefore, if either of the executions diverges, the result is indistinguishable to the attacker. If both executions terminate, they must do so with equal values for observable variables.

Definition 8 ($\sim_\ell$)**.** $m_1 \sim_\ell m_2$ *iff* $(m_1 = \bot) \vee (m_2 = \bot) \vee (m_1 =_\ell m_2)$.

This allows defining termination-insensitive noninterference as follows:

Definition 9 ($TINI(\ell)$)**.** *A program* c *is termination-*insensitively *noninterferent for level* ℓ *(written* $c \in TINI(\ell)$*) iff* $c \in NI(=_\ell, \sim_\ell)$.

In the *termination-sensitive* [64] setting, the attacker is assumed to be able to observe whether or not a program terminates. Hence, two final memory states are indistinguishable to the attacker if and only if they either both diverge, or they coincide on variables visible to the attacker.

Definition 10 ($\approx_\ell$)**.** $m_1 \approx_\ell m_2$ *iff* $(m_1 = \bot \wedge m_2 = \bot) \vee (m_1 =_\ell m_2)$.

By instantiating $\overset{i}{\sim}$ in Definition 4 to $\approx_\ell$, we obtain termination-sensitive noninterference:

Definition 11 ($TSNI(\ell)$)**.** *A program* c *is termination-*sensitively *noninterferent for level* ℓ *(written* $c \in TSNI(\ell)$*) iff* $c \in NI(=_\ell, \approx_\ell)$.

To illustrate the difference between the two notions in the context of opacity, consider $\varphi_\bot = \{m \mid m(h) \neq 1\}$ and a program $c_\bot =$ **if** $(h == 1)$ { *loop* } **else** { **skip** }. Since information is only leaked via termination, it holds that $\varphi_\bot \in Op(c_\bot, =_\ell, \sim_\ell)$ and $c_\bot \in TINI(\ell)$, but $\varphi_\bot \notin Op(c_\bot, =_\ell, \approx_\ell)$ and $c_\bot \notin TSNI(\ell)$.

Noninterference and Opacity: Theorems connecting noninterference (without any declassification) and opacity can then be obtained by instantiating $\overset{i}{\sim}$ with $=_\ell$ in Lemma 4:

Theorem 1 (Opacity and noninterference)**.** *Termination-(in)sensitive noninterference holds iff all informative predicates are termination-(in)sensitively opaque:*

$$c \in TINI(\ell) \Leftrightarrow \forall M \in \ker(=_\ell).\forall \varphi \subsetneq M.\varphi \in Op(c, =_\ell, \sim_\ell)$$
$$c \in TSNI(\ell) \Leftrightarrow \forall M \in \ker(=_\ell).\forall \varphi \subsetneq M.\varphi \in Op(c, =_\ell, \approx_\ell)$$

IV. I/O Programs

In an interactive setting, instead of assuming that the attacker can only see the final memory state of a program, they see a sequence of output events of the program. Inputs are modeled as memory states, a common simplification [2], similar to modeling environments as streams. Clark and Hunt [17] show that for the security of deterministic programs it makes no difference whether the environments are modeled as streams or strategies.

Instantiation 2. *Let $\mathbb{E} = \mathbb{M}$ and $\mathbb{R} = \mathbb{O}^*$ for some set of output events $\mathbb{O}$. We assume that each output event $o \in \mathbb{O}$ has an associated security level $\Gamma(o) \in \mathcal{L}$. We denote the empty trace by $\langle\rangle$ and appending trace t to output event o by $o.t$.*

Using this type of outputs we can reason about both *progress-insensitive* [2, 4, 10], *progress-sensitive* [4], and *timing-sensitive* [55] notions of noninterference by suitably choosing the relation $\overset{o}{\sim}$ in $NI(\overset{i}{\sim}, \overset{o}{\sim})$.

Progress-*insensitive* noninterference assumes that the attacker can not distinguish between the program silently diverging or the program producing more outputs in the future. Hence, for two traces to be progress-insensitively indistinguishable, they either have to coincide on low events or one of the traces has to diverge silently after producing a matching prefix of the other trace.

Definition 12. *We define the projection $t \restriction_\ell$ of trace t to level ℓ recursively by $\langle\rangle \restriction_\ell = \langle\rangle$ and $o.t \restriction_\ell = \begin{cases} o.(t \restriction_\ell) & \Gamma(o) \sqsubseteq \ell \\ t \restriction_\ell & \Gamma(o) \not\sqsubseteq \ell \end{cases}$*

Definition 13. $t_1 \sim_\ell t_2$ *iff* $(\exists t_1', t_1''.t_1' \restriction_\ell = t_2 \restriction_\ell \wedge t_1 = t_1'.t_1'') \vee (\exists t_2', t_2''.t_1 \restriction_\ell = t_2' \restriction_\ell \wedge t_2 = t_2'.t_2'')$

This allows us to define progress-insensitive noninterference using the generic notion of noninterference introduced in Section II.

Definition 14 ($PINI(\ell)$)**.** *A command c is* progress-insensitively *noninterferent for level ℓ (written $c \in PINI(\ell)$) iff $c \in NI(=_\ell, \sim_\ell)$.*

Progress-*sensitive* noninterference assumes that silent divergence is observable, so two traces have to match on all their low events:

Definition 15 ($\approx_\ell$)**.** $t_1 \approx_\ell t_2$ *iff* $t_1 \restriction_\ell = t_2 \restriction_\ell$.

As before, we define progress-sensitive noninterference by instantiating $\overset{o}{\sim}$ with $\approx_\ell$:

Definition 16 ($PSNI(\ell)$)**.** *A command c is* progress-sensitively *noninterferent for level ℓ (written $c \in PSNI(\ell)$) iff $c \in NI(=_\ell, \approx_\ell)$.*

A further strengthening of this property can be obtained by assuming that the attacker also observes the occurrence of *some* high-security event but without knowing which exact event it is:

Definition 17 ($\approxeq_\ell$)**.** $t_1 \approxeq_\ell t_2$ *is defined inductively by the following rules*

$$\frac{}{\langle\rangle \approxeq \langle\rangle} \qquad \frac{\Gamma(o_1) \sqsubseteq \ell \vee \Gamma(o_2) \sqsubseteq \ell \Rightarrow o_1 = o_2 \qquad t_1 \approxeq t_2}{o_1.t_1 \approxeq o_2.t_2}$$

Definition 18 ($TimeNI(\ell)$)**.** *A command c is* timing-sensitively *noninterferent for level ℓ (written $c \in TimeNI(\ell)$) iff $c \in NI(=_\ell, \approxeq_\ell)$.*

Time can be modeled using this definition by associating each computation with one or more $tick$ events that model passing of time. In such a setting, computations where execution time depends on a secret, will not be noninterferent.

Noninterference and Opacity: We can now use Lemma 4 to obtain a connection between the presented forms of noninterference and opacity:

Theorem 2 (Opacity and noninterference)**.** *For $\sim \in \{\sim_\ell, \approx_\ell, \approxeq_\ell\}$, a program is noninterferent wrt. $\sim$ if all informative predicates are opaque for $\sim$:*

$$c \in PINI(\ell) \Leftrightarrow \forall M \in \ker(\overset{i}{\sim}).\forall \varphi \subsetneq M.\varphi \in Op(c, =_\ell, \sim_\ell)$$

$$c \in PSNI(\ell) \Leftrightarrow \forall M \in \ker(\overset{i}{\sim}).\forall \varphi \subsetneq M.\varphi \in Op(c, =_\ell, \approx_\ell)$$

$$c \in TimeNI(\ell) \Leftrightarrow \forall M \in \ker(\overset{i}{\sim}).\forall \varphi \subsetneq M.\varphi \in Op(c, =_\ell, \approxeq_\ell)$$

V. Information Release vs. Information Hiding

This section investigates the relation of opacity to the *what* dimension of declassification [58], which, as mentioned in Section I, is most closely related when the focus is on protecting sensitive input.

As foreshadowed earlier, *partial release* [19, 41, 57, 54, 28, 43] policies specify what is released by splitting the domain of secrets into subdomains and only protecting secret variation within the subdomains. A convenient mechanism to specify partial release is via *escape hatch expressions* [54] that, intuitively, states that two initial environments are indistinguishable if and only if they agree on the values of the escape hatch expressions.

The accompanying policy of *delimited release* [54] specifies partial release by allowing to lower the security level of some expression, usually containing high variables, while prohibiting any other leaks beyond what is revealed by the declassified expression itself. Concretely, such policies can be expressed by adding expressions of the form **declassify**(e) to the language in question.

Delimited release can be obtained from Definition 4 by suitably instantiating $\overset{i}{\sim}$. To do so, we strengthen $=_\ell$ by requiring that two memory states not only coincide on observable variables, but also on the values of declassified expressions.

Definition 19 ($=_\ell^E$)**.** *Two memories m_1, m_2 are low equivalent for level ℓ and set of declassified expressions E iff*

$m_1 =_\ell m_2 \wedge (\forall e \in E. m_1(e) = m_2(e))$ *where* $m(e)$ *denotes evaluating expression* e *in memory* m*.*

Definition 20 ($DR(\overset{o}{\sim}, \ell, E)$)**.** *A program* c *satisfies* delimited release *for level* ℓ*, relation* $\overset{o}{\sim}$*, and declassified expressions* E *(written* $c \in DR(\overset{o}{\sim}, \ell, E)$*) iff* $c \in NI(=^E_\ell, \overset{o}{\sim})$*.*

The intuition behind this definition is that resulting memory states only need to look the same to the attacker the declassified expressions also have equal values.

Lemma 4 also allows deriving results connecting delimited release and noninterference.

Theorem 3 (Opacity and delimited release)**.** *For any reflexive relation* $\sim$*, we have* $c \in DR(\overset{o}{\sim}, \ell, E)$ *if and only if*

$$\forall M \in \ker(=^E_\ell). \forall \varphi \subsetneq M. \varphi \in Op(c, =^E_\ell, \overset{o}{\sim})$$

In the batch setting this theorem covers both termination-sensitive and termination-insensitive noninterference. In the I/O setting, progress-sensitive, progress-insensitive, and timing-sensitive notions of noninterference are captured by the theorem statement.

VI. Enforcement

A. Example Language

To illustrate our enforcement techniques, we introduce a simple *while*-language with arrays and output. The command ε denotes termination. Evaluation of a configuration $\langle c, m\rangle$ in one step to $\langle c', m'\rangle$ while producing trace t is denoted by $\langle c, m\rangle \xrightarrow{t} \langle c', m'\rangle$. $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$. We denote evaluation of an expression e in memory m by $m(e)$. Semantically, an array is modeled as a function mapping natural numbers to values. Program execution in this language is deterministic. The full definition and semantics of the language can be found in the extended version of the paper.

Without loss of generality, we consider only two security levels, **L** for public data and **H** for private data.

For the rest of this section, we instantiate the evaluation relation in the definitions of Section II as follows:

Instantiation 3. *Let* $\mathbb{R} = (\mathcal{L} \times \mathit{Val})^*$ *and* $\langle c, m\rangle \Downarrow t$ *iff* $\langle c, m\rangle \xrightarrow{t}{}^* \langle c', m'\rangle \wedge (\forall c'', m'', t'. \langle c', m'\rangle \xrightarrow{t'}{}^* \langle c'', m''\rangle \Rightarrow t' = \langle\rangle)$ *for some* m'*.*

B. Dynamic Monitoring

We present a dynamic enforcement mechanism inspired by common dynamic monitoring techniques for enforcing noninterference [24, 56, 35] and using concolic execution [59] for enforcing security properties [1].

The intuition is to keep track of the set of memories producing the same trace. Initially we consider the set $[m_1]_{\underset{\sim}{\iota}} \cap \overline{\varphi}$. This set becomes smaller over the course of execution, since outputs of the real execution need to be matched by another run. At each step of the program we check if the current set of possible starting memories is empty and, if so, stop execution before performing the next evaluation step.

To keep track of this set, we define a function τ computing this set for the next step of a configuration along with keeping track of dependencies of variables, given starting memory m_1 and the set so far. $\tau(m_1, \langle c, m\rangle, M, \delta)$ produces a pair (M', δ') where M' is a subset of memories in M producing the same trace as m_1 for the next step that $\langle c, m\rangle$ takes.

In order to calculate which initial memories result in the same trace during an execution, we symbolically keep track of how variables change during the execution of a program. Concretely, we express the value of a variable at a certain point during execution in terms of variables in the initial state. For example, when executing two steps in the program $x := y; x := x * 2$, we record the value of x as $y * 2$.

This is achieved by extending configurations with a function $\delta : \mathit{Var} \cup (\mathit{Var} \times \mathbb{N}) \rightarrow \mathbb{E} \times \mathcal{P}(\mathbb{E})$ that keeps track of variable dependencies in the following way: Evaluating the first component of $\delta(x)$ in m_1 yields the same result as $m_1'(x)$. The second component of $\delta(x)$ records which array-lookups the value of x depends on. Moreover, for any memory m_2 resulting in the same path through the program, it holds that the value of $m_2'(x)$ coincides with the first component of $\delta(x)$, provided that the same array lookups have been performed.

Similarly, δ keeps track of the dependencies of array elements $a[i]$. These notions are made precise by Lemma 8.

We extend δ to an arbitrary expression e by replacing all variables and array lookups occurring in e with their values in δ. We denote this extension of δ applied to e by $\delta\langle e, m\rangle$. Since our approach is value-sensitive wrt. the array indices, the dependencies of an expression e also depend on the current memory m.

Definition 21. *We define* $\delta\langle e, m\rangle \in \mathbb{E} \times \mathcal{P}(\mathbb{E})$ *for expression* e *and memory* m *recursively as follows:*

$$\delta\langle n, m\rangle = (n, \emptyset)$$
$$\delta\langle \textbf{\textit{true}}, m\rangle = (\textbf{\textit{true}}, \emptyset)$$
$$\delta\langle \textbf{\textit{false}}, m\rangle = (\textbf{\textit{false}}, \emptyset)$$
$$\delta\langle x, m\rangle = \delta(x)$$
$$\delta\langle a[e_i], m\rangle = (e', \{e_i'\} \cup E_i \cup E)$$
$$\quad \textit{where } (e', E) = \delta(a[m(e)]) \textit{ and } (e_i', E_i) = \delta\langle e_i, m\rangle$$
$$\delta\langle e_1 \otimes e_2, m\rangle = (e_1' \otimes e_2', E_1 \cup E_2)$$
$$\quad \textit{where } (e_1', E_1) = \delta\langle e_1, m\rangle \textit{ and } (e_2', E_2) = \delta\langle e_2, m\rangle$$
$$\delta\langle (e_1, e_2), m\rangle = ((e_1', e_2'), E_1 \cup E_2)$$
$$\quad \textit{where } (e_1', E_1) = \delta\langle e_1, m\rangle \textit{ and } (e_2', E_2) = \delta\langle e_2, m\rangle$$
$$\delta\langle \textbf{\textit{fst}}(e), m\rangle = (\textbf{\textit{fst}}(e'), E) \textit{ where } (e', E) = \delta\langle e, m\rangle$$
$$\delta\langle \textbf{\textit{snd}}(e), m\rangle = (\textbf{\textit{snd}}(e'), E) \textit{ where } (e', E) = \delta\langle e, m\rangle$$

$$\delta\langle !e, m\rangle = (!e', E)$$
$$\quad \textit{where } (e', E) = \delta\langle e, m\rangle$$
$$\delta\langle e \ ? \ e_1 \ : \ e_2, m\rangle = (e' \ ? \ e_1' \ : \ e_2', E \cup E_1 \cup E_2)$$
$$\quad \textit{where } (e', E) = \delta\langle e, m\rangle, \ (e_1', E_1) = \delta\langle e_1, m\rangle$$
$$\quad \textit{and } (e_2', E_2) = \delta\langle e_2, m\rangle$$

where $\otimes \in \{+, -, \times, \leq, \geq, \&\&, \|\}$.

Definition 22. *τ is defined by:*

$$\tau(m_1, \langle \varepsilon, m\rangle, M, \delta) = (M, \delta)$$
$$\tau(m_1, \langle \textbf{\textit{skip}}, m\rangle, M, \delta) = (M, \delta)$$
$$\tau(m_1, \langle x := e, m\rangle, M, \delta) = (M, \delta[x \mapsto \delta\langle e, m\rangle])$$
$$\tau(m_1, \langle a[e_1] := e_2, m\rangle, M, \delta) =$$
$$\quad (M \cap [m_1]_{\delta\langle e_1, m\rangle}, \delta[a[m(e_1)] \mapsto \delta\langle e_2, m\rangle])$$
$$\tau(m_1, \langle \textbf{\textit{out}} \ \ell' \ e, m\rangle, M, \delta) = \begin{cases} (M, \delta) & \ell' \not\sqsubseteq \ell \\ (M \cap [m_1]_{\delta\langle e, m\rangle}, \delta) & \ell' \sqsubseteq \ell \end{cases}$$
$$\tau(m_1, \langle \textbf{\textit{if}} \ e \ \{ \ c_1 \ \} \ \textbf{\textit{else}} \ \{ \ c_2 \ \}, m\rangle, M, \delta) =$$
$$\quad (M \cap [m_1]_{\delta\langle e, m\rangle}, \delta)$$
$$\tau(m_1, \langle \textbf{\textit{while}} \ e \ \textbf{\textit{do}} \ c, m\rangle, M, \delta) = (M \cap [m_1]_{\delta\langle e, m\rangle}, \delta)$$
$$\tau(m_1, \langle c_1; c_2, m\rangle, M, \delta) = \tau(m_1, c_1, M, \delta)$$

where

$$[m_1]_{(e,E)} = \{m_2 | m_1(e) = m_2(e) \wedge (\forall e' \in E. m_1(e') = m_2(e'))\}$$

For every case except assignments and array assignments, the mapping from variables to their dependencies is left unchanged. Producing an event on a public channel will constrain the set of possible starting memories to memories producing the same output for the expression.

Control-flow instructions, i.e. while and if statements are forced to take the same branches by allowing only memories in which the guard evaluates to the same value. Note that this still allows branching on low variables, as variables with low security levels are required to be equal to m_1 in the set of considered memories anyway.

We then define the monitor via an instrumented semantics (for a fixed m_1) in Figure 2: As common for dynamic monitoring for information flow properties [24, 56], we do not inspect branches of conditionals or loops that are not taken during the run that is being monitored. Therefore, our definition of τ ensures that the set of possible starting memories not satisfying φ takes the same branches as the run under consideration. Section VI-C describes an approach that avoids this loss of precision at the cost of an increased performance overhead.

The following lemma makes the connection between the instrumented semantics and soundness precise:

ENF-EVAL

$$\frac{\tau(m_1, \langle c, m\rangle, M, \delta) = (M', \delta') \qquad \langle c, m\rangle \xrightarrow{t} \langle c', m'\rangle \qquad M' \neq \emptyset \qquad (\forall c_1, c_2. c_1 \neq \varepsilon \Rightarrow c \neq c_1; c_2)}{\langle c, m, M, \delta\rangle \hookrightarrow^{t} \langle c', m', M', \delta'\rangle}$$

ENF-SEQ

$$\frac{\langle c_1, m, M, \delta\rangle \hookrightarrow^{t} \langle c_1', m', M', \delta'\rangle}{\langle c_1; c_2, m, M, \delta\rangle \hookrightarrow^{t} \langle c_1'; c_2, m', M', \delta'\rangle}$$

Figure 2. Instrumented semantics for monitoring

Lemma 8. *Whenever $\langle c, m_1, M, \delta_0\rangle \overset{t_1}{\hookrightarrow}^{*} \langle c', m_1', M', \delta'\rangle$ and $m_2 \in M'$, then there exist t_2, m_2' such that:*

$$\langle c, m_2\rangle \overset{t_2}{\rightarrow}^{*} \langle c', m_2'\rangle \wedge t_1 \approx_L t_2$$

and

$$\forall e \in \mathbb{E}. m_1(\pi_1(\delta\langle e, m_1'\rangle)) = m_1'(e)$$

and

$$\forall e \in \mathbb{E}. m_1 =_{\pi_2(\delta\langle e, m_1'\rangle)} m_2 \Rightarrow m_2(\pi_1(\delta\langle e, m_1'\rangle)) = m_2'(e)$$

where $\delta_0(x) = (x, \emptyset)$ and $\delta_0(a[i]) = (a[i], \emptyset)$ and π_1 and π_2 denote the first and second projections of a tuple.

This allows establishing the soundness of the presented enforcement technique.

Theorem 4 (Soundness of monitoring)**.** *If $\langle c, m_1, [m_1]_{\overset{i}{\sim}} \cap \overline{\varphi}, \delta_0\rangle \overset{t_1}{\hookrightarrow}^{*} \langle \varepsilon, m_1', M', \delta'\rangle$, then $\varphi \in Op(c, =_L, \approx_L, m_1, t_1)$.*

Similar to other dynamic enforcement mechanisms [56, 35], our soundness targets progress-insensitive security of monitored runs:

Theorem 5. *If $\langle c, m_1, [m_1]_{\overset{i}{\sim}} \cap \overline{\varphi}, \delta_0\rangle \overset{t_1}{\hookrightarrow}^{*} \langle c_1', m_1', M', \delta'\rangle$, $\langle c, m_2\rangle \overset{t_2}{\rightarrow}^{*} \langle c_2', m_2'\rangle$, $m_2 \in M'$, then $t_1 \sim_L t_2$, $m_1 =_L m_2$, and $m_2 \notin \varphi$.*

C. Sampling-based Enforcement

Being a fine-grained policy, opacity is not, in general, preserved by sequential composition of programs. In comparison, progress-sensitive noninterference is known to be compositional while progress-insensitive is not [51].

Consider predicate $\varphi_{seq} = \{m | m(h) \neq 5 \wedge m(h) \neq 6\}$ and programs $c_1 =$ (**if** $(h == 5 \ \| \ h == 4)$ { **out L** 1; } **else** { **out L** 2; }) and $c_2 =$ (**if** $(h == 6 \ \| \ h == 4)$ { **out L** 3; } **else** { **out L** 4; }). Both c_1 and c_2 are opaque for φ_{seq}: For c_1 and a memory $m_1 \in \varphi_{seq}$, we can match the trace by a memory $m_2 \notin \varphi_{seq}$ where $m_2(h) = 6$ if $m_1(h) \neq 4$ and $m_2(h) = 5$ otherwise.

Analogously we can satisfy opacity for c_2. Moreover, notice that φ_{seq} also satisfies *symmetric* opacity for both c_1 and c_2.

However, neither φ_{seq} nor $\overline{\varphi_{seq}}$ is opaque for the composition $c_1; c_2$: If $m_1 \in \varphi_{seq}$ with $m_1(h) \neq 4$, then the trace $t_1 = \langle 2, 4\rangle$ is produced. In order for a memory $m_2 \notin \varphi_{seq}$ to match the output 2, $m_2(h) = 6$ has to hold. For the output 4 to be matched, one needs to set $m_2(h) = 5$. Hence φ_{seq} is not opaque. Similarly if $m_1 \notin \varphi$, matching the traces produced requires both $m_2(h) = 4$ and $m_2(h) \neq 4$.

The lack of compositionality motivates us to propose a *blackbox* randomized procedure to detect whether for a given initial memory m_1 satisfying a predicate φ, there exists an equivalent run starting in a memory $m_2 \notin \varphi$. A high-level description of the algorithm is displayed below. This is a blackbox approach because the program code is not inspected, but certain outputs in the trace are used for the heuristics for the sampling process, detailed in Section VII.

Input: Program c, predicate φ, initial memory m_1, where $m_1 \in \varphi$.

Parameter: Sampling function $\mathcal{S} : \mathbb{C} \times \mathbb{M} \to \mathcal{P}([m_1]_{=_\ell})$. We assume that for all c and m_1 that $\mathcal{S}(c, m_1)$ is finite.

Output: Memory m_2 and t_2 satisfying $\langle c, m_2\rangle \Downarrow t_2$, $m_1 \overset{i}{\sim} m_2$, $t_1 \overset{o}{\sim} t_2$, and $m_2 \notin \varphi$.

for $m_2 \in \mathcal{S}(c, m_1)$ **do**
  if $[\![\langle c, m_1\rangle]\!] \overset{o}{\sim} [\![\langle c, m_2\rangle]\!] \wedge m_2 \notin \varphi$ **then**
    return $(m_2, [\![\langle c, m_2\rangle]\!])$
  end
end

Where $[\![\langle c, m\rangle]\!]$ denotes the trace produced by executing c starting in memory m. Note that this trace is uniquely defined since the language is deterministic.

The advantage of this approach over the dynamic monitoring technique described in Section VI-B is that it does not force all runs that are being considered to take the same branches for if-statements and execute while-loops the same number of times. However, this incurs a performance overhead as the program is executed multiple times. In our implementation, $\mathcal{S}$ is constructed using the random testing tool QuickCheck [16], as detailed in Section VII.

The following theorem establishes the soundness of this approach:

Theorem 6 (Soundness of Sampling-based Enforcement)**.** *If the above algorithm returns a pair (m_2, t_2), for initial memory m_1 resulting in trace t_1, then $\varphi \in Op(c, =_\ell, \overset{o}{\sim}, m_1, t_1)$.*

VII. Experiments

To demonstrate the practicality of the enforcement, we implement both the monitoring sampling-based approaches from Sections VI-B and VI-C in Haskell using BNFC [9] for parser generation. The source code is also available online.

To provide more realistic examples we add a facility to generate random numbers to the example language presented in Section VI-A. Statement **randomize**(x) assigns a random number to variable x. Note that this can be emulated by instead computing the outputs of a deterministic pseudo-random number generator programmatically, based on a private variable h_{seed} with $\Gamma(h_{seed}) \not\sqsubseteq \ell$, which is then updated after each use of the **randomize**(x) statement. Since we assume randomness to be unpredictable, we require that h_{seed} not occur in predicates.

For simplicity, we specify predicates as expressions in the language from Section VI-A, sufficient to express all predicates in the examples. The enforcement approaches are applicable to more complex languages for expressing predicates.

A. Location Privacy

We apply our enforcement to enforcing location privacy code (with the exception of Program 1, which is subsumed by Program 2).

1) Dynamic Monitoring.: To implement the sets of possible memories that are possible at each point, we collect constraints created by execution steps of a program and the predicate. We utilize state-of-the-art SMT solvers (Z3 [20] and CVC4 [7]) to ensure that the set of constraints is satisfiable, thereby showing that the set of memories producing the same trace is nonempty. We use multiple SMT solvers to cover cases where one solver might not be able to solve a particular problem. In the cases we examined however, both solvers were able to handle the generated formulas with Z3 often being faster.

As common for purely dynamic enforcement [56, 35] we do not inspect the branches not taken, which makes a difference for Program 2 with the clinic scenario for the monitoring and sampling techniques. We will see that the program is susceptible to sampling-based enforcement while monitoring takes the same branches of **if**-statements in the set of memories considered and hence has insufficient information to verify the program.

However, we can make this program amenable to dynamic enforcement by unconditionally computing random coordinates and then deciding which set of coordinates to output in the expression itself, as shown in Program 2a.

Note that this example also eliminates a possible timing leak introduced by performing the random number generation only if the user is located inside the clinic.

We introduce variables for the SMT solver for all variables occurring in the program and the predicate. The confidential information is whether the user is currently in the medical clinic, i.e. $\varphi_{loc} = \{m | m(hX) \geq clinicX_{min} \wedge m(hX) \leq clinicX_{max} \wedge m(hY) \geq clinicY_{min} \wedge m(hY) \leq clinicY_{max}\}$, where $clinicX_{min,max}$ refer to the start and

```
            /* Program 2a - Adapted */
   /* Location privacy with random output */
clinicXmin := 200; clinicXmax := 400;
clinicYmin := 50; clinicYmax := 150;
randomize(x);
while (x >= clinicXmin && x <= clinicXmax) {
      randomize(x);
}
randomize(y);
while (y >= clinicYmin && y <= clinicYmax) {
      randomize(y);
}
out L ((hX >= clinicXmin && hX <= clinicXmax &&
        hY >= clinicYmin && hY <= clinicYmax)
      ? (x, y)
      : (hX, hY));
```

end of the clinic location on the X-axis and $clinicY_{min,max}$ refer to the extent of the clinic location on the Y-axis.

If φ_{loc} is satisfied in the initial memory m_1, then the user is in the clinic. This fact should not be disclosed, and hence we need to find a memory m_2 such that $m_2 \notin \varphi_{loc}$. Therefore, we add $\neg(m(hX) \geq clinicX_{min} \wedge m(hX) \leq clinicX_{max} \wedge m(hY) \geq clinicY_{min} \wedge m(hY) \leq clinicY_{max})$ to the set of constraints (we assume that φ_{loc} holds in m_1, but the enforcement approach works in the same way for symmetric opacity).

Moreover, m_2 needs to coincide with m_1 on low variables, hence we require all low variables occurring in the program to be equal to their values in m_1. In this case, no low variable is used before being overwritten, so these constraints are vacuous. For brevity, we omit them here.

During evaluation of assignment statements the mapping δ is then updated with the dependencies of the variables. For a **randomize**(x) instruction, $\delta(x)$ is updated with a *fresh* random variable since the result of the random number generation is assumed to be unpredictable. In the case of an assignment $x := e$ we set the dependencies of x to $\delta\langle e, m\rangle$.

When encountering a **while**-loop with expression e as the guard, we then add the constraint that the initial memory must agree with m_1 on $\delta\langle e, m\rangle$. In particular, we generate constraints stating the guards on both loops coincide with m_1, i.e. that the random variables introduced for calls to **randomize**() result in the same number of loop iterations.

When evaluating the **out** statement we then add the constraint that the output of the run starting with m_1 has to be matched by a run starting in a memory m_2 satisfying the generated constraints.

In this case the run of the program starting in m_1 will result in the output (x, y), where x and y are randomly generated, but outside the clinic. For example, assume that (x, y) evaluates to $(23, 45)$ for m_1. Therefore we add the constraint $(23, 45) = (hX \geq 100 \wedge hX \leq 200 \ \ldots \ ? \ (r_1, r_2) \ : \ (hX, hY)$ where r_1 and r_2 are the fresh variables introduced by the **randomize**() instructions.

Whenever a constraint is added during monitoring, we translate the constraints into the syntax of the SMT solvers being used and check for satisfiability. If the solver returns that the set is satisfiable, we have found a memory producing the same trace which does not satisfy φ_{loc} and hence no information about φ_{loc} is revealed. In this particular case, the SMT solver will find a memory m_2 such that $m_2(hX) = 23$ and $m_2(hY) = 45$. r_1 and r_2 can have any coordinates outside the clinic.

Generally, we gain from SMT solvers to help with satisfying the constraints, with the tradeoff that when the set of constraints is unsatisfiable or the solver times out, then we would block the execution to prevent a possible opacity violation.

2) Sampling-based Enforcement.: Sampling-based enforcement consists of two steps: Running program c with the initial memory m_1 and trying random memories m_2 (with $m_1 \overset{i}{\sim} m_2$) to check if they produce the same trace where φ no longer holds. We use the QuickCheck tool [16] to generate random samples.

To handle randomness in our language, we add the following heuristic to our sampling-mechanism: If, during evaluation with memory m_1, we output values $v_1, \ldots, v_n$, we increase the likelihood of choosing $v_1, \ldots, v_n$ for variables $x \in vars(c)$ where $\Gamma(x) = \mathbf{H}$.

This heuristic allows us to check both the original clinic code in Program 2 (with **if**-statements to decide what to output), as well as the one adapted to dynamic monitoring.

Assuming that we start with a memory m_1 where $(m_1(hX), m_2(hY))$ is located within the medical clinic, we will output random coordinates (r_1, r_2) outside of the clinic. Using the heuristic described above, our sampling mechanism will consider memories m_2 with $m_2(hX) \in \{r_1, r_2\}$ and $m_2(hY) \in \{r_1, r_2\}$ with increased likelihood.

Hence, the presented approach finds a witness for opacity for this example after only a few tested memories.

3) Progress-sensitivity: Consider the example presented in Program 3 which does not produce any output if the user is located in a sensitive location. If the user is located outside of a sensitive area, their real coordinates are output.

This example satisfies progress-*insensitive* opacity, since the empty trace (produced if the user is in a sensitive location) is a prefix of all other traces.

However, progress-*sensitive* opacity is violated since the low output event that is generated if the user is located outside of sensitive areas cannot be matched in a run starting in a sensitive location.

Since this example relies on branching to achieve opacity, dynamic monitoring cannot be used to run this program. However, sampling-based enforcement is able to verify that the example satisfies progress-insensitive opacity, while progress-sensitive opacity correctly cannot be established.

B. Statistics Aggregation

We apply our enforcement to the healthcare statistics code in Program 4.

1) Dynamic Monitoring.: Assume a program run with initial memory m_1 results in 7. Assume the sensitive predicate $\varphi = \{m_1 \mid m_1(hHasDisease[3]) > 0\}$, i.e. whether the fourth patient is infected with a particular disease. Moreover, assume that $m_1(hHasDisease[3]) = 1$, i.e. that $m_1 \in \varphi$.

The monitor will initially add the constraints that φ must not be satisfied and that all low variables have the same values as in m_1. Among the constraints generated, when reaching the statement **out L** $result$, we add:

$$7 = 0 + (hHasDisease[0] > 0 \ ? \ 1 \ : \ 0)$$
$$+ \cdots + (hHasDisease[9] > 0 \ ? \ 1 \ : \ 0)$$

Running an SMT solver on the set of generated constraints yields that they are satisfiable: Since the sum of ten values of m_1 is 7, $m_1(hHasDisease[i]) = 0$ must hold for some $i \neq 3$. Therefore, we can satisfy the set of constraints by setting $m_2(hHasDisease[3]) = 0$ and $m_2(hHasDisease[i]) = i$ and $m_2(x) = m_1(x)$ otherwise. This memory yields the same observations and $m_2 \in \varphi$ does not hold.

Consider now instead a run with memory m_1' resulting in the output 10. In this case, the monitor will end up with an unsatisfiable set of constraints, since this implies that all patients are infected with the disease. Since the total number of patients is known to the attacker, they can infer that the fourth patient must also have this disease. Hence, this run is correctly terminated by our enforcement before the output takes place.

2) Sampling-based Enforcement.: As before, the sampling-based technique quickly finds a suitable witness for opacity due to heuristics employed by the *QuickCheck* library.

C. Discussion

The presented monitoring and sampling mechanisms offer different tradeoffs of precision and performance.

Monitoring avoids executing the program multiple times and utilizes SMT solving for problems that tend to be small, relative to the capabilities of SMT solvers. Moreover, the satisfiability only needs to be verified when encountering **out** expressions. On the other hand, programs are not allowed to branch on secrets, reducing precision. In some cases, these programs can be amended to allow for execution with the monitor, e.g. for Program 2.

Being blackbox, sampling-based enforcement is scalable to rich languages. While sampling-based enforcement is more precise, it uses heuristics for choosing an appropriate starting environment that leads to a higher rate of successful tries.

The table below summarizes the results for the two enforcement mechanisms concerning the examples from Section I. The checkmarks represent successful verification with respect to the provided indistinguishability relations.

Example	Sampling	Monitoring
Program 2	✓ (≈)	
Program 2a	✓ (≈)	✓ (≋)
Program 3	✓ (∼)	
Program 4	✓ (≈)	✓ (≋)

As a final note, our implementation is a proof-of-concept implementation, with a number of possible performance optimizations such as reducing the number of spawned processes when handling constraints. While these optimizations and scalability studies are promising directions of future work, we note that indicative performance overhead does not strike as unacceptable: sampling-based enforcement runs in a few milliseconds and dynamic monitoring within a few hundred milliseconds for each of the examples (run on an Intel i7-4600 processor using Linux 3.15.2 and compiled with GHC 7.8.3). While not insignificant, the proof-of-concept prototype can be a good fit for testing applications before deployment.

VIII. Related Work

The origins of opacity can be traced back to Sutherland's *nondeducibility* [60], with the intuition of keeping attacker-observable events consistent with possible variations of secret inputs. Nondeducibility has been criticized for failing to protect secret outputs [34] and address covert channels [66]. These criticisms have no bearing on the stream-based setting as in this paper, but need to be addressed when generalizing streams to strategies [66, 51].

Boisseau [11] and later Mazaré [46] introduce opacity in the context of cryptographic protocols.

Hughes and Shmatikov [39] develop an algebraic theory of *opaqueness* for reasoning about general knowledge functions. They present a protocol graph framework and study a hierarchy of anonymity system properties, noting that anonymity is neither necessary nor sufficient for privacy.

Bryans et al. study opacity in the setting of Petri nets [14] and in a general setting of transition systems [12, 13].

Ryan and Peacock [53] explore the relation between noninterference, noninference [50], nondeducibility [60], and nonleakage [65] in the setting of labeled transition systems expressed in CSP [37]. Although cast in a different setting and leaving out support for declassification and enforcement, the connection between opacity and noninterference is particularly relevant. They state that "formulating noninterference as opacity proves difficult, but we can show that noninterference implies opacity". Our work takes the next steps by connecting noninterference and opacity in both directions, parameterizing the results in the power of the attacker and declassification policies, and developing enforcement mechanisms.

Freni et al. [27] treat location privacy in social networks. Of particular interest is the definition of *absence privacy* to protect the fact that a user is absent at certain location points (reducing burglary risks). However, absence privacy requires

for all locations p in sensitive regions that the adversary cannot exclude that the user is located in p. With opacity, there is no need to demand excluding *all* locations: with home as a sensitive area, the fact that the user is absent in the kitchen is not dangerous when the user is present in the living room.

Although probabilistic behavior is not in the scope of our model, a worthwhile direction for further investigations is to extend it with probabilities. It would be interesting to combine it with the work by Bérard et al. [8] who study several probabilistic opacity properties in the setting of probabilistic automata.

In the context of databases, properties similar to opacity are desired for the *inference problem* [21] concerned with protecting individual data while revealing statistical aggregates. Mechanisms such as data swapping and query size control have been developed and their limitations identified. It is often possible to subvert these mechanisms by correlating the query results [23].

In an investigation of provenance security, Cheney [15] defines the *provenance obfuscation problem* for protecting database queries. The problem consists of inability of users to answers queries using their observations. A query cannot be answered if for any trace, there exists another trace with the same observation but so that the results of applying the query to the traces are different. Intuitively, this in line with symmetric opacity, although the exact relation is yet to be established by future work.

Del Tedesco et al. [61] study logical data erasure and develop a semantic hierarchy of erasure policies. When exploring the choices for ordering between knowledge sets that result from observing system behavior, they utilize a notion akin to opacity for modeling facts and queries over knowledge sets. While the main focus is on the expressiveness of erasure policies, verification of erasure policies is left for future work.

Griffis et al. [31] focus on the problem of personal data vaults to separate the capturing and sharing of data. In similar spirit to ours, they argue that common security definitions fall short of capturing location sharing policies whose granularity depends on the actual location. They propose to use *filters* to augment information release mechanisms by transforming sensitive information into coarse-grained approximations.

Gruska reasons about passive and active timing attacks in the context of opacity for timed process algebra [32] and explores opacity for both confidentiality and integrity [33] in a variant of CCS [47].

Hritcu et al. [38] utilize QuickCheck to aid the process of proving noninterference for a low-level abstract information-flow machine. The approach is directly suitable for checking unwinding conditions [30].

Wu and Lafortune [67] investigate a family of opacity policies for deterministic finite-state automata. They propose verification methods that are suitable for certifying opacity in the presence of a team of collaborating intruders.

The line of work on abstract noninterference [28, 40, 45] leads up to a powerful generalization by Mastroeni [45], formulated in a noninterference style with quantifying over pairs of runs with initial states indistinguishable by the attacker. The generalization is parametric in the indistinguishability on both inputs and outputs, which can be used for hiding differences between inputs.

Relation to declassification policies deserves discussion in the rest of the section. Let us come back to Program 1 where the goal is to make opaque whether the user is inside the hospital. Traditional declassification mechanisms require specifying what is released when the user is located in the clinic, i.e. the method of masking the user's location in that case is part of the policy. For example, declassification via escape-hatch expressions would allow expressing the policy in this scenario by downgrading the information using an expression of the form **declassify**$(hX >= clinicXmin \ldots \ ?\ (100, 200)\ :\ (hX, hY))$. From a policy standpoint however, it is unimportant how exactly the user's real location is concealed if he is located in a sensitive area, making opacity a more natural fit for this scenario.

More elaborate approaches for specifying conditional declassification policies have been proposed. Banerjee et al. [6] present an approach allowing to specify under which condition confidential information can be released while still providing delimited-release style guarantees that nothing else is leaked, using *flowspecs*. Nanevski et al. [49] propose a framework for expressing information flow policies based on dependent types. Their approach allows constructing functions that declassify information only under specified conditions.

Opacity policies however remain non-trivial to express in such frameworks. Fundamentally, the above frameworks allow expressing under which conditions output that is released to the attacker has to be equal in two runs that vary in the parts of secret input that may be released. In the case of Program 1, however, whenever the user is located outside of the clinic, the outputs of the program need not be equal between the two run. Moreover, the crucial comparison for opacity is between runs that differ on whether or not the user is located in the clinic; i.e. when the criterion for declassification is true in one run and violated in the other. Such policies are not natural to express under conditional release.

Generally, declassification deals with what *values* can be released, whereas opacity is concerned with *properties*. If properties are to be protected, it can be hard to see whether or not releasing a value will affect the attacker's knowledge about whether or not this property holds, especially if the property involves several variables.

IX. Conclusions

Driven by the research questions on understanding and enforcing opacity, we have presented a formal framework for opacity and demonstrated its differences and similarities to the common security definitions of noninterference, knowledge-based security, and information release. Our results give insight into the formal relation to the common policies, which can be achieved by quantifying over the system properties that need to be opaque. These results are parametric in the power of the attacker and formalized in Isabelle/HOL.

Our policy framework is accompanied by two enforcement strategies: a whitebox monitor and a blackbox sampling-based enforcement. We have established the soundness of the mechanisms and showed their usefulness by a prototype for the scenarios of location privacy and privacy-preserving aggregation.

Future work: A promising track for future work is exploring epistemic logic for expressing and enforcing opacity as well as applying it to reason about surreptitious code and program obfuscation [48]. The recent work on epistemic logic for information-flow security [5, 18] is a promising starting point.

In general, which properties should be opaque depends on the application and what properties of user input are desired to be protected. Weaker or stronger predicates might be appropriate, with no systematic way available a priori. Deriving opacity properties from code, to guide policy makers in tuning their policies, is an intriguing problem to investigate.

Our location privacy policies exemplify typical static policies, as commonly used in location privacy protocols [42, 62], and as used in Flickr's geofences. These policies need to be refined when disclosing location over time. For example, if a patient on the way to a hospital exposes a trajectory leading up to a geofence that surrounds the hospital, the attacker might conclude that the patient is at the hospital at a later time. Tracking location over time is a major challenge for much work on location privacy [42, 62]. Developing a sharper analysis that incorporates topological, spatial, and temporal sensitivity is much desired. We believe that opacity can be fruitfully applied to describe properties on trajectories

Acknowledgments: This work was funded by the European Community under the ProSecuToR and WebSand projects and the Swedish research agencies SSF and VR.

References

[1] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*, page 59, 2012.

[2] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS*, 2008.

[3] A. Askarov and A. Sabelfeld. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *S&P*, 2007.

[4] A. Askarov and A. Sabelfeld. Tight enforcement of information-release policies for dynamic languages. In *CSF*, 2009.

[5] M. Balliu, M. Dam, and G. L. Guernic. Epistemic temporal logic for information flow security. In *PLAS*, 2011.

[6] A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *IEEE Symposium on Security and Privacy*, pages 339–353. IEEE Computer Society, 2008.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *CAV*, 2011.

[8] B. Bérard, J. Mullins, and M. Sassolas. Quantifying opacity. In *QEST*, 2010.

[9] BNF Converter. http://bnfc.digitalgrammars.com/, 2014.

[10] A. Bohannon, B. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic. Reactive Noninterference. In *CCS*, Nov. 2009.

[11] A. Boisseau. *Abstractions pour la vérification de propriétés de sécurité de protocoles cryptographiques*. PhD thesis, École Normale Supérieure de Cachan, Sept. 2003.

[12] J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity Generalised to Transition Systems. In *FAST*, 2005.

[13] J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalised to transition systems. *Int. J. Inf. Sec.*, 2008.

[14] J. Bryans, M. Koutny, and P. Y. A. Ryan. Modelling opacity using petri nets. *Electr. Notes Theor. Comput. Sci.*, 2005.

[15] J. Cheney. A formal framework for provenance security. In *CSF*, 2011.

[16] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices*, 46(4), 2011.

[17] D. Clark and S. Hunt. Noninterference for deterministic interactive programs. In *Workshop on Formal Aspects in Security and Trust (FAST'08)*, Oct. 2008.

[18] M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *POST*, 2014.

[19] E. S. Cohen. Information transmission in sequential programs. In *Foundations of Secure Computation*. Academic Press, 1978.

[20] L. M. de Moura and N. Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[21] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[22] C. Dima, C. Enea, and R. Gramatovici. Nonde-

terministic noninterference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.
[23] C. Farkas and S. Jajodia. The inference problem: A survey. *SIGKDD Explorations*, 2002.
[24] J. S. Fenton. Memoryless subsystems. *Comput. J.*, 1974.
[25] Flickr. Flickr geoprivacy settings. http://www.flickr.com/account/geo/privacy/, 2011.
[26] R. Focardi and R. Gorrieri. Classification of security properties (part i: Information flow). In *FOSAD*, 2000.
[27] D. Freni, C. R. Vicente, S. Mascetti, C. Bettini, and C. S. Jensen. Preserving location and absence privacy in geo-social networks. In *CIKM*, 2010.
[28] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *POPL*, Jan. 2004.
[29] J. A. Goguen and J. Meseguer. Security policies and security models. In *S&P*, 1982.
[30] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *S&P*, 1984.
[31] E. Griffis, J. A. Vaughan, and T. Millstein. A platform for expressive and secure data sharing with untrusted third parties. Technical Report 120017, University of California, Los Angeles, 2011.
[32] D. P. Gruska. Observation based system security. *Fundam. Inform.*, 2007.
[33] D. P. Gruska. Informational analysis of security and integrity. *Fundam. Inform.*, 2012.
[34] J. D. Guttman and M. E. Nadel. What needs securing. In *CSFW*, 1988.
[35] D. Hedin and A. Sabelfeld. Information-flow security for a core of javascript. In *CSF*, 2012.
[36] J. Hintikka. *Knowledge and belief*. Cornell University Press, 1962.
[37] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985.
[38] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. A. de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *ICFP*, 2013.
[39] D. J. D. Hughes and V. Shmatikov. Information hiding, anonymity and privacy: a modular approach. *JCS*, 2004.
[40] S. Hunt and I. Mastroeni. The per model of abstract non-interference. In *SAS*, pages 171–185, 2005.
[41] R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3), 2000.
[42] J. Krumm. A survey of computational location privacy. *PUC*, 2009.
[43] P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *POPL*, 2005.
[44] H. Mantel. Possibilistic definitions of security - an assembly kit. In *CSFW*, 2000.
[45] I. Mastroeni. Abstract interpretation-based approaches to security - a survey on abstract non-interference and its challenging applications. In *Festschrift for Dave Schmidt*, pages 41–65, 2013.
[46] L. Mazaré. Using unification for opacity properties. In *WITS*, 2004.
[47] R. Milner. *Communication and concurrency*. Prentice Hall, 1989.
[48] J. Nagra and C. Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.
[49] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *IEEE Symposium on Security and Privacy*, pages 165–179. IEEE Computer Society, 2011.
[50] C. O'Halloran. A Calculus of Information Flow. In *ESORICS*, 1990.
[51] W. Rafnsson and A. Sabelfeld. Compositional information-flow security for interactive systems. In *CSF*, 2014.
[52] P. Ryan. Mathematical models of computer security—tutorial lectures. In *FOSAD*. Springer, 2001.
[53] P. Y. A. Ryan and T. Peacock. Opacity - further insights on an information flow property. Technical Report CS-TR-958, University of Newcastle upon Tyne, 2006.
[54] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS*, 2003.
[55] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 2003.
[56] A. Sabelfeld and A. Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *PSI*, 2009.
[57] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *HOSC*, 14(1), Mar. 2001.
[58] A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *JCS*, 17, 2009.
[59] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.
[60] D. Sutherland. A model of information. In *NCSC*, 1986.
[61] F. D. Tedesco, S. Hunt, and D. Sands. A semantic hierarchy for erasure policies. In *ICISS*, 2011.
[62] M. Terrovitis. Privacy preservation in the dissemination of location data. *SIGKDD Explorations*, 2011.
[63] D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *JCS*, 1996.
[64] D. M. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *CSFW*, 1997.
[65] D. von Oheimb. Information flow control revisited: Noninfluence = noninterference + nonleakage. In *ESORICS*, 2004.

[66] J. T. Wittbold and D. M. Johnson. Information flow in nondeterministic systems. In *IEEE Symposium on Security and Privacy*, 1990.
[67] Y.-C. Wu and S. Lafortune. Comparative analysis of related notions of opacity in centralized and coordinated architectures. *DEDS*, 2013.