51st IEEE Conference on Decision and Control
December 10-13, 2012. Maui, Hawaii, USA

Enforcement of Opacity Properties Using Insertion Functions

Yi-Chin Wu and Stéphane Lafortune

Abstract— **Opacity is a confidentiality property that arises in the analysis of security properties in networked systems. It characterizes whether a "secret" of a system can be inferred by an outside observer called an "intruder." We consider the problem of enforcing opacity in partially-observed discrete event systems modeled as automata. We propose a novel enforcement mechanism based on the use of insertion functions. An insertion function is a monitoring interface at the output of the system that changes the system's output behavior by inserting additional observable events. The insertion function must respond to the full system's output behavior. Also, the insertion function should not create new observed behavior but only replicate existing observable strings. We define the property of "i-enforceability," when there exists an insertion function that renders a non-opaque system opaque. To synthesize insertion functions that ensure opacity, we define and construct a new structure called the "All Insertion Structure" (AIS). The AIS can be used to verify if a given opacity property is i-enforceable. The AIS enumerates all i-enforcing insertion functions in a compact state transition structure. If a given opacity property has been verified to be i-enforceable, we show how to use the AIS to synthesize an i-enforcing insertion function.**

I. Introduction

Security and privacy are important concerns in today's networked systems. One class of security and privacy problems is confidentiality, i.e., preventing the disclosure of certain information to unauthorized identities. Some examples of information that require confidentiality are patients' medical history in online healthcare systems, identities of the sender and/or receiver in anonymous communication systems, and social security numbers in the electronic identification systems. The confidentiality property of interest in this paper is called *opacity*. It was introduced in the computer science community [10] and has been studied recently in Discrete Event Systems (DES) [1], [2], [11], [12]. Opacity is a confidentiality property that characterizes whether some secret information of a system can be inferred by outside observers. The ingredients of the DES formulation of an opacity problem are: (1) the system has a *secret*; (2) the system is only partially observable; (3) the *intruder* is an observer who has full knowledge of the system structure. The secret of the system is *opaque* if for every output behavior revealing the secret, there is an observationally-equivalent behavior that does not reveal the secret. We call the latter behaviors "non-secret." Thus, the intruder is not sure if the secret or the non-secret has occurred. The secret of the system is not restricted to events or states; it can be defined by any representation in the given DES model. For example, the secret can be defined in terms of the current state, the initial state, a sequence of K states, or a set of initial-final state pairs. Opacity properties corresponding to these various cases have been investigated in [1], [2], [11], [12], [15] using different DES modeling formalisms such as Petri nets, labeled transition systems, and automata.

In this paper, we consider the opacity problem in DES modeled as finite-state automata (FSA). Given an opacity property, methods for verifying if the secret is opaque or not have been investigated in [4], [6], [9], [11], [12]. When a secret is verified not to be opaque, the following question arises: How can we enforce the secret to be opaque? Many prior studies have designed the minimally-restrictive opacity-enforcing supervisory controller based on the supervisory control theory of DES; see, e.g., [5], [13]. The system behavior under supervisory control is restricted such that a behavior is disabled by feedback control if it is going to reveal the secret. While the secret under this controller is guaranteed to be opaque, this approach does not apply to situations where the system must execute its full behavior. Another approach to enforce opacity properties is to use a dynamic observer, which dynamically modifies the observability of every system event [4]. Unlike a supervisory controller, a dynamic observer allows the full system behavior, but it also erases some information that was to be output. Such erasure could create "new" observed strings that would have not been seen in the original system (under a static observable projection). When the intruder observes such a new string, it knows that a dynamic observer has been implemented and may choose to attack through other methods. Then, the system designer will have to modify the defense model. Another enforcement approach that allows the full system behavior is the run-time enforcement mechanism in [6]. This enforcement mechanism employs delays when outputting executions to enforce K-step opacity. However, this method applies only to secrets for which time duration is of concern.

The above limitations in the prior work have motivated us to propose and study a novel time-insensitive enforcement mechanism that allows the full system behavior without creating new observed behavior. As shown in Figure 1, the enforcement mechanism that we propose is based on the use of insertion functions at run-time. An insertion function allows the full system behavior because it is a monitoring interface placed at the output of the system. If an observed behavior t from the system is about to reveal the secret, then the insertion function remedies this behavior by inserting additional observable events and outputting t'. When making insertions, an insertion function should not create new ob-

Y. Wu and S. Lafortune are with the Department of EECS, University of Michigan, Ann Arbor, {ycwu;stephane}@umich.edu. This work was partially supported by the NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering (grant CCF-1138860).

served behavior. Thus, it cannot choose any random insertion strategy. An insertion strategy must be well-designed so that *every* behavior at the output of the insertion function must be observationally-equivalent to some non-secret behavior *at run-time*. We refer to such a characterization of insertion functions as the property of *i-enforceability*.

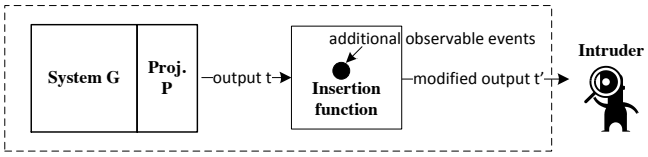


Fig. 1: The insertion mechanism

The i-enforceability property is a rather strong criterion for insertion function design. Not every secret of a system has a corresponding i-enforcing insertion function that enforces the secret to be opaque. Therefore, given a secret, we want to first *verify* if there exists an i-enforcing insertion function, i.e., if the opacity property is *i-enforceable*. One of the contributions of this paper is an algorithmic procedure to verify if a given opacity property is i-enforceable. We construct a structure called the "All Insertion Structure" (AIS), which enumerates in a compact manner all i-enforcing insertion functions. To verify if opacity is i-enforceable, it suffices to determine if the AIS is the empty structure or not. Furthermore, if opacity is i-enforceable, we show how to use the AIS to *synthesize* an i-enforcing insertion function. The algorithms for both the verification and the synthesis are general enough so that they apply to four opacity properties: current-state opacity, initial-state opacity, language-based opacity, and initial-and-final-state opacity. Other works in the computer science literature have also used insertion functions to enforce security properties; see e.g., [8], [14]. However, the class of security policies considered does not include opacity. To the best of our knowledge, our work is the first to address opacity enforcement using insertion functions.

The remaining sections of this paper are organized as follows. Section II introduces the system model and relevant definitions. Section III reviews the basics of the opacity problem. Section IV formally defines insertion functions and the i-enforceability property. In Section V, we present the 4-stage construction of the All Insertion Structure (AIS). The synthesis of an insertion function using the AIS is presented in Section VI. Finally, Section VII concludes the paper.

II. Preliminaries

A. Automata Models

An automaton $G = (X, E, f, X_0)$ has a set of states $X = \{0, 1, ..., N-1\}$, a set of events E, a deterministic state transition function $f : X \times E^* \to X$, and a set of initial-states X_0. In opacity problems, the initial state need not be known *a priori* by the intruder and thus we include a set of initial states X_0 in the definition of G. The language generated by G is the system behavior that is defined by $\mathscr{L}(G, X_0) := \{s \in E^* : (\exists i \in X_0)[f(i,s) \text{ is defined}]\}$. For simplicity, we will use $\mathscr{L}(G)$ if the set of initial states is clearly defined. In general, the system is partially observable. Hence, the event set is partitioned into an observable set E_o and an unobservable set E_{uo}. Given a string $t \in E^*$, its observation is the output of the natural projection $P : E^* \to E_o^*$, which is recursively defined as $P(te) = P(t)P(e)$ where $t \in E^*$ and $e \in E$. Projection of an event is $P(e) = e$ if $e \in E_o$ and $P(e) = \varepsilon$ if $e \in E_{uo} \cup \{\varepsilon\}$ where ε is the empty string.

B. Dashed Parallel Composition

We will use the *dashed parallel composition*, a special synchronization operator denoted as $||_d$. This composition synchronizes two types of automata: one with only solid transitions (e.g., the automaton in Figure 2c), and one with both solid and dashed transitions (e.g., the automaton in Figure 2d). In dashed parallel composition, the transitions of the two automata are synchronized on common events, like in the standard parallel composition. However, a common event is represented by a dashed transition if the corresponding transition in the second automaton is a dashed one; it is represented by a solid transition otherwise. For private events, the solid/dashed status of the transitions is preserved.

C. Projection P_{und} and Mask $\mathscr{M}_i$

The set of insertion events E_i is the set of observable events but every event has an extra insertion label denoted by subscript i. That is, $E_i = \{e_i : e \in E_o\}$. We now define the projection P_{und} and the mask $\mathscr{M}_i$. Each of P_{und} and $\mathscr{M}_i$ is an operation on strings consisting of insertion events and observable events. P_{und} is a natural projection that distinguishes between insertion events and observable events. Specifically, an insertion event under P_{und} is the empty string and an observable event remains observable: $P_{und}(e_i) = \varepsilon, e_i \in E_i$ and $P_{und}(e) = e, e \in E_o$. On there other hand, under $\mathscr{M}_i$, insertion events and observable events are indistinguishable. The mask removes the subscript i of events: $\mathscr{M}_i(e_i) = e, e_i \in E_i$ and $\mathscr{M}_i(e) = e, e \in E_o$.

III. Opacity Problems in Automata Formulations

A. Definition of Opacity Problems

We consider opacity properties in DES modeled as finite-state automata $G = (X, E, f, X_0)$. The settings of an opacity problem are: (1) G has a *secret*; (2) G is partially observable: $E = E_o \cup E_{uo}$; (3) The intruder is an observer of G; it has full knowledge of G, but only partially observes G through the natural projection P. That is, the intruder's observations are in $P[\mathscr{L}(G)]$. With the knowledge of G and its observations, the intruder infers the real system behavior by constructing estimates. The secret is said to be opaque if no intruder's estimate reveals the occurrence of the secret. That is, the system's secret is opaque if *for any secret behavior, there exists at least one other non-secret behavior that is observationally equivalent to the intruder*. Therefore, the intruder is not sure whether the secret or the non-secret has occurred. A formal definition will be given in Section III-C.

B. Four Notions of Opacity

Different definitions of the system's secret yield different notions of opacity. Here we consider four cases: current-state opacity (CSO), initial-state opacity (ISO), language-based

opacity (LBO), and initial-and-final-state opacity (IFO). The first three have been studied in [4], [9], [11]; IFO was introduced in our recent work [15]. Current-state opacity defines the secret as the current state of the system. Initial-state opacity defines the secret as the initial state. The secret of CSO or ISO is a subset of the state space, $X_S \subseteq X$. Language-based opacity defines the secret as a sublanguage of the system's language, $L_S \subseteq \mathscr{L}(G)$. Lastly, initial-and-final-state opacity defines the secret as pairs of initial and final states, $X_{sp} \subseteq X_0 \times X$. When considering these four opacity properties, we assume that the full (prefix-closed) system behavior $\mathscr{L}(G)$ is of concern and that the non-secret [language; state set; state pairs] is the complement of the secret [language; state set; state pairs]. These assumptions do not result in any essential loss of generality.

C. Verification of Opacity Properties

In [15], we have shown that CSO, ISO and IFO can be mapped to LBO by suitably (re)defining G and L_S. Thus, each opacity property can be discussed in terms of LBO, which we now formally define:

Definition 1 (LBO): Given $G = (X, E, f, X_0)$, P, and a secret language $L_S \subseteq \mathscr{L}(G, X_0)$, language-based opacity property holds if for every string $t_1 \in L_S$, there exists another string $t_2 \in L_{NS} = \mathscr{L}(G, X_0) \setminus L_S$ such that $P(t_1) = P(t_2)$.

In particular, an opacity property holds if the intruder *always* observes a string $t \in P[\mathscr{L}(G) \setminus L_S]$. The "largest" set containing such strings is the *supremal prefix-closed sublanguage* of $P(L_{NS})$, which we call the *safe language* and denote by L_{safe}. A string $t \in P[\mathscr{L}(G)]$ is said to be safe if $t \in L_{safe}$, and is unsafe otherwise. Using the result from [7], L_{safe} is characterized by the formula:

$$L_{safe} = P[\mathscr{L}(G)] \setminus (P[\mathscr{L}(G)] \setminus P(L_{NS}))\, E_o^* \qquad (1)$$

In [15], we have also shown that every opacity property can be verified by its corresponding *forward state estimator*[1]. A forward state estimator is an automaton where a state reached by string t is the intruder's [current-state; initial-state; initial-and-final-state] *estimate* when it observes t. Specifically, CSO and LBO can be verified by the standard *observer automaton* defined in Section 2.5.2 of [3]; ISO and IFO can be verified by the trellis-based Initial-State Estimator (ISE) introduced in [11]. *The existence of a forward state estimator for every opacity property of interest is key to the development of the verification algorithms in this paper.* For simplicity, we will call a forward state estimator an *estimator* and denote it by $\mathscr{E}$ hereafter. Given an opacity property, its estimator gives an estimate for every observed string $t \in P[\mathscr{L}(G)]$. The opacity property holds if no estimate contains only the secret information (specifically, current state, initial state, or initial-and-final-state pair). If there is an estimate that contains only the secret information, the secret is revealed when the corresponding observed string has occurred. In this case, the estimate corresponds to an *unsafe* observed string. We call an estimate as safe if it is reached by a safe string, and unsafe otherwise. It is clear that an estimate containing only the secret information is unsafe. However, for an estimate containing also the non-secret information, it could be unsafe if the corresponding observed string has an unsafe prefix (recall that L_{safe} is prefix-closed). In Section V-A, we will obtain an estimator with only safe estimates. For this purpose, we have to take the accessible part after removing the estimates that contain only the secret.

[1]This is in contrast to the alternative estimator for ISO proposed in [15], which is the observer of the reversed automaton of G and thus is not a "forward" state estimator.

IV. Enforcement of Opacity Using Insertion Functions

When a given secret is verifiably not opaque, one would like to know how to enforce opacity. We propose to enforce opacity using insertion functions. The insertion enforcement mechanism allows the full system behavior, does not create new observed behavior, and does not deploy delays.

A. Insertion Enforcement Mechanism

The insertion function is a special type of monitor that not only monitors the system but also inserts additional events to the system's observed behavior when necessary. It takes an observed *event* from the system, possibly inserts extra observable events, and outputs the resulting *string*. *In terms of the intruder, the extra inserted events are indistinguishable from the genuine system's observable events.* Therefore, observing at the output of the insertion function, the intruder cannot tell if the observed string includes inserted events or not.[2] While an inserted event and an observable event are indistinguishable, we need to clearly distinguish them for the sake of discussion. Hence, we attach a *virtual* insertion label to every inserted event. That is, we write that the insertion function inserts events $e_i \in E_i$ instead of $e \in E_o$. Because the virtual insertion label is not recognized by the intruder, we let the intruder observe the modified output under mask $\mathscr{M}_i$. We also use projection P_{und}, which erases inserted events, to reconstruct the original observed string.

B. Insertion Functions

The insertion enforcement mechanism depends on the design of insertion functions. Formally, we describe an insertion function as a (potentially partial) function $f_I : E_o^* \times E_o \to E_i^* E_o$ that outputs a string with insertions based on the system's past and current observed behavior. More precisely, given that the system has executed a string with observation $t \in E_o^*$ and the current observed event is $e_o \in E_o$, the insertion function is defined such that $f_I(t, e_o) = t_I e_o$ if string $t_I \in E_i^*$ is inserted before e_o. Hereafter we will use (t, e_o) to denote the system's observed behavior that defines the output of f_I.

The function f_I only defines the instantaneous insertion for every (t, e_o). To know the complete modified string from the insertion function, we define an equivalent string-based insertion function f_I^{str} from f_I: $f_I^{str}(\varepsilon) = \varepsilon$ and $f_I^{str}(s_n) = f_I(\varepsilon, e_1) f_I(e_1, e_2) \cdots f_I(e_1 e_2 \ldots e_{n-1}, e_n)$ where

[2]This ensures that opacity will hold even when the intruder knows the insertion function. When an intruder that knows the insertion function observes an output string, it is still not sure if the string is an unsafe string with insertion or a safe string, because inserted events are indistinguishable from genuine system observed events.

$s_n = e_1 e_2 \ldots e_n \in E_o^*$. Given G, the modified language output after the insertion function is $f_I^{str}(\mathscr{L}(G)) = \{t \in (E_i^* E_o)^* : t = f_I^{str}(s) \wedge s \in \mathscr{L}(G)\}$ and is denoted hereafter by L_{out}.

Example 1: Given G with $E_o = \{a,b\}$, the insertion function f_I that "inserts a before the first b" is defined as $f_I(\varepsilon,b) = a_i b, f_I(\varepsilon,a) = a, f_I(t,e) = e$ for $t \in E_o^* \setminus \{\varepsilon\}$ and $e \in \{a,b\}$.

We have defined a general insertion function. However, not all insertion functions enforce the given opacity property of the system. To enforce opacity, an insertion function must satisfy a property called *i-enforceability*.

C. I-Enforcing Insertion Functions

I-enforceability is a property of insertion functions that captures the fact that every system observable behavior is made to look non-secret under $\mathscr{M}_i$ after insertion. Therefore, given an i-enforcing insertion function, every behavior after insertion must be within the safe language and every system observable behavior must result in an insertion (possibly ε). Two properties are required to guarantee i-enforceability: *safety* and *admissibility*. An insertion function is *i-enforcing* if it is safe and admissible.

Definition 2: (Safe Insertion Functions) Given G, secret language L_S and non-secret language L_{NS}, an insertion function is safe if its modified language, L_{out}, is within the safe language under mask $\mathscr{M}_i$; that is, $\mathscr{M}_i(L_{out}) \subseteq L_{safe}$.

The safety property ensures that no string output by f_I reveals the secret. To make sure f_I inserts on every observable behavior, we define the admissibility property.

Definition 3: (Admissible Insertion Functions) Given G, an insertion function is admissible if it inserts (possibly ε) on every observable behavior from G. That is, $f_I(t,e_o)$ is defined for all $te_o \in P[\mathscr{L}(G)]$.

The admissible property means that an insertion function is a true *interface* that monitors the system.

Definition 4: (I-Enforcing Insertion Functions) Given G, secret language L_S and non-secret language L_{NS}, an insertion function is called *i-enforcing* if it is both safe and admissible. Moreover, the opacity property is called *i-enforceable* if there exists an i-enforcing insertion function.

V. Construction of the All Insertion Structure

We have proposed the use of insertion functions to enforce opacity properties. Then, one might ask two questions: (Q1) How to verify if a given opacity property for a system is i-enforceable? (Q2) How to synthesize an i-enforcing insertion function? These two questions can be answered by a special automaton called the *All Insertion Structure* (AIS). We will construct the AIS in this section and then answer (Q1). (Q2) will be answered in Section VI.

The All Insertion Structure (AIS) is an automaton that enumerates, in a compact transition structure, *all deterministic i-enforcing insertion functions for a given secret of the system*. The construction of the AIS is rather involved and comprises four stages: (1) *The i-verifier of the given opacity property of the system.* The i-verifier is the composition of two special estimators. It represents an insertion function that is *safe*, but *nondeterministic*. (2) *The meta-observer of the i-verifier.* The meta-observer is an observer-like automaton that is used to determine one t_I for every (t,e_o) from the nondeterministic function in stage 1. (3) *The unfolded i-verifier.* This structure "unfolds" the meta-observer in stage 2 and enumerates all deterministic insertion functions. (4) *The final structure AIS.* It is obtained by pruning inadmissible insertion functions from the unfolded i-verifier. We present the construction of the above four structures (which can all be viewed as automata) in the following four sections.

A. Stage 1: Construction of the i-verifier V

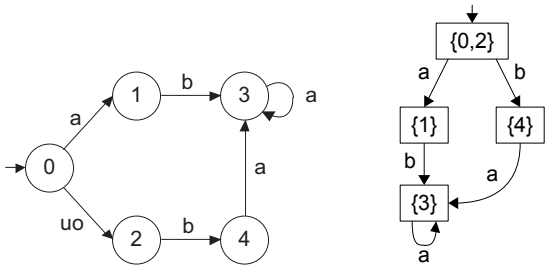
The i-verifier V represents an insertion function that is *safe*, *nondeterministic*, and *maximally-inserting*. This automaton is the *dashed parallel composition* of the desired estimator $\mathscr{E}^d$ and the feasible estimator $\mathscr{E}^f$, two automata obtained from the estimator $\mathscr{E}$ for the given secret of G.

The construction of the i-verifier V starts by building the *desired* estimator $\mathscr{E}^d$ and the *feasible* estimator $\mathscr{E}^f$. First, we build $\mathscr{E}^d$ from $\mathscr{E}$ by deleting all estimates containing only the secret information and taking the accessible part. The resulting estimator contains only safe estimates and thus generates the safe language L_{safe} by construction. Then, we build $\mathscr{E}^f$ by adding self-loops for every inserted event $e_i \in E_i$ (represented by dashed lines) at all states in $\mathscr{E}$. This estimator represents a nondeterministic insertion function that inserts the set E_i^* upon every (t,e_o). Furthermore, E_i^* is the maximal set for insertion because it includes all insertion strings of arbitrarily long length. Therefore, the function is also "maximally-inserting". Examples of $\mathscr{E}$, $\mathscr{E}^d$ and $\mathscr{E}^f$ are shown in Figures 2b, 2c and 2d, respectively. Finally, we synchronize the two estimators by the dashed parallel composition operation, and obtain the i-verifier $V = \mathscr{E}^d ||_d \mathscr{E}^f$ as shown in Figure 2e. Because the i-verifier is the synchronization of $\mathscr{E}^d$ and $\mathscr{E}^f$, it inherits the safe feature of $\mathscr{E}^d$, and the nondeterministic and maximally-inserting features of $\mathscr{E}^f$. The i-verifier represents a nondeterministic insertion function such that the insertion set upon every (t,e_o) includes any string $t_I \in E_I^*$ provided that the output inserted string is safe.

Formally, the i-verifier is an automaton denoted by $V = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$. A state $m_v \in M_v$ is a pair of estimates (m_d, m_f), where m_d is a (safe) estimate from $\mathscr{E}^d$ and m_f is the genuine (potentially unsafe) estimate from $\mathscr{E}^f$. In fact, m_d is the estimate of the intruder who eavesdrops with $\mathscr{M}_i$ and m_f is the estimate of the designer who reconstructs with P_{und}. The pair (m_d, m_f) means that although the genuine observed behavior of G leads to a possibly unsafe estimate m_f, the intruder is tricked into generating a safe estimate m_d.

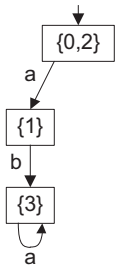
Example 2: Given G and the secret state set $\{4\}$ in Figure 2a, we build the estimator $\mathscr{E}$ in Figure 2b to verify current-state opacity (CSO). $\mathscr{E}$ shows that the secret is not opaque because estimate m_3 contains only the secret state 4. To determine if CSO is i-enforceable, we first construct the i-verifier V in this example and will complete the AIS in the following subsections.

To construct V, we first build $\mathscr{E}^d$ by removing m_3 from $\mathscr{E}$ and taking the accessible part; the resulting $\mathscr{E}^d$ generates the safe language and is shown in Figure 2c. Then, we build

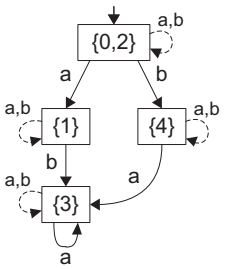


(a) The system G with secret state 4

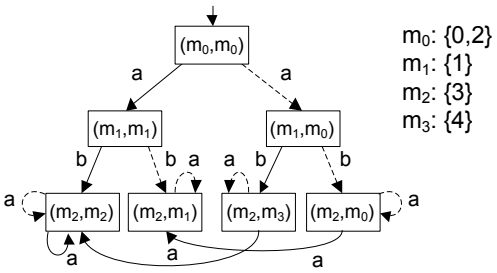
(b) The estimator $\mathscr{E}$



(c) The desired estimator $\mathscr{E}^d$



(d) The feasible estimator $\mathscr{E}^f$



(e) The i-verifier V

Fig. 2: The system, the estimators, and the i-verifier used in Example 2.

$\mathscr{E}^f$ by adding self-loops for a_i and b_i at every state in $\mathscr{E}$; the inserted events a_i and b_i are represented by dashed lines. Finally, V is the dashed parallel composition of $\mathscr{E}^d$ and $\mathscr{E}^f$, as shown in Figure 2e. We now use this example to see the features of the insertion function generated in V. First, the safe feature can be shown by the transitions from (m_0,m_0) to (m_1,m_0) and then to (m_2,m_3), which correspond to inserting a_i before the first b. This insertion tricks the intruder (who observes with $\mathscr{M}_i$) into thinking G outputs ab and generating safe estimate m_2, while the original observed string b corresponds to the unsafe estimate m_3. Second, the nondeterministic feature can be shown by the set $a_ib_ia_i^*$ inserted before the first a. Note that no modified string in $(a_ib_ia_i^*)a$ reveals the secret, and that no insertion not in $a_ib_ia_i^*$ can modify a to be a safe string. Therefore, the insertion function is maximally-inserting.

B. Stage 2: Construction of the meta-observer mObs(V)

The insertion function captured in V is nondeterministic. To determine one insertion response for every (t,e_o) from the nondeterministic insertion function, we first build the so-called *meta-observer* of the i-verifier, $mObs(V)$. The meta-observer has a structure similar to the observer automaton $Obs(V)$. It groups together all insertions corresponding to a given past observed string t. Given an i-verifier V, its meta-observer $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$ is a special observer of V, with respect to P_{und}, where the unobservable transitions (E_i) are preserved. The reason for preserving E_i transitions is to remember insertion responses. To preserve E_i transitions while grouping insertion responses, we define a meta-observer state $m_{v,mo} = (m_v, m_{v,obs}) \in M_{v,mo}$ to be a pair consisting of a state in V and a state in $Obs(V)$. Also, $mObs(V)$ has two transitions functions: $\delta_{v,obs}$ and $\delta_{v,mo}$. The former is the transition function of $Obs(V)$, which captures the system's observed behavior. The latter defines transitions between $M_{v,mo}$ states, which capture the effect of insertion responses. We show the meta-observer for Example 2 in Figure 3. Note that we only show the function $\delta_{v,mo}$, which is represented by the dashed and the solid transitions. The function $\delta_{v,obs}$ is not explicitly shown as it can be inferred from $\delta_{v,mo}$.

Formally, $mObs(V)$ is constructed from the i-verifier V and its observer automaton $Obs(V)$ with respect to P_{und}.

Input: $V = (M_v, E_o \cup E_i, \delta_v, m_{v,0})$
and $Obs(V) = (M_{v,obs}, E_o, \delta_{v,obs}, m_{v,obs,0})$
Output: $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$

1) $m_{v,mo,0} = (m_{v,0}, m_{v,obs,0})$. Set $M_{v,mo} = \{m_{v,mo,0}\}$
2) **for all** $\beta = (b, b_{obs}) \in M_{v,mo}$ whose dashed descendants have not been expanded **do**
   for all $e \in E_i$ **do**
   $\delta_{v,mo}(\beta, e) = (\delta_v(b,e), b_{obs})$ if $\delta_v(b,e)$ is defined.
   Add $\delta_{v,mo}(\beta, e)$ to $M_{v,mo}$
   end for
   end for
3) **for all** $\beta = (b, b_{obs}) \in M_{v,mo}$ whose dashed descendants have been expanded but whose solid descendants have not been expanded **do**
   for all $e \in E_o$ **do**
   $\delta_{v,mo}(\beta, e) = \big(\delta_v(b,e), \delta_{v,obs}(b_{obs}, e)\big)$ if $\delta_v(b,e)$ is defined.
   Add $\delta_{v,mo}(\beta, e)$ to $M_{v,mo}$
   end for
   end for
4) Go back to step 2 and repeat until all nodes in $M_{v,mo}$ have been completely expanded

Example 3: Given V in Figure 2e, we follow the above formal construction and build the meta-observer $mObs(V)$ shown in Figure 3. In step 1, the initial state is $((m_0,m_0),A)$, where (m_0,m_0) is the initial-state of V and $A = \{(m_0,m_0),(m_1,m_0),(m_2,m_0)\}$ is the initial state of $Obs(V)$. In step 2, $((m_0,m_0),A)$ expands to $((m_1,m_0),A)$ and then $((m_1,m_0),A)$ expands to $((m_2,m_0),A)$. This dashed-descendant expansion generates all $M_{v,mo}$ states whose $M_{v,obs}$ part is also A; these $M_{v,mo}$ states belong to the top-most group shown in the figure. In step 3, $((m_0,m_0),A)$ expands to $((m_1,m_1),B)$ with event a, $((m_1,m_0),A)$ expands to $((m_2,m_3),C)$ with b, and $((m_2,m_0),A)$ expands to $((m_2,m_1),B)$ with a. This time, the solid-descendant expansion generates all $M_{v,mo}$ states reached with E_o from the existing states. Each newly generated $M_{v,mo}$ state has a

$M_{v,obs}$ part that is different from A. Finally, we complete constructing $mObs(V)$ by recursively proceeding with steps 2 and 3 until all $M_{v,mo}$ states have been completely expanded.

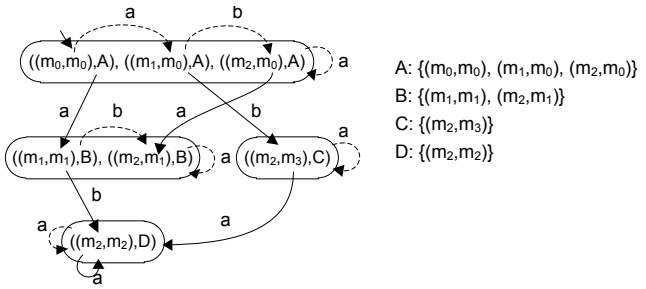


Fig. 3: The meta-observer of the i-verifier, $mObs(V)$. The solid ellipses around states group states reachable by dashed transitions, for the sake of illustration only.

C. Stage 3: Construction of the unfolded i-verifier V_u

The meta-observer groups all insertions to a given system's past output t; however, it does not consider the system's current output event e_o. To determine one insertion response for every (t, e_o), we build the unfolded i-verifier V_u that "unfolds" and enumerates all deterministic insertion functions captured in $mObs(V)$. In V_u, insertion functions are enumerated with sequences of alternating *actions*. Here, the actions are indeed the transitions in V_u. They are called *actions* to avoid being confused with event transitions of G. The actions alternate between actions of G (output events) and actions of an insertion function (insertion responses). Therefore, a sequence of actions fully defines an insertion strategy: If G outputs e_o, then f_I inserts t_I before e_o.

Formally, V_u is a bipartite graph that is defined as an automaton $V_u = (Y \cup Z, E_o \cup M_{v,mo}, f_{yz} \cup f_{zy}, y_0)$. An example of V_u is shown in Figure 4. The two disjoint sets of states are denoted by Y (square-shaped) and Z (ellipse-shaped). States along a sequence of actions alternate between Y states and Z states. We can describe V_u as a two-player "game." The first player is the system G; it makes actions at Y states and determines what events to output. The second player is the insertion function; it makes actions at Z states and determines what string to insert. More specifically, a state $y \in Y$ contains the information G needs for making actions; it is a meta-observer state, $y \in Y = M_{v,mo} \subseteq M_v \times M_{v,obs}$. A state $z \in Z$ contains the information f_I needs; it consists of its predecessor Y state and the corresponding incoming action of an observable event, $z \in Z \subseteq Y \times E_o = M_{v,mo} \times E_o$. An outgoing action of $y \in Y$ is an observable event in E_o. An outgoing action of $z \in Z$ is an $M_{v,mo}$ state that compactly represents a set of insertions which is given by a function called $Ins : M_{v,mo} \times M_{v,mo} \to 2^{E_i^*}$. Each insertion in this set reacts to the same observed behavior, leads to the same intruder-designer estimate pair, and has the same effect on future insertions. For example, Figure 4 shows the V_u built from the $mObs(V)$ in Figure 3. Given $z = (((m_0,m_0),A),a)$, its action $((m_2,m_0),A)$ is a set given by $Ins(((m_0,m_0),A),((m_2,m_0),A))$, which is defined by $\{t_I \in E_i^* : \delta_{v,mo}(((m_0,m_0),A),t_I) = ((m_2,m_0),A)\} = a_i b_i a_i^*$. The transition function from Y to Z is denoted by $f_{yz} : Y \times E_o \to Z$,

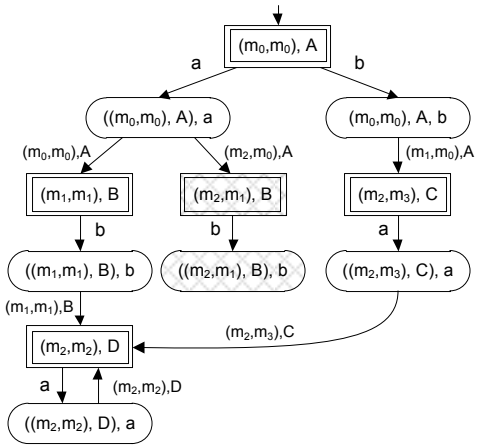


Fig. 4: The unfolded i-verifier V_u built from $mObs(V)$. The shaded states are pruned when constructing AIS.

and the transition function from Z to Y is denoted $f_{zy} : Z \times M_{v,mo} \to Y$. Lastly, as G is the first player, the initial state of V_u is defined as $y_0 = m_{v,mo,0} \in Y$.

The formal construction of V_u is presented below.

Input: $mObs(V) = (M_{v,mo}, E_o \cup E_i, \delta_{v,obs}, \delta_{v,mo}, m_{v,mo,0})$
Output: $V_u = (Y \cup Z, E_o \cup M_{v,mo}, f_{yz} \cup f_{zy}, y_0)$

1) $y_0 = m_{v,mo,0}$. Set $Y = \{y_0\}$
2) **for all** $y = (m_v, m_{v,obs}) \in Y$ that have not been examined **do**
   for all $e \in E_o$ **do**
   $f_{yz}(y,e) = (y,e)$ if $\delta_{v,obs}(m_{v,obs}, e)$ is defined
   Add $f_{yz}(y,e)$ to z.
   end for
   end for
3) **for all** $z = (m_{v,mo}, e) \in Z$ that have not been examined **do**
   for all $m \in M_{v,mo}$ **do**
   $f_{zy}(z,m) = \delta_{v,mo}(m,e)$ if $\exists t_I \in E_i^*$ such that $m = \delta_{v,mo}(m_{v,mo}, t_I)$ and $\delta_{v,mo}(m,e)$ is defined
   Add $f_{zy}(z,m)$ to Y.
   end for
   end for
4) Go back to step 2 and repeat until all accessible part has been built.

D. Stage 4: Construction of the All Insertion Structure (AIS)

The All Insertion Structure (AIS) is constructed by pruning inadmissible insertions in V_u. An insertion function is inadmissible if there is a system observed behavior that it cannot respond to. That is, there is $z \in Z$ that has no outgoing action for the insertion function (i.e., z is a deadlock state). An example of a deadlocked Z state is $(((m_2,m_1),B),b)$ in Figure 4. To prune inadmissible insertions, we prune $(((m_2,m_1),B),b)$ and its incoming action b. However, we cannot disable b because it is the action of the system at state $((m_2,m_1),B)$ and insertion functions have no control of system actions. Therefore, to avoid entering the troubled state $((m_2,m_1),B)$, an insertion function should decide not to respond with $((m_2,m_0),A)$ earlier at $(((m_0,m_0),A),a)$.

We formally solve the above pruning process by formulating it as an instance of the Supervisory Control Problem with

Blocking (SCPB), in the terminology of [3]. For this purpose, marked states and controllable/uncontrollable events need to be defined in V_u. We mark all Y states in V_u and leave all Z states unmarked. This is because a function completes its insertions at Y states, but is choosing an insertion response at Z states. Also, we model all outgoing actions of Y states (E_o) as uncontrollable and those of Z states ($M_{v,mo}$) as controllable because an insertion function has no control of the system output but is able to choose what string to insert. Finally, the minimally restrictive nonblocking supervisor of V_u (see [3]) for this SCPB formulation is the AIS. The AIS generates and only generates all admissible insertion functions. The formal construction is presented below:

1) Mark all the Y states in V_u.
2) Model all outgoing actions of Y states (E_o) as uncontrollable, and all outgoing actions of Z states ($M_{v,mo}$) as controllable.
3) Trim V_u and let the trimmed automaton V_u^{trim} be the specification automaton.
4) Obtain $[\mathscr{L}_m(V_u^{trim})]^{\uparrow C}$ w.r.t $\mathscr{L}(V_u)$ by following the standard $\uparrow C$ algorithm in [3]. The AIS is the resulting automaton representation of $[\mathscr{L}_m(V_u^{trim})]^{\uparrow C}$.

In Figure 4, step 3 will delete state $(((m_2,m_1),B),b)$ and step 4 will delete state $((m_2,m_1),B)$.

We now go back to question (Q1) posed at the beginning of this section and answer it with Theorem 1:

Theorem 1: Given a secret of the system G, the corresponding opacity property is i-enforceable if and only if the AIS of G is not the empty automaton.

Proof: (Only if) By a contrapositive argument, when the AIS is the empty automaton, it follows directly from the four-stage construction of the AIS that there is no i-enforcing insertion function, i.e., opacity is not i-enforceable. (If) We again use a contrapositive proof. Suppose opacity is not i-enforceable. Then there must be a system action in the AIS where no insertion function is able to react; i.e., the AIS has a deadlock state $z \in Z$. Because z is deadlocked and all incoming actions of z are uncontrollable, all Y states leading to z must also be pruned according to the $\uparrow C$ algorithm. Pruning all Y states leading to z will generate more deadlocked Z states, because otherwise opacity would be i-enforceable. After an iterative calculation, *all* states leading to z will eventually be pruned. Because z is reachable in the AIS, this pruning will result in the empty automaton. ■

VI. Synthesis of i-enforcing insertion functions

We now answer question (Q2) and use the AIS to synthesize an i-enforcing insertion function as follows:

1) At every state $z = (m_{mo}, e) \in Z$, select *one* outgoing action $m' \in M_{v,mo}$. Then choose *one* string $t_I \in Ins(m_{mo}, m')$ and replace label m' by t_I.
2) For every $y_1 \xrightarrow{e_o} z \xrightarrow{t_I} y_2$, define $f_I(t, e_o) = t_I e_o$ where t is any string leading to y_1 where all intermediate insertions (i.e., outgoing actions of intermediate Z states) are removed.

In step 1, the question of which outgoing action and string to choose arises. We can establish a selection criterion by designing an appropriate cost function, and formulate an optimal control problem. The AIS provides a structure over which such an optimal control problem can be formulated and solved, and where *all* i-enforcing insertion functions are considered. Solving such optimal control problems is beyond the scope of this paper.

VII. Conclusion

We have considered the problem of enforcing opacity using insertion functions at the interface between the system and the intruder. In this novel opacity-enforcement paradigm, the insertion function dynamically changes the system's observed behavior by inserting additional observable events. Such insertions must force the full system observed behavior to be observationally equivalent to some non-secret behavior. This property of insertion functions is called "i-enforceability." We have formally modeled the insertion mechanism and characterized the i-enforceability property of insertion functions. To verify if an opacity property is i-enforceable, we have constructed the All Insertion Structure (AIS) that enumerates in a compact state structure all i-enforcing insertion functions. We have shown that opacity is i-enforceable if and only if the AIS is not the empty structure. Furthermore, we have also shown how to construct i-enforcing insertion functions using the AIS.

References

[1] J. Bryans, M. Koutny, and P.Y.A. Ryan. Modeling opacity using petri nets. *Electronic Notes in Theoretical Computer Science*, 121:101–115, 2005.

[2] J.W. Bryans, M. Koutny, L. Mazaré, and P.Y.A. Ryan. Opacity generalized to transition systems. *International Journal of Information Security*, 7(6):421–435, 2008.

[3] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2008.

[4] F. Cassez, J. Dubreil, and H. Marchand. Dynamic observers for the synthesis of opaque systems. *Automated Technology for Verification and Analysis*, 5799:352–367, 2009.

[5] J. Dubreil, P. Darondeau, and H. Marchand. Supervisory control for opacity. *IEEE Transactions on Automatic Control*, 55(5):1089–1100, 2010.

[6] Y. Falcone and H. Marchand. Various notions of opacity verified and enforced at runtime. Technical Report 7349, INRIA, 2010.

[7] R. Kumar and V.K. Garg. *Modeling and control of logical discrete event systems*. Kluwer Academic Publishers, 1995.

[8] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[9] F. Lin. Opacity of discrete event systems and its applications. *Automatica*, 47(3):496–503, 2011.

[10] L. Mazaré. Using unification for opacity properties. *Proceedings of the 4th IFIP WG1*, 7:165–176, 2003.

[11] A. Saboori and C. N. Hadjicostis. Verification of initial-state opacity in security applications of DES. *Proc. of the 9th International Workshop on Discrete Event Systems*, pages 328–333, May 2008.

[12] A. Saboori and C. N. Hadjicostis. Verification of k-step opacity and analysis of its complexity. *IEEE Transactions on Automation Science and Engineering*, 8(3):549–559, 2011.

[13] A. Saboori and C. N. Hadjicostis. Opacity-enforcing supervisory strategies via state estimator constructions. *IEEE Transactions on Automatic Control*, 57(5):1155–1165, 2012.

[14] F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[15] Y.C. Wu and S. Lafortune. Comparative analysis of related notions of opacity in centralized and coordinated structures. *Discrete Event Dynamic Systems: Theory and Applications*, 2012.