2017 IEEE 56th Annual Conference on Decision and Control (CDC)
December 12-15, 2017, Melbourne, Australia

Enforcing Opacity by Publicly Known Edit Functions

Yiding Ji and Stéphane Lafortune

Abstract— **This paper extends prior work about the enforcement of opacity by insertion functions and applies a more general method called edit functions. Based on its observations, the edit function can insert or erase events to modify the outputs of the system and obfuscate the outside intruder. In this paper, a key assumption is that the intruder knows the implementation of the edit function, which requires the edit function to be "public-private enforcing". In order to capture the limitations of edit functions, state based edit constraints are introduced and may preclude some originally feasible edit choices, complicating the enforcement problem. The edit function in this work is deterministic and the enforcement problem is formulated as a two-player game between the edit function and the system. Our goal is to synthesize public-private enforcing edit functions without violating edit constraints. A new synthesis algorithm is proposed based on the game structure.**

I. Introduction

Opacity characterizes whether an outside malicious intruder can infer the secrets of a given dynamic system. It has received increasing attention in the literature on security and privacy since it was first introduced. The intruder is modeled as a passive outside observer with knowledge of the system's structure and it intends to access secrets by observing system's outputs. The system is called opaque if for every behavior induced by a secret (termed *secret behavior*), there is another observationally-equivalent behavior that is not induced by any secret (termed *non-secret behavior*).

In discrete event systems, various representations of secrets have been considered to study opacity, which leads to different opacity notions. In the context of finite state automata models, opacity has been formulated in the deterministic setting as: initial-state opacity, current-state opacity, language-based opacity, K-step opacity, infinite step opacity, initial-and-final state opacity [10], [11], [15], [21]; opacity has also been considered in stochastic settings [1], [12]. Also, opacity in infinite state systems is considered in [5]. Other works discuss opacity using Petri net models, such as [2], [14]. The reader is referred to [9] for a detailed and comprehensive review of current results from the above mentioned perspectives.

An important problem is the enforcement of opacity for a given system [6], [8]. One commonly used approach is to design a minimally restrictive supervisor [7], [13], which can disable undesired behaviors before secrets are disclosed. The work [20] provides a uniform approach of synthesizing maximally permissive supervisors from a bipartite transition system. In [22], the authors discuss opacity enforcement by supervisory control with maximal information release. Another popular framework is sensor activation [4], [19], where dynamic or maximally permissive observers are built.

In contrast to the above approaches, an insertion mechanism is proposed in [16], which inserts fictitious events at the system's output to obfuscate the intruder's observations. In particular, [17] investigates opacity enforcement under the assumption that the intruder may or may not know the implementation of the insertion function. In a more recent work [18], the authors extend insertion functions to a more general method called edit functions, which are able to modify system's outputs by inserting, erasing or replacing events. A new concept of utility constraint is defined to capture the limitations of edit functions.

In this work, we extend the works [17], [18] by considering *Public-Private (PP) enforceability* for edit functions. We assume that the edit function is made public by choice, or that the intruder uses learning techniques to infer the edit function. Also, we limit the capabilities of edit functions to insertion and erasure, since any event replacement is equivalent to event erasure followed by event insertion.

The main contributions of this paper are as follows. First, we formally characterize the edit mechanism and define public-private enforceability, under the framework of opacity enforcement by edit functions. Then we define edit constraints in a state-based manner and show how they restrict edit functions' behaviors. To capture all possible choices of edit functions, we construct a bipartite structure called All Edit Structure (AES) and obtain its reduced substructure under constraints, denoted as AES_c. We show that PP-enforcing edit functions may not always exist due to edit constraints. Finally, we propose a new synthesis algorithm for PP-enforcing edit functions, which is based on the *reachability tree* of AES_c.

The remaining sections of this paper are organized as follows. Section II briefly reviews basic notations and introduces the notion of opacity used in this paper. Section III defines edit functions and formally characterizes the property of public-and-private enforceability. Section IV presents the construction procedure of the All Edit Structure and reduces it under edit constraints. Section V identifies relevant concepts and properties of AES under constraints and builds its reachability tree. Section VI presents the new synthesis algorithm with illustrative examples. Finally, Section VII concludes the paper along with directions for future work.

This work was partially supported by NSF grants CCF-1138860 (Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering) and CNS-1421122, and by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

Y. Ji and S. Lafortune are with the Department of Electrical Engineering and Computer Science at the University of Michigan. {jiyiding;stephane}@umich.edu

II. Opacity Notions in Automata Models

We consider opacity in the framework of finite-state discrete event systems modeled as finite state automata: $G = (X, E, f, X_0)$, where X is the finite set of states, E is the finite set of events, f is the partial state transition function $f : X \times E \to 2^X$, and $X_0 \subseteq X$ is the set of initial states [3]. Specifically, we denote $X_S \subset X$ as the set of *secret states*, which characterizes the system's secrets. We allow G to be nondeterministic so that the codomain of f is the power set of X. The transition function is extended to domain $X \times E^*$ in the standard manner and we still denote the extended function by f. Also, we use the notation $s \preceq u$ to denote that string s is a prefix of string u. The language generated by G is the set of system behaviors that is defined by $\mathscr{L}(G, X_0) := \{t \in E^* : \exists x \in X_0,\ f(x,t) \text{ is defined}\}$ and from now on, we denote it as $\mathscr{L}(G)$ for short. The system G 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 observable projection is the output of the natural projection $P : E^* \to E_o^*$, which is recursively defined as $P(t) = P(t'e) = P(t')P(e)$ where $t' \in E^*$ and $e \in E$. The 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.

The intruder is modeled as an observer w.r.t. system G, denoted by $\mathscr{E} = (X_e, E_o, \delta, x_{e0})$. The standard construction procedure of building observers can be found in Section 2.5.2 of [3]. The intruder knows the structure of G and observes the outputs in $P[\mathscr{L}(G)]$. Then, it can combine its knowledge of G and online observations to infer secrets. Opacity holds if the intruder can not assert a *secret behavior* from its estimate and the system is opaque if for every *secret behavior*, there is another observationally-equivalent *non-secret behavior*.

Definition 1 (Current-State Opacity (CSO)): Given system G, projection P, and the set of secret states X_S, G is current state opaque if $\forall t \in L_S := \{t \in \mathscr{L}(G, X_0) : \exists x_0 \in X_0, f(x_0,t) \cap X_S \neq \emptyset\}$, $\exists t' \in L_{NS} := \{t \in \mathscr{L}(G, X_0) : \exists x_0 \in X_0, f(x_0,t) \cap (X \setminus X_S) \neq \emptyset\}$ such that $P(t) = P(t')$.

CSO can be verified by building the corresponding *observer automaton* and checking whether any observer state contains solely secret states [9].

III. Edit Mechanism for Opacity Enforcement

In this work, we apply a similar framework as in [18]: the edit function is an interface between the system and the outside intruder; it receives the system's output, inserts fictitious events or erases observed events, then outputs the modified strings. We assume that all events in E_o are allowed to be edited, and the intruder is unable to distinguish between an inserted event and its genuine counterpart.

A. Edit functions

An edit function is defined as a (potentially partial) function $f_e : E_o^* \times E_o \to E_o^*$ that outputs a string with edited events based on the past observed behavior and the current observed event. Given an observable string $se_o \in P[\mathscr{L}(G)]$, an edit function is defined such that:

$$f_e(s, e_o) = \begin{cases} s_I e_o & f_e \text{ inserts } s_I \text{ before } e_o \\ \varepsilon & f_e \text{ erases } e_o \end{cases}$$

Notice that s_I may be ε so that f_e can leave the last observed event intact. Here we see that if the edit function first erases a certain event and then inserts a different event (upon the next event occurrence), the effect is equivalent to *replacing* an event. So in this work, we only explicitly consider event *insertion* and *erasure*; this makes the exposition simpler. With a slight abuse of notation, we also define a string based edit function f_e recursively as: $f_e(\varepsilon) = \varepsilon$ and $f_e(se_o) = f_e(s)f_e(s, e_o)$. Given G, the modified language output by edit function f_e is denoted by $f_e(P[\mathscr{L}(G)]) = \{\tilde{s} \in E_o^* : \exists s \in P[\mathscr{L}(G)], f_e(s) = \tilde{s}\}$.

B. Private-Public Enforceability

As was mentioned earlier, the key issue in this paper is that we assume the intruder *knows* the edit function employed by the system. Similarly to the case of insertion functions treated in [17], the edit function should be *public-private enforcing*, i.e. admissible, privately safe, and publicly safe. We now explain these properties. Admissibility is an input property for edit functions: the edit function f_e should be able to modify any string in $P[\mathscr{L}(G)]$, i.e. $\forall se_o \in P[\mathscr{L}(G)]$, $f_e(s, e_o)$ is defined. Private safety is an output property of edit functions: what the intruder observes should be consistent with the system's transition structure. Besides, every modified output behavior should not lie out of original non-secret behaviors of the system. Because of these requirements, every modified output string should be a string in the safe language L_{safe}, which is the supremal prefix-closed sublanguage of $P(L_{NS})$ (defined in Definition 1) and is calculated by the equation:

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

Hereafter, we call a string $s \in P[\mathscr{L}(G)]$ *safe* if it is in L_{safe} and *unsafe* otherwise, so $L_{unsafe} = P[\mathscr{L}(G)] \setminus L_{safe}$. Also, all the continuations of an unsafe string are unsafe.

Definition 2 (Private Safety): Consider G with P, L_{safe} and L_{unsafe}. An edit function f_e is privately safe if $\forall s \in P[\mathscr{L}(G)]$, $f_e(s) \in L_{safe}$; equivalently, $f_e(P[\mathscr{L}(G)]) \subseteq L_{safe}$.

Furthermore, public safety is an output property and the idea behind it is that no matter what the insertion function outputs, the output could also have been obtained from a non-secret string; hence system secrets would not be disclosed, even if the intruder has knowledge of the implementation of the edit function.

Definition 3 (Public Safety): Consider G with P, L_{safe} and L_{unsafe}, an edit function f_e is publicly safe if $\forall s \in L_{unsafe}, \exists t \in L_{safe}$ s.t. $f_e(t) = f_e(s)$; equivalently, $f_e(L_{unsafe}) \subseteq f_e(L_{safe})$ or $f_e(P[\mathscr{L}(G)]) = f_e(L_{safe})$.

Definition 4 (PP-Enforceability): Edit function f_e is PP-enforcing if it is admissible, privately safe, and publicly safe.

It is easy to construct a trivial PP-enforcing edit function, which erases all the observable events, reducing the whole language to $\{\varepsilon\}$. However, this solution does not make too much sense in practice and it will not be considered as a feasible solution in this work. Thus we introduce the concept of edit constraints using a state based binary function that eliminates certain edit choices.

Definition 5 (Edit Constraint): The edit constraint is a binary function $\varphi : X_e \times X_e \to \{0,1\}$, a state pair (x_d, x_f) satisfies the edit constraint if $\varphi(x_d, s_f) = 1$.

Edit constraint is a general form of the utility constraint in [18] and it is specified by requiring that certain state pairs not appear when we design edit functions. In the remainder of this paper, we assume that the edit constraints are given and problem-dependent.

IV. All Edit Structure under Constraints

This section presents the procedure of building the All Edit Structure under constraints (AES$_c$). The procedure is similar to that of building the corresponding All Insertion Structures in [17], [18]. AES$_c$ is a bipartite game-like structure (discrete transition system) between the system and the edit function, with two sets of states defined as Y and Z states. When it is the system's turn to play, a certain observable event e_0 occurs at the current Y-state, which is observed by the edit function and leads to a Z-state. Then, it is the edit function's turn to play and some edit decision is made at the current Z-state. The outcome of this edit decision will be observed by the intruder. Also, AES$_c$ embeds in its transition structure *all* feasible privately enforcing edit functions under constraints [18].

As in [18], there are three steps in building AES$_c$: (1) construct the verifier; (2) construct the unfolded verifier; (3) check the constraints and obtain AES$_c$. In step (1), we first build the *desired estimator* $\mathscr{E}^d$ by deleting all the states where the secret is revealed from $\mathscr{E}$ and taking the accessible part of it. Here the estimator is just the standard observer of G and $\mathscr{E}^d$ generates exactly the safe language L_{safe}. The transitions in $\mathscr{E}^d$ are denoted as δ_d. Next, we build the *feasible estimator* $\mathscr{E}^f$, which includes *all* possible edit choices. We insert a self-loop at each state for every observable event, unless that self-loop is already defined at that state in $\mathscr{E}$. We also add an ε transition between two states as long as there is a transition with an observable event defined between them. Therefore, the feasible estimator is defined as $\mathscr{E}^f = (X_f, E_o, \delta_{fo}, \delta_{fi}, \delta_{fe}, \Gamma, x_{f0})$. Specifically, we denote $\delta_f = \delta_{fo} \cup \delta_{fi} \cup \delta_{fe}$, where δ_{fo} includes the normal transitions; δ_{fi} includes the above-mentioned inserted self-loop transitions of the form $\delta_{fi}(x_f, e) = x_f$; and δ_{fe} includes the ε transitions (event erasure transitions) defined as $\delta_{fe}(x_f, e \to \varepsilon) = x'_f$ if $\delta_{fo}(x_f, e) = x'_f$. Here we denote $E_o^\varepsilon = \{e \to \varepsilon : e \in E_o\}$ as set of event erasures. Finally, we synchronize $\mathscr{E}^d$ and $\mathscr{E}^f$ by a special parallel composition called *verifier parallel composition* (cf. [16]), resulting in a new structure called *verifier*, which embeds all privately safe and admissible edit choices. Correspondingly, there are three types of transitions in the verifier: (1) the normal transitions f_{vo}; (2) the inserted event transitions f_{vi}; and (3) the erased event transitions f_{ve}.

Definition 6 (Verifier Parallel Composition): The verifier parallel composition $\|_v$ is a special kind of parallel composition between automata $\mathscr{E}^d$ and $\mathscr{E}^f$. The verifier is defined as $V = (X_v, E_o, E_o^\varepsilon, f_v, \Gamma_v, x_{v0})$, where X_v denotes the state space and three types of transition functions $f_v = f_{vo} \cup f_{vi} \cup f_{ve}$ are defined: $f_{vo} : X_v \times E_o \to X_v$, $f_{vi} : X_v \times E_o \to X_v$ and $f_{ve} : X_v \times E_o \to X_v$.

$$V := \mathscr{E}^d \|_v \mathscr{E}^f = Ac(X_d \times X_f, E_o, E_o^\varepsilon, f_{vo}, f_{vi}, f_{ve}, \Gamma_v, (x_0, x_0))$$

where Ac stands for the accessible part and X_d, X_f denote the state spaces of $\mathscr{E}^d$, $\mathscr{E}^f$, respectively. The transition functions work as follows, where ! stands for "is defined":

$$f_{vo}(x_v, e) := (\delta_d(x_d, e), \delta_{fo}(x_f, e)), \text{ if } e \in \Gamma(x_d), \delta_{fo}(x_f, e))!$$
$$f_{vi}(x_v, e) := (\delta_d(x_d, e), x_f), \text{ if } e \in \Gamma(x_d), \delta_{fi}(x_f, e)!$$
$$f_{ve}(x_v, e) := (x_d, \delta_{fe}(x_f, e \to \varepsilon)), \text{ if } \delta_{fo}(x_f, e))!$$

The first transition corresponds to a normal transition labeled by e in both $\mathscr{E}^d$ and $\mathscr{E}^f$; the second transition corresponds to a normal transition labeled by e in $\mathscr{E}^d$ and an inserted self-loop transition also labeled by e in $\mathscr{E}^f$; the third transition corresponds to an ε transition in $\mathscr{E}^f$.

Then we unfold all deterministic edit choices from the verifier and obtain a game structure between the system player and the edit function player. This structure is called the *unfolded verifier*, denoted by V_u. Its construction procedure, shown in Algorithm 1, is similar to the procedure of building V_u in [16]. As required in [17], loops should not be inserted in building V_u, so that there are only a finite number of edit choices at each Z state. We define $\Gamma_{uv} : Z \to E_o^* \cup E_o^\varepsilon$ as the set of edit choices, specially, if we concatenate an edit choice $\gamma = e_o \to \varepsilon$ with e_o, then $\gamma e_o = \varepsilon$.

Algorithm 1: Build Unfolded Verifier

```
Input  : $V = (X_v, E_o, E_o^\varepsilon, f_{vo}, f_{vi}, f_{ve}, \Gamma_v, x_{v0})$
Output: $V_u = (Y, Z, E_o, E_o^\varepsilon, f_{yz}, f_{zy}, \Gamma_{uv}, y_0)$
1  $y_0 := x_{v0}, Y := \{y_0\}$;
2  for $y := x_v = (x_d, x_f) \in Y$ that has not been examined do
3      for $e \in E_o$ do
4          if $f_{vo}(x_v, e)$ or $f_{ve}(x_v, e)$ is defined then
5              $f_{yz}(y, e) := (y, e)$;
6              $Z := Z \cup \{(y, e)\}$;
7  for $z := (y, e) = (x_v, e) = ((x_d, x_f), e) \in Z$ that has not been examined do
8      $\Gamma_{uv}(z) = \emptyset$;
9      if $\exists x'_v = (x'_d, x'_f) \in X_v$, s.t. $\exists s_i \in E_o^*, x'_v = f_{vi}(x_v, s_i)$ and $f_{vo}(x'_v, e)$ is defined then
10         $f_{zy}(z, s_i) := f_{vo}(x'_v, e)$;
11         $\Gamma_{uv}(z) = \Gamma_{uv}(z) \cup \{s_i\}$;
12         $Y := Y \cup \{f_{zy}(z, s_i)\}$;
13     if $f_{ve}(x_v, e) = x'_v$ then
14         $f_{zy}(z, e \to \varepsilon) = x'_v$;
15         $\Gamma_{uv}(z) = \Gamma_{uv}(z) \cup \{e \to \varepsilon\}$;
16         $Y := Y \cup \{f_{zy}(z, e \to \varepsilon)\}$;
17 Go back to step 2 and repeat until all accessible part has been built and return $V_u$;
```

After building V_u, we prune away the *deadlock* Z states where no outgoing transitions are defined as well as Y states that do not satisfy the edit constraints. This process can be interpreted as a supremal controllable sublanguage calculation and more details can be found in [16]. Then we obtain AES$_c$ in Algorithm 2.

```
Algorithm 2: Build AES_c
  Input  : $V_u = (Y, Z, E_o, E_o^\varepsilon, f_{yz}, f_{zy}, \Gamma_{uv}, y_0)$
  Output: $AES_c = (Y, Z, E_o, E_o^\varepsilon, f^c_{AES,yz}, f^c_{AES,zy}, \Gamma_c, y_0)$
1 Mark all the $Y$-states in $V_u$;
2 Let $f_{yz}$ be the set of uncontrollable transitions and $f_{zy}$
  be the set of controllable transitions;
3 Prune away all $Y$ states that do not satisfy the edit
  constraints and denote the substructure as $\tilde{V}_u$;
4 $\tilde{V}_u^{trim} := Trim(\tilde{V}_u)$, obtain $\mathscr{L}_m(\tilde{V}_u^{trim})]^{\uparrow C}$ w.r.t $\mathscr{L}(V_u)$ by
  following the standard $\uparrow C$ algorithm;
5 Return the subautomaton that generates $[\mathscr{L}_m(\tilde{V}_u^{trim})]^{\uparrow C}$
  as the AES$_c$;
```

V. Analysis of AES under constraints

In this section, we first define some relevant concepts for AES$_c$ and then exploit its properties. PP-enforcing edit functions may not always exist even when privately safe edit functions exist. In order to verify the existence of PP-enforcing edit functions, we need to build a reachability-tree-like structure of AES$_c$ for further analysis. Since observable events and edit decisions alternate in the AES$_c$, we define the concept of *run* and characterize its generated string.

Definition 7 (Run): A run in AES$_c$ is a sequence of alternating states and events. $\Omega = \{\omega : \omega = \langle y_0 \xrightarrow{e_0} z_0 \xrightarrow{\gamma_0} y_1 \xrightarrow{e_1} \cdots y_{n-1} \xrightarrow{e_{n-1}} z_{n-1} \xrightarrow{\gamma_{n-1}} y_n \rangle\}$ where $n \in \mathbb{N}$, y_0 is the initial state of AES$_c$, $e_i \in E_o$, $f^c_{AES,yz}(y_i, e_i) = z_i$, $\gamma_i \in \Gamma_c(z_i)$, s.t. $f^c_{AES,zy}(z_i, \gamma_i) = y_{i+1}, \forall i,\ 0 \le i < n$.

Definition 8 (String Generated by a Run): Given a run $\omega = \langle y_0 \xrightarrow{e_0} z_0 \xrightarrow{\gamma_0} y_1 \xrightarrow{e_1} \cdots y_{n-1} \xrightarrow{e_{n-1}} z_{n-1} \xrightarrow{\gamma_{n-1}} y_n \rangle$, the string generated by it is defined recursively: $l_0(\omega) = \varepsilon$, $l_1(\omega) = l_0(\omega)\gamma_0 e_0 \cdots$, $l_n(\omega) = l_{n-1}(\omega)\gamma_{n-1}e_{n-1}$, and $l(\omega) = l_n(\omega)$.

In order to extract the unedited strings from runs in AES$_c$, we define *edit projection*, which removes the edit choices from runs.

Definition 9 (Edit Projection P_e): Given a run $\omega = \langle y_0 \xrightarrow{e_0} z_0 \xrightarrow{s_0} y_1 \xrightarrow{e_1} \cdots y_{n-1} \xrightarrow{e_{n-1}} z_n \xrightarrow{s_{n-1}} y_n \rangle$, the edit projection P_e returns the string $s = e_0 e_1 \cdots e_{n-1}$.

For synthesis purposes, one critical problem is whether PP-enforcing edit functions always exist in any given AES_c. The answer is negative and we provide a counterexample.

Example 1: Let the system G be with $E_o = \{a, b, c, d\}$, language $\mathscr{L}(G) = \overline{\{dabc, abc, b\}}$ and $L_{unsafe} = \{abc, b\}$. The edit constraint function φ is not explicitly stated here. Let us suppose it results in only one privately safe edit function f_e, which maps b to ab, abc to $dabc$, and leaves $dabc$ and its prefixes intact. However, it turns out that $\overline{f_e(L_{unsafe})} = \overline{\{dabc, ab\}}$ is not a subset of $f_e(L_{safe}) = \overline{\{dabc\}}$, so this edit function is not PP-enforcing and there does not exist a PP-enforcing edit function.

Proposition 1: Private enforceability does not always imply PP-enforceability under edit constraints.

Then it is natural to ask when there exists a PP-enforcing edit function in the given AES$_c$. To answer this question, we need to ensure that *every unsafe string shares the same edited behavior with some safe string*. In AES$_c$, one Y state may appear in multiple runs and different strings may be edited to the same string. In order to facilitate the following discussion, we split the states apart and build the *reachability tree* of AES$_c$, which is denoted by AES_t.[1] Its construction procedure is shown in Algorithm 3.

```
Algorithm 3: Build Labeled Reachability Tree of AES_c
  Input: $AES_c = (Y, Z, E_o, E_o^\varepsilon, f^c_{AES,yz}, f^c_{AES,zy}, \Gamma_c, y_0)$
  Output: $AES_t = (Y_t, Z_t, E_o, E_o^\varepsilon, f^t_{AES,yz}, f^t_{AES,zy}, \Gamma_t, y_0)$
  1: Do breadth-first search from $y_0$, view it as the root
     state, keep expansion until all the states in $AES_c$ are
     checked or termination criteria are satisfied;
  2: For each new node, i.e. $y \in Y_t$ or $z \in Z_t$, check if
     $f^c_{AES,yz}(y, e_o)$ or $f^c_{AES,zy}(z, \gamma)$ is defined at this node;
   (1) If $f^c_{AES,yz}(y, e_o)$ is not defined, then add $y$ as a leaf
       state;
   (2) If $f^c_{AES,yz}(y, e_o)$ or $f^c_{AES,zy}(y, \gamma)$ is defined, create a
       new node $z' = f^c_{AES,yz}(y, e_o)$ or $y' = f^c_{AES,zy}(z, \gamma)$;
   (3) If $y'$ is identical to a node in the path from $y_0$ to
       $z$, terminate expansion and add $y'$ as a leaf state;
  3: For every $y_t \in Y_t$, specify the run $\omega_t$ from $y_0$ to $y_t$;
  4: Get the string $l(\omega_t)$ generated by $\omega_t$, take the edit
     projection $P_e(\omega_t)$ to get the original string of $\omega_t$, use
     $(l(\omega_t), P_e(\omega_t))$ to label $y_t$;
  5: return AES$_t$.
```

In AES$_t$, states are completely split in terms of state and string components: every Y state consists of a state pair as well as a string pair where one is unedited and the other is edited. In line 2.(1) and 2.(3) of Algorithm 3, since all runs in AES$_c$ terminate in Y states by definition, all the leaf states of AES$_t$ are Y states. In line 2.(3), if one particular state appears again on the path, it means that a loop is formed in AES$_c$. There may be an infinite number of strings if there are loops in the original automaton, however, edit functions are assumed to be memoryless and always specify the same edit choice at each information state. So no information is lost if we only consider edit choices in AES$_t$.

In the reachability tree, some Y_t states consist of a secret state as well as a non-secret state while others consist of solely non-secret states. Based on this observation, we partition Y_t states in AES$_t$ as: (1) $Y_t^1 = \{((y_d, y_f), (t, s)) \in Y_t : t \in L_{safe}, s \in L_{unsafe}\}$; (2) $Y_t^2 = \{((y_d, y_f), (t, s)) \in Y_t : t, s \in L_{safe}\}$.

We also define the *last preserved Y_t^2 state* as: $Y_{lp}^2 = \{y_t^2 \in Y_t^2 : \exists y_t^1 \in Y_t^1, e_o \in E_o, \gamma \in \Gamma_t,$ s.t. $y_t^1 = f^t_{AES,zy}(f^t_{AES,yz}(y_t^2, e_o), \gamma)\}$. By definition, last preserved Y_t^2 states contain only safe string components but their successor states contain unsafe string components. Then we define $Y_{leaf}^1 = Y_{leaf} \cap Y_t^1$, $Y_{leaf}^2 = Y_{leaf} \cap Y_t^2$ respectively and collect unsafe strings appearing in Y_{leaf} as $L^u_{leaf} = \{l \in L_{unsafe} :$

[1]The terminology of *reachability tree* is from the Petri net literature; it is employed here as it is well-suited to the construction procedure in this paper.

$\exists y^1_{leaf} = ((x_d,x_f),(t,s)) \in Y^1_{leaf},\ s.t.\ s = l\}$. We number each string in L^u_{leaf} as $l_1, l_2, \cdots l_n$ and each $l_i \in L^u_{leaf}$ may appear in multiple leaf states in the tree. Similarly, we collect safe strings in Y_{leaf} as $L^s_{leaf} = \{l \in L_{safe} : \exists y^2_{leaf} = ((x_d,x_f),(t,s)) \in Y^2_{leaf},\ s.t.\ s = l\}$. Also, the set of safe strings in Y^2_{lp} is defined as: $L^s_{lp} = \{l \in L_{safe} : \exists y^2_{lp} = ((x_d,x_f),(t,s)) \in Y^2_{lp},\ s.t.\ s = l\}$. Furthermore, we group Y_t states by their state and string components: (1) $Y^1_{leaf}(l) = \{((y_d,y_f),(t,s)) \in Y^1_t : s = l \in L^u_{leaf}\}$; (2) $Y^2_{leaf}(l') = \{((y_d,y_f),(t,s)) \in Y^2_t : s = l' \in L^s_{leaf}\}$; (3) $Y^2_{lp}(l'') = \{((y_d,y_f),(t,s)) \in Y^2_{lp} : s = l'' \in L^s_{lp}\}$. Besides, we define $Y^2_l = Y^2_{leaf} \cup Y^2_{lp}$ and $Y^2_l(\tilde{l}) = \{((y_d,y_f),(t,s) \in Y^2_{leaf} \cup Y^2_{lp} : s = \tilde{l} \in L^s_{lp} \cup L^s_{leaf}\}$.

By the definition of L_{safe}, if an edit function maps a string s to a safe string t, then all prefixes of s are mapped to some safe strings. Formally speaking, the following theorem holds and we can focus on L^u_{leaf} for synthesis purposes.

Theorem 1: Consider any edit function f_e, if $s,t \in E^*_o$ satisfy $f_e(s) = f_e(t)$, then $\forall s' \preceq s, \exists t' \preceq t$, s.t. $f_e(s') = f_e(t')$.

VI. Synthesis of PP-enforcing Edit Functions

In this section, we address the problems of verifying the existence of PP-enforcing edit functions and synthesizing such a function if one exists.

The essence of verifying the existence of a PP-enforcing edit function is to ensure that every unsafe string in L^u_{leaf} shares the same behavior with certain strings in $L^s_{leaf} \cup L^s_{lp}$ under the effect of an edit function. Suppose there are n strings in $L^u_{leaf} : l_1, l_2, \cdots l_n$ and each set of $Y^1_{leaf}(l_i)$ contains n_i states. We pick one leaf state from each set of $Y^1_{leaf}(l_i)$ and form a *combination of unsafe strings* in AES$_t$. Each combination is denoted as $c^1_j = [y^1_{1j_1}, y^1_{2j_2}, \cdots, y^1_{nj_n}], y^1_{ij_i} \in Y^1_{leaf}(l_i)\ 1 \le i \le n, 1 \le j_i \le n_i, 1 \le j \le \tilde{n}$ where $\tilde{n} = \prod_{i=1}^{n} n_i$ is the total number of combinations of unsafe strings. Similarly, suppose there are m safe strings in $L^s_{leaf} \cup L^s_{lp}$ and each set of $Y^2_l(l_q)$ contains m_q states. We pick one state from each $Y^2_l(l_q)$ and form a *combination of safe strings* in AES$_t$. Each combination is denoted as $c^2_k = [y^2_{1k_1}, y^2_{2k_2}, \cdots, y^2_{mk_m}], y^2_{qk_q} \in Y^2_l(l_q), 1 \le q \le m, 1 \le k_q \le m_q, 1 \le k \le \tilde{m}$ where $\tilde{m} = \prod_{q=1}^{m} m_q$ is the total number of combinations of safe strings. Both types of combinations contain the information of how a string is edited and thus can be used to evaluate if an edit function is PP-enforcing. The reason why we also need to consider L^s_{lp} is as follows: if an unsafe string has a safe prefix, then an edit function may erase its unsafe suffix to make it safe. In order to synthesize a PP-enforcing edit function, we try to find one *PP-enforcing combination pair* in AES$_t$, which determines the existence of PP-enforcing edit functions.

Definition 10 (PP-enforcing combination pair): Given a combination c^1_j of unsafe strings and a combination c^2_k of safe strings, they form a PP-enforcing combination pair if $\forall y^1_{ij_i} = ((x_{obsd1}, x_{obsf1}),(t_1,s_1)) \in c^1_j,\ \exists y^2_{qk_q} = ((x'_{obsd1}, x'_{obsf1}),(t'_1,s'_1)) \in c^2_k$, s.t. $t_1 \preceq t'_1$.

Theorem 2: A PP-enforcing edit function exists if and only if a PP-enforcing combination pair exists in the AES$_c$.

The proof is omitted here and based on this theorem, we propose the following algorithm to synthesize PP-enforcing edit functions and give an illustrative example hereafter.

Algorithm 4: Synthesize Deterministic PP-enforcing Edit Functions

```
Input  : $AES_t = (Y_t, Z_t, E_o, E^\varepsilon_o, f^t_{AES,yz}, f^t_{AES,zy}, \Gamma_t, y_0)$
Output: A deterministic PP-enforcing edit function or
        non-existence of such functions
1  Enumerate all possible candidate combinations for
   unsafe strings: $c^1_j = [y^1_{1j_1}, y^1_{2j_2}, \cdots, y^1_{nj_n}]$
2  Enumerate all possible candidate combinations for safe
   strings: $c^2_k = [y^2_{1k_1}, y^2_{2k_2}, \cdots, y^2_{mk_m}]$.
3  for $j = 1 : \tilde{n}$ do
4      Consider $c^1_j = [y^1_{1j_1}, y^1_{2j_2}, \cdots, y^1_{nj_n}]$
5      for $k = 1 : \tilde{m}$ do
6          if $\exists c^2_k$ that forms a PP-enforcing combination
           pair with $c^1_j$. then
7              Mark all the states on runs from $y_0$ to states
               in $c^1_j \cup c^2_q$.
8              Return the marked substructure as a
               PP-enforcing edit function.
9  if No marked substructure is returned then
10     No PP-enforcing edit function exists.
```

Example 2: In this example, we show the whole process of synthesizing PP-enforcing edit functions. Suppose G has observable events $E_o = E = \{a,b,c,d\}$ and $X_S = \{5\}$. Since $E_o = E$, G coincides with its current state estimator $\mathscr{E}$ in Figure 1. First, we build the desired estimator $\mathscr{E}^d$ by removing state 5 from G and taking the accessible part. Then we build the feasible estimator $\mathscr{E}^f$ by adding self-loops at each state and ε transitions along every defined transition. Then we do the verifier parallel composition to get verifier V, which is not shown due to space limitations.

Next, we unfold V to obtain V_u and finally obtain AES$_c$ in Figure 2 after pruning away some edit choices under edit constraints $\varphi(2,5) = 0$ and $\varphi(3,0) = 0$. There are two types of states in AES$_c$: the square Y states where the system plays and the oval Z states where the edit function plays. The transitions from Y states are events observed by the edit function and the events from Z states are choices of the edit function. The game is initialized at state $(0,0)$ where events a,b,d are the system's observable outputs. After a is observed, the game reaches state $((0,0),a)$ and the edit function begins to play and it inserts d before a. Thus the intruder will observe da. All the transitions in the structure can be interpreted in a similar manner.

With AES$_c$ built, we proceed to the step of building the reachability tree in Figure 3. There are 3 leaf states in the tree and 2 of them are Y^1_{leaf} states, which are indicated by red dash lines. In this example, Y^1_{leaf} states are grouped as: $Y^1_{leaf}(bc) = \{((0,0),(dabc,bc)))\}$, $Y^1_{leaf}(abc) = \{((0,0),(dabc,abc))\}$. So we get the combination for unsafe strings as $c^1_1 = [((0,0),(dabc,bc)),((0,0),(dabc,abc))]$. Also, Y^2_{leaf} states are grouped as: $Y^2_{leaf}(dabc) = \{((0,0),(dabc,dabc))\}$ and the only Y^2_{lp} state is $Y^2_{lp}(a) = \{((2,4),(da,a))\}$. So we get the combination for safe strings as $c^2_1 = [((0,0),(dabc,dabc)),((2,4),(da,a))]$. Then we consider these two combinations: since $\exists((0,0),(dabc,dabc)) \in c^2_1$, s.t. $dabc \preceq dabc$, c^1_1 and c^2_1 form a PP-enforcing combination pair. So there exists a PP-enforcing edit function f_e in this example, which inserts da every time b occurs from state 0 and inserts d every time a occurs from state 0.

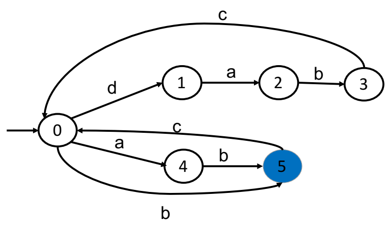

Fig. 1. Observer of the system

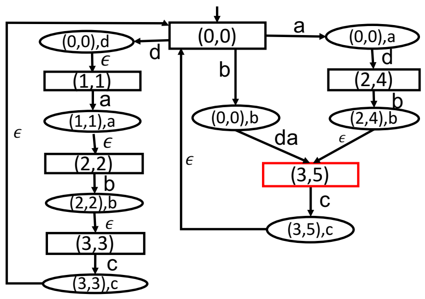

Fig. 2. AES under constraints–AES$_c$

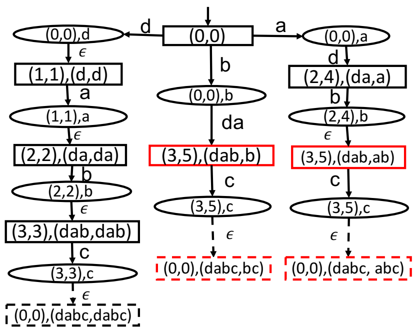

Fig. 3. Reachability tree labeled with strings

VII. Conclusion

This paper extends our prior method of privately safe enforcing edit functions to public-private enforcing edit functions, which can enforce opacity in a more adverse situation. We formally characterize PP-enforcing edit functions and constrain their functionality by introducing edit constraints to avoid trivial solutions. We further show that edit constraints may cause the non-existence of PP-enforcing edit functions. An algorithm is proposed to verify the existence of PP-enforcing edit functions and synthesize them if they exist. In future work, it would be of interest to consider other types of edit functions, such as non-deterministic ones.

References

[1] B. Bérard, J. Mullins, and M. Sassolas. Quantifying opacity. *Mathematical Structures in Computer Science*, 25(Special issue 2):361–403, 2015.

[2] J. W. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalised 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 – 2nd Edition*. Springer, 2008.

[4] F. Cassez, J. Dubreil, and H. Marchand. Synthesis of opaque systems with static and dynamic masks. *Formal Methods in System Design*, 40(1):88–115, 2012.

[5] S. Chédor, C. Morvan, S. Pinchinat, and H. Marchand. Diagnosis and opacity problems for infinite state systems modeled by recursive tile systems. *Discrete Event Dynamic Systems: Theory and Applications*, 25(1-2):271–294, 2015.

[6] P. Darondeau, H. Marchand, and L. Ricker. Enforcing opacity of regular predicates on modal transition systems. *Discrete Event Dynamic Systems: Theory and Applications*, 25(1-2):251–270, 2015.

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

[8] Y. Falcone and H. Marchand. Enforcement and validation (at runtime) of various notions of opacity. *Discrete Event Dynamic Systems: Theory and Applications*, 25(4):531–570, 2015.

[9] R. Jacob, J.-J. Lesage, and J.-M. Faure. Overview of discrete event systems opacity: Models, validation, and quantification. *Annual Reviews in Control*, 2016.

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

[11] A. Saboori and C. N. Hadjicostis. Notions of security and opacity in discrete event systems. *Proc. of the 46th IEEE Conference on Decision and Control*, pages 5056–5061, Dec 2007.

[12] A. Saboori and C. N. Hadjicostis. Current-state opacity formulations in probabilistic finite automata. *IEEE Transactions on Automatic Control*, 59(1):120–133, 2014.

[13] S. Takai and Y. Oka. A formula for the supremal controllable and opaque sublanguage arising in supervisory control. *SICE Journal of Control, Measurement, and System Integration*, 1(4):307–311, 2008.

[14] Y. Tong, Z. Li, C. Seatzu, and A. Giua. Verification of state-based opacity using Petri nets. *IEEE Transactions on Automatic Control*, 62(6):2823–2837, 2017.

[15] Y.-C. Wu. and S. Lafortune. Comparative analysis of related notions of opacity in centralized and coordinated architectures. *Discrete Event Dynamic Systems: Theory and Applications*, 23(3):307–339, 2013.

[16] Y.-C. Wu. and S. Lafortune. Synthesis of insertion functions for enforcement of opacity security properties. *Automatica*, 50(5):1336–1348, 2014.

[17] Y.-C. Wu and S. Lafortune. Synthesis of opacity-enforcing insertion functions that can be publicly known. In *Proceedings of the 54th IEEE Conference on Decision and Control*, pages 3506–3513, 2015.

[18] Y.-C. Wu, V. Raman, S. Lafortune, and S. A. Seshia. Obfuscator synthesis for privacy and utility. In *Proceedings of the 8th NASA Formal Methods Symposium (NFM)*, pages 239–248, June 2016.

[19] X. Yin and S. Lafortune. A general approach for solving dynamic sensor activation problems for a class of properties. In *Proceedings of the 54th IEEE Conference on Decision and Control*, pages 3610–3615, 2015.

[20] X. Yin and S. Lafortune. A uniform approach for synthesizing property-enforcing supervisors for partially-observed discrete-event systems. *IEEE Transactions on Automatic Control*, 61(8):2140–2154, 2016.

[21] X. Yin and S. Lafortune. A new approach for the verification of infinite-step and K-step opacity using two-way observers. *Automatica*, 80:162–171, 2017.

[22] B. Zhang, S. Shu, and F. Lin. Maximum information release while ensuring opacity in discrete event systems. *IEEE Transactions on Automation Science and Engineering*, 12(3):1067–1079, 2015.