Obfuscator Synthesis for Privacy and Utility*

Yi-Chin Wu[1,2], Vasumathi Raman[3],
Stéphane Lafortune[2], and Sanjit A. Seshia[1]

[1] UC Berkeley yichin.wu@berkeley.edu, sseshia@eecs.berkeley.edu
[2] University of Michigan stephane@umich.edu
[3] United Technologies Research Center ramanv@utrc.utc.com

Abstract. We consider the problem of synthesizing an obfuscation policy that enforces privacy while preserving utility with formal guarantees. Specifically, we consider plants modeled as finite automata with pre-defined secret behaviors. A given plant generates event strings for some useful computation, but meanwhile wants to hide its secret behaviors from any outside observer. We formally capture the privacy and utility specifications using the automaton model of the plant. To enforce both specifications, we propose an obfuscation mechanism where an edit function "edits" the plant's output in a reactive manner. We develop algorithmic procedures that synthesize a correct-by-construction edit function satisfying both privacy and utility specifications. To address the state explosion problem, we encode the synthesis algorithm symbolically using Binary Decision Diagrams. We present EdiSyn, an implementation of our algorithms, along with experimental results demonstrating its performance on illustrative examples. This is the first work, to our knowledge, to successfully synthesize controllers satisfying both privacy and utility requirements.

1 Introduction

Many systems transmit information to the outside world during their operation. For example, location-based services require devices such as smartphones to transmit location information to other devices or to servers in the cloud. Similarly, in defense and aerospace applications, a network of drones may need to broadcast location information to a variety of agents, including other drones, ground personnel, and remote base stations. These settings often involve nodes that are resource-constrained or connected in ad-hoc, dynamically-changing networks. Some of the transmitted information may reveal secrets about the system or its users; therefore, privacy is an important design consideration. At the same time, the agents to which this information is being sent must have enough information to provide relevant services or perform other actions. Thus, the transmission of information from the system to the outside world needs to balance the contrasting requirements of privacy and utility.

Consider the following illustrative example:

*This work was supported in part by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and in part by the National Science Foundation under grants CCF-1138860 and CCF-1139138 (NSF Expeditions in Computing project ExCAPE: Expeditions in Computer Augmented Program Engineering) and CNS-1421122.

Example 1. We consider a user Alice moving in a building. Information about Alice's location needs to be sent to a server and other agents in order to perform some useful actions; e.g., adjusting the heating system based on Alice's location and other occupancy levels in the building, or directing her to the closest coffee machine. Suppose also that there are some "secret" locations, and that Alice does not want others to know when or whether she visits these locations. An example could be a room containing highly sensitive data, such that the mere act of being able to visit it discloses compromising information that Alice wishes to protect (i.e., there are only a handful of people who can visit this room, and their identity is to be kept secret). However, Alice also wants the server to be able to compute some information that is useful based on her location, because otherwise Alice is always cold and uncaffeinated.

Suppose that an "event generator" (e.g., on Alice's phone) generates events based on her movements, and broadcasts these events to other agents. Suppose further that the quality of the service that requires tracking Alice's reported location degrades based on the Euclidean distance from her true location. How can one generate an output event stream that does not reveal whether Alice visited a secret location while also providing sufficient accuracy for determining her location for the relevant services?

Following the terminology used in supervisory control of discrete event systems [9], we refer to the combination of the event generator and the process it is based on (Alice, in our example) as the *plant*. Our goal is to introduce an element of decision-making into the event generator so that it can modify the events to be output before relaying them in order to meet both the privacy and utility requirements. We refer to this decision-making as an *obfuscation policy*.

In this paper, we present a formalization of this problem, along with an algorithm to synthesize an obfuscation policy. We are given a plant modeled as a finite automaton, with formally specified secret behaviors and a specification of utility. The plant must generate event strings that provide sufficient utility while hiding its secret behaviors from an outside observer. The privacy and utility specifications are captured as automata-theoretic requirements on the model of the plant. To enforce both specifications, we propose an obfuscation mechanism whereby the plant *edits* its output in a reactive manner, such that all resulting output strings provably satisfy the specifications. The presented algorithm synthesizes a correct-by-construction edit function that maps true executions of the plant to ones that achieve the privacy and utility specifications.

The paper is structured as follows. We first define the obfuscation problem in Section 2. In Section 3 we describe our algorithm for automatically editing reported values. The treatment in this section is "explicit", i.e., in terms of graph operations on discrete game structures. To address the state explosion problem, we encode the synthesis algorithm symbolically using Binary Decision Diagrams (BDDs) [1], as described in Section 4. We then demonstrate our approach empirically in Section 5, using EdiSyn, an open source Python toolkit we developed for this purpose. We conclude after a discussion about related work and future directions.

2 Preliminaries and Problem Statement

2.1 Preliminaries

A Nondeterministic Finite Automaton (NFA) is a tuple $G = (Q, \Sigma, \delta, Q_0)$ with a finite set of states Q, a finite set of events Σ, a partial state transition function $\delta : Q \times \Sigma \to$

2^Q, and a set of initial states $Q_0 \subseteq Q$. An NFA G is called a Deterministic Finite Automaton (DFA) when $|Q_0| = 1$ and $|\delta(q, e)| \leq 1$ for every state $q \in Q$ and event $e \in \Sigma$. More explicitly, for a DFA as $G = (Q, \Sigma, \delta, q_0)$, the single initial state is $q_0 \in Q$ and the transition function is $\delta : Q \times \Sigma \to Q$, which deterministically defines the next state given the current state and the event.

Given an NFA transition function $\delta : Q \times \Sigma \to 2^Q$, we extend it to $\delta^* : Q \times \Sigma^* \to 2^Q$ recursively as follows: $\delta^*(q, \varepsilon) = \{q\}$, $\delta^*(q, e) = \delta(q, e)$, $\delta^*(q, e_1 e_2 \cdots e_n) = \cup_{q' \in \delta^*(q, e_1)} \delta^*(q', e_2 \cdots e_n)$, where Σ^* is the set of finite strings of events and ε denotes the empty string. The *language* generated by G is the set of strings defined by $\mathcal{L}(G) := \{t \in \Sigma^* : \exists q_0 \in Q_0 \text{ s.t. } \delta^*(q_0, t) \neq \emptyset\}$. A DFA transition function $\delta : Q \times \Sigma \to Q$ is extended to $\delta^* : Q \times \Sigma^* \to Q$ in a similar manner. Also, the language of a DFA is defined similarly. Given string t, we use $t' \preccurlyeq t$ to denote that string t' is a prefix of t, and use $t^{1:k}$ to denote the length-k prefix of t. Finally, $|t|$ denotes the length of t.

In this paper, the system of interest, called the *plant*, is modeled as a DFA $G = (Q, \Sigma, \delta, q_0)$. In our model, the state of the plant cannot be observed directly. However, upon each transition, an event is emitted and can be observed by an outside observer. Hence, an outside observer can infer the state of the plant based on the observation of the string of events emitted upon transitions.

2.2 Threat Model

We consider a scenario where the plant G has a set of *secret states* $Q_S \subset Q$ that need to be kept hidden from the outside observer. The observer of the plant's output strings is a passive-but-curious adversary that has a copy of G, and can see all strings output by the plant; the observer can mimic transitions in its copy of G based on the output strings. We assume that the observer is also a legitimate recipient in the sense that the plant emits strings in order to deliver some information to the observer. However, the plant also wants to hide from the observer whether it is ever in a secret state.

In the following, we will call a string $t \in \mathcal{L}(G)$ a *secret string* if $\delta^*(q_0, t) \in Q_S$ and a *public string* otherwise.

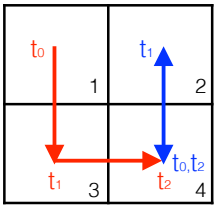
Example 2. Alice and Bob are trying to arrange a secret meeting to exchange a top secret package in a $m \times n$ grid world. We model the generator of Alice and Bob's movements as a plant $G = (Q, \Sigma, \delta, q_0)$ with secret states Q_S, where

- The set of states is $Q = Loc^2 \cup \{init\}$ where $Loc = \{1, \cdots, m \times n\}$ contains the set of all locations on the grid word.
- The set of events is $\Sigma = \{a_{ij} b_{kl} : i, j, k, l \in Loc\}$, where $a_{ij} b_{kl}$ specifies Alice's movement from i to j and Bob's movement from k to l.
- The transition function δ is defined such that, for both Alice and Bob, only moving to neighboring locations or staying at the current location is allowed.
- The set of secret states is $Q_S = \{(i, k) \in Loc : i = k\}$, where Alice and Bob are in the same location.

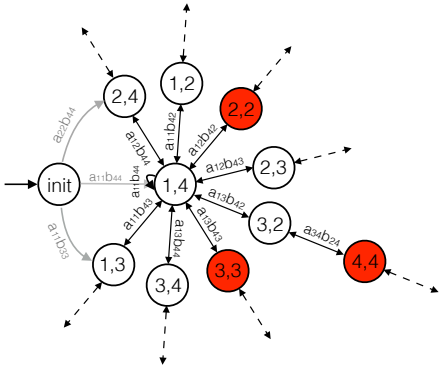
We show in Figure 1 the 2×2 grid world and a partial plant automaton of G representing the generator of Alice and Bob's movements. The full model of G contains 17 states and 144 events. Because of space limitations, we do not draw all the states and transitions, and only show a partial plant automaton of G. State $init$ is introduced to model the

initial moment when no locations from Alice and Bob have been reported. For each state $(i, j) \in Loc^2$, there is a transition from $init$ to (i, j) with event label $a_{ii}b_{jj}$. For each state (i, j), there is a transition from (i, j) to (k, l) with event label $a_{ik}b_{jl}$ as long as k is a neighboring location of i and l is a neighboring location of j. The red states in G are secret where Alice and Bob meet in the same location.

Let the quality of the service degrade with the L_1 distance from Alice's and Bob's true locations. That is, the quality loss from state (i, j) to state (k, l) is $||(i_x, i_y, j_x, j_y) - (k_x, k_y, l_x, l_y)||_1 = |i_x - k_x| + |i_y - k_y| + |j_x - l_x| + |j_y - l_y|$, where i_x and i_y are the x-coordinate and y-coordinate of location i, and similarly for locations j, k, l. Hence, in reporting the locations of Alice and Bob, we would like to maintain the L_1-distance between the real and the reported locations within some allowable range. This could be because we want an external observer to be able to track the progress of Alice and Bob towards their goal of meeting, while not knowing exactly when or where they meet.



(a) The 2×2 grid world and samples traces of Alice (red) and Bob (blue). Alice and Bob meet at time t_2.



(b) A partial plant automaton of G modeling the generator of Alice and Bob's movements. The secret states are colored in red.

Fig. 1. The 2×2 grid world and a partial plant automaton G representing the generator of Alice and Bob's movements.

2.3 Edit Functions

To defend against attacks as described in the previous section, we propose to add an interface at the output of the plant that hides secret strings while preserving the utility of the original string. The interface *edits* the plant's original string t as it is produced ("online"), such that the resulting string $\tilde{t}$ after editing never reveals the secret, and yet preserves the utility of t within an allowable range. As this interface is a function that maps each plant output event to another event or string, we refer to it as an *edit function*.

We permit edit functions $f_e : \Sigma^* \times \Sigma \to \Sigma^*$ that map an output event to another event or string with *one* replacement, deletion, or insertion operation. Given past output string t, $f_e(t, e) = o$ means that the plant's output event e is edited to o. Note that $o \in \Sigma^*$ in general because we allow event insertion as well as deletion. If event e is deleted, then the output is $o = \epsilon$; on the other hand, if a string t_I is inserted before e, then the output is $o = t_I e$. Every edit function is *causal*: it can only edit the current output event e and not any previous output. For convenience of notation, we also define

a *string-based* edit function $\hat{f}_e : \Sigma^* \to \Sigma^*$ recursively from f_e such that $\hat{f}_e(\varepsilon) = \varepsilon$ and $\hat{f}_e(te) = \hat{f}_e(t)f_e(t, e)$. Note that, in general, $\hat{f}_e$ is a partial function, and $\hat{f}_e(t)$ may only be defined for selected $t \in \Sigma^*$. Also, since an edit function is causal, its string-based version is *prefix-preserving*: $\forall t_1, t_2 \in \Sigma^*, t_1 \preccurlyeq t_2 \Rightarrow f_e(t_1) \preccurlyeq f_e(t_2)$. An edit function of the above form can be implemented by a deterministic, potentially infinite-state automaton, which we call the *edit automaton*, and denote by $\mathcal{EA} = (S, \Sigma, Trans, s_0)$. The elements of the $\mathcal{EA}$ tuple are the set of states S, the set of events Σ, the transition relation $Trans \subseteq S \times \Sigma \times \Sigma^* \times S$, and the initial state s_0. Each transition in $Trans$ is a tuple (s, e, o, s') of the starting state s, the input event e, the output string o, and the target state s'. Given an edit function f_e, there is a corresponding transition relation $Trans$ with $(s, e, o, s') \in Trans$ iff $f_e(t, e) = o$ and s is the state reached on input string t. The transition relation for f_e is deterministic: $\forall s \in S, \forall e \in \Sigma, |\{s' : (s, e, o, s') \in Trans, o \in \Sigma^*\}| = 1$.

Throughout the paper, we use "edit" to collectively refer to any replacement, deletion, and insertion operation. We will call the output string from the plant as the *original string* t, and call the string after editing as the *obfuscated string* $\tilde{t}$.

2.4 Problem Formulation

Our goal is to synthesize an edit function f_e that hides the plant's secret strings while preserving the utility of the original strings within some allowable range. We capture the *utility* of each original string t by the final state that is reached by t in the plant DFA, and define the utility loss in mapping t to $\tilde{t}$ by the utility difference between the states reached by t and $\tilde{t}$. Without loss of generality, we model the utility loss by an integer-valued distance metric $D : Q \times Q \to \mathbb{N}$. The formal statement of the synthesis problem is as follows:

Problem 1 (Edit Synthesis) *Given a plant modeled as DFA $G = (Q, \Sigma, \delta, q_0)$ with a set of secret states $Q_S \subset Q$, utility distance $D : Q \times Q \to \mathbb{N}$, and accuracy budget $W \in \mathbb{N}$, construct an edit automaton $\mathcal{EA} = (S, \Sigma, Trans, s_0)$ implementing an edit function f_e such that:*

(1) $\forall t \in \mathcal{L}(G)$, $\hat{f}_e(t)$ *is defined*
(2) $\forall t \in \mathcal{L}(G), \delta^*(q_0, \hat{f}_e(t)) \neq \emptyset$ *and* $\delta^*(q_0, \hat{f}_e(t)) \notin Q_S$ *(privacy specification)*
(3) $\forall te \in \mathcal{L}(G)$ *where* $t \in \Sigma^*$ *and* $e \in \Sigma$, $D\big(\delta^*(q_0, te), \delta^*(q_0, \hat{f}_e(t)o)\big) \leq W$ *and* $D\big(\delta^*(q_0, t), \delta^*(q_0, \hat{f}_e(t)o^{1:k})\big) \leq W$ *where* $o = f_e(t, e)$, *for* $k = 1, \ldots, |o| - 1$ *(utility specification)*

Remark 1. Note that the privacy specification is a safety property on the output of the edit function.

3 Edit Synthesis Algorithm

We solve Problem 1 by formulating it as a safety game between the edit function and the plant. Such game formulations are common for program synthesis, where the program is modeled as a protagonist playing against the adversarial environment, with the goal of satisfying a given specification. For the edit synthesis problem, the edit function is

the "program" to be synthesized, and the plant is the environment that provides inputs to the edit function: adversarialism here means that the plant can evolve arbitrarily, and the edit function must satisfy the specification under all possible evolutions.

3.1 Edit Patterns Satisfying the Specifications

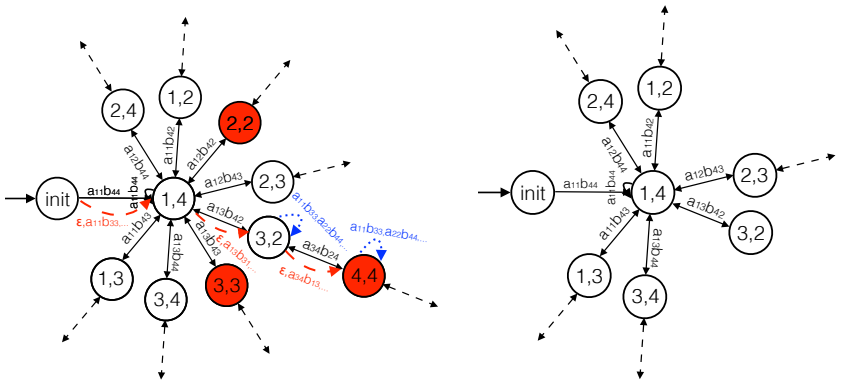
To construct the game, we first want to easily determine whether an edit pattern satisfies the privacy and utility specifications. One challenge is that determining whether an edit pattern satisfies the privacy and utility specifications requires examining not only the obfuscated string, but also its distance from the original string. Fortunately, we can construct an NFA that recognizes all valid edit patterns.

Lemma 1 *There exists an NFA $\mathcal{PA}$, with state space $O(|Q|^2)$, that recognizes all edit patterns satisfying the privacy specification in Problem 1.*

Proof Sketch. Given G, we first build the "edit-pattern" NFA $G_e = (Q, \Sigma \cup \{\varepsilon\}, \delta_e, q_0)$ that recognizes all edit patterns, by adding transitions to G. Transition function δ_e is defined with respect to decomposition $\delta_e := \delta \cup \delta_r \cup \delta_i$. More concretely, consider the plant G in Figure 1(b). The corresponding G_e is built as shown in Figure 2(a), such that (i) all original transitions exist, as depicted by the (black) solid arrows; (ii) the replacement transitions δ_r are defined by adding a replacement transition for every event in parallel with the original transition, as depicted by the (red) dashed arrows; and (ii) the insertion transitions δ_i are defined by adding a self loop for every event at every state, as depicted by the (blue) dotted arrows. No replacement or insertion transition is added if an original transition for the given event already exists. Deletion is subsumed by replacement, as deleting an event is the same as replacing the event by the empty string ε. We then construct in Figure 2(b) the "public-behavior" DFA $G_p = (Q, \Sigma, \delta_p, q_0)$ from G, by pruning away all secret states. Finally, to find all edit patterns satisfying the privacy specification, we compose G_e and G_p and build the product automaton $\mathcal{PA}$. Specifically, the composition synchronizes δ and δ_p (the original transitions), δ_r and δ_p (the replacement transitions), and δ_i and δ_p (the insertion transitions), thereby preserving the edit choices. In sum, since G_e recognizes all edit patterns and G_p recognizes all public behaviors, $\mathcal{PA}$ recognizes each edit pattern for which no obfuscated string ever visits secret states. □

Note that, as an interface at the output of the plant, the edit function does not change the plant's original dynamics. This feature is captured in our construction of G_e: neither insertion nor replacement transition changes the real plant state in G_e. Consider an edit pattern with $t, \tilde{t}$, and edit operations. We can uniquely determine a path because each edit transition function is deterministic. Given an edit pattern from t to $\tilde{t}$, by the construction of $\mathcal{PA}$, the ending state of the trace of this edit pattern in $\mathcal{PA}$ is a state pair (q_e, q_p) where $q_e = \delta^*(q_0, t)$ is the plant's real state and $q_p = \delta^*(q_0, \tilde{t})$ is the state perceived by the outside observer based on $\tilde{t}$. Hence, with $\mathcal{PA}$ capturing the pair $(q_e, q_p) = (\delta^*(q_0, t), \delta^*(q_0, \tilde{t}))$ for every t, we can now build from $\mathcal{PA}$ an NFA that recognizes all edit patterns satisfying both the privacy and the utility specifications.

Lemma 2 *There exists an NFA $\mathcal{A}$, with state space $O(|Q|^2)$, that recognizes all edit patterns satisfying the privacy and the utility specifications in Problem 1.*



(a) The edit-pattern NFA G_e (b) The public-behavior DFA G_p

Fig. 2. Partial automata of the edit-pattern NFA G_e and the public-behavior DFA G_p for the plant G in Figure 1(b). In G_e, the solid black arrows depict the original transitions, the dotted blue arrows depict the insertion transitions, and the dashed red arrows depict the replacement transitions.

Proof. Consider $\mathcal{PA}$ from Lemma 1 that recognizes all editing patterns satisfying the privacy specification. Because the distance function D is defined with respect to state pairs, we can determine if the given editing pattern violates the utility specification based on the reached state pair in $\mathcal{PA}$. That is, we can build $\mathcal{A}$ from $\mathcal{PA}$ by pruning all (q_e, q_p) where $D(q_e, q_p) > W$. □

3.2 Safety Game Formulation

The edit synthesis problem is formulated as a safety game between the edit function and the plant. In the safety game, the outputs of the plant are the inputs to the edit function, and the edit function must react to its inputs (i.e., the plant's outputs) and satisfy both the privacy and the utility specifications. If the edit function can reactively satisfy the specifications regardless of what the plant does, then its reactions form a winning strategy in the formulated safety game. Conversely, a winning strategy in the safety game can be converted into an edit function that solves the edit synthesis problem.

Formally, a two-player safety game structure is $\mathcal{GS} = (V_1, V_2, \Sigma, \rho_1, \rho_2, v_0)$ where V_1 and V_2 are sets of game positions, Σ is the action set, $\rho_1 : V_1 \times \Sigma \to V_2$ and $\rho_2 : V_2 \times \Sigma^* \to (V_1 \cup \bot)$ are the transition functions, and $v_0 \in V_1$ is the initial position. We note that, actions of the edit function are strings in Σ^* and ρ_2 has a domain of $V_2 \times \Sigma^*$ because the edit function can react by inserting a string. The game starts with player 1, and subsequent plays alternate between players 1 and 2. Position $\bot$ is a special position where player 2 loses and player 1 wins.

In the game corresponding to the edit synthesis problem, player 1 is the plant, who moves on positions in V_1 according to transition function ρ_1, and player 2 is the edit function, who moves on positions in V_2 according to ρ_2. A play of $\mathcal{GS}$ is a sequence of positions $v_0 v_1 v_2 \cdots \in (V_1 V_2)^*$ that starts from the initial position. Given a play, the edit function wins if $\bot$ is never visited, and the plant wins otherwise.

Consider the automaton $\mathcal{A} = (Q^2, \Sigma \cup \{\varepsilon\}, \delta_A, q_{A,0})$ in Lemma 2. Recall from Lemma 1 that the transition function of G_e is decomposed, and the synchronous composition used to obtain $\mathcal{PA}$ distinguishes edit choices. Hence, δ_A can also be decomposed into the original transition function $\delta_{A,o}$, the replacement transition function $\delta_{A,r}$, and the insertion transition function $\delta_{A,i}$. We build the safety game structure $\mathcal{G}S = (V_1, V_2, \Sigma, \rho_1, \rho_2, v_0)$ between the edit function and the plant from $\mathcal{A}$ as follows.

- $V_1 = Q^2 \cup \{\bot\}, V_2 = Q^2 \times \Sigma$
- $\rho_1 : V_1 \times \Sigma \to V_2$ is defined such that $\forall (q_e, q_p) \in V_1, \forall e \in \Sigma$, $\rho_1((q_e, q_p), e) = ((q_e, q_p), e)$ if $\delta(q_e, e)$ is defined
- $\rho_2 : V_2 \times \Sigma^* \to V_1$ is defined such that $\forall ((q_e, q_p), e) \in V_2, \forall o \in \Sigma^*$, we have the following four cases:
  - (i) $\rho_2\big(((q_e, q_p), e), o\big) = \delta_{A,o}((q_e, q_p), e)$ if $o = e$ and $\delta_{A,o}((q_e, q_p), e)$ is defined
  - (ii) $\rho_2\big(((q_e, q_p), e), o\big) = \delta_{A,r}((q_e, q_p), o) = (q'_e, q'_p)$ if $o \in (\Sigma \setminus \{e\}) \cup \{\varepsilon\}$, $\delta_{A,r}((q_e, q_p), o)$ is defined, and $q'_e = \delta(q_e, e)$
  - (iii) $\rho_2\big(((q_e, q_p), e), o\big) = \delta_{A,o}\big(\delta^*_{A,i}((q_e, q_p), t_I), e\big)$ if $o = t_I e$, $t_I \in \Sigma^*$, and $\delta_{A,o}\big(\delta^*_{A,i}((q_e, q_p), t_I), e\big)$ is defined.
  - (iv) $\rho_2\big(((q_e, q_p), e), o\big) = \bot$ if none of cases (i)-(iii) holds
- $v_0 = q_{A,0} = (q_0, q_0)$

Transition functions ρ_1 and ρ_2 define all possible actions of the plant and the edit function, respectively. Specifically, ρ_1 captures the plant dynamics and is determined by the plant's transition function δ. On the other hand, ρ_2 defines all edit actions. Cases (i)-(iii) are edit actions defined in $\mathcal{A}$, which by Lemma 2 satisfy both the private and the utility specifications. In particular, the edit function outputs the original event from the plant in case (i), replace or delete the plant's original output event in case (ii), insert events before the plant's original output event in case (iii). In case (iv), the edit action cannot satisfy the specifications and leads to the losing position $\bot$. For every plant's output event, the edit function reacts with *one* edit operation.

4 Symbolic Encoding of Edit Synthesis

So far we have assumed that the plant automaton model in the edit synthesis problem is given explicitly, i.e., as an explicit list of states and transitions. However, in practice, such explicit representations lead to what is known as the state explosion problem: a system with n variables that take k possible values requires at least k^n states to model, and thus these models quickly become impractical. In order to mitigate the state explosion problem, we represent the plant model symbolically using sets of states and sets of transitions, both represented compactly as implicit solutions to logical equations. We can then analyze the state space symbolically using Binary Decision Diagrams (BDDs) [1]. By using BDDs to reason about propositional formulas representing the state space, we avoid building the state graph explicitly.

In this section, we present our encoding of the given plant automaton symbolically using propositional formulae. We will explain how the safety game can be constructed symbolically, as well as how to extract a winning edit strategy from the symbolic encoding of the safety game.

4.1 Symbolic Automata

Given an explicit DFA $G = (Q, \Sigma, \delta, Q_0)$, we encode G symbolically as $(B^Q, B^\Sigma, \Delta_\delta, \overline{b^{q_0}})$, where $B^Q = \{y_1^q, \cdots, y_n^q\}$ is the set of Boolean variables that encode the states, $B^\Sigma = \{y_1^e, \cdots, y_m^e\}$ is the set of Boolean variables encoding the events, $\Delta_\delta : B^Q \times B^\Sigma \times B^{Q'} \to \{0,1\}$ is the propositional formula representing the transition function δ, and $\overline{b^{q_0}}$ is the Boolean encoding of the initial state. The *primed set* $B^{Q'} = \{y_1^{q'}, \cdots, y_n^{q'}\}$ is the Boolean variables that encode the target states in transitions. Given Boolean variable set $\{y_1, \cdots, y_n\}$, we use $\bar{y}$ to denote the variable tuple $(y_1, \cdots, y_n)$. We will write $\chi(\bar{b})$ if a function χ over variables $\bar{y}$ is evaluated with the Boolean vector $\bar{b} = (b_1, \cdots, b_n)$. For a function χ of variables $\bar{y}$, we use $\chi\{\bar{y} \leftarrow \bar{z}\}$ to denote the new function obtained from χ with the variable y_i renamed to z_i.

With a slight abuse of notation, we write $\overline{b^q} \in Q$ if $\overline{b^q}$ is the Boolean encoding of a state in Q and use $\overline{b^q}$ directly to refer to the given state; similar notation applies for events and primed states. We use Δ to denote the propositional formulae for transition functions. Propositional formula Δ_δ is defined such that $\Delta_\delta(\overline{b^q}, \overline{b^\Sigma}, \overline{b^{q'}}) = 1$ iff $\overline{b^{q'}} \in \delta(\overline{b^q}, \overline{b^\Sigma})$.

To symbolically solve the edit synthesis problem, it remains for us to encode the privacy and utility specifications. We encode the secret state set Q_S as a Boolean function $\chi_{Q_S} : B^Q \to \{0,1\}$ such that $\chi_{Q_s}(\overline{b^q}) = 1$ iff state $\overline{b^q} \in Q_S$. Given the utility distance function D and the accuracy budget W, we construct a propositional function $\Delta_{D_W} : B^Q \times B^{Q'} \to \{0,1\}$ such that $\Delta_{D_W}(\overline{b^q}, \overline{b^{q'}}) = 1$ iff $D(\overline{b^q}, \overline{b^{q'}}) \leq W$; i.e., the accuracy loss in obfuscating state $\overline{b^q}$ to state $\overline{b^{q'}}$ is bounded by the given budget.

4.2 Symbolic Game Structure

We are now ready to solve the edit synthesis problem symbolically. Consider the plant modeled as a symbolic automaton $G = (B^Q, B^\Sigma, \Delta_\delta, \overline{b^{q_0}})$, the symbolic encoding of secret states χ_{Q_S}, and the propositional formula for the utility specification Δ_{D_W}. We follow the procedures in Section 3, first building symbolic intermediate automata G_e, G_p, $\mathcal{PA}$, and $\mathcal{A}$ and then build the symbolic game structure.

First, we construct the symbolic edit-pattern NFA $G_e = (B^e, B^\Sigma, \Delta_{\delta_e}, \overline{b^{e_0}})$ where:

- $B^e = \{y_1^q, \cdots, y_n^q\}$ are the Boolean variables for the *original* plant states.
- $\Delta_{\delta_e} = \Delta_\delta \vee \Delta_{\delta_r} \vee \Delta_{\delta_i}$ where
  - Δ_δ defines the original transitions.
  - $\Delta_{\delta_r} = (\exists \overline{y^\Sigma}.\Delta_\delta) \wedge \neg\Delta_\delta$ defines the replacement transitions.
  - $\Delta_{\delta_i} = (\overline{y^q} \Leftrightarrow \overline{y^{q'}}) \wedge \neg\Delta_\delta$ defines the insertion transitions.
- $\overline{b^{e_0}} = \overline{b^{q_0}}$

We can similarly build the symbolic public-behavior DFA $G_p = (B^p, B^\Sigma, \Delta_{\delta_p}, \overline{b^{p_0}})$, where B^p are the Boolean variables for the *fake* states and Δ_{δ_p} prunes all secret states. Next, we build the product automaton $\mathcal{PA}$ from G_e and G_p, and then prune the state pairs that violate the utility specification to obtain $\mathcal{A} = (B^A, B^\Sigma, \Delta_{\delta_A}, \overline{b^{A_0}})$. Here $\Delta_{\delta_A} = \Delta_{\delta_e} \wedge \Delta_{\delta_p} \wedge \chi_{D_W}\{\overline{y^{q'}} \leftarrow \overline{y^p}\} = \Delta_{\delta_{A,o}} \vee \Delta_{\delta_{A,r}} \vee \Delta_{\delta_{A,i}}$ is decomposed into the original transitions $\Delta_{\delta_{A,o}}$, the replacement transitions $\Delta_{\delta_{A,r}}$, and the insertion

transitions $\Delta_{\delta_{A,i}}$ for technical convenience later. Symbolic automaton $\mathcal{A}$ recognizes all edit patterns satisfying the privacy and utility specifications.

Finally, we build the symbolic game structure $\mathcal{G}S = (B^V, B^I, B^O, \Delta_{\rho_1}, \Delta_{\rho_2}, \overline{b^{v_0}})$. Let $\overline{y^A} = (y_1^q, \cdots, y_n^q, y_1^p, \cdots, y_n^p)$, we will use $\overline{y^A_{\Downarrow q}}$ to denote the projection of of $\overline{y^A}$ onto variables y_i^q. That is, $\overline{y^A_{\Downarrow q}} = (y_1^q, \cdots, y_n^q)$. Similarly, $\overline{y^A_{\Downarrow p}} = (y_1^p, \cdots, y_n^p)$.

- $B^V = B^A$ are the Boolean variables encoding the game positions.
- $B^I = B^\Sigma$ are the Boolean variables for the plant's actions. Superscript I means they are input variables to the edit function.
- $B^O = \{y_1^O, \cdots, y_m^O\}$ are the Boolean variables for the edit function's actions. Superscript O means they are output variables of the edit function.
- $\Delta_{\rho_1} : B^V \times B^I \to \{0,1\}$ such that $\Delta_{\rho_1}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^{A'}}) = \Delta_\delta(\overline{y^A_{\Downarrow q}}, \overline{y^\Sigma}, \overline{y^{A'}_{\Downarrow q}})$
- $\Delta_{\rho_2} : B^V \times B^I \times B^O \times B^{V'} \to \{0,1\}$ that is decomposed into $\Delta_{\rho_2,or} \vee \Delta_{\rho_2,i}$
  - $\Delta_{\rho_2,or}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^O}, \overline{y^{A'}}) =$
    $\Big(\Delta_\delta(\overline{y^A_{\Downarrow q}}, \overline{y^\Sigma}, \overline{y^{A'}_{\Downarrow q}}) \wedge \Delta_{\delta_{A,o}}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^{A'}})\{\overline{y^\Sigma} \leftarrow \overline{y^O}\}\Big) \vee$
    $\Big(\Delta_\delta(\overline{y^A_{\Downarrow q}}, \overline{y^\Sigma}, \overline{y^{A'}_{\Downarrow q}}) \wedge \Delta_{\delta_{A,r}}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^{A'}})\{\overline{y^\Sigma} \leftarrow \overline{y^O}\}\Big)$
  - $\Delta_{\rho_2,i}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^O}, \overline{y^{A'}}) =$
    $\exists \overline{y^{A''}}. \Big(\overline{y^{A''}} \in Reach_i(\overline{y^A}) \wedge \Delta_{\delta_{A,o}}(\overline{y^{A''}}, \overline{y^\Sigma}, \overline{y^{A'}})\Big)$, where
    - $Post_i(Z) = \{\overline{y^{A'}} \mid \exists \overline{y^\Sigma} \exists \overline{y^A}.(\overline{y^A} \in Z) \wedge \Delta_{\delta_{A,i}}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^{A'}})\}$
    - $Reach_i(\overline{y^A}) = \mu Z. Post_i(\overline{y^A}) \vee Post_i(Z)$
- $\overline{b^{v_0}} = \overline{b^{A_0}}$

Observe that Δ_{ρ_2} is decomposed into two parts, one containing the original and the replacement actions $\Delta_{\rho_2,or}$, and another containing only the insertion actions $\Delta_{\rho_2,i}$. We make this partition because the outputs for insertion actions are in general strings whose lengths are not known in advance. To symbolically encode all such output strings, we would need to introduce a potentially unbounded number of Boolean variables corresponding to all possible events and intermediate states on allowed output strings. To avoid this, we only encode in the game construction whether it is possible for the edit function to react with an insertion action. That is, a transition $(\overline{b^V}, \overline{b^I}, \overline{b^O}, \overline{b^{V'}}) \models \Delta_{\rho_2,i}$ if it is possible to move from position $\overline{b^V}$ to position $\overline{b^{V'}}$ with insertion. Here, the output $\overline{b^O}$ is unconstrained as it is not used: the actual insertion string will be computed explicitly in the synthesis algorithm in Section 4.3. We use μ-calculus [6, 3] to formulate the problem of determining whether it is possible to apply insertion actions. The μ-calculus formula $\mu Z. Post_i(\overline{y^A}) \vee Post_i(Z)$ is the least fixed point that computes all positions that are reachable from $\overline{y^A}$ via a non-zero length insertion string.

When computing $\Delta_{\rho_2,i}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^O}, \overline{y^{A'}})$, the intermediate steps of the fixpoint computation $Reach_i(\overline{y^A})$ encode the insertions themselves, and are stored in a data structure ins to be used later to extract the edit function. Informally, we store in $ins_{\overline{y^A}, \overline{y^\Sigma}}[i]$ the set of positions reachable from $\overline{y^A}$ via an insertion string of length i followed by an input event $\overline{y^\Sigma}$.

4.3 Synthesis

With the game structure $\mathcal{GS}$, we now compute the set of winning positions for the edit function and synthesize a winning edit strategy in Algorithm 1. We characterize the set of winning positions $\mathcal{W}$ using a μ-calculus formula. Specifically, in step 1, the μ-calculus formula $\nu Z.Pre(Z)$ is the greatest fixed point containing all positions where the edit function can continuously react to the plant with a winning edit action. If $\mathcal{W}$ does not contain the initial position $\overline{b^{v_0}}$, then Algorithm 1 returns that the edit synthesis problem is not feasible; i.e., Problem 1 has no solution. Otherwise, there exists a winning edit strategy and we synthesize, starting from step 3, a winning edit automaton $\mathcal{EA}$ by breadth-first search on the winning positions. The initial state of $\mathcal{EA}$ is the initial position $\overline{b^{v_0}}$ of $\mathcal{GS}$. Steps 6-18 compute concrete winning actions and construct the corresponding explicit edit automaton. χ_s is the set of explored positions in $\mathcal{GS}$. In each iteration, we take newly-reached positions $\chi_{s,diff}$ and compute in step 9 the one-step winning actions act from $\chi_{s,diff}$, using function `Winning_Actions`. With act being computed, in the inner while loop, we extract concrete transitions in act. In step 12, function `Extract_One` extracts one concrete transition. Then, in step 18, we subtract from act all transitions with the same game position $\overline{b^V}$ and plant output event $\overline{b^I}$, as an edit action for that position and event has already been found. This inner while loop terminates until act is empty. In each iteration, if the extracted transition is an insertion action, then we compute the output string for the insertion action using function `Compute_Insert_Out`. Otherwise, the output is the event b^O in the extracted transition.

Function `Compute_Insertion` returns a string o of local insertion events leading from position $\overline{b^V}$ to $\overline{b^{V'}}$ on input $\overline{b^I}$, and we describe it here informally. Recall the data structure ins stored during the fixpoint computation that defines $\Delta_{\rho_2,i}$ in Section 4.2. Since $(\overline{b^V}, \overline{b^I}, \overline{b^O}, \overline{b^{V'}}) \models \Delta_{\rho_2,i}$, we have $\overline{b^{V'}} \in ins_{\overline{b^V},\overline{b^I}}[i]$ for some $i \geq 0$. Informally, $\overline{b^{V'}}$ is reachable from $\overline{b^V}$ via an insertion string of length i followed by $\overline{b^I}$. Note that we will want to find a *shortest* such i, for which $\overline{b^{V'}} \in ins_{\overline{b^V},\overline{b^I}}[i]$ but $\overline{b^{V'}} \notin ins_{\overline{b^V},\overline{b^I}}[i-1]$. Now, we can reconstruct a path of insertions from $\overline{b^V}$ to $\overline{b^{V'}}$ by working backwards from $ins_{\overline{b^V},\overline{b^I}}[i]$ as follows. Set $o_i = \overline{b^I}$. At each iteration, we extract an insertion action o_{i-1} that leads from $ins_{\overline{b^V},\overline{b^I}}[i-1]$ to $ins_{\overline{b^V},\overline{b^I}}[i]$. We repeat this until we arrive at $ins_{\overline{b^V},\overline{b^I}}[0] = \overline{b^V}$. The resulting $o = o_0 o_1 ... o_i$ is the output after insertion.

Theorem 1. *Given a plant G with a set of secret states Q_S, utility distance D, and accuracy budget W, Algorithm 1 returns a finite edit automaton $\mathcal{EA} = (S, Trans, s_0)$ that solves Problem 1, if one exists, and declares infeasibility otherwise.*

Proof. Recall that the game structure $\mathcal{GS}$ is constructed from G_e that recognizes all edit patterns. Hence, the symbolic $\mathcal{GS}$ enumerates all edit strategies in a finite structure that satisfy the privacy and utility specifications before potentially reaching a losing position. Because the winning set $\mathcal{W}$ is a set of positions where the edit function can *continuously* react to the plant with an edit action satisfying the specifications, we can synthesize an edit automaton that solves Problem 1 iff the initial game position is winning. A winning edit strategy can in general require memory: it can choose different edit actions based on the history. But because the game is a safety game, we can convert any such strategy to a winning memoryless strategy by repeatedly selecting the same edit action every time it visits the same game position. In fact, Algorithm 1 considers only

Algorithm 1: Edit function synthesis

input : $G = (Q, \Sigma, \delta, q_0)$, $Q_S \subset Q$, $D : Q \times Q \to \mathbb{N}$, $W \in \mathbb{N}$
output: $\mathcal{EA} = (S, Trans, s_0)$

Construct $\mathcal{GS} = (B^V, B^I, B^O, \Delta_{\rho_1}, \Delta_{\rho_2}, \overline{b^{v_0}})$ per Section 4.2
Compute winning set $\mathcal{W} = \nu Z.Pre(Z)$ where
$Pre(Z) =$
$\{\overline{y^A} \mid \forall \overline{y^\Sigma}\, \forall \overline{y^{A'}_{\Downarrow q}}\, \exists \overline{y^{A'}_{\Downarrow p}}\, \exists \overline{y^O}.\, \left[\Delta_{\rho_1}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^{A'}}) \Rightarrow \overline{y^{A'}} \in Z \wedge \Delta_{\rho_2}(\overline{y^A}, \overline{y^\Sigma}, \overline{y^O}, \overline{y^{A'}})\right]\}$
if $\mathcal{W} \wedge \overline{b^{v_0}} =$ `False` **then**
  return `Infeasible`
$s_0 := \overline{b^{v_0}}$, $S \leftarrow \{s_0\}$
$\chi_s \leftarrow \overline{b^{v_0}}$, $\chi_{s,old} \leftarrow$ `False`
while $\chi_s \neq \chi_{s,old}$ **do**
  $\chi_{s,diff} \leftarrow \chi_s \wedge \neg\chi_{s,old}$
  $\chi_{s,old} \leftarrow \chi_s$
  $act \leftarrow$ `Winning_Actions`$(\chi_{s,diff}, \Delta_{\rho_1}, \Delta_{\rho_2}, \mathcal{W})$
  $\chi_s \leftarrow \chi_{s,old} \vee act_{\Downarrow A}\{\overline{y^{A'}} \leftarrow \overline{y^A}\}$
  while $act \neq$ `False` **do**
    $(\overline{b^V}, \overline{b^I}, \overline{b^O}, \overline{b^{V'}}) \leftarrow$ `Extract_One`(act)
    $S \leftarrow S \cup \{\overline{b^{V'}}\}$
    if $(\overline{b^V}, \overline{b^I}, \overline{b^O}, \overline{b^{V'}}) \models \Delta_{\rho_2,i}$ **then**
      $o \leftarrow$ `Compute_Insertion`$((\overline{b^V}, \overline{b^I}, \overline{b^O}, \overline{b^{V'}}))$
    else
      $o \leftarrow \overline{b^O}$
    $Trans \leftarrow Trans \cup \{(\overline{b^V}, \overline{b^I}, o, \overline{b^{V'}})\}$
    $act \leftarrow act \wedge \neg(\overline{b^V} \wedge \overline{b^I})$
return $(S, Trans, s_0)$

memoryless edit strategies. Therefore, the synthesized edit automaton is guaranteed to be finite.

4.4 Complexity

Computing the winning set $\mathcal{W}$, which is expressed as a μ-calculus formula of alternation depth 1, can be solved with effort $O(N)$ where N is the number of states the game structure $\mathcal{GS}$, which is $O(n^2)$ if n is the number of states in the plant G. Here effort is measured in symbolic steps, i.e., in the number of preimage computations in the fixpoint in step 1 of the Algorithm. Extracting an edit function also takes $O(N)$, and hence Algorithm 1 has complexity = $O(N) = O(n^2)$. However, constructing the game structure $\mathcal{GS}$ has additional complexity $O(N^2) = O(n^4)$ because of the least fixpoint computation for every state in the game structure when computing $\Delta_{\rho_2,i}$. In all, Algorithm 1 has complexity of $O(n^4)$.

5 Case Studies and Experiments

We demonstrate our approach empirically using EdiSyn, an open source Python toolkit we developed for this purpose[1]. EdiSyn implements the synthesis algorithm based on Binary Decision Diagrams (BDDs), and relies on the CUDD BDD library [11] and DD [5], an open source Python binding to CUDD. We ran EdiSyn with Example 2 introduced in Section 2.2. The utility distance is defined based on the L_1 distance, as defined in Section 2.2. Finally, we let the accuracy budget be 2.

We shown in Figure 3 the real and the obfuscated moving traces of Alice and Bob. t_i's denote the time points where their locations are reported. The left figure depicts the real moving traces. At time t_2, Alice and Bob meet at location $(2, 2)$, which corresponds to a secret state. The right figure depicts the traces output from the edit function. The edit function obfuscates their traces such that Alice and Bob are never reported to be in the same location. Furthermore, the distance between the original and the obfuscated locations always remain within 2.

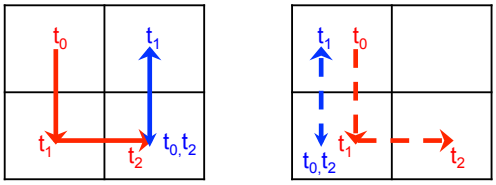


Fig. 3. Left: The original moving traces of Alice (in blue) and Bob (in red). Right: The obfuscated moving traces output from the synthesized edit function.

We also ran EdiSyn with examples in the same settings but increasing grid sizes. $\text{Grid}_{k\times k}$ is the example where Alice and Bob move in the $k \times k$ grid world and want to hide their secret meetings. The accuracy budget is set to 2 in all examples. The results of this experiment are summarized in Table 1. For each grid example, Table 1 shows the number of plant variables (i.e., the number of states plus the number of events), the computation time of the symbolic implementation, the peak number of BDD nodes, and the memory used during the synthesis computation. The experiment was performed on an Intel Core i5 (2.4 GHz, 4 GB) machine running Mac OS X 10.10.5 Yosemite. While we also implemented the explicit (non-symbolic) algorithm, the explicit implementation threw `outOfMemory` errors on all of the grid examples.

These are preliminary results based on a simple, unoptimized implementation. In addition to optimizing the implementation in terms of memory usage and efficiency of operations, comparison of various variable ordering strategies for the BDDs and reuse of intermediate stages of the fixpoint computations while constructing the game structure are also acknowledged as worthy of further exploration. Finally, observe that the synthesis algorithm is computed offline. Once an edit function is synthesized, it can be used to efficiently edit the plant's output online.

[1] EdiSyn is available at https://bitbucket.org/yichinwu/edisyn

Table 1. Scalability test for the grid world example, with a timeout of 60 minutes.

Example	Variables	Synthesis Time (min)	Peak Nodes	Memory (MB)
$\text{Grid}_{2\times2}$	161	0.05	21274	30.8
$\text{Grid}_{3\times3}$	1171	2.90	261446	151.3
$\text{Grid}_{4\times4}$	4353	42.37	1103200	637.7
$\text{Grid}_{5\times5}$	11651	Timeout	N/A	N/A

6 Related Work

This work combines perspectives and techniques from computer security and formal synthesis. Our threat model and problem formulation are inspired by the definition of differential privacy [2], where the consumers of data include both legitimate receivers and adversaries, and the goal is to provide privacy while preserving utility with respect to the desired data analytics. Informally, ϵ-differential privacy guarantees that the resulting output is insensitive (up to a factor dependent on ϵ) to the modification, deletion or addition of any single record in the original dataset. Utility of a differentially private mechanism is evaluated using query-dependent measures of the deviation between results obtained from the original dataset obtained by applying the mechanism to the original dataset. The edit functions in this work can be viewed as a discrete logic counterpart of differentially private mechanisms; privacy and accuracy here are captured by logical conditions on the edited executions of the plant in comparison with the real executions. Additionally, most traditional approaches to providing privacy rely on cryptographic primitives; however, such schemes require an infrastructure to create and distribute secret keys. In the settings we consider, especially those involving ad-hoc and dynamic networks, and resource-constrained devices, a non-cryptographic solution such as ours may be preferred.

There has been some previous work on the synthesis of artifacts enforcing privacy requirements in the discrete logic setting. Specifically, synthesis for a privacy notion called opacity has been explored by researchers in discrete event systems; see e.g., [10, 4, 12]. The edit mechanism in this paper is related to but more powerful than the insertion mechanism developed in [12]. The most distinguishing feature of this work is the threat model. All existing works on synthesis for opacity consider an threat model where every outside observer of the system is malicious. In contrast, the malicious outside observer in this paper is also endowed with some legitimate observational needs. As a consequence of this different threat model, none of the above works addresses questions of the preserving utility of observations. To the best of our knowledge, this paper is the first attempt to formulate the synthesis problem for both privacy and utility. Our work is also distinguished by the presentation of a symbolic encoding of the solution. We encode the synthesis problems symbolically using Binary Decision Diagrams, and are thereby better equipped to address the state explosion problem.

In addition to the field of discrete event systems, we draw inspiration from recent work in robotics that considers the design of discrete filters satisfying privacy and utility constraints provided as pairwise distinguishability (and indistinguishability) requirements on states [8]. Our work is most similar in spirit to this effort, but our privacy and utility constraints are specified as automata theoretic winning conditions instead of pairwise requirements on states. In [8], the requirements are satisfied via graph colorings:

states that must be indistinguishable have the same color and ones to be distinguished are colored differently. Our edit mechanism is more general, in that it also allows inserting fictitious events.

Finally, the idea of editing event labels on automaton transitions is also employed in [7], where the authors considered a selfish environment that edits the inputs to the plant automaton. However, the focus in [7] is on deciding whether the plant is resilient to such a selfish environment rather than on synthesizing an edit strategy with privacy and utility objectives.

7 Conclusion and Future Work

We have defined the problem of synthesizing an obfuscation policy that enforces privacy specifications while preserving utility. The specifications in this work were captured as automata-theoretic requirements on a finite state model of the plant's outputs. Our method allows plants to generate and broadcast event strings for some useful computation, while simultaneously hiding certain secret behaviors from an outside observer. To enforce the privacy and utility specifications, we automatically synthesized an edit function that reacts to the plant's outputs and transforms them in a way that meets both requirements. Our synthesis algorithm was encoded symbolically, improving the efficiency of obtaining a solution. This is, to our knowledge, the first work to consider synthesis for both privacy and utility specifications.

In this work, we considered simple privacy and utility specifications: in fact, our privacy requirement is a safety guarantee. In the future, we will explore the use of more complicated specifications to express these desirables. For example, temporal logics are expressive tools for stating requirements. Formulating privacy and utility as temporal logic formulae would allow a much richer set of specifications. Also, so far our utility specification has taken the form of an accuracy budget constraining the distance between the real and released states. In the future, we will tackle an optimization problem that asks the question, what is the smallest budget for which the problem in this paper becomes feasible?

References

1. Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
2. C. Dwork. Differential privacy. In *International Conference on Automata, Languages and Programming*, pages 1–12. 2006.
3. E. A. Emerson. Model checking and the mu-calculus. *DIMACS Series in Discrete Mathematics*, 31:185–214, 1997.
4. Y. Falcone and H. Marchand. Runtime enforcement of K-step opacity. In *52nd IEEE Conference on Decision and Control*, 2013.
5. I. Filippidis. https://github.com/johnyf/dd.
6. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27(3):333–354, 1983.
7. O. Kupferman and T. Tamir. Coping with selfish on-going behaviors. In *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6355, pages 501–516, 2010.
8. J. M. O'Kane and D. A. Shell. Automatic design of discreet discrete filters. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 353–360, 2015.

9. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.
10. A. Saboori and C. N. Hadjicostis. Opacity-enforcing supervisory strategies via state estimator constructions. *IEEE Transactions on Automatic Control*, 57(5):1155–1165, 2012.
11. F. Somenzi. CUDD: CU decision diagram package release 2.3.0. *University of Colorado at Boulder*, 1998.
12. Y.-C. Wu and S. Lafortune. Synthesis of insertion functions for enforcement of opacity security properties. *Automatica*, 50(5):1336–1348, 2014.