Acta Informatica (2025) 62:30
https://doi.org/10.1007/s00236-025-00502-1

RESEARCH

Gray-box runtime enforcement of hyperproperties

Tzu-Han Hsu[1] · Ana Oliveira da Costa[2] · Andrew Wintenberg[3,5] · Ezio Bartocci[4] · Borzoo Bonakdarpour[1]

Received: 4 December 2024 / Accepted: 21 July 2025

Abstract
Enforcement of information-flow policies has been extensively studied by language-based approaches over the past few decades. In this paper, we propose an alternative, novel, general, and effective approach using enforcement of *hyperproperties*– a powerful formalism for expressing and reasoning about a wide range of information-flow security policies. We study *black*- vs. *gray*- vs. *white-box* enforcement of hyperproperties expressed by nondeterministic finite-word hyperautomata (NFH), where the enforcer has null, some, or complete information about the implementation of the system under scrutiny. Given an NFH, in order to generate a runtime enforcer, we reduce the problem to controller synthesis for hyperproperties and subsequently to the satisfiability problem for quantified Boolean formulas (QBFs). The resulting enforcers are transferable with low-overhead. We conduct a rich set of case studies, including information-flow control for JavaScript code, as well as synthesizing obfuscators for control plants.

Tzu-Han Hsu, Ana Oliveira da Costa and Andrew Wintenberg have contributed equally to this work.

✉ Tzu-Han Hsu
tzuhan@msu.edu

✉ Ana Oliveira da Costa
ana.costa@ist.ac.at

Andrew Wintenberg
awintenb@umich.edu

Ezio Bartocci
ezio.bartocci@tuwien.ac.at

Borzoo Bonakdarpour
borzoo@msu.edu

[1] Department of Computer Science and Engineering, Michigan State University, East Lansing 48824, MI, USA

[2] IST Austria, Klosterneuburg, Austria

[3] Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor 48109, MI, USA

[4] TU Wien, Vienna, Austria

[5] Present address: ICTEAM, UCLouvain, Louvain-la-Neuve, Belgium

Published online: 09 August 2025

1 Introduction

Runtime enforcement (RE) is a technique to ensure the correctness of systems at runtime, especially in corner cases that static analysis and testing fail to identify. In RE, an *enforcer* monitors and actively ensures that the system's executions satisfy a formal specification. Monitoring can be agnostic (black-box), partially aware (gray-box), or fully aware (white-box) of the system implementation under scrutiny. RE can perform operations (e.g., insertion or suppression of events) to correct the system's observable behavior. Ideally, an enforcer changes the system's behavior only when strictly necessary (precision) while ensuring the desired property satisfaction (soundness).

RE is especially useful in systems that cannot tolerate even a small or transient violation of their specification. Another benefit of using RE is to provide formal assurance for systems that need to be kept alive while also fulfilling critical properties. This is vital for policies where a system shutdown to avoid violations is not an adequate course of action. For example, policies such as privacy and termination-sensitive non-interference [3] cannot be dealt with by simply shutting down the system, as halting operations may leak information to an attacker. Instead, RE ensures that the composition of the enforcer with the system fulfills the intended specification. Then, with well-specified requirements, it guarantees that the enforcing mechanism does not add additional information leakage that is forbidden by the specification.

In this work, we tackle the challenge of enforcing information-flow policies where the enforcer may not have complete access to the implementation of the system under monitoring. Information flow control (IFC) requires reasoning over multiple executions. For example, an attacker may infer secret information by comparing the system's observable behavior across different runs for the same public input. For this reason, from a language-theoretic standpoint, information-flow policies define *hyperproperties* [18]– i.e., they define broad, system-wide requirements– rather than *trace properties* (which only specify requirements on individual executions). The shift from trace to system-level requirements introduces significant challenges in synthesizing effective policy enforcers. To address these challenges, we propose an approach that leverages knowledge about the systems to be monitored to synthesize sound and effective enforcers for hyperproperties.

1.1 Motivating example

Consider the JavaScript program in Fig. 1, which implements login functionality for a web page (lines 1–12) with dynamic advertisement loading from a remote script (lines 14–16), that we will use throughout the paper as a running example. Our goal is to guarantee at runtime that (1) the user's secret information does not leak to an untrusted agent (confidentiality) and (2) secure communication channels are not misused (integrity). The JavaScript program is exposed to the following attacks:

Attack 1 (confidentiality): In this example, the attacker uses a DOM update to expose the user credentials to an external server. In particular, the attacker attaches the user credentials cred to the source of DOM attribute imgSrc (see Attack 1 in Fig.1), triggering a request to the insecure "evil.com" which includes the credentials. Here, the password Pwd in the

A simple login web page in JavaScript:

```
<script type="javascript">
    var baseUrl;
    var settings = function(s){baseUrl = s;} // base URL setup
    if(readCookie("isAdmin")) {
      settings("my.com/admin/login.php");
    } else {
      settings("my.com/login.php");
    }
</script>

<text id="User"> <text id="Pwd">
<button id="BTNLogin" onclick="login()">

<div id="AdNode">
    <script src="ads.com/display.js"> // dynamic advertisement loading
</div>
```

Attack 1: credentials leak to `imgSrc` (confidentiality)

```
var cred = doc.getElement("User").text + doc.getElement("Pwd").text;
var imgSrc = "http://evil.com/img.jpg" + "?t=" + escape(cred);
doc.getElementById("img").src = imgSrc;
```

Attack 2: `baseUrl` is set to `evil.com` (integrity)

```
var dir = "evil.com"; settings(dir);
```

Fig. 1 Confidentiality (attack 1) and integrity (attack 2) vulnerabilities in a simple login web page written in JavaScript with dynamic advertisement loading.

user credentials has a high confidentiality level (i.e., it is a secret variable), while both the user name User and the DOM attribute imgSrc has low confidentiality level (i.e., they are public variables). Attaching the user password, Pwd, to imgSrc violates a *non-interference* property: secret values should not interfere with publicly observable behavior of a system. In this example, we can specify non-interference as *observational determinism* (*OD*) [66]: all pairs of executions with the same public input must also produce the same public output. OD is a 2-safety property; i.e., we need at most two finite executions of the system to witness its violation. In the context of hyperproperties, we may refer to it as a universal hyperproperty (and, in particular, as a $\forall\forall$ hyperproperty), because it only requires universal quantification over the system's executions.

Attack 2 (integrity): An attacker may target the integrity of the system by redirecting the user post request in settings to an untrusted server "evil.com" (see Attack 2 in Fig 1). That is, the baseURL no longer points to a secure communication channel. In this example, we assume that the specification of secure URLs (trusted channels) is unavailable offline. However, secure URLs can be inferred at runtime by observing the system behavior when no advertisement is loaded (i.e. when the system operates without interference from a malicious agent). We observe that OD is inadequate for specifying the integrity requirement we want to enforce because baseURL may take on different values (i.e., non-determinism). In this scenario, we express the integrity requirement as *non-inference* (NI) [44, 64]: That is, for all executions, the link assigned to baseUrl must be in at least one of the system executions without advertisements. Non-inference requires a trace quantifier-alternation ($\forall\exists$); thus, it is not a safety hyperproperty, which poses real challenges for its enforcement.

1.2 What is missing?

Type-theoretic solutions [47] and *taint-tracking* information-flow control would halt the program execution at line 3 in Fig. 1 (baseUrl = s), because this is a flow from an untrusted source (s) to trusted sink (baseUrl). Such an enforcing technique, while effective at preventing the attack, has undesired side effects: (1) terminating a program is not always an ideal strategy, and (2) a program where s matches the correct URL, is actually safe, but will be forcefully terminated.

Secure multi-execution (SME) [22, 38, 65] primarily deals with secrecy. An SME approach executes the monitored program multiple times: one execution for each security level with special rules for I/O interaction between the levels. Outputs can only be produced within the context of their security label. At the same time, a default value replaces input values in all executions except those labeled with the same level or higher. For the simple case where there are only two security levels– High and Low (or secret and public), the High-labeled execution sees all input values and only changes secret outputs, while the Low-labeled execution receives secret inputs set to a default value and it can affect only public outputs. In our example, if we consider only two security levels and label the baseUrl as public, then the advertisement can change the baseUrl value and affect future logins. We remark that while baseUrl is public, it has high integrity (i.e., it is trusted). One could augment SME to consider both secrecy and integrity and add the trusted and untrusted labels. This will require four copies of the program executing, resulting in significant runtime overhead. Moreover, SME as well as more recent approaches such as *multi-faceted* execution [7, 42] would treat our $\forall\exists$ policy as a $\forall\forall$ which is overly conservative.

Finally, existing techniques are typically designed only for specific policies and systems and hence, lack the generality of a framework that can deal with a rich set of security requirements. Logic-based approaches (as our approach in this paper) offer such generality and furthermore are *transferable*, meaning that an enforcer for a policy can be used in different systems.

1.3 Our approach

Objectives
We aim to develop an RE technique for information-flow policies such as confidentiality and integrity. Our goal is to develop an approach that is:
(1) ***transferable***: independent of the enforcing system,
(2) ***general***: logic-based, and capable of handling a wide range of information-flow security policies,
(3) ***low-overhead***: imposing only light-weight runtime operations, and
(4) ***sound*** and ***precise***: introduce no undesired behaviors, and does not modify secure systems.
These objectives clearly distinguish our objectives from the prior work on enforcement of information-flow policies.

With this motivation, we propose a novel RE technique for information-flow (IF) policies expressed by *hyperproperties*. A hyperproperty [18] is a set of sets of traces, where each set describes a system that satisfies the policy expressed by the hyperproperty. Enforcing of hyperproperties enables us to deal with a wide range of IF polices beyond traditional non-interference. While there has been significant progress on developing logic-based RE

methods for trace properties (those that prescribe the behavior of individual executions) [8, 24, 25, 33, 45, 63], to the best of our knowledge, the work on RE for hyperproperties is limited to [20], where the authors propose a black-box enforcement approach that ensures soundness but makes no guarantees on precision. Furthermore, the work in [20] is limited to alternation-free hyperproperties which leaves out many important policies such as non-inference [40], generalized non-interference [39], or non-interference for concurrent programs, etc. In this work, we propose a solution for RE that can handle hyperproperties with quantifier alternations. Enforcing such general hyperproperties is challenging as it requires integrating system knowledge both during enforcer synthesis and operation.

1.4 Contributions

Our first contribution is formalizing a general notion of RE parameterized by some knowledge of the system that covers *black-*, *gray-*, and *white-box* RE, where the enforcer has null, some, or complete information about the system implementation under scrutiny, respectively. Hyperproperties for both *terminating* and *non-terminating* programs are represented using *nondeterministic finite-word hyperautomata* (NFH) [15]. We show that, in the context of hyperproperties: (1) black-box RE may lead to a sound but not a precise solution, where correct executions are unnecessarily altered, and (2) having certain a-priori knowledge about the system (i.e., gray-box) effectively assists in developing sound and precise enforcers.

Our second contribution is a reduction from the RE problem to the *controller synthesis* problem for hyperproperties [16]. Controller synthesis problem aims at identifying a *controller* that steers the execution of a *plant*, modeled by a finite transition system, ensuring the controlled plant satisfies a requirement. The choice of reduction to controller synthesis is motivated by the fact that in RE, we are often faced by constraints over what the enforcer is allowed to take action (i.e., insert or suppress). Subsequently, we solve the controller synthesis problem by mapping it to the satisfiability problem for *quantified Boolean formulas* (QBF). To the best of our knowledge, this is the first implementation of controller synthesis for hyperproperty specifications.

Our technique is fully implemented end-to-end, meaning from code to a concrete enforcer for the code.[1] We present experiments over a rich set of case studies, including JavaScript runtime security enforcement, and privacy enforcement for obfuscation-aware eavesdroppers. Our implementation integrates existing tools such as ExpoSE [36], Jalangi [53], and QuAbS [58]. We also experiment with various policies and hyperproperties including declassification, non-interference, non-inference, privacy, etc. Our experimental results demonstrate the effectiveness of our approach in synthesizing enforcers with minimal overhead, from as low as 3.5% to at most 28%.

1.4.1 Organization

Preliminaries and our running example are introduced in Sec. 2. Section 3 presents our definition of RE and formalizes white/gray/black enforcers. Reductions from RE to controller synthesis and subsequently to the satisfiability for QBF are presented in Sections 4 and 5, respectively. Implementation, evaluation, and experimental results are in Section 6. Related

[1] Available at: https://github.com/hyperenforce/artifact

work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8.

2 Preliminaries

Since we reason about the *runtime* behavior of systems, which can only exhibit finite observations, our approach focuses on finite-word hyperproperties. Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{\mathrm{AP}}$ be the *alphabet*. We call the elements of Σ *letters*. A *(finite) word* w over alphabet Σ is a finite sequence of letters $w = w(0)w(1)\cdots w(n)$, for some $n \in \mathbb{N}$. We denote by $w[i\ldots]$ the sequence of letters of w from position i, i.e., $w[i\ldots] = w(i)\cdots w(n)$. We denote the *empty word* by ε and the set of *all* finite words by Σ^*. From the set theoretic point of view, a *finite-word property* W over a set of propositional variables AP is a set of finite words (i.e., $W \subseteq \Sigma^*$). The *restriction* of a letter in $l \in \Sigma$ by a subset $\mathrm{AP}' \subseteq \mathrm{AP}$ is $l|_{\mathrm{AP}'} = l \cap \mathrm{AP}'$. We extend naturally this notion to a restriction of a word w and a set W of words to a subset $\mathrm{AP}' \subseteq \mathrm{AP}$: $w|_{\mathrm{AP}'} = w(0)|_{\mathrm{AP}'}\cdots w(n)|_{\mathrm{AP}'}$ and $W|_{\mathrm{AP}'} = \{w|_{\mathrm{AP}'} \mid w \in W\}$.

A *finite-word hyperproperty* φ is a set of sets of finite words. i.e., $\varphi \subseteq 2^{\Sigma^*}$. Intuitively, while a property prescribes the behavior of individual words, a hyperproperty can express *system-wide* security requirements such as confidentiality and integrity. In the sequel, we refer to finite-word hyperproperties just as hyperproperties and finite words as words. We denote hyperproperties using $\varphi, \psi, \ldots$ and sets of words using $W, U, \ldots$. A set of words $W \subseteq \Sigma^*$ satisfies a hyperproperty $\varphi \subseteq 2^{\Sigma^*}$ iff $W \in \varphi$.

2.1 Nondeterministic finite hyperautomata

We specify hyperproperties with *nondeterministic finite hyperautomata* (NFH) [15]. Using NFH is a design choice and one can also define the target hyperproperties in HyperLTL [19].

Definition 1 A *nondeterministic finite-word automaton* (NFA) is a tuple $A = (\Sigma, Q, \hat{q}, F, \delta)$, where Σ is the alphabet, Q is a nonempty finite set of states with $\hat{q} \in Q$ being the initial state and $F \subseteq Q$ a set of accepting states; and $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation.

A *run* of an NFA $A = (\Sigma, Q, \hat{q}, F, \delta)$ on a word $w \in \Sigma^*$ is a finite sequence of states $q_0 \cdots q_n$, where $q_0 = \hat{q}$, and all steps $0 < i \leq n$, we have $(q_{i-1}, w(i), q_i) \in \delta$. The run is *accepting* if $q_n \in F$. An NFA A accepts a word w if there exists an accepting run of A on w. The language of A, denoted $\mathcal{L}(A)$, where $\mathcal{L}(A) \subseteq \Sigma^*$ is the set of all words accepted by A.

We now lift NFAs to NFHs. A *hyperword* W over Σ is a set of words over Σ, $W \subseteq \Sigma^*$. A *nondeterministic finite hyperautomaton* (NFH) (also referred to as a hyperautomaton) reads *hyperwords*. Syntactically, a hyperautomaton is defined as an NFA that reads from the alphabet Σ^n prefixed with a sequence of quantifiers over a finite set of *word variables* $\Pi = \{\pi_1, \ldots, \pi_n\}$.

Definition 2 A *nondeterministic finite-word hyperautomaton* (NFH) over alphabet Σ and word variables $\Pi = \{\pi_1, \ldots, \pi_n\}$ is a tuple $\mathbb{A} = (\Sigma, \Pi, Q, \hat{q}, F, \delta, \alpha)$, where

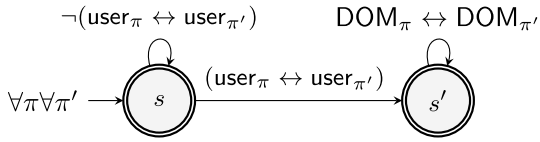


Fig. 2 NFH for non-interference, $\mathbb{A}_{\mathrm{OD}}$

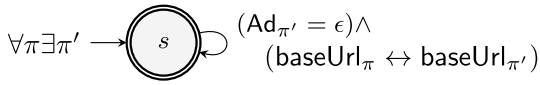


Fig. 3 NFH for non-inference, $\mathbb{A}_{\mathrm{NI}}$

$\tilde{\mathbb{A}} = (\Sigma^n, Q, \hat{q}, F, \delta)$ defines the underlying NFA, and $\alpha = \mathbb{Q}_1\pi_1 \cdots \mathbb{Q}_n\pi_n$ is a sequence of word quantifiers, i.e., for all $1 \leq i \leq n$, $\mathbb{Q}_i \in \{\forall, \exists\}$.

The zip function $\mathrm{zip} : (\Sigma^*)^n \to (\Sigma^n)^*$ zips tuples of words over Σ into a word over $(\Sigma \cup \{\#\})^n$, where words are padded as needed with $\#$ to the same length. A hyperword $W \subseteq \Sigma^*$ is accepted by a hyperautomaton $\mathbb{A} = (\Sigma \cup \{\#\}, \Pi, Q, \hat{q}, F, \delta, \alpha)$, with $\alpha = \mathbb{Q}_1\pi_1 \cdots \mathbb{Q}_n\pi_n$, iff:

$$\mathbb{Q}_1\pi_1 \in W,\ \mathbb{Q}_2\pi_2 \in W,\ \mathbb{Q}_3\pi_3 \in W, \cdots,\ \mathbb{Q}_n\pi_n \in W.\ \mathrm{zip}(\pi_1, \cdots, \pi_n) \in \mathcal{L}(\tilde{\mathbb{A}}).$$

Example 1 Consider the simple login page in Fig. 1. In our example, each trace represents a distinct web session, with each login attempt initiating a new session. The values at each point in the trace capture the state of the webpage at that moment. We observe that the username used for the login attempt that initiates the session (which we will refer to simply as *user*) is established at the beginning of the trace and remains unchanged. Under these assumptions, we expresses non-interference requirement for Fig. 1 with the NFH in Fig. 2: for each user, the DOM objects (e.g., imgsrc) should not change for login attempts with the same username. Without loss of generality, we may annotate transitions by Boolean expressions that express combinations of letters in Σ^n. Similarly, the NFH in Fig. 3 specifies non-inference policy for Attack 2 in Fig 1: the absence of the advertisement should not affect the URL. For simplicity, we only depict the accepting paths.

2.2 Plants

We represent systems under scrutiny as a transition system with transitions labeled either controllable or uncontrollable, which we refer to as *plants*.

Definition 3 A *plant* over AP is a tuple $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$, where S is a finite set of states with $\hat{s}$ being the initial state; $\mathfrak{c}, \mathfrak{u} \subseteq S \times S$ are respectively sets of *controllable* and *uncontrollable* transitions, where $\mathfrak{c} \cap \mathfrak{u} = \emptyset$, and $L : S \to 2^{\mathrm{AP}}$ is a *labeling* function.

We define the set of all transitions in $\mathcal{P}$ as $\delta_{\mathfrak{c},\mathfrak{u}} = \mathfrak{c} \cup \mathfrak{u}$ and the set of all *terminal* states as $\mathrm{Term}(\mathcal{P}) = \{s' \in S \mid \forall s \in S\,(s', s) \notin \delta_{\mathfrak{c},\mathfrak{u}}\}$. A *path* of a plant $\mathcal{P}$ is a finite sequence of states $s_0 \cdots s_n \in S^+$ such that $s_0 = \hat{s}$ and, for all steps $0 \leq i < n$, $(s_i, s_{i+1}) \in \delta_{\mathfrak{c},\mathfrak{u}}$. A *path is terminated* if it ends in a terminal state, i.e. $s_n \in \mathrm{Term}(\mathcal{P})$. The set of all *reachable states* in

$\mathcal{P}$ is defined as $\mathrm{Reach}(\mathcal{P}) = \bigcup\{s_0, \ldots, s_n \mid s_0 \cdots s_n \text{ is a path in } \mathcal{P}\}$. A *trace* of a path $s_0 \cdots s_k$ with respect to $\mathcal{P}$, denoted $L(s_0 \cdots s_k)$, is the sequence $L(s_0) \cdots L(s_k)$, where L is the labeling function in $\mathcal{P}$. The *trace set* of a plant $\mathcal{P}$ is the set of traces of all of its paths, denoted by $\mathrm{Tr}(\mathcal{P})$.

We say that $\mathcal{P}$ satisfies a hyperproperty φ, denoted $\mathcal{P} \models \varphi$, if its set of traces $\mathrm{Tr}(\mathcal{P})$ as a hyperword is accepted by $\mathbb{A}$; formally, $\mathrm{Tr}(\mathcal{P}) \in \mathcal{L}(\mathbb{A})$. Without loss of generality, we will assume that plants contain a sink state corresponding to runs padded with # to simplify checking hyperproperty satisfaction. In this case, the following result states that we only need to consider paths of the plant with a sufficiently long length.

Lemma 1 *Given a plant $\mathcal{P}$ and hyperautomaton $\mathbb{A}$, there exists $d \in \mathbb{N}$ such that $\mathcal{P}$ satisfies the hyperproperty encoded by $\mathbb{A}$ if and only if $\mathbb{A}$ accepts $Tr_d(\mathcal{P}) = \{t \in Tr(\mathcal{P}) \| |t| = d\}$.*

Proof We begin by recalling that a transition system such as $\mathcal{P}$ satisfies a linear-time property represented by an automaton $\mathbb{A}$ with $\alpha = \forall\pi_1$ or $\alpha = \exists\pi_1$, if and only if paths of $\mathcal{P}$ with length at most $d = |S||Q|$ satisfy the property. With the plant fixed, we can inductively construct an equivalent hyperautomaton eliminating the last quantifier until only one remains at which point previously stated fact can be applied to derive a sufficient d. Let $\alpha = \mathbb{Q}_1\pi_1 \cdots \mathbb{Q}_n\pi_n$ denote the quantifiers of $\mathbb{A}$, let $\tilde{\mathbb{A}} = (\Sigma^n, Q, \hat{q}, F, \delta)$ denote the underlying NFA of $\mathbb{A}$, and let $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ be the plant. We begin by assuming $n > 1$ and $\mathbb{Q}_n = \exists$. Define $\tilde{\mathbb{A}}_\exists = (\Sigma^{n-1}, Q \times S, (\hat{q}, \hat{s}), F \times S, \delta_\exists)$ where:

$$\delta_\exists = \{((q,s),(\sigma_1, \cdots, \sigma_{n-1}),(q',s')) \mid \exists\sigma_n \in \Sigma.\\ (q,(\sigma_1, \cdots, \sigma_n),q') \in \delta \ \wedge \ (s,\sigma_n,s') \in \mathfrak{c} \cup \mathfrak{u}\}.$$

By construction $\mathcal{P}$ satisfies $\mathbb{A}$ if and only if it satisfies the hyperautomaton corresponding to $\tilde{\mathbb{A}}_\exists$ which has at most $|Q||S|$ many states. To handle universal quantifiers, we can perform the same construction on the negation of the hyperproperty represented by the complement hyperautomaton with an exponential number of states. In this case the resulting $\tilde{\mathbb{A}}_\forall$ will have at most $2^{|Q||S|}$ many states. Iterating this procedure results in an hyperautomaton equivalent to $\mathbb{A}$ over $\mathcal{P}$ with a single quantifier. □

3 Enforcing hyperproperties

This section introduces a general definition of RE of hyperproperties. Additionally, soundness and precision of enforcers are defined independently of the choice of the specification language and monitored system.

3.1 Runtime enforcers for hyperproperties

We are interested in reasoning over finite observations of systems since the enforcer can only observe finite behaviors at runtime. An observation is a finite set of finite traces, while the set of all *finite observations of the system* $\mathcal{S}$, denoted $\mathrm{Obs}(\mathcal{S})$, is the prefix-closed set of all its finite and possibly non-terminating behaviors and, so, $\mathrm{Obs}(\mathcal{S}) \subseteq \Sigma^*$. From now on, when we refer to a system, we mean its finite observation set.

An enforcer $\mathbb{E}$ of a hyperproperty φ (specified by a hyper language) modifies the traces of a system $\mathcal{S}$ at runtime, generating as output a set of traces $\mathbb{E}(\mathcal{S})$.

Definition 4 An *enforcer* is a total function $\mathbb{E}$ that takes a system and produces a new system. Formally:

$$\mathbb{E} : 2^{\Sigma^*} \to 2^{\Sigma^*}.$$

Example 2 A strategy to enforce non-interference, as defined in Section 2 with the hyperautomaton $\mathbb{A}_{\mathrm{OD}}$ (see Fig. 2), is to simply force the same valuation for DOM in all executions. Enforcers $\mathbb{E}_1$ and $\mathbb{E}_2$, defined in Table 1, force DOM to be constantly 0 or 1, respectively. Note that, for a time point i represented by the word t_i, we say that DOM is 1 iff $\mathrm{DOM} \in t_i$. Both $\mathbb{E}_1$ and $\mathbb{E}_2$ are examples of black-box enforcers, as the enforcer uses no knowledge of the system.

3.2 Black- Vs. Gray-, Vs. White-box RE

In order to characterize *black-* vs. *gray-*, vs. *white-*box RE, we define the notion of *system class*. A *class of systems*, denoted by ψ, defines a set of systems, i.e., it is a hyperproperty $\psi \subseteq 2^{\Sigma^*}$. Then, a system $\mathcal{S}$ belongs to the class ψ iff $\mathcal{S} \in \psi$. An enforcer that has full knowledge of a system $\mathcal{S}$ is white-box, the one that has no knowledge of $\mathcal{S}$ is black-box, and is gray-box, otherwise.

Definition 5 Given a system $\mathcal{S}$, a system class ψ, and an enforcer $\mathbb{E}$:

- if $\psi = \{\mathcal{S}\}$, then $\mathbb{E}$ is a *white-box* enforcer for $\mathcal{S}$;
- if $\psi = \mathrm{true}$ (i.e., the universal set of systems), then $\mathbb{E}$ is a *black-box* enforcer;
- otherwise, $\mathbb{E}$ is a *gray-box* enforcer.

Example 3 To illustrate the benefit of incorporating system knowledge, we examine enforcers $\mathbb{E}_3$ and $\mathbb{E}_4$ in Table 1. These enforcers are parameterized by a set of traces M used to decide on how to enforce $\mathbb{A}_{\mathrm{OD}}$ at runtime. For systems where M is an adequate model (or abstraction), $\mathbb{E}_3$ and $\mathbb{E}_4$ use M to effectively minimize their interference. Enforcers $\mathbb{E}_3$ and $\mathbb{E}_4$ begin by checking whether M can be used decide the DOM value for the trace t. That is, whether there is at least one trace in M agreeing with the user in t, by using the function

Table 1 Enforcer candidates for $\mathbb{A}_{\mathrm{OD}}$, for a given set of traces M

$\mathbb{E}_1(\mathcal{S}) = \{t'_0 t'_1 \cdots \mid t_0 t_1 \cdots \in \mathcal{S} \text{ and } \forall i \in \mathbb{N} : t'_i = (t_i \setminus \{\mathrm{DOM}\})\}$;
$\mathbb{E}_2(\mathcal{S}) = \{t'_0 t'_1 \cdots \mid t_0 t_1 \cdots \in \mathcal{S} \text{ and } \forall i \in \mathbb{N} : t'_i = (t_i \cup \{\mathrm{DOM}\})\}$;
$\mathbb{E}_3(\mathcal{S}) = \{t'_0 t'_1 \cdots \mid t \in \mathcal{S}, \text{if } \mathrm{badModel}(t, M), \text{ then } \forall i \in \mathbb{N} : t'_i = t(i), \text{otherwise}$
$\mathrm{firstUser}(t, M) = i, \forall 0 \le j \le i : t'(j) = t(j), \text{ and}$
$\exists m \in M : t(i)\vert_{\mathrm{user}} = m(i)\vert_{\mathrm{user}} \text{ and } \forall i < k \le \vert t\vert : t'_k = \mathrm{update}(t(k), m(k))\}$.
$\mathbb{E}_4(\mathcal{S}) = \{t'_0 t'_1 \cdots \mid t \in \mathcal{S}, \text{if } \mathrm{badModel}(t, M), \text{ then } \forall i \in \mathbb{N} : t'_i = t_i \setminus \{\mathrm{DOM}\}, \text{otherwise}$
$\mathrm{firstUser}(t, M) = i, \forall 0 \le j \le i : t'(j) = t(j), \text{ and}$
$\exists m \in M : t(i)\vert_{\mathrm{user}} = m(i)\vert_{\mathrm{user}} \text{ and } \forall i < k \le \vert t\vert : t'_k = \mathrm{update}(t(k), m(k))\}$

$\mathrm{firstUser}(t, M)$ returning the earliest position in all traces in M where one of them has the same value of user as t and defined below:

$$\mathrm{firstUser}(t, M) = \min(\{i \mid m \in M \text{ and } t(i)|_{\mathrm{user}} = m(i)|_{\mathrm{user}}\} \cup \{\infty\}).$$

If $\mathrm{firstUser}$ returns a trace position (i.e., a natural number), they further check if there are conflicting traces in M for that user. Formally:

$$\mathrm{badModel}(t, M) \stackrel{\mathrm{def}}{=} (\mathrm{firstUser}(t, M) = \infty) \ \vee \big(\mathrm{firstUser}(t, M) = i \wedge \exists m, m' \in M : (m(i)|_{\mathrm{user}} = m'(i)|_{\mathrm{user}} \ \wedge \ m[i\ldots]|_{\mathrm{DOM}} \neq m'[i\ldots]|_{\mathrm{DOM}})\big)$$

If the set M is deemed adequate for a given trace t, then both $\mathbb{E}_3$ and $\mathbb{E}_4$ pick a trace $m \in M$ and update DOM values in t with DOM values from m:

$$\mathrm{update}(l, l') = \begin{cases} l \cup \{\mathrm{DOM}\} & \text{if } \mathrm{DOM} \in l' \\ l \setminus \{\mathrm{DOM}\} & \text{otherwise.} \end{cases}$$

The only difference between $\mathbb{E}_3$ and $\mathbb{E}_4$ is that when M is not adequate for a trace t, then $\mathbb{E}_3$ does not change t while $\mathbb{E}_4$ updates all DOM values in t to 0.

3.3 The runtime enforcement problem

An enforcer for a hyperproperty φ must define a system that satisfies φ, i.e., it must be *sound* for φ. For example, enforcers $\mathbb{E}_1$ and $\mathbb{E}_2$ defined above are sound, as they ensure satisfaction of $\mathbb{A}_{\mathrm{OD}}$. However, an enforcer should additionally achieve soundness by modifying the input system $\mathcal{S}$ *as little as possible* and only when necessary. In the context of trace properties enforcers, this requirement is often referred as transparency [25], where modifications to the current monitored execution $t \in \mathcal{S}$, are performed *as late as* possible in t. The notion of 'past' for trace properties (and consequently the idea of 'as late as possible') is straightforward because it only needs information from the current system execution. Transparency, however, is problematic for hyperproperties because we need to reason about *multiple* executions with no presumption on the order we observe them.

With this motivation, we adopt the notion of *precision* [43]: an enforcer is precise for a hyperproperty φ if it does not change the observable behavior of systems that satisfy φ. This aligns with the standard meaning [13] where transparency refers to the ability not to change secure executions (within a possibly insecure program), while precision is about not changing secure programs. Obviously, enforcers $\mathbb{E}_1$ and $\mathbb{E}_2$ in Example 2 are not precise as they unnecessarily change the behavior of correct systems. This is an inherent deficiency in black-box enforcement of hyperproperties[2]. We are particularly interested in *gray-box* enforcers, where the enforcer may use some knowledge about the system under scrutiny.

[2] Although SME guarantees transparency for black-box under the language-based view, our black-box notion is stricter. In our approach, the enforcer has no assumption about the monitored system regarding its runtime behavior. In contrast, in SME, the enforcer runs multiple copies of the system, actively controlling their communication and scheduling.

Definition 6 Let φ and ψ be hyperproperties. An enforcer $\mathbb{E}$ is an *enforcer for a hyperproperty φ and systems in class ψ* iff for all systems $\mathcal{S} \in \psi$, $\mathbb{E}(\mathcal{S})$ is:

- *Sound:* $\mathbb{E}(\mathcal{S}) \in \varphi$; and
- *Precise:* If $\mathcal{S} \in \varphi$, then $\mathbb{E}(\mathcal{S}) = \mathcal{S}$.

Enforcement decision problem
Let AP be a set of atomic propositions, and φ and ψ be two hyperproperties over AP expressing the specification and a class of systems.
Does there exist a sound and precise enforcer $\mathbb{E}$ for φ and systems $\mathcal{S}$ in class ψ?

Example 4 We now examine the enforcers defined in Table 1 and discuss whether they are sound and precise enforcers for $\mathbb{A}_{\mathrm{OD}}$ as defined in Sec. 2. Table 2 summarizes our discussion below. Both $\mathbb{E}_1$ and $\mathbb{E}_2$ (Example 2) are black-box enforcers. They are both sound for the language defined by $\mathbb{A}_{\mathrm{OD}}$. They both trivially satisfy non-interference because, for all systems $\mathcal{S}$, then all traces in the set of traces defined by $\mathbb{E}_1(\mathcal{S})$ and $\mathbb{E}_2(\mathcal{S})$ have the same value for DOM (0 or 1, respectively). However, they are not precise, because they force a DOM value regardless of the system being monitored.

Recall that gray-box enforcers $\mathbb{E}_3$ and $\mathbb{E}_4$ (Example 3) use a set of traces M to decide on how to change the DOM value in the traces of any given system $\mathcal{S}$. Then, their soundness and precision depends on how the set of traces M relates to the system under scrutiny. For the white-box instances, i.e., the set of traces M is the same as the system ($\psi = \{\mathcal{S} \mid \mathcal{S} = M\}$), both $\mathbb{E}_3$ and $\mathbb{E}_4$ define precise enforcers for $\mathbb{A}_{\mathrm{OD}}$. White-box $\mathbb{E}_3$, however, does not define a sound enforcer for $\mathbb{A}_{\mathrm{OD}}$. Consider the case where $\mathcal{S} \notin \mathcal{L}(\mathbb{A}_{\mathrm{OD}})$. Then, there are two traces $t, t' \in \mathcal{S}$, that have the same username, $t(0)|_{\mathrm{user}} = t'(0)|_{\mathrm{user}}$, with a different DOM, i.e., $t(0)|_{\mathrm{DOM}} \neq t'(0)|_{\mathrm{DOM}}$. Hence, both $\mathrm{badDom}(t(0)|_{\mathrm{user}}, \mathcal{S})$ and $\mathrm{badDom}(t'(0)|_{\mathrm{user}}, \mathcal{S})$ hold. By definition of $\mathbb{E}_3(\mathcal{S})$, both t and t' will not be changed by $\mathbb{E}_3$. As t and t' witness a violation of $\mathbb{A}_{\mathrm{OD}}$, then $\mathbb{E}_3(\mathcal{S}) \notin \mathcal{L}(\mathbb{A}_{\mathrm{OD}})$. On the contrary, $\mathbb{E}_4$ is sound, as it updates DOM to be always 0 on all traces that witness a violation of the hyperproperty specified by $\mathbb{A}_{\mathrm{OD}}$.

Now, we look at a class systems ψ where the set of traces M is a strict under-approximation of all systems in ψ, i.e. $M \subset \mathcal{S}$ for all $\mathcal{S} \in \psi$. $\mathbb{E}_4$ and $\mathbb{E}_3$ are sound and not sound, respectively, for the same reason as for the white-box instance. However, for this class of systems, $\mathbb{E}_4$ is not precise. Consider a system that satisfies the hyperproperty specified by

Table 2 Soundness and precision of $\mathbb{E}_3$ and $\mathbb{E}_4$, from Table 1 and defined over a set of traces M, for $\mathbb{A}_{\mathrm{OD}}$.

System class	$\mathbb{E}_3(\mathcal{S})$		$\mathbb{E}_4(\mathcal{S})$	
ψ	Sound	Precise	Sound	Precise
true	✗	✓	✓	✗
$\{\mathcal{S} \parallel M \subset \mathcal{S}\}$	✗	✓	✓	✗
$\{\mathcal{S} \mid \mathcal{S} = M\}$	✗	✓	✓	✓
$\{\mathcal{S} \mid M \subseteq \mathcal{S}$ and $\forall t \in \mathcal{S}$ $\exists^1 m \in M : t(0)|_{\mathrm{user}} = m(0)|_{\mathrm{user}}\}^{(\dagger)}$	✓	✓	✓	✓

$^{(\dagger)}\exists^1$ stands for *there exists at most one*

$\mathbb{A}_{\mathrm{OD}}$, $\mathcal{S} \in \mathcal{L}(\mathbb{A}_{\mathrm{OD}})$, and has a trace, $t \in \mathcal{S}$, such that: (i) there is no trace in the set M with the same user as in t and (ii) DOM is not always 0 in t. By (i), $\mathrm{noUser}(t(0)|_{\mathrm{user}}, M)$ holds and $\mathbb{E}_4$ forces DOM to be always 0 in t. Then, $\mathbb{E}_4(\mathcal{S}) \neq \mathcal{S}$ even though the system satisfies non-interference, $\mathcal{S} \in \mathcal{L}(\mathbb{A}_{\mathrm{OD}})$.

$\mathbb{E}_4$ is precise for the class of systems ψ where for each system $\mathcal{S} \in \psi$ the set of traces M under-approximates it ($M \subseteq \mathcal{S}$) and M has a trace for each user defined in $\mathcal{S}$. $\mathbb{E}_3$ is sound, if there exists at most one trace in M for each user value defined in a system $\mathcal{S} \in \psi$.

4 Solving RE by controller synthesis

We reduce the enforcement decision problem, described in Section 3, to *weighted controller synthesis*. We modify the controller synthesis problem for hyperproperties, as in [15], to sets of finite traces and extend it with weighted transitions.

4.1 Weighted controller synthesis

Given a system represented by a plant $\mathcal{P}$ and a specification φ, the controller synthesis problem is to find a controller, denoted $\mathcal{P}'$, that satisfies φ. A controller $\mathcal{P}'$ simply restricts controllable transitions within $\mathcal{P}$.

Definition 7 A *controller* of a plant $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ is another plant $\mathcal{P}' = (S, \hat{s}, \mathfrak{c}', \mathfrak{u}, L)$ with $\mathfrak{c}' \subseteq \mathfrak{c}$.

We use *weighted controller synthesis* to find an enforcer that satisfies a hyperproperty by removing the minimum cost set of transitions from the input plant.

Weighted controller synthesis problem, breakable

Let AP be a set of atomic propositions, $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ be a plant, and $\mathbb{A}$ be an NFH, both over AP. Let $W : \mathfrak{c} \to \mathbb{Z} \cup \{\infty\}$ be a *weight function* over the controllable transitions in $\mathcal{P}$ and $\mathrm{Term}(\mathcal{P})$ be the set of terminal states in $\mathcal{P}$.

Does there exist a controller $\mathcal{P}' = (S, \hat{s}, \mathfrak{c}', \mathfrak{u}, L)$ of $\mathcal{P}$ s.t.:

(1) *(Acceptance)*$\mathcal{P}'$ is accepted by $\mathbb{A}$;

(2) *(No deadlocks)* No change in terminal states, $\mathrm{Term}(\mathcal{P}) = \mathrm{Term}(\mathcal{P}')$;

(3) *(Minimum cost)* For all plants $\mathcal{P}''$ with controllable transitions $\mathfrak{c}''$ that satisfy (1) and (2): $\sum_{e \in \mathfrak{c} \setminus \mathfrak{c}'} W(e) \leq \sum_{e \in \mathfrak{c} \setminus \mathfrak{c}''} W(e)$.

We denote by $\mathrm{C}(\mathcal{P}, \mathbb{A}, W)$ the set of all controllers solving the weighted controller synthesis problem for a given plant $\mathcal{P}$, NFH $\mathbb{A}$ and weights W.

4.2 Overall idea– sketch of the reduction

Figure 4 depicts our overall approach for synthesizing an runtime enforcer $\mathbb{E}$ and, if one is found, applying it to a plant $\mathcal{P}$, which represents the system. The knowledge given to the enforcer about the set of target systems (i.e., our gray-box setting) is encoded by another plant $\mathcal{P}_0$ (i.e., a partial view of the target plant $\mathcal{P}$).

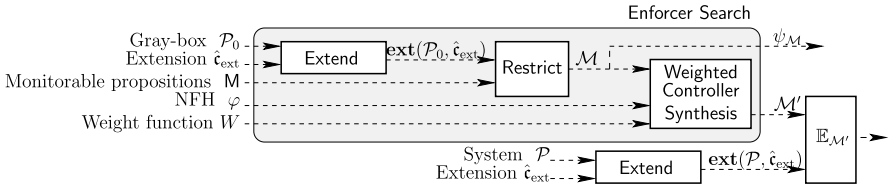


Fig. 4 Overview of our approach

Fig. 5 Abstraction of Fig. 1, where choose$\{e_1; e_2\}$ runs e_1 or e_2 non-deterministically

```
1 if(isIntern) {
2   baseUrl=str1;
3 } else { baseUrl=str2;}
4 if (advertisement) {
5   choose{
6     baseUrl=str3;
7     changeDom(); }
8 } else{ skip; }
```

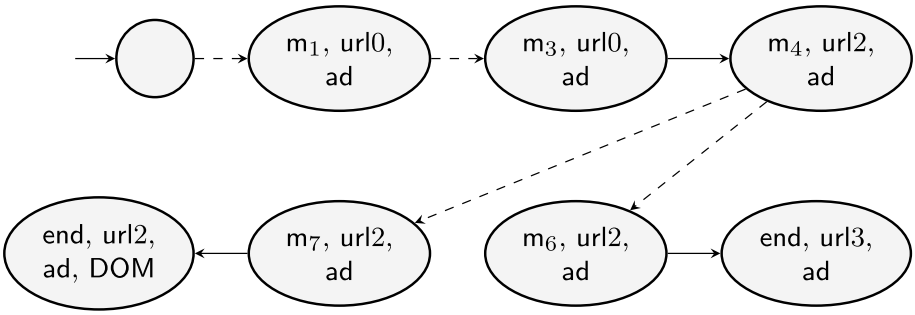


Fig. 6 Plant for program in Fig. 5, where m_i stands for the program line and $\text{url}j$ stands for $\text{url} = \text{str}_j$

Example 5 We rewrite the JavaScript code introduced in Section 1.1, into the program in Fig. 5. Then, one can apply some form of static analysis to generate $\mathcal{P}_0$. A branch of $\mathcal{P}_0$ is shown in Fig. 6, where url0 is a shorthand for the initial empty value of baseURL.

Step 1 (Extend): In our formulation of controller synthesis in Section 4.1, a controller is obtained by only removing a subset of controllable transitions. Although, removal of transitions mimics suppression of action, it may quickly result in deadlocks for deterministic programs and is not likely to result in a powerful enforcer alone. To extend the possible enforcement strategies, we allow the user to specify a set $\hat{\mathfrak{c}}_{\text{ext}}$ of additional controllable transitions encoding different enforcement strategies which are integrated with $\mathcal{P}_0$ in the step Extend in Fig. 4, which spits out an extended plant $\text{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\text{ext}})$. Notice that this procedure may add new behaviors to the plant and, hence, change the semantics of the monitored system. This may, in turn, violate some other specification. If such other critical specification exists, it has to be given as input to our enforcement algorithm.

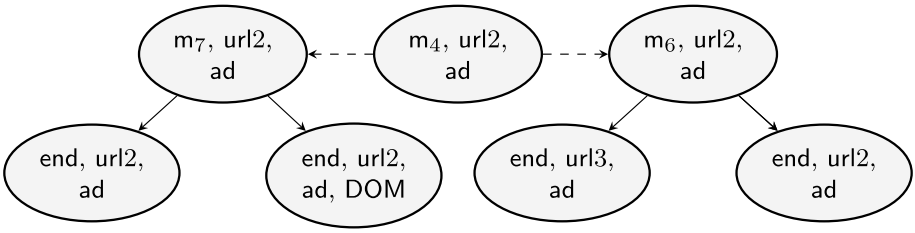


Fig. 7 Extension of plant in Fig. 6 with possibility to suppress execution of lines 6 and 7 in Fig. 5

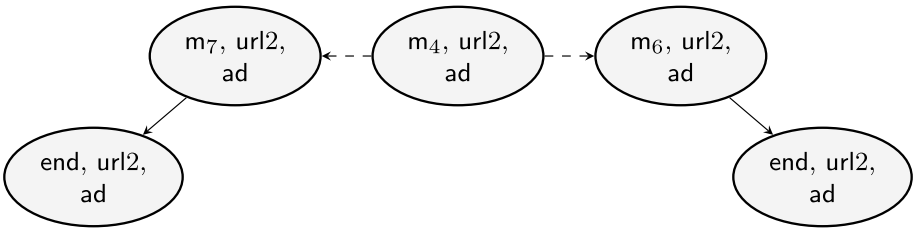


Fig. 8 The controller found by *Enforcer Search* for the extended plant in Fig. 7 and hyperproperties φ_{OD} and φ_{NI}

Example 6 Figure 7 depicts the branch from Fig. 6 extended with the possibility to suppress the execution of steps in lines 6 and 7 in Fig. 5. Since we allow the user to add transitions, states unreachable from the initial state may become reachable after the extension step. Hence, unreachable states are as relevant as reachable states in this setting.

Step 2 (Restrict): The initial plant $\mathcal{P}_0$ is not necessarily a precise description of our target systems. Hence, there may exist propositions that are not in the alphabet of both $\mathcal{P}_0$ and the target system $\mathcal{P}$. In order to design an enforcer based upon $\mathcal{P}_0$ that will be applied to some unknown $\mathcal{P}$, we must identify a set of *monitorable propositions* M that are common to $\mathcal{P}_0$ and $\mathcal{P}$. For example, if we are enforcing non-interference φ_{OD} (as defined in Fig. 2), then $\mathrm{M} = \{\mathrm{user}, \mathrm{DOM}\}$. Thus, after the extension step we restrict the output plant to the alphabet 2^{M}.

Step 3 (Controller synthesis): The output of $\mathrm{Restrict}$, denoted $\mathcal{M}$ and referred to as the *model*, is an input to the controller synthesis algorithm together with a NFH φ and a weight function W. If there exists a solution to the controller synthesis problem, we output it as the controller $\mathcal{M}'$. $\mathcal{M}'$ enforces φ over target plants that are the same as $\mathcal{M}$ after we apply the extension and the restriction to the alphabet 2^{M}. This class of plants is specified by $\psi_{\mathcal{M}}$.

Example 7 Figure 8 shows a partial view of the controller found by Enforcer Search in Fig. 4, where $\mathcal{P}_0$ is the extended plant in Fig. 7 and φ is the conjunction of φ_{OD} and φ_{NI} (see Section 2.1). While monitoring the JavaScript program in Fig. 1, with an advertisement loaded dynamically, the enforcer checks for each step whether it defines a transition in the controller $\mathcal{M}'$. For example, when the monitored JavaScript program reaches line 6 in the abstract program in Fig. 5, then the controller is in a state labeled $\{\mathrm{m}_6, \mathrm{url2}, \mathrm{ad}\}$. The advertisement changing baseUrl defines the transition from $\{\mathrm{m}_6, \mathrm{url2}, \mathrm{ad}\}$ to $\{\mathrm{end}, \mathrm{url3}, \mathrm{ad}\}$,

which is not a transition in $\mathcal{M}'$. Then, the enforcer forces the program to comply to the only controllable transition available, i.e., to $\{\mathrm{end}, \mathrm{url2}, \mathrm{ad}\}$, achieved by suppressing the execution of the baseUrl update by the advertisement script.

4.3 Detailed reduction to controller synthesis

We consider a set M of *monitorable propositions* that is common to both $\mathcal{P}_0$ and $\mathcal{P}$. In a white-box setting $\mathrm{M} = \mathrm{AP}$, whereas in a black-box setting $\mathrm{M} = \{\}$. In our framework, we use a *gray-box* setting, which considers the enforcer with partial knowledge of the system, i.e., $\mathrm{M} \subset \mathrm{AP}$. Note that in a gray-box setting, the gray-box plant $\mathcal{P}_0$ may not be a full representation of the target system $\mathcal{P}$. Both the gray-box plant $\mathcal{P}_0$ and the target plant $\mathcal{P}$ will have their alphabets restricted to 2^{M}. We need to guarantee that the plant obtained after an alphabet restriction to 2^{M} is still a plant and all information related to propositions in M is preserved. Plants that satisfy these requirements are called *monitorable*. All plants in our reduction (i.e., both the gray-box and the target systems) must be monitorable with respect to the given set M. For convenience, for a transition $(s, s') \in 2^{\mathrm{AP}} \times 2^{\mathrm{AP}}$, we define the restriction $(s, s')|_{\mathrm{M}} = (s|_{\mathrm{M}}, s'|_{\mathrm{M}})$. We also extend this notion to the restriction of sets by $T|_{\mathrm{M}} = \{t|_{\mathrm{M}} \mid t \in T\}$.

Definition 8 A plant $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ over AP is *monitorable* for the set $\mathrm{M} \subseteq \mathrm{AP}$ iff:

1. It has unique states that are all the possible values of the propositions AP:

$$S = 2^{\mathrm{AP}} \text{ and } \forall s \in S.\, L(s) = s,$$

2. Restriction over M preserves *transitions*:

$$\forall (s, s') \in \mathfrak{c} \cup \mathfrak{u}.\ \forall \tilde{s} \in S.\ \exists \tilde{s}' \in S.\ (s|_{\mathrm{M}} = \tilde{s}|_{\mathrm{M}}) \Rightarrow (s'|_{\mathrm{M}} = \tilde{s}'|_{\mathrm{M}} \ \wedge\ (\tilde{s}, \tilde{s}') \in \mathfrak{c} \cup \mathfrak{u})\,;$$

3. Restriction over M preserves *controllability*:

$$\forall (s, s'), (\tilde{s}, \tilde{s}') \in \mathfrak{c} \cup \mathfrak{u}.\ ((s, s')|_{\mathrm{M}} = (\tilde{s}, \tilde{s}')|_{\mathrm{M}}) \Rightarrow ((s, s') \in \mathfrak{c} \ \Leftrightarrow\ (\tilde{s}, \tilde{s}') \in \mathfrak{c})\,.$$

When a plant over AP is monitorable for its own alphabet, we refer to it just as a monitorable plant over AP. We define below the notion of restricting the alphabet of a monitorable plant $\mathcal{P}$ over AP to a subset of propositions $\mathrm{M} \subseteq \mathrm{AP}$. This is the operation performed at $\mathrm{Restrict}$ in Fig. 4. We prove that if the plant $\mathcal{P}$ is monitorable for M, then the restriction defines a monitorable plant as well.

Definition 9 Given a plant $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$, where $\forall s \in S. L(s) = s$, the *restriction* of $\mathcal{P}$ to $\mathrm{M} \subseteq \mathrm{AP}$ defines the plant $\mathcal{P}|_{\mathrm{M}} = (S_{\mathrm{M}}, \hat{s}_{\mathrm{M}}, \mathfrak{c}_{\mathrm{M}}, \mathfrak{u}_{\mathrm{M}}, L_{\mathrm{M}})$ where $S_{\mathrm{M}} = 2^{\mathrm{M}}$, $\hat{s}_{\mathrm{M}} = \hat{s}|_{\mathrm{M}}$, $\forall s \in S_{\mathrm{M}}.\ L_{\mathrm{M}}(s) = s$, $\mathfrak{c}_{\mathrm{M}} = \mathfrak{c}|_{\mathrm{M}}$ and $\mathfrak{u}_{\mathrm{M}} = \mathfrak{u}|_{\mathrm{M}}$.

Lemma 2 *Let $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ be a plant over AP monitorable for $M \subseteq AP$, then $\mathcal{P}|_M = (S_M, \hat{s}_M, \mathfrak{c}_M, \mathfrak{u}_M, L_M)$ is a monitorable plant over M.*

Proof Consider arbitrary plant $\mathcal{P}=(S,\hat{s},\mathfrak{c},\mathfrak{u},L)$ over AP monitorable for $\mathrm{M}\subseteq\mathrm{AP}$. We prove that $\mathcal{P}|_{\mathrm{M}}$ defines a plant. In particular, we show that $\mathfrak{c}_{\mathrm{M}}$ and $\mathfrak{u}_{\mathrm{M}}$ are disjoint sets. Assume towards a contradiction that there exists a transition $(s,s')\in\mathfrak{c}_{\mathrm{M}}\cap\mathfrak{u}_{\mathrm{M}}$. Then, by $\mathcal{P}$ being a plant, it follows that there exists $(s_c,s'_c)\in\mathfrak{c}$ and $(s_u,s'_u)\in\mathfrak{u}$ that are the same when projected over M: $(s_c,s'_c)\mid_{\mathrm{M}}=(s_u,s'_u)\mid_{\mathrm{M}}=(s,s')$. This contradicts our assumption that $\mathcal{P}$ is a monitorable for M (c.f. condition 3 in Def. 8). $\mathcal{P}|_{\mathrm{M}}$ being monitorable for M follows directly from $\mathcal{P}$ being monitorable for M. □

We prove next that the restriction of a plant $\mathcal{P}$ monitorable for M to the alphabet 2^{M} preserves all information related to propositions in M. Formally, the restriction $\mathcal{P}|_{\mathrm{M}}$ defines the same traces as restricting to M the set of traces defined by $\mathcal{P}$, i.e. $\mathrm{Tr}(\mathcal{P})|_{\mathrm{M}}=\mathrm{Tr}(\mathcal{P}|_{\mathrm{M}})$.

Lemma 3 *Let $\mathcal{P}=(S,\hat{s},\mathfrak{c},\mathfrak{u},L)$ be a plant monitorable for $M\subseteq AP$. Then, $Tr(\mathcal{P})|_M=Tr(\mathcal{P}|_M)$.*

Proof We prove that $\mathrm{Tr}(\mathcal{P})|_{\mathrm{M}}\subseteq\mathrm{Tr}(\mathcal{P}|_{\mathrm{M}})$. By condition 1 in Def. 8, every trace $s_0\cdots s_n$ in $\mathrm{Tr}(\mathcal{P})$ defines the path $s_0\cdots s_n$ in $\mathcal{P}$. Then, $s_0|_{\mathrm{M}}\cdots s_n|_{\mathrm{M}}$ is a path in $\mathcal{P}|_{\mathrm{M}}$ and $s_0|_{\mathrm{M}}\cdots s_n|_{\mathrm{M}}$ is a trace in $\mathcal{P}|_{\mathrm{M}}$. Now, we prove that $\mathrm{Tr}(\mathcal{P})|_{\mathrm{M}}\supseteq\mathrm{Tr}(\mathcal{P}|_{\mathrm{M}})$. Consider an arbitrary path $s_0\cdots s_n$ in $\mathcal{P}|_{\mathrm{M}}$. We want to prove that there exists a path $s'_0\cdots s'_n$ in $\mathcal{P}$ such that $s_i=s'_i|_{\mathrm{M}}$, for all $0\leq i<n$. By definition of restriction $s_0|_{\mathrm{M}}=\hat{s}|_{\mathrm{M}}$. Now consider (s_j,s_{j+1}), for arbitrary $0\leq j<n$. Pick an arbitrary s'_j s.t. $s'_j|_{\mathrm{M}}=s_j$. Then, by condition 2 in Def. 8, there exists s'_{j+1} s.t. $s'_{j+1}|_{\mathrm{M}}=s_{j+1}$ and $(s_j,s_{j+1})\in\mathfrak{c}\cup\mathfrak{u}$. □

While the controllable transitions of the input plant $\mathcal{P}$ represent non-determinism that can be removed, we allow the user to provide a set of transitions $\hat{\mathfrak{c}}_{\mathrm{ext}}\subseteq<\mathit{spanclass}='\mathit{crossLinkCiteEqu}'>2</\mathit{span}>^{\mathrm{AP}}\times 2^{\mathrm{AP}}$ that specify other possible strategies available to the enforcer. We define below the extension of a plant $\mathcal{P}$ with $\hat{\mathfrak{c}}_{\mathrm{ext}}$, represented as Extend in Fig. 4. We need to be careful to avoid turning uncontrollable transitions into controllable during the extension.

Definition 10 Given a plant $P=(S,\hat{s},\mathfrak{c},\mathfrak{u},L)$ over AP and an extension $\hat{\mathfrak{c}}_{\mathrm{ext}}\subseteq 2^{\mathrm{AP}}\times 2^{\mathrm{AP}}$, $\mathcal{P}$ *extended with* $\hat{\mathfrak{c}}_{\mathrm{ext}}$ defines the plant $\mathrm{ext}(\mathcal{P},\hat{\mathfrak{c}}_{\mathrm{ext}})=(S,\hat{s},\mathfrak{c}_{\mathrm{ext}},\mathfrak{u},L)$ where $\mathfrak{c}_{\mathrm{ext}}=\mathfrak{c}\cup(\hat{\mathfrak{c}}_{\mathrm{ext}}\setminus\mathfrak{u})$.

The enforcement problem introduced in Section 3 is parameterized by a formula ψ that describes a class of systems. Here, we concretely define ψ with respect to a *model*, denoted $\mathcal{M}$, that provides some knowledge about the target plants. For a given plant $\mathcal{P}_0$ and extension $\hat{\mathfrak{c}}_{\mathrm{ext}}$ such that $\mathrm{ext}(\mathcal{P}_0,\hat{\mathfrak{c}}_{\mathrm{ext}})$ is a monitorable plant, the model $\mathcal{M}$ is defined as $\mathrm{ext}(\mathcal{P}_0,\hat{\mathfrak{c}}_{\mathrm{ext}})|_{\mathrm{M}}$, which is the restriction to the alphabet 2^{M} of the extension of $\mathcal{P}_0$. The class denoted $\psi_{\mathcal{M}}$ consists of all monitorable plants with model $\mathcal{M}$, i.e., $\psi_{\mathcal{M}}=\{\mathcal{P}\mid\mathcal{M}=\mathrm{ext}(\mathcal{P},\hat{\mathfrak{c}}_{\mathrm{ext}})|_{\mathrm{M}}\}$. Our enforcers are controllers $\mathcal{M}'$ of a given plant $\mathcal{M}$. Applying an enforcer $\mathcal{M}'$ to a plant $\mathcal{P}\in\psi_{\mathcal{M}}$, amounts to restricting the controllable transitions in $\mathcal{P}$ to those in $\mathcal{M}'$.

Definition 11 Let $\mathrm{M}\subseteq\mathrm{AP}$ be a set of monitorable propositions. Let $\mathcal{P}_0$ be a plant over AP and $\hat{\mathfrak{c}}_{\mathrm{ext}}$ be an extension so that $\mathrm{ext}(\mathcal{P}_0,\hat{\mathfrak{c}}_{\mathrm{ext}})$ is monitorable over M. Let $\mathcal{M}=\mathrm{ext}(\mathcal{P}_0,\hat{\mathfrak{c}}_{\mathrm{ext}})|_{\mathrm{M}}$ be the model and $\mathcal{P}$ a plant in $\psi_{\mathcal{M}}$. Let $\mathcal{M}'=(S,\hat{s},\mathfrak{c}',\mathfrak{u},L)$ be a controller of $\mathcal{M}$. For

a plant $\mathcal{P}$, the output of the enforcer parameterized by $\mathcal{M}'$ and applied to $\mathcal{P}$, denoted $\mathbb{E}_{\mathcal{M}'}(\mathcal{P})$, is a controller of $\mathrm{ext}(\mathcal{P}, \hat{\mathfrak{c}}_{\mathrm{ext}}) = (S, \hat{s}, \mathfrak{c}_{\mathrm{ext}}, \mathfrak{u}, L)$ with controllable transitions: $\mathfrak{c}'_{\mathrm{ext}} = \{(s, s') \in \mathfrak{c}_{\mathrm{ext}} \mid (s, s')|_{\mathrm{M}} \in \mathfrak{c}'\}$.

For a given specification φ and weight function W, such controllers $\mathcal{M}'$ can be found as solutions to the controller synthesis problem. Here, the specification must be represented as an NFH $\mathbb{A}$ over the observable propositions M. Formally, we consider the hyperproperty $\varphi_{\mathbb{A},\mathrm{M}}$ that is satisfied by exactly the monitorable plants $\mathcal{P}$ such that $\mathrm{Tr}(\mathcal{P})|_{\mathrm{M}}$ is accepted by $\mathbb{A}$. In this case we find solutions $\mathcal{M}' \in \mathrm{C}(\mathcal{M}, \mathbb{A}, W)$. We next prove that enforcers obtained with controller synthesis are sound.

Theorem 1 *Let $\mathcal{P}_0$ be a plant over AP and $\hat{\mathfrak{c}}_{ext}$ an extension so that $\mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{ext})$ is monitorable for $M \subseteq AP$. Let $\mathcal{M} = \mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{ext})|_M$ be the resulting model of $\mathcal{P}_0$. Let $\mathcal{M}'$ be a solution to the controller synthesis problem for $\mathcal{M}$ and NFH $\mathbb{A}$, defining the enforcer $\mathbb{E}_{\mathcal{M}'}$. Then, for every $\mathcal{P} \in \psi_{\mathcal{M}}$, the output $\mathbb{E}_{\mathcal{M}'}(\mathcal{P})$ satisfies $\varphi_{\mathbb{A},M}$.*

Proof Recall $\mathbb{E}_{\mathcal{M}'}(\mathcal{P})$ is a controller of $\mathrm{ext}(\mathcal{P}, \mathfrak{c}_{\mathrm{ext}})$, so we can write $\mathbb{E}_{\mathcal{M}'}(\mathcal{P}) = (S, \hat{s}, \mathfrak{c}', \mathfrak{u}, L)$. Thus to show that $\mathbb{E}_{\mathcal{M}'}(\mathcal{P})|_{\mathrm{M}} = \mathcal{M}'$, it suffices to show that $\mathfrak{c}'|_{\mathrm{M}} = \mathfrak{c}'_M$. By construction $\mathfrak{c}'|_{\mathrm{M}} = \mathfrak{c}_{\mathrm{ext}}|_{\mathrm{M}} \cap \mathfrak{c}'_M$. Additionally as $\mathcal{P} \in \psi_{\mathcal{M}}$, $\mathfrak{c}_{\mathrm{ext}}|_{\mathrm{M}}$ is exactly the set $\mathfrak{c}_M$ of controllable transitions of M as $\mathcal{P} \in \psi_{\mathcal{M}}$. Thus $\mathfrak{c}'|_{\mathrm{M}} = \mathfrak{c}'_M$ and $\mathbb{E}_{\mathcal{M}'}(\mathcal{P})|_{\mathrm{M}} = \mathcal{M}'$, so by definition $\mathrm{Tr}(\mathcal{P}')|_{\mathrm{M}} = \mathrm{Tr}(\mathcal{M}')$. As $\mathcal{M}'$ is accepted by $\mathbb{A}$, then by definition $\mathbb{E}_{\mathcal{M}'}(\mathcal{P}) \models \psi_{\mathbb{A},\mathrm{M}}$. □

If precision is required, we can solve the more general weighted controller synthesis problem for an appropriate weighting function. To guarantee that original transitions from $\mathcal{P}$ are only removed if necessary, we require that they are given a positive non-zero weight while extensions to $\mathcal{P}$ are given a negative weight.

Theorem 2 *Let $\mathcal{P}_0 = (S_0, \hat{s}_0, \mathfrak{c}_0, \mathfrak{u}_0, L_0)$ be a monitorable plant inducing model $\mathcal{M} = \mathrm{ext}(\mathcal{P}_0)|_M$. Let W be a weight function for $\mathcal{M}$ such that $W(e) < 0$ for added transitions $e \in \mathfrak{c}_{ext} \setminus \mathfrak{c}_0|_M$ and $W(e) > 0$ otherwise for original transitions. Let $\mathcal{M}'$ be a solution to the weighted controller synthesis problem for $\mathcal{M}$, W, and NFH $\mathbb{A}$ defining the enforcer $\mathbb{E}_{\mathcal{M}'}$. If $\mathcal{P}_0 \models \varphi_{\mathbb{A},M}$ then $\mathbb{E}_{\mathcal{M}'}(\mathcal{P}_0) = \mathcal{P}_0$.*

Proof As $\mathcal{P}_0$ is monitorable and $\mathcal{P}_0 \models \varphi_{\mathbb{A},\mathrm{M}}$, then $\mathcal{P}_0|_{\mathrm{M}}$ is a solution to the weighted controller synthesis problem. As all transitions added in $\mathfrak{c}_{\mathrm{ext}}$ have negative cost to remove and all original transitions have positive cost, the total cost is uniquely minimized by removing all added transitions and including all original transitions, i.e, for the solution $\mathcal{P}_0|_{\mathrm{M}}$. Hence $\mathcal{M}' = \mathcal{P}_0|_{\mathrm{M}}$ is the unique minimal solution. Then, $\mathbb{E}_{\mathcal{M}'}(\mathcal{P}_0) = \mathcal{P}_0$. □

4.3.1 From controllers to trace sets

The reduction explained above is for the enforcement problem over monitorable plants. However, we are interested in the enforcement problem for sets of traces that describe monitorable plants. To extend to such sets, we need to define a bijective mapping from sets of traces to plants. Then, the results in Section 4.3 transfer naturally to trace sets.

We start by defining how to go from a plant to an appropriate set of traces. The translation needs to include enough information to allow rebuilding the plant and successfully apply an enforcer (as defined in Section 4) to the derived plant. In particular, the set of traces must include traces representing the unreachable part of the plant (as they may become reachable after the extension step) and, for each step, whether it was defined by a controllable or uncontrollable transition. For a given set of propositions AP, we extend it with proposition $\mathrm{unreach}$ to distinguish reachable from unreachable traces, and the proposition $\mathrm{control}$ to mark controllable steps in a trace.

Let $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ be a plant over AP monitorable for $\mathrm{M} \subseteq \mathrm{AP}$. For any two states $s, s' \in S$, then its control labeling is defined as $L^{\mathfrak{c}}_{s'}(s) = L(s) \cup \{\mathrm{control}\}$ if $(s, s') \in \mathfrak{c}$, and $L^{\mathfrak{c}}_{s'}(s) = L(s)$ otherwise. The set of all (reachable) traces with control annotations is defined as $\mathrm{Tr}^c(\mathcal{P}) = \{L^{\mathfrak{c}}_{s_1}(s_0) L^{\mathfrak{c}}_{s_2}(s_1) \cdots L(s_n) \mid s_0 \cdots s_n \text{ is a terminated path of } \mathcal{P}\}$. We now define the set of unreachable traces. The set of all *starting states* (states without incoming transitions) is defined as $\mathrm{Start}(\mathcal{P}) = \{s' \in S \mid \forall s \in S : (s, s') \notin \mathfrak{c} \cup \mathfrak{u}\}$. An *unreachable path* of a plant $\mathcal{P}$ is a finite sequence of states $s_0 \cdots s_n \in S^+$ such that s_0 is a starting state different from the initial state (i.e. $s_0 \in \mathrm{Start}(\mathcal{P}) \setminus \{\hat{s}\}$); for all steps $0 \leq i < n$, $(s_i, s_{i+1}) \in \mathfrak{c} \cup \mathfrak{u}$; and it ends in a terminated state (i.e. $s_n \in \mathrm{Term}(\mathcal{P})$). The set of unreachable paths is defined over $\mathrm{AP} \cup \{\mathrm{unreach}, \mathrm{control}\}$, as $\mathrm{Tr}_{\mathrm{un}} = \{L^{\mathfrak{c}}_{s_1}(s_0) \cup \{\mathrm{unreach}\} \cdots L(s_n) \cup \{\mathrm{unreach}\} \mid s_0 \cdots s_n$ is an unreachable path of $\mathcal{P}\}$.

Definition 12 Let $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$ be a monitorable plant. The complete set of traces of $\mathcal{P}$ is: $\mathrm{AllTr}(\mathcal{P}) = \mathrm{Tr}^c(\mathcal{P}) \cup \mathrm{Tr}^c_{\mathrm{un}}(\mathcal{P})$.

A *complete set of traces* $\mathcal{T}$ over AP satisfies the following conditions: (i) for each trace in $\mathcal{T}$ either $\mathrm{unreach}$ is always true or it is always false; (ii) all traces where $\mathrm{unreach}$ is false start with the same letter $\hat{s}_{\mathcal{T}} \in 2^{\mathrm{AP}}$, i.e. for all traces $t \in \mathcal{T}$ with $\mathrm{unreach} \notin t(i)$, for all $0 \leq i < |t|$, then $t(0) = \hat{s}_{\mathcal{T}}$. We define now the plant induced by a complete set of traces.

Definition 13 Let $\mathcal{T}$ be a complete set of traces. The plant $\mathcal{P}_{\mathcal{T}}$ induced by $\mathcal{T}$ is defined as:

- $S = 2^{\mathrm{AP}}$ and $\forall s \in S : L(s) = s$;
- $\hat{s} = \hat{s}_{\mathcal{T}}$;
- $\mathfrak{c} = \{(s, s') \mid s_0 \cdots s_n \in \mathcal{T}$ and $\exists 0 \leq j < n$ such that $s = s_j$, $s' = s_{j+1}$ and $\mathrm{control} \in s\}$;
- $\mathfrak{u} = \{(s, s') \mid s_0 \cdots s_n \in \mathcal{T}$ and $\exists 0 \leq j < n$ such that $s = s_j$, $s' = s_{j+1}$ and $\mathrm{control} \notin s\}$;

We say that a set of traces is monitorable iff $\mathrm{AllTr}(\mathcal{P}_{\mathcal{T}}) = \mathcal{T}$.

4.4 Discussion

In order to enforce hyperproperties over plants with potentially infinite state-space, such as JavaScript programs, in our approach we apply controller synthesis to a model $\mathcal{M}$, which is an abstraction or approximation of the extended plant. For simplicity, in this work, the model is the *restriction* of the extension of some base plant to a user-specified set of monitorable propositions. We establish the notion of the extended plant being monitorable (in Definition 8), which is a sufficient condition to guarantee the soundness of our controller synthesizes approach. However, this condition must be imposed carefully; otherwise, it

might raise *limitations* in some instances. We now discuss this limitation in detail using the example in Fig. 9.

4.4.1 Imposing restriction after extension

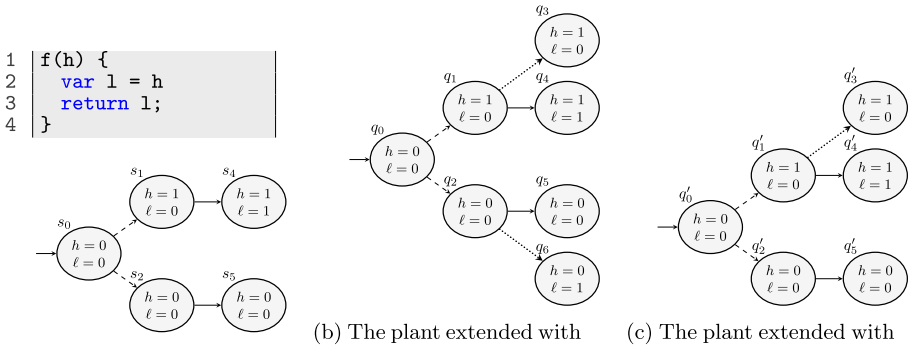
It is worth noting that it is advantageous to impose monitorability after extending the original plant. Consider the simple JavaScript function and its corresponding plant in Fig. 9a, where l is initialized to 0. The program has low-security ℓ and high-security h variables. For the set of monitorable propositions $\mathrm{M} = \{\ell\}$, the plant in Fig. 9a (before any extension) is not monitorable (condition 2 in Definition 8 is not satisfied). However, the plant may become monitorable *after* extension. For example, consider the extension that adds the possibility of negating the assignment (i.e., var l =!h) in line 2, presented as the extended plant in Fig. 9b. In this case, the extension added the behavior missing from the restriction to satisfy the monitorability conditions (i.e., the extended plant is now monitorable for $\mathrm{M} = \{\ell\}$). Our approach requires monitorability on the extended plant to avoid this potential limitation (see Def. 11 and Theorem 1).

4.4.2 Targeting specific class of hyperproperties

In practice, it may be challenging to identify an appropriate set of monitorable propositions M. The set M must be large enough to ensure monitorability while not being too large to result in a model too complex for synthesis. Alternatively, it may be possible to apply our approach to more easily computed under- or over-approximations of the plant while maintaining soundness for a reduced class of hyperproperties. Indeed, this is the case for under-approximations and existential hyperproperties. Formally, given a base plant $\mathcal{P}_0$, the set of plants under-approximated by $\mathcal{P}_0$ is:

$$\psi_{\mathcal{P}_0,\text{under}} = \{\mathcal{P} \mid \text{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\text{ext}}) \sqsubseteq \text{ext}(\mathcal{P}, \hat{\mathfrak{c}}_{\text{ext}})\},$$

where $\sqsubseteq$ indicates a *subplant* relation (i.e., the left side plant is defined by a subset of states and transitions of the right side plant). It is readily shown that a controller synthesized



(a) The function and corresponding plant.

(b) The plant extended with negation, i.e., changing line 2 to `var l = !h`.

(c) The plant extended with suppressed assignments, i.e., suppressing line 2.

Fig. 9 A Javascript function, the corresponding plant, and two possible extended plants. Dotted edges denote controllable transitions added in the extension while dashed edges denote uncontrollable transitions

for $\mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\mathrm{ext}})$ and applied to plants in $\psi_{\mathcal{P}_0,\mathrm{under}}$ is sound for existential hyperproperties. However, the dual of this statement is not valid for over-approximations and universal hyperproperties. Formally, given a base plant $\mathcal{P}_0$, the set of plants over-approximated by $\mathcal{P}_0$ is:

$$\psi_{\mathcal{P}_0,\mathrm{over}} = \{\mathcal{P} \mid \mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\mathrm{ext}}) \sqsupseteq \mathrm{ext}(\mathcal{P}, \hat{\mathfrak{c}}_{\mathrm{ext}})\}\,.$$

A controller synthesized for $\mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\mathrm{ext}})$ may result in deadlocks when applied to a plant $\mathcal{P} \in \psi_{\mathcal{P}_0,\mathrm{over}}$. For example, consider enforcement for the function and plant $\mathcal{P}$ in Fig. 9a that allows the enforcer to *suppress* line 2. This enforcement strategy defines an extension that adds only one transition: from a state with $h = 1$ and $\ell = 0$ (q_1' in Fig. 9c), to the state that does not change these values (q_3' in Fig. 9c). Now consider that as a base plant for our approach (i.e., $\mathcal{P}_0$), we are given the plant in Fig. 9b. Note that the above extension does not add new transitions to Fig. 9b, and the plant in Fig. 9c is over-approximated by it. If we apply controller synthesis to enforce noninterference to $\mathrm{ext}(\mathcal{P}_0, \hat{\mathfrak{c}}_{\mathrm{ext}})$ (i.e., plant in Fig. 9b), one possible solution is to disable transitions to states with $\ell = 0$ in the final step (i.e., removing transitions (q_1, q_3) and (q_2, q_5)). However, applying this controller to the plant in Fig. 9c (i.e., removing transitions (q_1', q_3') and (q_2', q_5')) results in a deadlocked state q_2'. In this example, this means that the program is terminated before the return statement. Hence the synthesized controller is not a valid controller for the plant in Fig. 9c. We remark that this stems from the requirement of liveness (i.e. avoiding deadlocks) being an existential property which cannot be enforced using an over-approximation.

5 QBF encoding

While the controller synthesis problem for hyperproperties has Nonelementary complexity in general, for plants given by directed acyclic graphs the complexity is PSPACE-complete [16] (the optimization objective in our problem does not change this complexity, since optimization itself is a hyperproperty). Solutions to the minimal controller synthesis problem can then be extracted from certificates for QBF satisfaction and be used in the implementation of enforcers.

5.1 Encoding inputs and outputs

Given a plant $\mathcal{P} = (S, \hat{s}, \mathfrak{c}, \mathfrak{u}, L)$, a controller $\mathcal{P}'$ is uniquely determined by the choices of controllable transitions $\mathfrak{c}' \subseteq \mathfrak{c}$. Thus, we define a set of Boolean variables $\boldsymbol{v}_{\mathfrak{c}\,s,s'}$ for every $(s, s') \in \mathfrak{c}$, which the value of $\boldsymbol{v}_{\mathfrak{c}\,s,s'}$ indicates if $(s, s') \in \mathfrak{c}'$.

While the values of these controller variables serve as the output of synthesis, the user must provide QBFs describing the plant $\mathcal{P}$ and NFH $\mathbb{A}$ as inputs. These formulas are defined with respect to a fixed encoding of the states, defined as v_s, of $\mathcal{P}$ and $\mathbb{A}$ in a logarithmic number of bits. The plant is described by two formulas: $I_{\mathcal{P}}(v_s)$ satisfied when v_s encodes the initial state, and $\Delta_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, v_s, {v_s}')$ satisfied when the transition from v_s to ${v_s}'$ is present in the controller encoded by $\boldsymbol{v}_{\mathfrak{c}}$. For example, if $[\![s]\!]_S$ denotes the encoding of state s such a $\Delta_{\mathcal{P}}$ is depicted in Table 3. Additionally, for simplicity we will assume that the encoding of a state s is given by the values of its label $L(s)$. Likewise the hyperautomaton is described

by three formulas: $I_{\mathbb{A}}(v_q)$ and $F_{\mathbb{A}}(v_q)$ satisfied when v_q encodes an initial or accepting state, respectively, and $\Delta_{\mathbb{A}}(v_q, {v_s}^1, \cdots, {v_s}^n, {v_q}')$ satisfied when there is a transition from v_q to ${v_q}'$ labeled by the tuple encoded by $({v_s}^1, \cdots, {v_s}^n)$.

5.2 Feasibility

We now construct a QBF spec which is satisfied by feasible solutions (i.e., no deadlocks and acceptance by the NFH) to the controller synthesis problem:

$$\text{spec}(\boldsymbol{v}_{\mathfrak{c}}) = \neg\text{deadlock}(\boldsymbol{v}_{\mathfrak{c}}) \ \wedge \ \text{accept}(\boldsymbol{v}_{\mathfrak{c}})\,. \tag{1}$$

Recall that a deadlocked state s is one that is not terminal in the plant, i.e., $s \notin \text{Term}(\mathcal{P})$, but has no outgoing transition in the controller $\mathcal{P}'$. That is,

$$\text{deadlock}(\boldsymbol{v}_{\mathfrak{c}}) = \bigvee_{s \in S \setminus \text{Term}(\mathcal{P})} \ \bigwedge_{\substack{s' \in S \\ s.t.(s,s') \in \mathfrak{c}}} \neg \boldsymbol{v}_{\mathfrak{c}\, s,s'}\,. \tag{2}$$

Next, we define accept. This requirement can be expressed directly with a QBF with paths encoded as sequences $\boldsymbol{v}_s$ of state variables of length d, the length of maximal paths in $\mathcal{P}$ if it is acyclic, or the value from Lemma 1 in general. To restrict quantification to only paths of the controller, we use the formula $\text{reach}_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_s)$ in Table 3 which is satisfied by sequences $\boldsymbol{v}_s$ which are paths of the controller encoded by $\boldsymbol{v}_{\mathfrak{c}}$. As in [23, 30], this formula takes advantage of quantification to avoid unrolling transitions like:

$$I_{\mathcal{P}}(\boldsymbol{v}_{s1}) \ \wedge \ \Delta_{\mathcal{P}}(\boldsymbol{v}_{s1}, \boldsymbol{v}_{s2}) \ \wedge \ \Delta_{\mathcal{P}}(\boldsymbol{v}_{s2}, \boldsymbol{v}_{s3}) \ \cdots \tag{3}$$

Likewise, we express acceptance of a path of the NFH with formula $\text{reach}_{\mathbb{A}}$ in Table 3. In detail, $\text{reach}_{\mathbb{A}}({\boldsymbol{v}_s}^1, \cdots, {\boldsymbol{v}_s}^n, \boldsymbol{v}_q)$ is satisfied when $\boldsymbol{v}_q$ encodes an accepting sequence of states of the underlying NFA $\mathbb{A}$ over the zip of the labels by the paths encoded by ${\boldsymbol{v}_s}^1, \cdots, {\boldsymbol{v}_s}^n$. Finally, recall that the word formed ${\boldsymbol{v}_s}^1, \cdots, {\boldsymbol{v}_s}^n$ by these labels is accepted if there exists some accepting path $\boldsymbol{v}_q$. Together, these formulas allow us to express that the traces of a controller encoded by $\boldsymbol{v}_{\mathfrak{c}}$ satisfy condition (2.1) iff it satisfies:

Table 3 Formulas for the QBF encoding of the controller synthesis problem

$$\Delta_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, v_s, {v_s}') = \left(\bigvee_{s,s' \in \mathfrak{u}} [\![s]\!]_S = v_s \ \wedge \ [\![s']\!]_S = {v_s}'\right) \vee$$
$$\left(\bigvee_{s,s' \in \mathfrak{c}} [\![s]\!]_S = v_s \ \wedge \ [\![s']\!]_S = {v_s}' \ \wedge \ \boldsymbol{v}_{\mathfrak{c}\, s,s'}\right)$$
$$\text{reach}_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_s) = I_{\mathcal{P}}(\boldsymbol{v}_{s0}) \ \wedge \forall z_s, {z_s}'.$$
$$\left(\bigvee_{i=1}^{d} z_s = \boldsymbol{v}_{si} \ \wedge \ {z_s}' = \boldsymbol{v}_{si+1}\right) \ \rightarrow \ \Delta_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, z_s, {z_s}')$$
$$\text{reach}_{\mathbb{A}}({\boldsymbol{v}_s}^1, \cdots, {\boldsymbol{v}_s}^n, \boldsymbol{v}_q) = I_{\mathbb{A}}(\boldsymbol{v}_{q_0}) \ \wedge F_{\mathbb{A}}(\boldsymbol{v}_{q_d}) \ \wedge \ \forall {z_s}^1, \cdots, {z_s}^n, z_q, {z_q}'.$$
$$\left(\bigvee_{i=1}^{d} z_q = \boldsymbol{v}_{q_i} \ \wedge \ {z_q}' = \boldsymbol{v}_{q_{i+1}} \ \wedge \ \bigwedge_{j=1}^{n} {z_s}^j = {\boldsymbol{v}_s}_i^j\right) \rightarrow \ \Delta_{\mathbb{A}}(z_q, {z_s}^1, \cdots, {z_s}^n, {z_q}')$$

$$\begin{aligned}\text{accept}(\boldsymbol{v}_{\mathfrak{c}}) =& \mathbb{Q}_1 \boldsymbol{v}_s^1.\ \text{reach}_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_s^1) \circ_1 \dots \\ & \mathbb{Q}_n \boldsymbol{v}_s^n.\ \text{reach}_{\mathcal{P}}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_s^n) \circ_n \exists \boldsymbol{v}_q.\ \text{reach}_{\mathbb{A}}(\boldsymbol{v}_s^1, \cdots, \boldsymbol{v}_s^n, \boldsymbol{v}_q)\,,\end{aligned}$$

where $\circ_i = \wedge$ if $\mathbb{Q}_i = \exists$, and $\circ_i = \rightarrow$ if $\mathbb{Q}_i = \forall$ for all $i \leq n$ in the sequence of quantifiers in the input NFH. So in total the variables $\boldsymbol{v}_{\mathfrak{c}}$ encode a feasible solution to the controller synthesis problem if they satisfy spec.

5.3 Minimality

We first express the comparison of the costs of two controllers encoded by $\boldsymbol{v}_{\mathfrak{c}}$ and $\boldsymbol{v}_{\mathfrak{c}}{}'$:

$$\sum_{e \in \mathfrak{c}} W(e) \boldsymbol{v}_{\mathfrak{c}e} \leq \sum_{e \in \mathfrak{c}} W(e) \boldsymbol{v}_{\mathfrak{c}e}'\,. \tag{4}$$

Such binary linear inequalities and related cardinality constraints can be encoded into a Boolean formula in a variety of ways [60]. Thus, we use the totalizer encoding [10] to create a unary representation of each sum as a sequence of variables. The comparison of the unary representations is straightforward and we construct the QBF $\text{less cost}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_{\mathfrak{c}}{}')$ which is satisfied when the cost of the controller encoded by $\boldsymbol{v}_{\mathfrak{c}}$ is no more than that encoded by $\boldsymbol{v}_{\mathfrak{c}}{}'$. Then given the QBF spec encoding feasibility, the controller encoded by $\boldsymbol{v}_{\mathfrak{c}}$ is a solution to the minimal controller synthesis if it satisfies:

$$\text{minspec}(\boldsymbol{v}_{\mathfrak{c}}) = \forall \boldsymbol{v}_{\mathfrak{c}}{}'.\ \text{spec}(\boldsymbol{v}_{\mathfrak{c}}) \ \wedge\ \text{spec}(\boldsymbol{v}_{\mathfrak{c}}{}') \rightarrow\ \text{less cost}(\boldsymbol{v}_{\mathfrak{c}}, \boldsymbol{v}_{\mathfrak{c}}{}')\,. \tag{5}$$

While existing QBF solvers check the truth of formulas without free variables, they can also return a certificate consisting of a satisfying variable assignment to the outermost quantifier if it is existential similar to SAT. As such, we check the truth of the QBF $\exists \boldsymbol{v}_{\mathfrak{c}}.\ \text{minspec}(\boldsymbol{v}_{\mathfrak{c}})$ from which the controller can be extracted from the certificate for $\boldsymbol{v}_{\mathfrak{c}}$ as follows. Since $\boldsymbol{v}_{\mathfrak{c}}$ are variables that encode the controllable transitions (i.e., controllers actions), so the certificate for $\boldsymbol{v}_{\mathfrak{c}}$ indicates how the program should proceed at run time from the current state, which gives us the controller. We can further use this controller information to build the *enforcer* by monitoring-and-enforce the runtime value of the program (e.g., to conduct the suppressing action or not) accordingly.

6 Case studies and experimental evaluation

We have implemented our algorithm into a proof-of-concept tool[3]. Our QBF translation to solve weighted controller synthesis (implemented in Python) generates QCIR formulas and is integrated with the QBF solver QuAbS [58]. The output is a plant, i.e., a transition system describing the controller for enforcement. The detailed proof of soundness of our tool-chain is presented in Section 4.3. We now introduce two applications and their empirical evaluations: (1) JavaScript runtime security enforcement, and (2) privacy enforcement for obfuscation-aware eavesdroppers. Our security policies range over a diverse set of

[3] Available at: https://github.com/hyperenforce/artifact

hyperproperties including alternation-free and alternating quantifiers, which are generally not possible using SME or taint-tracking.

6.1 JavaScript runtime security enforcement

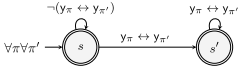
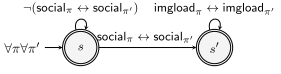
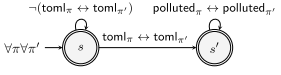
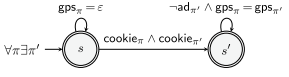
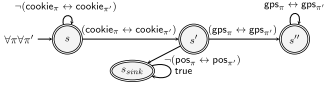
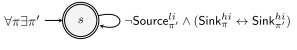
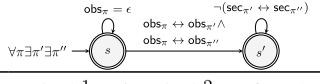
We evaluate our implementation with a rich set of JavaScript programs including some benchmarks from [55]. Our case study selection criteria primarily focus on choosing applications with requirements expressed by hyperproperties, particularly those involving quantifier alternations. Additionally, we aim to include a diverse set of cases to demonstrate the generality of our approach. We use two existing tools to work with JavaScript programs: (1) ExpoSE [36], a dynamic symbolic execution engine, to explore the state space of the program and output its traces; and (2) Jalangi [53], a dynamic analysis framework for JavaScript, to instrument the programs.

We begin by manually extending the target program with user-specified set of controllable transitions and extensions for enforcement. In our use cases, we extend the programs with the possibility to suppress assignments to controllable variables. We then employ ExpoSE to explore the resulting behaviors and construct the extended plant. Note that the extended plant generated by ExpoSE is a gray-box representation of the extended program, since ExpoSE provides an approximation of the program's behavior. The script to synthesize the enforcer has the extended plant and the NFH (specifying the hyperproperty to enforce) as input. Specifying the NFH is a manual step; however, one could define the target hyperproperties as HyperLTL and use standard translations of LTL to NFA to generate the NFH. If the script finds an enforcer, the respective controller is outputted. Finally, we translate the output controller into a Jalangi analysis class to enforce the input hyperproperty. As ExpoSE is built on top of Jalangi, the translation from the output controller to a Jalangi class is seamless. We now introduce the policies for the investigated cases:

Information-flow Security. Confidentiality policies forbid the flow of secret sources (high confidentiality - hc) to public sinks (low confidentiality - lc); while integrity states that no information from untrusted sources (low integrity - li) should flow to a trusted sink (high integrity - hi). In addition to the confidentiality and integrity examples described in earlier sections, we adapted a use-case of social media fingerprinting, as reported in [35]. In this attack, an attacker can bypass Same Origin Policy (SOP) restrictions and obtain a list of the social networks a user is logged in by observing how different images are loaded. To enforce no leakage of login information, we require that for all pair of executions (denoted π and π'), as soon as the image source refers to the same social network (i.e., $\mathrm{social}_{\pi} \leftrightarrow \mathrm{social}_{\pi'}$), the webpage observable behavior for these images should be the same (i.e., $\mathrm{imgload}_{\pi} \leftrightarrow \mathrm{imgload}_{\pi'}$).

Declassification. In practice, we may need to relax non-interference, which does not allow revealing anything about a secret, to *declassification* [48], revealing only specific pieces of sensitive information (e.g., the last four digits of credit card number). We formulate this requirement with the NFH $\mathbb{A}_{D_{\mathrm{OD}}}$ in Tab. 4 requiring for any two executions with the same low confidentiality inputs (In^{lc}) and *declassified information* (Decl), the low confidentiality outputs (Out^{lc}) should agree as well. All declassification policies in our use cases are examples of *delimited release* [48, 52], but one can also express other types of declassification such as robust declassification [59] and gradual release [4] using an NFH.

Table 4 Summary of experiments for JavaScript case studies, where **LoC** is the lines of code in the original program, **#V/E** are the numbers of vertices and edges of the synthesized controller, **syn** is the synthesis time, **enf.** and **unenf.** (in milliseconds) indicate the execution time of the program with and without the enforcer

	Prop.	Model	Hyperautomaton	LoC	#V/E	syn[s]	enf.	unenf.	Overhead
1	OD	login	Figure 2	27	20/23	217.36	2297	1765	23.16 %
2	NI	login	Figure 3	27	20/23	155.85	2220	1584	28.64 %
3	OD	impl^1		11	4/3	2.05	1798	1492	17.0 %
		impl^2		12	8/8	8.81	1836	1496	18.51 %
4	OD	socleak^1		50	12/12	120.98	2340	1904	18.60 %
		socleak^2		58	23/24	514.73	2311	1649	28.64 %
5	OD	toml		50	12/11	11.82	1583	1199	24.25 %
6	D_{NI}	gps		48	21/25	64.42	2304	1858	19.35 %
7	D_{OD}	gps		48	21/25	38.77	2141	1835	14.2 %
8	CI	printer		64	19/18	290.32	21040	20292	3.5 %
		m-log		72	18/17	87.72	1096	998	8.9 %
9	*Den*	wallet		35	11/11	3.82	2026	1640	19.05 %

Cases impl^1 and impl^2 are from [7], socleak^1 and socleak^2 are from [35], toml is from [14], printer and m-log are from [55], wallet is from [9] and the rest are home-grown

Conditional declassification. We also investigate *conditional declassification*, revealing declassified information only when certain conditions are fulfilled. We combine it with non-inference to specify *conditional partial declassification* by identifying a subset of trusted executions that describe what information can be released. For example, in a GPS application, which reveals the user location only after the cookies are confirmed (i.e., GPS tracking is enabled). Revealing location should, however, be partial (i.e., coordinates are round up) to protect the user's privacy. Then, the cookie being enabled is the condition for declassification and the allowed coordinates can be described by a set of trusted executions. As in Tab. 4, we depict an NFH $\mathbb{A}_{D_{\text{NI}}}$ specifying conditional partial declassification.

Code injection. *Code injection* attacks affect the integrity of a web page and are widely considered as an important aspect of JavaScript security [17, 31, 55]. The enforcement goal here is to prevent untrusted data from flowing into a trusted sink [55]. The automaton in Fig. 3 and case #8 in Tab. 4, are instances of this policy.

Deniability. *Deniability* [12, 50] is a confidentiality policy that that has recently gained interest. Unlike non-inference which requires that observations do not reveal if a trace was secret, deniability limits the information that observations reveal about the value of a secret. Formally, this requires for every such observation, there exists a number of traces (in our example, we require two distinct traces) producing the same observation, each possessing a different secret value.

6.2 Enforcing privacy for obfuscation-aware eavesdroppers

Many cyber-physical systems rely on communication of sensitive information for control and monitoring. In many cases, the enforcement of privacy from passive eavesdroppers is critical. For example, users of location-based services (LBS) for navigation may desire to hide their exact location from the service. Privacy has been studied in the area of discrete-event systems (DES) using the information flow-property of *opacity* which is similar to non-inference. There, one proposed mechanism for enforcement over automata models is supervisory control, where behaviors that reveal sensitive information are restricted [57]. However, for many systems it is infeasible to impose such restrictions, e.g., restricting the movement of LBS users. In this case, *obfuscation* of communications [62] has been proposed to enforce privacy by editing the output stream of the system in a way that is undetectable by eavesdroppers: *the obfuscated outputs must mimic the original system*.

However when using obfuscation, we must maintain *utility* ensuring that sensitive information is available to the intended users of the system. To this end, [61] proposed the use of an *inference function* designed alongside the obfuscator using distributed synthesis for LTL properties [26]. The inference function acts as a *key*, as in cryptography, that is shared only with intended users at initialization which enable them to interpret obfuscated outputs. However, this approach only guarantees privacy under the assumption that eavesdroppers are unaware of the goals of the obfuscation. This is in conflict with the typical requirement that security should be maintained when everything about the system is known, except the key. To address this, we consider a set of obfuscators and inference functions acting as key pairs as depicted in Fig. 10. In this case, privacy from obfuscation-aware eavesdroppers and the distributed structure obfuscator and inference function can be formulated as hyperproperties. In this way, privacy enforcement with obfuscation can be formulated as an instance of the enforcement problem considered in this work.

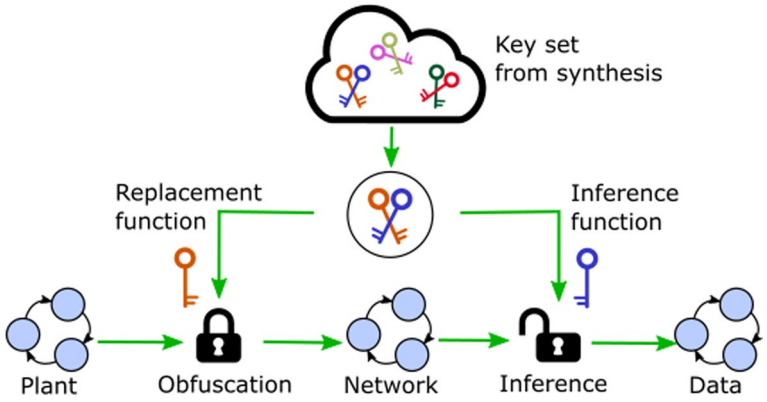
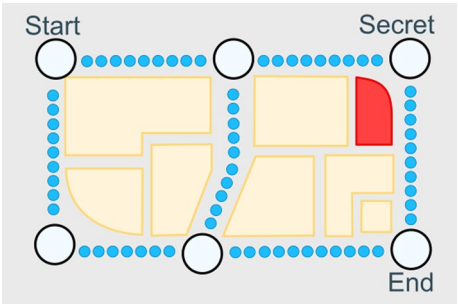


Fig. 10 Privacy-enforcing obfuscation

Fig. 11 User visits to the secret locations hidden by obfuscation



As an example, we consider the enforcement of location privacy for a user on the map depicted in Fig. 11 as they move from the start location to the end location in three discrete steps. For navigational purposes they must report their approximate location to LBS, but wish to hide or cloak visits to *secret* locations such as a hospital or bank, depicted on the map as Secret. At the same time, they want to make this secret information available to another trusted service, for example sharing their location with friends, using the LBS as a proxy. To this end we consider obfuscation with *replacement functions*, reporting an altered location each step, while securely providing the trusted service an inference function to recover visits to the secret location. Critically, the obfuscated locations must correspond to a feasible path over the map. To formulate this as a white-box enforcement problem, we construct a plant over propositions encompassing user locations loc, the secret $\mathrm{sec} \in \mathrm{loc}$, key selection key, and obfuscated locations obf. The obfuscator must *react* to the user movement, modeled in the plant by alternation between uncontrollable user movement and controllable obfuscation outputs. This input plant simply reports true user locations, while the extension adds obfuscated outputs consistent with the map.

Over this plant, we consider hyperproperties for enforcement modeled by the NFH. Opacity/privacy is modeled by the NFH $\mathbb{A}_{\mathrm{priv}}$ (depicted in Fig. 12, (up)). Next, the requirement that the obfuscator outputs depend only on the observed user locations and key is modeled by the NFH $\mathbb{A}_{\mathrm{loc}\cup\mathrm{key}\to\mathrm{obf}}$. Likewise the requirement that the secret can be inferred from the obfuscator outputs can be expressed by the NFH $\mathbb{A}_{\mathrm{obf}\cup\mathrm{key}\to\{\mathrm{sec}\}}$. Acceptance of the

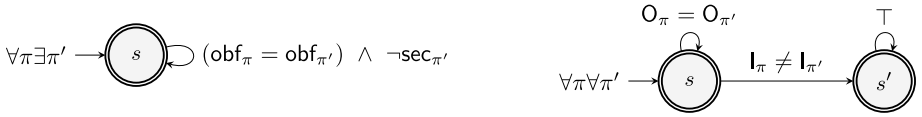


Fig. 12 NFH for opacity/privacy requirement $\mathbb{A}_{\text{priv}}$ (up) and variable dependence $\mathbb{A}_{I\to O}$ (down)

controller by these NFHs guarantees for each key, we can extract a corresponding replacement function and inference function (both can be depicted as $\mathbb{A}_{I\to O}$ in Fig. 12, (down)).

6.3 Evaluation and analysis

All cases studies shown in Table 4 are run on a MacBook Pro with Apple M1 Max chip and 64 GB of memory. The table summarizes the results of enforcing different hyperproperties on a set of JavaScript code from [7, 55].

We run each case 1 million times to ensure robustness of measurement using Jalangi and calculate the overhead in the last column. The input JavaScript code is originally exposed to an attack (i.e., does not satisfy the given property) and our goal is to build enforcers for each program to satisfy the target property. We were able to successfully build enforcers for a variety of hyperproperties, including the ones that are not 2-safety. Our implementation is also efficient when synthesizing an enforcer for non-trivial programs and complex hyperautomata (synthesis time within at most a few minutes). Note that the size of controllers (i.e., numbers of vertices and edges) are small because we only consider the controllable parts in a program and do not do line-by-line tracking.

To measure the runtime overhead after applying our synthesized enforcer to the original program, we use a simple driver script to execute each program both with and without enforcers, using randomly generated inputs. The overhead of our benchmarks ranges from 3.5% to 28.64%, depending on the ratios of the original program versus the parts that we actually enforce. For example, in the **login** program, every assignment of baseUrl needs to always be monitored, hence, a higher overhead. However, in cases such as the **printer** program, many operations and variable updates are not affecting the given property, that is, the enforcer is triggered less often, hence; a lower overhead. Our experiments do not show obvious differences on overheads between violating and satisfying runs. The reason is that our implementation in Jalangi only evaluates monitorable parts when a controllable variables is about to update. In other words, for both violating or satisfying runs, the same evaluation are done similarly.

We are able to synthesize enforcers for all our benchmarks. Unfortunately, we were unsuccessful in replicating implementations of SME and Faceted SME for comparison and contrast, but we believe the overhead incurred by our approach is well below SME since it has to run multiple copies of the same program. We also synthesized a solution for obfuscation of the LBS system over the map depicted in Fig. 11. The constructed plant used as input contained 39 vertices and 54 transitions. Our solver for controller synthesis was able to find a feasible controller in 16 seconds, representing two obfuscator and inference function pairs (one for each key) enforcing privacy and utility. Our empirical evaluation covers both gray-box (JavaScript cases), and white-box (obfuscation cases) enforcement. That is, we are able to synthesize effective enforcers with either partial or full knowledge about the

programs. Our selected benchmarks in this paper serve as instances for our proof-of-concept implementation.

7 Related work

In this section, we present the related work on the enforcement of security properties. To better compare with existing enforcing techniques from different research fields, we distinguish between *language-based approaches* (taking advantage of programming language features and working at the application level), and *logic-based approach* (focusing on general solutions independent of the system to be enforced).

Security Automata Schneider [51] was first to explore enforceability of security policies by studying the class of *execution monitoring* (EM) enforcers and defining *security automata* as an EM. EMs are enforcement mechanisms that monitor a system and terminate the current execution when a violation of the policy is detected. Ligatti et al. introduced *edit automata* [34], which extends security automata with the ability to change executions that are not compliant with the policy they enforce (instead of just terminating them). Martinell and Matteucci [37] propose a synthesis technique for EM from a μ-calculus specification. As proved in [2], there are hyperproperties that cannot be expressed by μ-calculus. A survey on enforceable security policies can be found in [32], while [25] presents a classification of properties that can be enforced at runtime.

Language-based Type systems can enforce non-interference properties at compile time by enforcing a stronger safety property to rule out potentially dangerous programs [47]. Dynamic monitoring approaches typically do not need a programmer to add types to the source code, which place less burden on the developers, and usually allow more safe programs to execute [5, 8, 21, 28, 49, 56]. These two approaches can be combined into a hybrid solution where the runtime monitor works together with static analysis [6, 27, 29, 41, 46, 54]. Both static and dynamic approaches for information-flow control mentioned before suffer from high numbers of false alarms on real-world programs.

In secure multiple execution (SME) [22, 65] and multi-facets (MF) [7, 42], a program is executed multiple times with a changed semantics to enforce non-interference and the output of the program is secure without changing programs that were correct to start with. SME defines black-box enforcers and inherently incurs high runtime overhead [1], which our approach does not. MF defines white-box enforcers: it uses the source code of program to enforce non-interference, but it lacks the generality of our approach.

Language-based enforcement mechanisms are typically tailored for their target security condition. By employing hyperproperties, our approach develops a uniform framework for enforcing a variety of confidentiality and integrity policies. Finally, our notion of black-box RE is stricter than SME. For example, while SME does not have access to the code of a program (black-box in the language-based view), it does influence the program runs, particularly how the different security run contexts are scheduled and interact. In the black-box definition we introduce here, we require enforcers to be *transferable*, i.e., they need to be sound and precise for all systems in all contexts, regardless of the programming language. Additionally, many hyperproperties cannot be addressed by language-based approaches because they require reasoning about relationships across an unbounded number of executions. For instance, enforcing opacity requirements, typically expressed as a $\forall\exists$ hyperprop-

erty, requires comparing each observed execution against all possible system behaviors simultaneously. Since the set of possible behaviors is often infinite, this comparison goes beyond the capabilities of standard trace-language frameworks.

Logic-based While there is extensive work on logic-based enforcement of standard safety properties [8, 24, 25, 33, 45, 63], To the best of our knowledge, [20] is the only on enforcement of hyperproperties. However, [20]. It is restricted to: (1) black-box systems, which as we showed earlier cannot always ensure precision; (2) the universal fragment of HyperLTL [20], and (3) the following classes of input systems: parallel, with a finite number of traces progressing synchronously; and sequential with terminated history where only one execution progresses and all executions observed before are terminated.

Gray-box enforcement Lower and upper bounds for the class of policies that are enforceable by black-box mechanisms are studied in [43]. They show that any policy that is black-box enforceable must be hypersafety. With a gray-box approach, we are able to enforce policies outside of the hypersafety class. In [11], Ligatti et al. study security automata in non-uniform runtime contexts. A non-uniform context is a restriction on the set of possible executions an edit automaton may encounter while running.

8 Conclusion and future work

In this paper, we focused on RE of hyperproperties, expressed by nondeterministic finite-word hyperautomata (NFH), as a means to ensure the continued satisfaction of information-flow security properties. Since our approach is logic-based, it is transferable and general, and exhibits low-overhead at run time. We characterized different types of runtime enforcers, namely, black-, gray-, and white-box for hyperproperties, where the enforcement monitor has null, some, or complete information about the implementation of the system under scrutiny. Our approach is based on a sound and precise reduction of the enforcement problem to controller synthesis and subsequently to QBF solving. We demonstrated the power, low-overhead, and generality of our technique by conducting case studies on enforcing information-flow control such as non-interference and declassification on JavaScript code as well as privacy enforcement for communication using obfuscation.

As for future work, the first important step is to extend our work to automata extended with expressions over satisfiability modulo theories (SMT) to cover more programs with infinite behavior. Another important extension is to concurrent programs that make no assumption about the behavior of the scheduler. Generating runtime probabilistic enforcers is also intriguing in the context of designing safety nets for AI-enabled systems.

9 Supplementary information

The implementation of this paper (as a proof-of-concept tool), including code w.r.t. model translation using ExpoSE, QBF encoding use home-grown python code, and sources of all cases presented in Tab. 4, are public available on GitHub[4] as a preliminary artifact for enforcing hyperproperties.

[4] Available at: https://github.com/hyperenforce/artifact

Acknowledgements This project was funded in part by the Austrian Science Fund (FWF) SFB project SpyCoDe F8502, Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018] (ProbInG) and WWTF project ICT22-023 (TAIGER), National Science Foundation (NSF) CPS Award 1837680, NSF award ECCS-2144416 and NSF SaTC Award 2245114.

Author Contributions All authors contributed to the conceptualization and writing of the main manuscript, with Tzu-Han Hsu (TH), Ana Oliveira da Costa (AOC), and Andrew Wintenberg (AW) making equal contributions. AOC, BB, and AW led the development of the theoretical framework, while TH and AW designed the tool and conducted the experiments. Ezio Bartocci provided valuable oversight and reviewed the manuscript.

Funding Open access funding provided by Institute of Science and Technology (IST Austria).

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests This project was funded in part by the Austrian Science Fund (FWF) SFB project SpyCoDe F8502, Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018] (ProbInG) and WWTF project ICT22-023 (TAIGER), NSF CPS Award 1837680, NSF award ECCS-2144416 and NSF SaTC Award 2245114.

References

1. Algehed, M., Flanagan, C.: Transparent IFC enforcement: Possibility and (in)efficiency results. In: 2020 IEEE 33rd Computer Security Foundations Symposium (CSF), pp. 65–78 (2020). doi:10.1109/CSF49147.2020.00013
2. Alur, R., Cerný, P., Zdancewic, S.: Preserving secrecy under refinement. In: Proceedings of the 33rd Automata, Languages and Programming, International Colloquium (ICALP), pp. 107–118 (2006)
3. Askarov, A., Hunt, S., Sabelfeld, A., Sands, D.: Termination-insensitive noninterference leaks more than just a bit. In: Proceedings of 13th European Symposium on Research in Computer Security Computer Security (ESORICS), pp. 333–348 (2008)
4. Askarov, A., Sabelfeld, A.: Gradual release: Unifying declassification, encryption and key release policies. In: 2007 IEEE Symposium on Security and Privacy (SP'07), pp. 207–221 (2007). IEEE
5. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: 2009 22nd IEEE Computer Security Foundations Symposium, pp. 43–59 (2009)
6. Askarov, A., Chong, S., Mantel, H.: Hybrid monitors for concurrent noninterference. In: 2015 IEEE 28th Computer Security Foundations Symposium, pp. 137–151 (2015)
7. Austin, T.H., Flanagan, C.: Multiple facets for dynamic information flow. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 165–178 (2012). https://doi.org/10.1145/2103656.2103677
8. Austin, T.H., Flanagan, C.: Efficient purely-dynamic information flow analysis. In: ACM Transactions on Programming Languages and Systems, pp. 113–124 (2009)
9. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy, pp. 141–153. IEEE Computer Society, USA (2009). https://doi.org/10.1109/SP.2009.18

10. Bailleux, O., Boufkhad, Y.: Efficient cnf encoding of boolean cardinality constraints. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003, pp. 108–122. Springer, Berlin, Heidelberg (2003)
11. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Proceedings of the Workshop on Foundations of Computer Security (FCS'02), Copenhagen, Denmark (2002)
12. Bindschaedler, V., Shokri, R., Gunter, C.A.: Plausible deniability for privacy-preserving data synthesis **10**(5), 481–492 https://doi.org/10.14778/3055540.3055542
13. Bielova, N., Rezk, T.: Spot the difference: Secure multi-execution and multiple facets. In: Askoxylakis, I., Ioannidis, S., Katsikas, S., Meadows, C. (eds.) 21st European Symposium on Research in Computer Security (ESORICS), pp. 501–519 (2016). https://doi.org/10.1007/978-3-319-45744-4_25
14. Bhuiyan, M.H.M., Parthasarathy, A.S., Vasilakis, N., Pradel, M., Staicu, C.-A.: Secbench.js: An executable security benchmark suite for server-side javascript. In: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), pp. 1059–1070 (2023). https://doi.org/10.1109/ICSE48619.2023.00096
15. Bonakdarpour, B., Sheinvald, S.: Finite-word hyperlanguages. In: Proceedings of the 15th International Conference on Language and Automata Theory and Applications (LATA), pp. 173–186 (2021)
16. Bonakdarpour, B., Finkbeiner, B.: Controller synthesis for hyperproperties. In: Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF), pp. 366–379 (2020)
17. Bugliesi, M., Calzavara, S., Focardi, R.: Formal methods for web security. Journal of Logical and Algebraic Methods in Programming **87**, 110–126 (2017)
18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. Journal of Computer Security **18**(6), 1157–1210 (2010)
19. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Proceedings of the 3rd Conference on Principles of Security and Trust POST, pp. 265–284 (2014)
20. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J., Schillo, Y.: Runtime enforcement of hyperproperties. In: Proceedings of the 19th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 283–299 (2021)
21. De Groef, W., Devriese, D., Nikiforakis, N., Piessens, F.: Flowfox: A web browser with flexible and precise information flow control. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. CCS '12 (2012)
22. D, D., Piessens, F.: Noninterference through secure multi-execution. In: Proceedings of the 31st IEEE Symposium on Security and Privacy, S &P, pp. 109–124 (2010)
23. Dershowitz, N., Hanna, Z., Katz, J.: Bounded model checking with QBF. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing, pp. 408–414. Springer, Berlin, Heidelberg (2005)
24. Falcone, Y., Mounier, L., Fernandez, J.-C., Richier, J.-U.: Runtime enforcement monitors: composition, synthesis, and enforcement abilities. Formal Methods in System Design **38**(3), 223–262 (2011)
25. Falcone, Y., Fernandez, J., Mounier, L.: What can you verify and enforce at runtime? Int. J. Softw. Tools Technol. Transfer (STTT) **14**(3), 349–382 (2012)
26. Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05), pp. 321–330 (2005). https://doi.org/10.1109/LICS.2005.53
27. Guernic, G.L.: Automaton-based confidentiality monitoring of concurrent programs. In: 20th IEEE Computer Security Foundations Symposium (CSF'07), pp. 218–232 (2007)
28. Hedin, D., Birgisson, A., Bello, L., Sabelfeld, A.: Jsflow: Tracking information flow in javascript and its apis. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing. SAC '14 (2014)
29. Hedin, D., Bello, L., Sabelfeld, A.: Value-sensitive hybrid information flow control for a javascript-like language. In: 2015 IEEE 28th Computer Security Foundations Symposium, pp. 351–365 (2015)
30. Hsu, T.-H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: Proceedings of the 27th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS) (2021). To appear
31. Jang, D., Jhala, R., Lerner, S., Shacham, H.: An empirical study of privacy-violating information flows in javascript web applications. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, pp. 270–283 (2010)
32. Khoury, R., Tawbi, N.: Which security policies are enforceable by runtime monitors? A survey. Computer Science Review **6**(1), 27–45 (2012). https://doi.org/10.1016/j.cosrev.2012.01.001
33. Könighofer, B., Alshiekh, M., Bloem, R., Humphrey, L.R., Könighofer, R., Topcu, U., Wang, C.: Shield synthesis. Formal Methods in System Design **51**(2), 332–361 (2017)
34. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. Int. J. Inf. Secur. **4**(1), 2–16 (2005)

35. Linus, R.: Social Media Fingerprint. https://robinlinus.github.io/socialmedia-leak/ (2023 (accessed August 3, 2023))
36. Loring, B., Mitchell, D., Kinder, J.: Expose: practical symbolic execution of standalone javascript. In: Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, pp. 196–199 (2017)
37. Martinell, F., Matteucci, I.: Through modeling to synthesis of security automata. Proceedings of the Second International Workshop on Security and Trust Management (STM 2006) **179**, 31–46 (2007). https://doi.org/10.1016/j.entcs.2006.08.029
38. McCall, M., Bichhawat, A., Jia, L.: Compositional information flow monitoring for reactive programs. In: 2022 IEEE 7th European Symposium on Security and Privacy (EuroS &P), pp. 467–486 (2022)
39. McCullough, D.: Noninterference and the composability of security properties. In: Proceedings of the 1988 IEEE Symposium on Security and Privacy, pp. 177–186 (1988)
40. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: Proc. IEEE Symposium on Security and Privacy, pp. 79–93 (1994)
41. Moore, S., Chong, S.: Static analysis for efficient hybrid information-flow control. In: 2011 IEEE 24th Computer Security Foundations Symposium, pp. 146–160 (2011)
42. Ngo, M., Bielova, N., Flanagan, C., Rezk, T., Russo, A., Schmitz, T.: A better facet of dynamic information flow control. In: Companion Proceedings of The Web Conference 2018, pp. 731–739 (2018). https://doi.org/10.1145/3184558.3185979
43. Ngo, M., Massacci, F., Milushev, D., Piessens, F.: Runtime enforcement of security policies on black box reactive programs. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 43–54. Association for Computing Machinery (2015). https://doi.org/10.1145/2676726.2676978
44. O'Halloran, C.: A calculus of information flow. Proceedings of the European Symposium on Research in Computer Security (1990)
45. Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., Marchand, H.: Predictive runtime enforcement. Formal Methods in System Design **51**(1), 154–199 (2017)
46. Russo, A., Sabelfeld, A.: Dynamic vs. static flow-sensitive security analysis. In: 2010 23rd IEEE Computer Security Foundations Symposium, pp. 186–199 (2010)
47. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2006)
48. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: 18th IEEE Computer Security Foundations Workshop (CSFW'05), pp. 255–269 (2005). https://doi.org/10.1109/CSFW.2005.15
49. Sabelfeld, A., Russo, A.: From dynamic to static and back: Riding the roller coaster of information-flow control research. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics (2010)
50. Sahai, S., Subramanyan, P., Sinha, R., Lahiri, S.K., Wang, C.: Verification of quantitative hyperproperties using trace enumeration relations. In: Computer Aided Verification, pp. 201–224. Springer, Cham (2020)
51. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3**, 30–50 (2000)
52. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) Software Security - Theories and Systems, pp. 174–191. Springer (2004)
53. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 488–498 (2013)
54. Shroff, P., Smith, S., Thober, M.: Dynamic dependency monitoring to secure information flow. In: 20th IEEE Computer Security Foundations Symposium (CSF'07), pp. 203–217 (2007)
55. Staicu, C.-A., Schoepe, D., Balliu, M., Pradel, M., Sabelfeld, A.: An empirical study of information flows in real-world javascript. In: Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, pp. 45–59 (2019)
56. Stefan, D., Russo, A., Mitchell, J.C., Mazières, D.: Flexible dynamic information flow control in haskell. In: Proceedings of the 4th ACM Symposium on Haskell. Haskell '11 (2011)
57. Tong, Y., Ma, Z., Li, Z., Seatzu, C., Giua, A.: Supervisory enforcement of current-state opacity with uncomparable observations. In: 2016 13th International Workshop on Discrete Event Systems (WODES), pp. 313–318 (2016). https://doi.org/10.1109/WODES.2016.7497865
58. Tentrup, L.: CAQE and QuAbS: Abstraction based QBF solvers. Journal of Satisfiability Boolean Modeling and Computation **11**(1), 155–210 (2019)
59. Vanhoef, M., De Groef, W., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: 2014 IEEE 27th Computer Security Foundations Symposium, pp. 293–307 (2014). IEEE

60. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Inf. Process. Lett. **68**(2), 63–69 (1998). https://doi.org/10.1016/S0020-0190(98)00144-6
61. Wintenberg, A., Blischke, M., Lafortune, S., Ozay, N.: A dynamic obfuscation framework for security and utility. In: 2022 ACM/IEEE 13th International Conference on Cyber-Physical Systems (ICCPS), pp. 236–246 (2022). https://doi.org/10.1109/ICCPS54341.2022.00028
62. Wu, Y.-C., Raman, V., Rawlings, B.C., Lafortune, S., Seshia, S.A.: Synthesis of obfuscation policies to ensure privacy and utility. J. Autom. Reason. **60**(1), 107–131 (2017). https://doi.org/10.1007/s10817-017-9420-x
63. Wu, M., Zeng, H., Wang, C., Yu, H.: Safety guard: Runtime enforcement for safety-critical cyber-physical systems. In: Proceedings of the Design Automation Conference, pp. 84–1846 (2017)
64. Zakinthinos, A., Lee, E.S.: A general theory of security properties. In: Proceedings. 1997 IEEE Symposium on Security and Privacy, pp. 94–102 (1997). https://doi.org/10.1109/SECPRI.1997.601322
65. Zanarini, D., Jaskelioff, M., Russo, A.: Precise enforcement of confidentiality for reactive systems. In: 2013 IEEE 26th Computer Security Foundations Symposium, pp. 18–32 (2013). https://doi.org/10.1109/CSF.2013.9
66. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW), pp. 29–43 (2003)