Rewriting-Based Runtime Verification for Alternation-Free HyperLTL

Noel Brett, Umair Siddique, and Borzoo Bonakdarpour

Department of Computing and Software
McMaster University, Canada

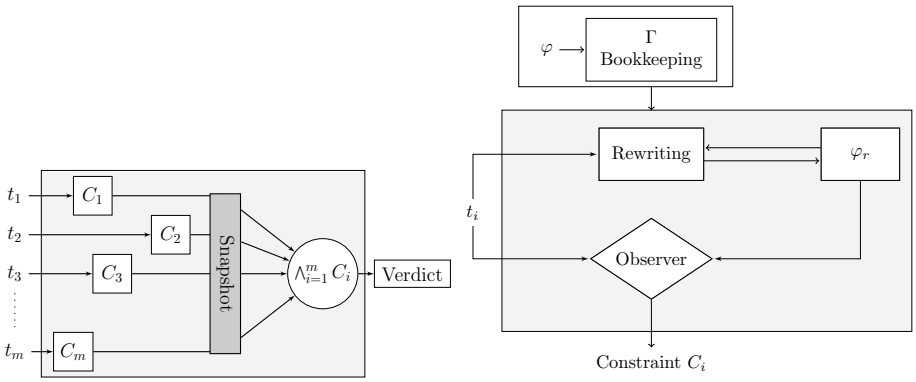
Abstract. Analysis of complex security and privacy policies (e.g., information flow) involves reasoning about multiple execution traces. This stems from the fact that an external observer may gain knowledge about the system through observing and comparing several executions. Monitoring of such policies is in particular challenging because most existing monitoring techniques are limited to the analysis of a single trace at run time. In this paper, we present a rewriting-based technique for runtime verification of the full alternation-free fragment of HyperLTL, a temporal logic for specification of hyperproperties. The distinguishing feature of our proposed technique is its space complexity, which is independent of the number of trace quantifiers in a given HyperLTL formula.

1 Introduction

Dependability and reliability are two crucial aspects of any computing system that deals with *cybersecurity*. This is because even a short transient violation of security or privacy policies may result in leaking private or highly sensitive information, compromising safety, or lead to the interruption of vital public or social services. One approach to gain confidence about the well-being of such a system is to continuously monitor it with respect to a set of formally specified requirements that system should meet at all times. This approach is commonly known as *runtime verification* (RV).

We start with the premise that existing RV techniques cannot monitor a large but vital class of the security and privacy polices, e.g., information flow. Take, for instance, the *non-interference* policy [12], where a low user should not be able to acquire any information about the activities (if any) of the high user by observing independent execution traces. Monitoring this policy would require observing and reasoning about multiple execution traces, whereas existing RV techniques are limited to evaluating only one trace at run time.

In order to specify security and privacy policies, we focus on HyperLTL [8], a temporal logic for expressing *hyperproperties* [9]. A hyperproperty is a set of sets of execution traces. HyperLTL adds explicit and simultaneous quantification over multiple traces to the standard LTL. HyperLTL significantly extends the range of security policies under consideration, including complex information-flow properties like generalized non-interference, declassification, and quantita-



(a) Monitor Overview (b) Constraint Generation for Single Trace

Fig. 1: RV Framework for HyperLTL

tive non-interference. For example, the following is a HyperLTL formula:

$$\varphi = \forall\pi.\forall\pi'.\ a_\pi \ \rightarrow\ \mathbf{F} b_{\pi'}$$

It states that for any pair of traces π and π', if proposition a holds in the initial state of π, then proposition b should eventually hold in trace $\pi^{'}$. To describe the challenges in monitoring HyperLTL specifications, consider formula φ and two traces $t = cde$ and $t' = acddb$. These traces individually (e.g., if π and π' are both instantiated by t), satisfy the formula, but collectively (e.g., if π is instantiated by t and π' by t') do not. If a monitor first observes trace t and then t', it has to somehow remember that b never occurred in t and declare violation as soon as it observes a in the initial state of t'. Thus, a HyperLTL monitor has to be memeoryful; i.e., the monitoring algorithm has to be able to memorize the status of propositions of interest in the past traces to be able to reason about current and future traces.

With this motivation, in this paper, we introduce a novel RV algorithm for monitoring the alternation-free fragment of (i.e., $\forall^*$ and $\exists^*$) HyperLTL (in Section 4, we will argue that alternating formulas cannot be monitored using a runtime technique only). Our algorithm takes as input a formula φ and a finite but unbounded-size set T of finite traces (see Fig. 1(a)). The traces in T can be produced by multiple sequential terminating or concurrent executions of a system under inspection. This means that the traces in T can grow in number and/or length at run time. The algorithm works as follows (see Fig. 1(b)):

- First, given φ, it identifies the propositions and possibly simple Boolean expressions that need bookkeeping using a function Γ.
- Then, for each trace $t_i \in T$, by incorporating the elements returned by Γ, the monitor generates a constraint C_i. This constraint basically encapsulates two things. It

1. encodes what the monitor has observed in t_i with respect to the elements returned by Γ, so it can reason about new incoming traces as well as existing traces growing in length, and
2. rewrites the inner LTL formula in φ using Havelund and Rosu's algorithm [13] and obtains a formula φ_r.

Hence, the resulting constraint C_i encodes the full memory of all relevant things that has occurred in t_i.

– At any point of time, the conjunction $\bigwedge_{i=1}^{m} C_i$ where m is the number of traces being monitored, determines the current RV verdict (see Fig. 1(a)). That is, the result of simplification of the conjunction shows whether φ has been satisfied, violated, or currently impossible to tell (i.e., it can go either way in the future).

Finally, we note that although the number and length of the generated constraints are theoretically unbounded, this can be prevented by making practical assumptions. One example is to incorporate a synchronization mechanism that ensures that the difference in length of traces do not grow over a certain bound. Furthermore, the complexity of our algorithm is detached from the number of trace quantifiers in a given HyperLTL formula.

Organization The rest of the paper is organized as follows. Section 2 presents the syntax and semantics of HyperLTL. In Section 3, we introduce our finite semantics for HyperLTL. Section 4 discusses challenges in monitoring HyperLTL formulas. Subsequently, the components of our RV algorithm are presented in Sections 5 and 6. Related work is discussed in Section 7. Finally, we make concluding remarks and discuss future work in Section 8. Additional examples and all proofs appear in the appendix.

2 Background

Let AP be a finite set of *atomic propositions* and $\Sigma = 2^{AP}$ be the finite *alphabet*. We call each element of Σ a *letter* (or an *event*). Throughout the paper, Σ^ω denotes the set of all infinite sequences (called *traces*) over Σ, and Σ^* denotes the set of all finite traces over Σ. For a trace $t \in \Sigma^\omega$ (or $t \in \Sigma^*$), $t[i]$ denotes the i^{th} element of t, where $i \in \mathbb{Z}_{\geq 0}$. Also, $t[0, i]$ denotes the prefix of t up to and including i, and $t[i, \infty]$ is written to denote the infinite suffix of t beginning with element i. By, $|t|$ we mean the length of (finite or infinite) trace t.

Now, let u be a finite trace and v be a finite or infinite trace. We denote the concatenation of u and v by $\sigma = uv$. Also, $u \leq \sigma$ denotes the fact that u is a prefix of σ. Finally, if U is a set of finite traces and V is a finite or infinite set of traces, then the prefix relation $\leq$ on sets of traces is defined as:

$$U \leq V \;\equiv\; \forall u \in U.\, (\exists v \in V.\, u \leq v)$$

Note that V may contain traces that have no prefix in U.

2.1 HyperLTL

Clarkson and Schneider [9] proposed the notion of *hyperproperties* as a means to express security policies that cannot be expressed by traditional properties. A hyperproperty is a set of sets of execution traces. Thus, a hyperproperty essentially defines a set of systems that respect a policy. HyperLTL [8] is a logic for syntactic representation of hyperproperties. It generalizes LTL by allowing explicit quantification over multiple execution traces simultaneously.

Syntax The set of HyperLTL formulas is inductively defined by the grammar as follows:

$$\varphi ::= \exists \pi.\varphi \mid \forall \pi.\varphi \mid \phi$$
$$\phi ::= a_\pi \mid \neg\phi \mid \phi \vee \phi \mid \phi \,\mathbf{U}\, \phi \mid \mathbf{X}\phi$$

where $a \in AP$ and π is a trace variable from an infinite supply of variables $\mathcal{V}$. Similar to LTL, $\mathbf{U}$ and $\mathbf{X}$ are the 'until' and 'next' operators, respectively. Other standard temporal connectives are defined as syntactic sugar as follows: $\varphi_1 \rightarrow \varphi_2 = \neg\varphi_1 \vee \varphi_2$, $\varphi_1 \wedge \varphi_2 = \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\texttt{true} = a_\pi \vee \neg a_\pi$, $\texttt{false} = \neg\texttt{true}$, $\mathbf{F}\phi = \texttt{true}\,\mathbf{U}\,\phi$, and $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$. Quantified formulas $\exists\pi$ and $\forall\pi$ are read as 'along some trace π' and 'along all traces π', respectively.

Semantics A formula φ in HyperLTL satisfied by a set of traces T is written as $\Pi \models_T \varphi$, where trace assignment $\Pi : \mathcal{V} \rightarrow \Sigma^\omega$ is a partial function mapping trace variables to traces. $\Pi[\pi \rightarrow t]$ denotes the same function as Π, except that π is mapped to trace t. The validity judgment for HyperLTL is defined as follows:

$$\begin{array}{lll}
\Pi \models_T \exists\pi.\varphi & \text{iff} & \exists t \in T.\, \Pi[\pi \rightarrow t] \models_T \varphi \\
\Pi \models_T \forall\pi.\varphi & \text{iff} & \forall t \in T.\, \Pi[\pi \rightarrow t] \models_T \varphi \\
\Pi \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\
\Pi \models_T \neg\phi & \text{iff} & \Pi \not\models_T \phi \\
\Pi \models_T \phi_1 \vee \phi_2 & \text{iff} & (\Pi \models_T \phi_1) \vee (\Pi \models_T \phi_2) \\
\Pi \models_T \mathbf{X}\phi & \text{iff} & \Pi[1,\infty] \models_T \phi \\
\Pi \models_T \phi_1 \,\mathbf{U}\, \phi_2 & \text{iff} & \exists i \geq 0.\, (\Pi[i,\infty] \models_T \phi_2 \wedge \\
 & & \forall j \in [0,i).\, \Pi[j,\infty] \models_T \phi_1)
\end{array}$$

where the trace assignment suffix $\Pi[i,\infty]$ denotes the trace assignment $\Pi' = \Pi(\pi)[i,\infty]$ for all π. If $\Pi \models_T \phi$ holds for the empty assignment Π, then T satisfies ϕ.

Example Non-interference (NI) security policy requires any pair of traces with the same initial low observation to remain indistinguishable for low users, yet low inputs will be unaltered, irrespective of the the high inputs. This policy can be specified by the following HyperLTL formula:

$$\forall\pi.\forall\pi'.(\mathbf{G}\lambda_H(\pi') \wedge \mathbf{G}\neg(\bigwedge_{a\in H} a_\pi \leftrightarrow a_{\pi'})) \;\rightarrow\; \mathbf{G}(\bigwedge_{a\in L} a_\pi \leftrightarrow a_{\pi'})$$

Where $\mathbf{G}\lambda_H(\pi')$ denotes all the high variables in π' that hold the value λ, and H and L are the high and low variables in their respected security levels.

3 Finite Semantics for HyperLTL

In this section, we present our finite semantics for HyperLTL, inspired by the finite semantics of LTL [15]. For a finite trace t, let $t[i,j]$ denote the subtrace of t from position i up to and including position j:

$$t[i,j] \;=\; \begin{cases} \epsilon & \text{if} \quad i > |t| \\ t[i, \min(j, |t|{-}1)] & \text{otherwise} \end{cases}$$

where ϵ is the empty trace. We let $t[i,..]$ denote $t[i,|t|{-}1]$.

Let trace assignment $\Pi_F : \mathcal{V} \to \Sigma^*$ be a partial function mapping trace variables to *finite* traces. Similar to the infinite semantics, $\Pi_F[\pi \to t]$ denotes the same function as Π_F, except that π is mapped to finite trace t. We consider two truth values for the finite semantics: $\top$ and $\bot$. To distinguish finite from infinite semantics, we use $[\Pi_F \models_T \varphi]$ to denote the valuation of HyperLTL formula φ for a set T of finite traces. The finite semantics for Boolean operators '$\vee$' and '$\neg$' as well as for the trace quantifiers '$\forall$' and '$\exists$' are identical to those of infinite semantics. We define the finite semantics of HyperLTL for temporal operators as follows:

$$\big[\Pi_F \models_T \forall/\exists\pi.\varphi\big] = \begin{cases} \top & \text{if} \quad \forall/\exists t \in T.[\Pi_F[\pi \to t] \models_T \varphi] = \top \\ \bot & \text{otherwise} \end{cases}$$

$$\big[\Pi_F \models_T \phi_1 \vee \phi_2\big] = \begin{cases} \bot & \text{if} \quad [\Pi_F \models_T \phi_1] = \bot \wedge [\Pi_F \models_T \phi_2] = \bot \\ \top & \text{otherwise} \end{cases}$$

$$\big[\Pi_F \models_T \neg\phi\big] = \begin{cases} \bot & \text{if} \quad [\Pi_F \models_T \phi] = \top \\ \top & \text{otherwise} \end{cases}$$

$$\big[\Pi_F \models_T \mathbf{X}\,\varphi\big] = \begin{cases} [\Pi_F[1,..] \models_T \varphi] & \text{if} \quad \Pi[1,..] \neq \epsilon \\ \bot & \text{otherwise} \end{cases}$$

$$\big[\Pi_F \models_T \bar{\mathbf{X}}\,\varphi\big] = \begin{cases} [\Pi_F[1,..] \models_T \varphi] & \text{if} \quad \Pi[1,..] \neq \epsilon \\ \top & \text{otherwise} \end{cases}$$

$$\big[\Pi_F \models_T \varphi_1\,\mathbf{U}\,\varphi_2\big] = \begin{cases} \top & \text{if} \quad \exists i \geq 0 : \Pi_F[i,..] \neq \epsilon \wedge [\Pi_F[i,..] \models_T \varphi_2] = \top \wedge \\ & \qquad \forall j \in [0,i) : [\Pi_F[j,..] \models_T \varphi_1] = \top \\ \bot & \text{otherwise} \end{cases}$$

where $\bar{X}$ denotes the 'weak next' operator.

Example Consider formula $\phi = \forall\pi_1.\forall\pi_2.\ a_{\pi_1}\ \mathbf{U}\ b_{\pi_2}$ and $T = \{t_1 = aaab, t_2 = aab, t_3 = aab\}$. Although traces t_1, t_2, and t_3 individually satisfy the formula ϕ,

we have $[\Pi_F \models_T \varphi] = \bot$, as there does not exist a position, where each pair of traces agree on the position of b. Now consider formula $\varphi' = \forall\pi_1.\forall\pi_2.\mathbf{F}a_{\pi_1} \wedge \mathbf{F}b_{\pi_2}$ and let $T' = \{**a*b,\ \ *b**a\}$. We have $[\Pi_F \models_{T'} \varphi'] = \top$.

4 Challenges in Monitoring HyperLTL Formulas

Let us assume we are to monitor a finite but unbounded-size set T of finite traces with respect to a HyperLTL formula φ. The traces in T can be produced by multiple sequential terminating or concurrent executions of a system under inspection. This means that traces in T can grow in number and/or length at run time. Unlike conventional runtime monitoring techniques, where verification decision only depends upon one current execution, monitoring T for φ may depend on the past, future, or concurrent evolution of the traces in T. Thus, a monitor for φ needs to bookkeep the occurrence (and even not occurrence) of certain events to be able to reason about φ at run time. In the following, we outline a set of challenges which need to be addressed in order to develop a monitoring algorithm.

Alternating formulas. Let $\varphi = \forall\pi.\exists\pi'.\psi$. Verifying this formula requires us to show that *for all* traces in T, there exists a trace that satisfies ψ. However, since the number of traces in T may grow, a runtime monitor can never prove or disprove φ. This argument holds in general for $\forall^*\exists^*$ and $\exists^*\forall^*$ formulas. This is the main reason that in the remainder of this paper, we will only focus on the alternation-free fragment of HyperLTL. Observe that for $\forall^*$ (respectively, $\exists^*$) formulas, it is possible to compute verdict $\bot$ (respectively, $\top$) at run time.

Inter-trace dependencies. Reasoning about φ by observing individual traces in T is clearly not sufficient. Progression through traces in T requires to keep information about the past or concurrent traces in T. One root cause of this is due to the existence of a disjunction in φ involving two distinct trace variables. For example, let $\phi = \forall\pi_1.\forall\pi_2.\ a_{\pi_1} \rightarrow \mathbf{F}b_{\pi_2}$. Now, consider two traces $t_1 = dcf$ and $t_2 = aeb$, where $AP = \{a, b, c, d, e, f\}$. Note that traces t_1 and t_2, individually satisfy φ, but they collectively violate φ, as event b does not occur in t_1.

Time of occurrence of events. Reasoning about some formulas requires bookkeeping the time of occurrence of some propositions in each trace. For example, consider formula $\varphi_1 = \forall\pi_1.\forall\pi_2.\ a_{\pi_1}\,\mathbf{U}\,b_{\pi_2}$ and traces $t_1 = aab$, $t_2 = ab$, and $t_3 = aaaab$. Although, each trace individually satisfies the formula, any pair of them violates the formula, as event b occurs at different times. This can become even more complex when the occurrence of some propositions needs to agree across multiple traces and multiple times. An example of such a formula is $\varphi_2 = \forall\pi_1.\forall\pi_2.\forall\pi_3.\ (a_{\pi_1}\ \mathbf{U}\ b_{\pi_2})\ \mathbf{U}\ c_{\pi_3}$, where the first occurrence of c and every occurrence of b need to be agreed across all traces in T. For example, for traces $t_1 = (ab)a(ac)(ac)b$, $t_2 = (ab)a(ac)(a)(b)$, and $t_3 = a(ac)(ac)b$, traces t_1 and t_2 agree on times of occurrence of b and c, but trace t_3 violates this agreement, thus violating formula φ_2. Yet other examples are formula

$\varphi_3 = \forall\pi_1.\forall\pi_2.\ \mathbf{G}(a_{\pi_1} \rightarrow a_{\pi_2})$ (which requires all traces to agree on each occurrence of a) and the non-interference formula discussed in Section 2.

5 Identifying Propositions of Interest

The challenges and examples outlined in Section 4 suggest that monitoring a HyperLTL formula requires the identification of propositions which shape the trace agreement to be followed amongst distinct traces. We call this process *bookkeeping*, denote $\mathcal{BK}$ as a set of all elements which require bookkeeping, and Γ as the function that computes $\mathcal{BK}$.

We note that only the structure of the HyperLTL formula contributes to the elements of $\mathcal{BK}$. More precisely, the 'until' operator is the main contributor to $\mathcal{BK}$, as its semantics (in particular, the existential quantifier) may delineate the existence of an index for satisfaction of some propositions across multiple traces. Moreover, we may need to bookkeep Boolean expressions (and not just atomic propositions). We may prefix elements of $\mathcal{BK}$ by either # or $\mathbf{X}$. Prefixing an element by # means that only the first occurrence of the element needs to be bookkept. Prefixing by $\mathbf{X}$ means that bookkeeping starts from the next state.

Examples In formula $\forall\pi_1.\forall\pi_2.\forall\pi_3.(a_{\pi_1}\,\mathbf{U} b_{\pi_2})\,\mathbf{U} c_{\pi_3}$, we will have $\mathcal{BK} = \{b, \#c\}$, meaning every occurrence of b and only the first occurrence of c should be memorized. For formula $\forall\pi_1.\forall\pi_2.a_{\pi_1}\,\mathbf{U}\,(b_{\pi_2} \vee c_{\pi_2})$, we have $\mathcal{BK} = \{\#(b \vee c)\}$. However, for formula $\forall\pi_1.\forall\pi_2.\forall\pi_3.a_{\pi_1}\,\mathbf{U}\,(b_{\pi_2} \vee c_{\pi_3})$, we have $\mathcal{BK} = \{\#b, \#c\}$. Finally, for formula $\forall\pi.\forall\pi'.\mathbf{X}(a_\pi\,\mathbf{U} b_{\pi'})$, we will have $\mathcal{BK} = \{\mathbf{X}\#b\}$.

Our bookkeeping recursive function Γ takes as input a HyperLTL formula, a set of trace variables $\mathcal{V}$ (initially empty), and a Boolean value (initially *false*), and it returns as output the set $\mathcal{BK}$, defined in Fig. 2. The function works as follows. The first three cases are straightforward, as a HyperLTL formula involving only a proposition requires bookkeeping if it is under the scope of an 'until' operator, whereas operators $\neg$ and $\mathbf{X}$ allow the recursive application of Γ function to the formula ϕ. The symbol $\odot$ denotes the application of unary operators ($\neg$, # and $\mathbf{X}$) to the elements of set $\mathcal{BK}$ (e.g., $\neg \odot \{a, b\} = \{\neg a, \neg b\}$).

The next case $\phi_1 \mathbf{U} \phi_2$, we require further matching on the structure of both ϕ_1 and ϕ_2, as follows:

- **(Case 1: Both operands are propositions)** In this case, Γ returns $\{\#b\}$ if π and π' are bound by different quantifiers or removing π' from $\mathcal{V}$ does not result in an empty set. Otherwise, Γ returns the empty set. For example, consider two formulas $\forall\pi_1.a_{\pi_1}\,\mathbf{U}\,b_{\pi_1}$ and $\forall\pi_1.\forall\pi_2.a_{\pi_1}\,\mathbf{U}\,b_{\pi_2}$. The first formula does not require any trace agreement whereas the second does require a trace agreement due to the scope of the trace quantifiers.
- **(Case 2: Only the left operand is a proposition)** In this case, we store the trace variable associated with a in set $\mathcal{V}$ and invoke Γ recursively to formula ϕ_2. We also set the value of Boolean variable k to *true* which indicates that the original formula ϕ includes an 'until' operator. For example,

$$\Gamma(a_\pi, \mathcal{V}, k) = \begin{cases} \{\#a\} & \text{if} \quad (k = true \wedge \mathcal{V} - \{\pi'\} \neq \emptyset) \\ \{\} & \text{otherwise} \end{cases}$$

$$\Gamma(\mathbf{X}\phi, \mathcal{V}, k) = \mathbf{X} \odot \Gamma(\phi, \mathcal{V}, k)$$
$$\Gamma(\neg\phi, \mathcal{V}, k) = \neg \odot \Gamma(\phi, \mathcal{V}, k)$$

$$\Gamma(\phi_1 \,\mathbf{U}\, \phi_2, \mathcal{V}, k) =$$
$$\textbf{match } \phi_1 \quad \phi_2 \quad \textbf{with}$$
$$\mid \; a_\pi \quad b'_\pi \quad \rightarrow \begin{cases} \{\#b\} & \text{if} \quad (\mathcal{V} - \{\pi'\} \neq \emptyset \vee \pi \neq \pi') \\ \{\} & \text{otherwise} \end{cases}$$
$$\mid \; a_\pi \quad - \quad \rightarrow \; \Gamma(\phi_2, \mathcal{V} \cup \{\pi\}, k := true)$$
$$\mid \; - \quad - \quad \rightarrow \begin{cases} \Gamma(\phi_2, \mathcal{V} \cup trace\_vars(\phi_1), k := true) & \text{if } \phi_1 \notin \textsc{HyperLTL}_1(\mathbf{U}) \\ \#^{-1} \odot \Gamma(\phi_1, \mathcal{V}, k := true) \cup & \\ \# \odot \Gamma(\phi_2, \mathcal{V} \cup trace\_vars(\phi_1), k := true) & \text{otherwise} \end{cases}$$

$$\Gamma(\phi_1 \vee \phi_2, \mathcal{V}, k) =$$
$$\textbf{match } \phi_1 \quad \phi_2 \quad \textbf{with}$$
$$\mid \; a_\pi \quad b'_\pi \quad \rightarrow \begin{cases} \{a \vee b\} & \text{if} \quad k = true \wedge \pi = \pi' \\ \{a\} \cap \{b\} & \text{if} \quad k = true \wedge \pi \neq \pi' \\ \{\} & \text{otherwise} \end{cases}$$
$$\mid \; a_\pi \quad - \quad \rightarrow \begin{cases} \{a\} \cup \Gamma(\phi_2, \mathcal{V}, k) & \text{if} \quad k = true \\ \Gamma(\phi_2, \mathcal{V}, k) & \text{otherwise} \end{cases}$$
$$\mid \; - \quad b'_\pi \quad \rightarrow \begin{cases} \Gamma(\phi_1, \mathcal{V}, k) \cup \{b\} & \text{if} \quad k = true \\ \Gamma(\phi_1, \mathcal{V}, k) & \text{otherwise} \end{cases}$$
$$\mid \; - \quad - \quad \rightarrow \Gamma(\phi_1, \mathcal{V}, k) \cup \Gamma(\phi_2, \mathcal{V}, k)$$

Fig. 2: Bookkeeping Function Γ

for formula $\forall\pi.a_\pi \,\mathbf{U}\, (b_\pi \mathbf{U} c_\pi)$, recursing through Γ will result in an empty set since there were no variations in the trace variables, whereas for formula $\forall\pi_1.\forall\pi_2.a_{\pi_1} \,\mathbf{U}\, (b_{\pi_1} \,\mathbf{U}\, c_{\pi_2})$, the Γ function will simply return $\{\#c\}$.

– **(Case 3: None of the operands are propositions)** In this case, we recurse through ϕ_1 only if it contains an 'until' operator, where $trace\_vars(\phi)$ denotes the set of trace variables found in ϕ. Furthermore, we recurse through ϕ_2 and indicate that any elements produced need to be tracked only once (i.e., their first occurrence). Moreover, we prefix the recursion of Γ on ϕ_1 by symbol $\#^{-1}$, which helps to remove the prefix # for elements which require tracking more than once. The result will consist of the union of both produced sets. For example, for formula $\forall\pi_1.\forall\pi_2.\forall\pi_3.\forall\pi_4.(a_{\pi_1} \mathbf{U} b_{\pi_2}) \mathbf{U} (c_{\pi_3} \mathbf{U} d_{\pi_4})$, we have $\mathcal{BK} = \{b, \#d\}$. Note that expressions $\#^{-1}\#a$ and $\#\#b$ are equivalent to a and $\#b$, respectively.

The last inductive case includes an 'or' ($\vee$), which also requires further matching on the structure of formulas ϕ_1 and ϕ_2. Here, we consider the condition of k, which reflects the case when $\phi_1 \vee \phi_2$ is under the scope of an 'until' operator. For example, formula $\forall\pi_1.\forall\pi_2.a_{\pi_1}\,\mathbf{U}\,(b_{\pi_2} \vee c_{\pi_2})$. The application of Γ function will result in $\Gamma(b_{\pi_2} \vee c_{\pi_2}, \mathcal{V}, k := \mathit{true})$, which further results in $\{\#(b \vee c)\}$. On the contrary, the case of formula $\forall\pi_1.\forall\pi_2.\forall\pi_3.a_{\pi_1}\,\mathbf{U}\,(b_{\pi_2} \vee c_{\pi_3})$, the Γ function will return $\{\#b, \#c\}$ due to the disparity of trace variables.

Theorem 1 (Soundness and optimality of Γ function). *Given a HyperLTL formula φ and assuming we have set T such that $[\Pi_F \models_T \varphi] = \top$ then*

- Γ *function returns all the propositions required for bookkeeping.*
- *Given the set $\mathcal{BK}$, every element $k \in \mathcal{BK}$ is included in some trace agreement described by φ.*

6 Monitoring Algorithm

6.1 Algorithm Sketch

Given an alternation-free HyperLTL formula φ of the form $\forall^*$, our algorithm consists of the following elements:

1. *Monitor:* In order to monitor φ, we begin by intaking an event for a particular trace and begin to generate the constraints. At any point of time, we can take a snapshot of our system and utilize our satisfaction function `SAT` to find the RV verdict (see Fig. 1(a)).
2. *Constraint Handler:* Next, we manipulate φ according to its structure. Disjunctions are divided and treated separately to detect which half prompted the satisfaction. Each sub-formula of the disjunction is then subject to `ConstraintRewriting`. Temporal formulas without disjunction do not undergo any manipulation before being sent to `ConstraintRewriting`.
3. *Constraint Rewriting:* Initially, φ is stripped of its quantifiers. This allows for rewriting using the technique in [22] to evaluate the altered formula φ_r. The events are examined against the propositions or Boolean expressions in $\mathcal{BK}$ and the satisfaction of φ_r to generate the corresponding constraints.
4. *Satisfaction of Function* `SAT`*:* On each invocation of the `SAT` function, we compute the conjunction of all the constraints collectively. If `SAT` returns `false`, then φ is violated. Otherwise, the constraints are further checked for possible refinement by checking the membership of other generated constraints.

Observe that a formula of the form $\forall^*$ cannot be evaluated to $\top$. This would require the full set of all possible system traces, which is not possible at run time. We note that monitoring a formula of the form $\exists^*$ can be achieved by simply monitoring its negation which would be of the form $\forall^*$.

6.2 Algorithm Details

We utilize the following HyperLTL formula as a running example to demonstrate the steps of our proposed algorithm.

$$\forall\pi_1.\forall\pi_2.\forall\pi_3.\forall\pi_4.\ ((a_{\pi_1} \vee b_{\pi_2})\,\mathbf{U}\,c_{\pi_3}) \vee d_{\pi_4}$$

where $AP = \{a, b, c, d\}$. We now describe the algorithm in detail which leads to the overview of Fig. 1.

Algorithm 1 (HyperLTL Monitor). This is our main monitoring algorithm which is comprised of a while loop. We continue to iterate as long as new events associated with a trace come in and until we find a violation. On Lines 2-3, we check for a new trace and then add it to our set of traces M. Given that the incoming event is associated with some trace t_j, at Line 4, we call `ConstraintsHandler` for t_j, which returns constraint C_j. Lines 5-6 deal with the process of taking a snapshot of our system to determine the RV verdict using function `SAT`. Finally, if the returned value from function `SAT` is `false` (Lines 7-9), then we have found a violation and return $\perp$ (Line 10). Otherwise, we continue to iterate through the while loop.

Algorithm 2 (Constraint Handler). In this algorithm, we treat the given HyperLTL formula according to its structure. The algorithm is recursively applied to the given formula based on different cases. The first block of the algorithm (Lines 1-10) handles the case ($\varphi = \phi_1 \vee \phi_2$), where the given (sub-)formula is a disjunction. In particular, we call `ConstraintsHandler` function for both ϕ_1 and ϕ_2 (Lines 2-3). We also need to pass the information about the elements of $\mathcal{BK}$ which are associated with ϕ_1 and ϕ_2 (as given by $\mathcal{BK}_{\phi_i}$). In our running example, we have $\phi_1 = ((a_{\pi_1} \vee b_{\pi_2})\,\mathbf{U}\,c_{\pi_3})$ and $\phi_2 = d_{\pi_4}$. In case both values from previous steps are `false`, then we have found a violation and the algorithm returns `false` (Lines 4-5). On the other hand, if one of the values from Lines 2 and 3 is a constraint, then we return the corresponding constraint (Lines 6-9). Moreover, if both values have generated constraints, we return them both (Lines 10) meaning that any one of them can influence the verdict in future.

Next block in the algorithm (Lines 12-22) handles the case when the input formula contains an 'until' operators with a disjunction on the left operand with a disparity in corresponding trace quantifiers. We invoke `ConstraintsHandler` function for both operands of '$\vee$'; i.e., ϕ_L and ϕ_R (Lines 13-14). In our running example, $\phi_1 = ((a_{\pi_1} \vee b_{\pi_2})\,\mathbf{U}\,c_{\pi_3})$ matches this case and a_{π_1} and b_{π_2} will go through `ConstraintsHandler`. If both values in Lines 13 and 14 result in `false`, then the formula has been violated and we return `false`.

Algorithm 1: HyperLTL Monitor

Input: HyperLTL formula ϕ, $\mathcal{BK}$, set of incoming traces M
Output: $\lambda = \{\bot, ?\}$

1 **while** `getEvent`(e_i, t_m) **do**
2 **if** `newIncomingTrace`(t_m) **then**
3 $M \leftarrow M \cup \{t_m\}$
4 $C_m \leftarrow$ `ConstraintsHandler` $(\phi, \mathcal{BK}, e_i)$
5 Take a snapshot for constraints $\mathcal{C} = \{C_1, C_2, \cdots, C_m\}$ at time instant
6 $\beta \leftarrow$ `SAT`$(\mathcal{C})$
7 **if** $(\beta =$ `false`$)$ **then**
8 $\lambda \leftarrow \bot$
9 break
10 **return** (λ)

Algorithm 2: `ConstraintsHandler`

Input: HyperLTL formula ϕ, $\mathcal{BK}$, event e_i
Output: {`false`, Set of Constraints}

1 **if** $(\phi = \phi_1 \vee \phi_2)$ **then**
2 $\psi_1 \leftarrow$ `ConstraintsHandler` $(\phi_1, \mathcal{BK}_{\phi_1}, e_i)$
3 $\psi_2 \leftarrow$ `ConstraintsHandler` $(\phi_2, \mathcal{BK}_{\phi_2}, e_i)$
4 **if** $(\psi_1 =$ `false` $\wedge\ \psi_2 =$ `false`$)$ **then**
5 **return** (`false`)
6 **else if** $(\psi_1 =$ `false`$)$ **then**
7 **return** (ψ_2)
8 **else if** $(\psi_2 =$ `false`$)$ **then**
9 **return** (ψ_1)
10 **else**
11 **return** (ψ_1, ψ_2)
12 **else if** $(\phi := \phi_1\ \mathbf{U}\ \phi_2 \wedge ((\phi_1 := \phi_L \vee \phi_R) \wedge \neg($`samequantifiers`$(\phi_L, \phi_R))))$ **then**
13 $\psi_1 \leftarrow$ `ConstraintsHandler` $(\phi_L \mathbf{U}\ \phi_2, \mathcal{BK}, e_i)$
14 $\psi_2 \leftarrow$ `ConstraintsHandler` $(\phi_R \mathbf{U}\ \phi_2, \mathcal{BK}, e_i)$
15 **if** $(\psi_1 =$ `false` $\wedge\ \psi_2 =$ `false`$)$ **then**
16 **return** (`false`)
17 **else if** $(\psi_1 =$ `false`$)$ **then**
18 **return** $(\psi_2,$ `false`$)$
19 **else if** $(\psi_2 =$ `false`$)$ **then**
20 **return** $($`false`$, \psi_1)$
21 **else**
22 **return** (ψ_2, ψ_1)
23 **else**
24 $r \leftarrow$ `ConstraintRewriting`$(\phi, \mathcal{BK}, e_i)$
25 **if** $(r =$ `false`$)$ **then**
26 **return** `false`
27 **else**
28 **return** r

Algorithm 3: `ConstraintRewriting`

Input: HyperLTL formula φ, $\mathcal{BK}$, e_i
Output: Constraints r

1 $r \leftarrow$ `true`
2 $\varphi_r \leftarrow$ `quantifier-elimination`(φ)
3 $\varphi_r \leftarrow$ `REWRITE` (e_i, φ_r)
4 **if** $(\varphi_r =$ `false`$)$ **then**
5 **return** φ_r
6 **for** *(each* $a \in \mathcal{BK}$ *s.t.* $e_i \vDash a$*)* **do**
7 $r \leftarrow r \wedge \mathbf{X}^i a$
8 **if** $(a = \#a')$ **then**
9 $\mathcal{BK} \leftarrow \mathcal{BK} \setminus \{a\}$
10 **for** *(each* $a \in \mathcal{BK}$ *s.t.* $a = \boldsymbol{X}a'$*)* **do**
11 $\mathcal{BK} \leftarrow (\mathcal{BK} \setminus \{a\}) \cup \{a'\}$
12 **return** r

However, if only one of the sides returns some constraints, then we return `false` and alternating constraint for further refinement (Lines 17-20). Finally, if both sides satisfy the formula, then we return a combination of the returned values of Lines 13 and 14. This allows us to refine the constraints from the function `SAT` in Algorithm 4.

The last part of the algorithm (Lines 24– 28) invokes the `ConstraintRewriting` function which return the constraints for other types of formulas. For example, formula $\forall\pi_1.\forall\pi_2.\forall\pi_3.\forall\pi_4.(a_{\pi_1}\mathbf{U}b_{\pi_2})\ \mathbf{U}\ (c_{\pi_3}\ \mathbf{U}\ d_{\pi_4}))$ will directly undergo constraint generation.

Algorithm 3 (Constraints Rewriting). This algorithm generates the constraints (denoted by r) by utilizing the elements of $\mathcal{BK}$. We set the initial value of r to `true` as we have no violation in the start of the monitoring process. We

strip off the quantifiers of our formula φ to convert into its corresponding LTL form φ_r (Line 2). For example, $\forall\pi_1.\forall\pi_2.(a_{\pi_1}\,\mathbf{U}\,b_{\pi_2})$ will be converted to $(a\,\mathbf{U}\,b)$. Then, we apply `REWRITE` function to formula φ_r with the given event e_i (Line 3). This function is essentially the rewriting algorithm by Havelund and Rosu [13] (see Algorithm 5). If the event violates our formula then we immediately return the violation (Lines 4-5).

If ϕ is not violated and if the event satisfies any object $a \in \mathcal{BK}$, then a is considered for our constraints (Line 6). Given the position of the event is i in a trace, in Line 7 we administer $\mathbf{X}^i$ on a (i.e., $\mathbf{X}^i a$) . The elements of $\mathcal{BK}$ which are prefixed by "#" are removed from $\mathcal{BK}$ as we have indicated that their first appearance is significant (Lines 8-9). In our running example, the invocation of `ConstraintRewriting` for $a_{\pi_1}\,\mathbf{U}\,c_{\pi_3}$ with set $\mathcal{BK} = \{\#c\}$ and consecutive events of traces $t_1 = (ab)(ab)a(ad)c$, $t_2 = a(abcd)$, $t_3 = c$ will result in $r_1 = \mathbf{X}^4 c$, $r_2 = \mathbf{X}c$ and $r_3 = c$, respectively.

The elements of $\mathcal{BK}$ with "$\mathbf{X}$" operators are considered for upcoming events by stripping one instance of "$\mathbf{X}$" on that element (Lines 10-11). Indeed, the presence of $\mathbf{X}$'s in the elements of $\mathcal{BK}$ delays the observation and expose the corresponding proposition to be observed for constraint generation in the subsequent rounds. Finally, we return our generated constraint r.

Algorithm 4: `SAT`

```
   Input: Constraint Matrix C
   Output: λ = {false, ?}
1  Function SAT (C)
2      Initialize m'
3      columns ← max{|x| | x ∈ C}
4      existsConstrains ← false
5      for (j ← 0; j < columns; j++) do
6          β ← ⋀_{m=1}^{|M|} C_m[j]
7          if (β = false) then
8              dropColumn
9          else
10             m' ←
               largest constraint of column j
11             if
               (∃t ∈ C_(t,j).¬memberof(t, m'))
               then
12                 dropColumn
13             else
14                 existsConstrains ← true
15     if (existsConstraints = false) then
16         return (false)
17     else
18         return (?)
```

Algorithm 5: `REWRITE`

```
   Input: φ_r, e
   Output: {true, false, φ}
1  match (φ_r) with
2  | (a) :
3      if (a ∈ e) then
4          return (true)
5      else if (a ∉ e) then
6          return (false)
7  | (true) :
8      return (true)
9  | (false) :
10     return (false)
11     | (φ1 ∨ φ2) :
12     return
       (REWRITE(φ1, e) ∨ REWRITE(φ2, e))
13     | (φ1 U φ2) :
14     if (lastevent (e)) then
15         return (REWRITE(φ2, e))
16     else
17         return (REWRITE(φ2, e) ∨
           (REWRITE(φ1, e) ∧ (φ1 U φ2)))
18 | (Xφ) :
19     if (lastevent (e)) then
20         return (false)
21     else
22         return (REWRITE(φ, e))
```

Algorithm 4 (Satisfaction Function). The input of the `SAT` function is a set consisting of the constraints associated with each trace, i.e., $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$.

We can imagine all these constraints as rows of a matrix. For our running example, we will have $\mathcal{C}_i = [C_i^{(a_{\pi_1} \mathbf{U} c_{\pi_3})}, C_i^{(b_{\pi_2} \mathbf{U} c_{\pi_3})}, C_i^{d_{\pi_4}}]$ where i corresponds to i^{th} trace in M. We iterate through the columns for each of the traces and conjunct together their constraints. If they evaluate to `false`, then we can drop the column as traces have found a disagreement (Lines 3-8). If the conjunction is not `false`, we acquire the longest constraint m' of the corresponding column. We then check to see that no constraints associated by other traces disagree by confirming that they are members of m' (Lines 10-11). If one of the constraints disagrees, then we drop the column, or else we have found an agreement of constraints between the traces (Lines 12-14). Finally, we return a violation if we were unable to find any agreement within the constraints between traces (Lines 15-18).

Note that the process of dropping columns indeed results in a refined set of constraints. Since the incoming traces can progress at various speeds, we confirm that the constraints for "slower" traces are in-fact a member of the "fastest" trace's constraints. If no traces contradict the "fastest trace", then this suggests that no disagreement has yet emerged in the system. We resume taking snapshots of the system until a violation is detected.

Theorem 2 (Correctness of Algorithm 1). *Let φ be a HyperLTL formula. Algorithm 1 returns $\bot$ for an input set of traces T iff $[\Pi_F \models_T \varphi] = \bot$.*

6.3 Discussion

Our algorithms reflect that the decision of appropriate consideration for propositions or Boolean expressions, paired with the effective structural division of a HyperLTL formula, and provides an effective way to monitor complex HyperLTL formulas. Additionally, we encode only the minimum information to check that the agreement between traces is delineated according to the observed locations of propositions or Boolean expressions.

A potential drawback of our RV technique is its theoretical unbounded memory requirement. However, this requirement does not influence the cases where the verification is done offline. For online RV we can still use our algorithms for by making practical assumptions. For example, we can incorporate a synchronization mechanism amongst traces to ensure that the difference in length of traces is not beyond some bound. We note that the worst case complexity of Algorithm 1 is $\mathcal{O}(|t| \cdot |T|)$, where $|t|$ is the length of the longest trace in set T. Interestingly, this complexity is independent from the number of trace quantifiers in a given HyperLTL formula. Indeed, the set $\mathcal{BK}$ computed pre-runtime by Γ function provides the means to avoid dependence on the trace quantifiers, which otherwise is polynomial on the order of numbers of quantifiers. We believe that our proposed algorithm is efficient enough to be adopted for the monitoring of security policies in real-world applications.

Note that our proposed algorithm can only be used to monitor alternation-free fragment (i.e., $\forall^*$ and $\exists^*$) of HyperLTL, which can express a wide class of

security policies including non-interference and declassification. However, specification of some security policies require alternation in the trace quantifiers. For example, *noninference* [17] specifies that the behavior of low-variables should not change when all high variables are replaced by an arbitrary variable λ, given as follows:

$$\forall\pi.\exists\pi'.(\mathbf{G}\lambda_H(\pi') \wedge \mathbf{G}(\bigwedge_{a\in L} a_\pi \leftrightarrow a_{\pi'})$$

Similarly, generalized non-interference (GNI) [16] also requires alternation in trace quantifiers as it allows non-determinism in the low variables of the system.

7 Related Work

Static Analysis Sabelfeld et al. [24] survey the literature focusing on static program analysis for enforcement of security policies. In some cases, with compilers using Just-in-time compilation techniques and dynamic inclusion of code at run time in web browsers, static analysis does not guarantee secure execution at run time. Type systems, frameworks for JavaScript [6] and ML [21] are some approaches to monitor information flow. Several tools [18, 11, 19] add extensions such as statically checked information flow annotations to Java language. Clark et al. [7] present verification of information flow for deterministic interactive programs. On the other hand, our approach is capable of monitoring the subset of hyperproperties described by alternation-free HyperLTL and not just information flow without assistance from static analyzers. In [2], the authors propose a technique for designing runtime monitors based abstract interpretation of the system under inspection.

Dynamic analysis Russo et al. [23] concentrate on permissive techniques for the enforcement of information flow under flow-sensitivity. It has been shown that in the flow-insensitive case, a sound purely dynamic monitor is more permissive than static analysis. However, they show the impossibility of such a monitor in the flow-sensitive case. A framework for inlining dynamic information flow monitors has been presented by Magazinius et al. [14]. The approach by Chudnov et al. [5] uses hybrid analysis instead and argues that due to JIT compilation processes, it is no longer possible to mediate every data and control flow event of the native code. They leverage the results of Russo et al. [23] by inlining the security monitors. Chudnov et al. [4] again use hybrid analysis of 2-safety hyperproperties in relational logic. In [1], the authors propose an automata-based RV technique for monitoring only a disjunctive fragment of alternation-free HyperLTL.

Austin et al. [3] implement a purely dynamic monitor, however, restrictions such as "no-sensitive upgrade" were placed. Some techniques deploy taint tracking and labelling of data variables dynamically [26, 20]. Zdancewic et al. [25] verify information flow for concurrent programs. Most of the techniques cited

above aim to monitor security policies described solely with two trace quantifiers (without alternation), on observing a single run, whereas, our work is for any hyperproperties that can be described with alternation-free HyperLTL, when multiple runs are observed.

SME Secure multi-execution [10] is a technique to enforce non-interference. In SME, one executes a program multiple times, once for each security level, using special rules for I/O operations. Outputs are only produced in the execution linked to their security level. Inputs are replaced by default inputs except in executions linked to their security level or higher. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads. This approach is sound in a deterministic language.

While there are small similarities between SME and our work, there are fundamental differences. SME only focuses on non-interference and aims to enforce it, but there are many critical hyperproperties that differ from non-interference that our method is able to monitor. Thus, SME enforces a security policy at the cost of restricting what it can enforce, whereas our technique monitors a much larger set of policies.

8 Conclusion

In this paper, we introduced an algorithm for monitoring alternation-free fragment of HyperLTL [8], a temporal logic that allows for expressing complex information-flow properties like generalized non-interference, declassification, and quantitative non-interference. The main challenge in designing an RV algorithm for HyperLTL formulas is that reasoning about the formula involves analyzing multiple traces (as opposed to a single trace in traditional RV techniques). Our algorithm has three components: (1) a function that identifies propositions that have to be bookkept across multiple traces, (2) a constraint generator that encodes the occurrence of propositions of interest, and (3) a rewriting module based on the algorithm in [22] that incorporates formula progression with respect to incoming events for traces. In our view, our algorithm is a significant step forward in monitoring sophisticated information-flow security and privacy policies.

Our first step to extend this work will be to implement our algorithm and test it for real-world applications, e.g., in smartphones. For future work, one may consider RV algorithms based on monitor synthesis (as opposed to rewriting). We are also planning to develop techniques for monitoring alternating HyperLTL formulas. We believe dealing with such formulas is not possible without assistance from a static analyzer.

References

1. S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*, pages 239–252, 2016.

2. M. Assaf and D. A. Naumann. Calculational design of information flow monitors. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*, pages 210–224, 2016.
3. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM Transactions on Programming Languages and Systems*, pages 113–124, 2009.
4. A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014, Vienna, Austria, 19-22 July, 2014*, pages 48–62, 2014.
5. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of CSF*, pages 200–214, 2010.
6. R. Chugh, J. A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of PLDI*, pages 50–62, 2009.
7. D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *Proceedings of Formal Aspects in Security and Trust*, pages 50–66, 2008.
8. M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal Logics for Hyperproperties. In *Principles of Security and Trust (POST)*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014.
9. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
10. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P*, pages 109–124, 2010.
11. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, P. L. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*
12. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
13. K. Havelund and G. Rosu. Monitoring Programs Using Rewriting. In *Automated Software Engineering (ASE)*, pages 135–143, 2001.
14. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
15. Z. Manna and A. Pnueli. *Temporal verification of reactive systems - safety.* Springer, 1995.
16. D. McCullough. Noninterference and the composability of security properties. In *IEEE Symposium on Security and Privacy*, pages 177–186, 1988.
17. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 79–93, 1994.
18. A. C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.
19. Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels, 1998.
20. S. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. 197(1):3–16, 2008.
21. F. Pottier and V. Simonet. Information flow inference for ml. In *Proceedings of Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 319–330, 2002.

22. G. Rosu and K. Havelund. Rewriting-Based Techniques for Runtime Verification. *Automated Software Engineering*, 12(2):151–197, 2005.
23. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the XXrd IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199, 2010.
24. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
25. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–, 2003.
26. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical report, EECS Department, University of California, Berkeley, Oct 2009.

9 Additional Examples

9.1 Example of Γ Function

.

Example: We demonstrate a step by step application of Γ function. Consider the following HyperLTL$_1$ formula.

$$\phi = \forall\pi_1.\forall\pi_2.\forall\pi_3.\ \mathbf{X}(a_{\pi_1}\,\mathbf{U}\,b_{\pi_3})\mathbf{U}\ \overbrace{\neg(true\,\mathbf{U}\,\neg c_{\pi_2})}^{\mathbf{G}c}$$

- Applying Γ function: $\Gamma(\mathbf{X}(a_{\pi_1}\,\mathbf{U}\,b_{\pi_3})\mathbf{U}\ \neg(true\,\mathbf{U}\,\neg c_{\pi_2}), \mathcal{V}, k)$

- Using the last subcase of $\phi_1\mathbf{U}\phi_2$ in the definition of Γ function: $\#^{-1}\odot\Gamma(\mathbf{X}(a_{\pi_1}\,\mathbf{U}\,b_{\pi_3}), \mathcal{V}, k := true)\cup\#\odot\Gamma(\neg(true\,\mathbf{U}\,\neg c_{\pi_2}), \mathcal{V}\cup\{\pi_1,\pi_3\}, k := true)$

- Using second case ($\mathbf{X}\phi$) and third case ($\neg\phi$) in the definition of Γ function: $\#^{-1}\odot\mathbf{X}\odot\Gamma((a_{\pi_1}\,\mathbf{U}\,b_{\pi_3}), \mathcal{V}, k := true)\cup\#\odot\neg\odot\Gamma((true\,\mathbf{U}\,\neg c_{\pi_2}), \{\pi_1,\pi_3\}, k := true)$

- Using first and second subcases of $\phi_1\mathbf{U}\phi_2$ in the definition of Γ function: $\#^{-1}\odot\mathbf{X}\odot\{\#b\}\cup\#\odot\neg\odot\Gamma((\neg c_{\pi_2}), \{\pi_1,\pi_3\}, k := true)$

- Using the definition of $\odot$ and first case in the definition of Γ function: $\mathbf{X}\odot\{b\}\cup\#\odot\neg\odot\neg\odot\Gamma((c_{\pi_2}), \{\pi_1,\pi_3\}, k := true)\ = \{\mathbf{X}b\}\cup\{\#c\} = \{\mathbf{X}b, \#c\}$

9.2 Examples for Constraint Generation

Main utilization of the bookkeeping set $\mathcal{BK}$ and structure of the formula is to generate the constraints during the progression of each trace. The key idea behind constraint generation is to extract useful information required to check the compliance of all traces with respect to the given HyperLTL$_1$ formula. We demonstrate the constraint generation process by the following examples.

– Consider the formula $\phi = \forall\pi_1.\forall\pi_2.\ a_{\pi_1}\ \mathbf{U}\ b_{\pi_2}$ and traces $t_1 = aaaaab$ and $t_2 = aab$. In this case, we can find the corresponding $\Gamma(\phi) = \{b\}$ using definition of the Γ function (Fig. 2). Here, proposition b is required in the trace agreement and our task is to keep the information of its occurrence in each trace. The progression of first trace results in the constraint $\mathcal{C}_{t_1} = \mathbf{X}^5 b$, i.e., proposition b is required to happen at location $t_1[6]$. Similarly, the constraint generated for the second trace is $\mathcal{C}_{t_2} = \mathbf{X}^2 b$ which clearly disagrees with $\mathcal{C}_{t_1}$. Note that we only generate constraints if the incoming trace does not violate the formula ϕ. For example, we will not generate any constraints for trace $t_3 = acaab$ as it violates ϕ at the second location.

– Consider the formula $\phi = \forall\pi_1.\forall\pi_2.\forall\pi_3.\ (a_{\pi_1}\mathbf{U}b_{\pi_2})\mathbf{U}c_{\pi_3}$ and traces $t_1 = (ab)a(ac)(ac)b$, $t_2 = (ab)a(ac)(a)(b)$ and $t_3 = a(ac)(ac)b$. We can find the corresponding $\Gamma(\phi) = \{b, \#c\}$ using the definition of the Γ function (Fig. 2). Here the set $\mathcal{BK}$ indicates that we need to keep the information about the first occurrence of proposition c and every occurrence of proposition b. This leads to the constraints $\mathcal{C}_{t_1} = b \wedge \mathbf{X}^2 c \wedge \mathbf{X}^4 b$, $\mathcal{C}_{t_2} = b \wedge \mathbf{X}^2 c \wedge \mathbf{X}^4 b$ and $\mathcal{C}_{t_3} = \mathbf{X}^2 c \wedge \mathbf{X}^3 b$ for traces t_1, t_2 and t_3, respectively. Based on these constraints, we note that traces t_1 and t_2 are in accordance with trace agreement, however, trace t_3 violates the trace agreement.
– Consider the formula $\phi = \forall\pi_1.\forall\pi_2.\ a_{\pi_1} \rightarrow \mathbf{F}b_{\pi_2}$ and traces $t_1 = dcb$, $t_2 = eee$. Our first step is to transform the formula in disjunctive form as follows: $\phi = \forall\pi_1.\forall\pi_2.\ \neg a_{\pi_1} \vee \mathbf{F}b_{\pi_2}$. In this case, our bookkeeping function $\Gamma(\phi)$ returns an empty set as the semantics of the HyperLTL does not enforce a or b to occur on the same location in each trace. However, we still need to keep the information that each incoming trace satisfies ϕ due the presence of $\neg a$, $\mathbf{F}b$, or both. For each trace, we generate two constraints with respect to $\neg a$ and $\mathbf{F}b$, e.g., constraints for trace t_1 are $\mathcal{C}_{t_1}^{\neg a} = \neg a$ and $\mathcal{C}_{t_1}^{\mathbf{F}b} = \mathbf{F}b$. These constraints indicate that the trace t_1 satisfies both parts of the disjunction, i.e., $\neg a$ and $\mathbf{F}b$. Similarly, the constraint for the trace t_2 is $\mathcal{C}_{t_1}^{\neg a} = \neg a$ and $\mathcal{C}_{t_1}^{\mathbf{F}b} = \texttt{false}$, as there is no b in the second trace. Combining constraints for both t_1 and t_2, we can deduce two important points: 1) the formula ϕ is satisfied with respect to t_1 and t_2 and; 2) for incoming traces, the only constraint is $\mathcal{C}_{t_i}^{\neg a} = \neg a$. Since no b was never observed in t_2, the satisfaction came from the fact that the antecedent evaluated to `false`. Furthermore, the presence of a in other traces will result in a violation in ϕ as t_2 does not contain b. We call this process as *constraint refinement*.

9.3 Example Matrix For Function *SAT*

Consider the formula: $\varphi = \forall\pi_1.\forall\pi_2.\forall\pi_3.\forall\pi_4.((a_{\pi_1} \vee b_{\pi_2})\mathbf{U}c_{\pi_3}) \vee d_{\pi_4}$ and the following traces: $t_1 = (da)aaac$, $t_2 = (da)aa$, $t_3 = (dab)$, $t_4 = bbb$, $t_5 = (db)bbbc$. Note that the traces have not necessarily halted.

We compute the following matrix:

	$a_{\pi_1}\mathbf{U}c_{\pi_3}$	$b_{\pi_2}\mathbf{U}c_{\pi_3}$	d_{π_4}
t_1	$\mathbf{X}^4 c$	`false`	`true`
t_2	`true`	`false`	`true`
t_3	`true`	`true`	`true`
t_4	`false`	`true`	`false`
t_5	`false`	$\mathbf{X}^4 c$	`true`

Each row represents the current generated constraints for each trace, and the columns depict which constraints to follow. Having `true` as a constraint means that the trace has not violated ϕ with respect to that constraint.

10 Proofs

Proof of Theorem 1

Given a HyperLTL formula φ and assuming we have set T such that $[\Pi_F \models_T \varphi] = \top$ then

- Γ function returns all the propositions required for bookkeeping.
- Given the set $\mathcal{BK}$, every element $k \in \mathcal{BK}$ is included in some trace agreement described by φ.

For the first part, the proof is done by a structural induction on $\varphi \in$HyperLTL:

Base case: $\varphi \in \{a_\pi \in AP\}$

- **Case** $a_\pi \in AP$.
  - Let us suppose that we have $k =$ `false`. k indicates whether we have previously observed an **U** operator. Since this is not the case, we simply return $\{\}$
  - Let us suppose that we have $k =$ `true`. Having observed an **U** operator, we are able to return our value $\{a\}$

Induction Case: $\varphi \in \{\neg\phi', \mathbf{X}\phi', \phi_1 \vee \phi_2, \phi_1 \mathbf{U} \phi_2\}$. Our induction hypothesis states that applying Γ will result in the propositions that require bookkeeping.

- **Case** $\neg\phi'$. We propagate "$\neg$" to the result of iterating through the Γ of ϕ'
- **Case** $\mathbf{X}\phi'$. Similar to the previous case, except we propagate "$\mathbf{X}$"
- **Case** $\phi_1 \vee \phi_2$. Applying Γ on $\phi_1 \vee \phi_2$ will influence the results according to the state of k and whether or not either operand is not a proposition themselves:
  - Let us suppose that neither of the operands are propositions, we can then apply Γ on both operands and find their respected values.
  - Let us suppose that we have $k =$ `false`. If both of the operands are propositions, then we simply return $\{\}$. Else we apply Γ on the operands that are propositions, and return their respected values.

- Let us suppose that we have $k = \texttt{true}$. We can distinguish between two different sub-cases:
  - Both of the operands are propositions. We choose differently according to their trace variable. Given that they belong to the same trace then we consider their disjunction collectively for for bookkeeping. But, if they belong to different traces, then either side of the disjunction can individually satisfy the agreement between traces allowing for individual consideration of both sides.
  - Either one of the operands is a proposition. Irrespective of their trace variables, we return the union of the result of the application of Γ on the whichever of the operands is not a proposition and the remaining proposition.

– **Case** $\phi_1 \mathbf{U} \phi_2$. We have to consider the next three sub-cases:
- Both of the operands are propositions. Given that the trace variables differ, we return the # of the right operand as this is the value that satisfies the formula. Else there is no trace agreement and simply return $\{\}$.
- Only the left operand is a proposition. We iterate through the right operand with Γ, including the trace variables of the left operand to be matched for the next iterations, and return the appropriate result.
- The left operand is not a proposition. Whether or not the left operand contains another **U** operator, will vary the outcome. If the left operand does not contain an **U** operator, then this is as the previous case. Else, we call Γ on the left operand, but discard the succeeding # by including $\#^{-1}$ to the result, next the outcome is added with the results of the previous case.

The second part of the proof is done by contradiction.

Consider the HyperLTL formula $\varphi = \forall \pi_1 . \forall \pi_2 . \, a_{\pi_1} \mathbf{U} b_{\pi_2}$.
Let's assume that the $\mathcal{BK} = \{a, \#b\}$. Moreover, for any set T of finite traces, and $t_1, t_2 \in T$, we can rewrite φ according to the semantics of HyperLTL as follows:

$$\forall t_1, t_2 \in T : \Pi[\pi_1 \to t_1, \pi_2 \to t_2] \models_T a_{t_1} \mathbf{U} \, b_{t_2} \tag{1}$$

$$\begin{array}{l} \exists i \geq 0 . (\forall t_2 \in T : \Pi[\pi_2 \to t_2[i, \infty]] \models_T b_{t_2} \wedge \\ \forall j \in [0, i) . \forall t_1 \in T : \Pi[\pi_1 \to t_1[j, \infty]] \models_T a_{t_1}) \end{array} \tag{2}$$

$$\begin{array}{l} \exists i \geq 0 . (\forall t_2 \in T : \Pi[\pi_2 \to t_2[i, \infty]] \models_T b_{t_2} \wedge \\ \forall j \in [0, i) . \forall t_1 \in T : \Pi[\pi_1 \to t_1[j, \infty]] \models_T a_{t_1}) \end{array} \tag{3}$$

We can observe that the index imposed by the existential quantifier creates a bound for the traces t_1 and t_2, enforcing an agreement at location i. Furthermore, this suggest that the location of b is critical for their agreement, as it must be required that b is satisfied at i for both t_1 and t_2. Since no particular restriction is imposed on a, then we should not include a in $\mathcal{BK}$. Therefore, our $\mathcal{BK}$ should

only contain $\{\#b\}$. An example of two traces that have an agreement described by φ but oppose that $\mathcal{BK} = \{a, \#b\}$: $t_1 = aaab$ and $t_2 = aaa(ab)$. Both t_1 and t_2 satisfy φ, but through the satisfaction of b at index 4 and no agreement established for a. ■

Proof of Theorem 2

Let φ be a HyperLTL formula. Algorithm 1 returns $\perp$ for an input set of traces T *iff* $[\Pi_F \models_T \varphi] = \perp$.

- ($\Rightarrow$) For an input set $T \in \mathcal{P}^*(\Sigma^*)$, where $[\Pi_F \models_T \varphi] = \perp$. By contradiction let us assume that Algorithm 1 returns ?. The antecedent implies that there exist at least one trace ($t \in T$) such that any extension of t violates φ. If the algorithm has not yet returned $\perp$, there is at least one trace which has not been observed so far. In all cases (either a given trace violates the φ (Algorithm 3) or creates disagreement amongst other trace (Algorithm 4)) which result in $[\Pi_F \models_T \varphi] = \perp$, Algorithm 1 returns $\perp$, which is a contradiction. Therefore, if $[\Pi_F \models_T \varphi] = \perp$ then Algorithm 1 returns $\perp$.
- ($\Leftarrow$) If Algorithm 1 returns $\perp$, by contradiction, let us assume that $[\Pi_F \models_T \varphi] \neq \perp$. The antecedent implies that there is at least one trace $t \in T$ such that $[\Pi_F \models_T \varphi] \neq \perp$. But the violation of φ for a single trace leads to the fact that $[\Pi_F \not\models_T \varphi]$. This contradicts the assumption that $[\Pi_F \models_T \varphi] = \perp$. Hence, if Algorithm 1 returns $\perp$ then $[\Pi_F \models_T \varphi] = \perp$. ■