Temporal Logics for Hyperproperties

Michael R. Clarkson[1], Bernd Finkbeiner[2], Masoud Koleini[1],
Kristopher K. Micinski[3], Markus N. Rabe[2], and César Sánchez[4]

[1]George Washington University; [2]Universität des Saarlandes;
[3]University of Maryland, College Park; [4]IMDEA Software Institute

Abstract. Two new logics for verification of hyperproperties are proposed. Hyperproperties characterize security policies, such as noninterference, as a property of sets of computation paths. Standard temporal logics such as LTL, CTL, and CTL* can refer only to a single path at a time, hence cannot express many hyperproperties of interest. The logics proposed here, HyperLTL and HyperCTL*, add explicit and simultaneous quantification over multiple paths to LTL and to CTL*. This kind of quantification enables expression of hyperproperties. A model checking algorithm for the proposed logics is given. For a fragment of HyperLTL, a prototype model checker has been implemented.

1 Introduction

Trace properties, which developed out of an interest in proving the correctness of programs [35], characterize correct behavior as properties of individual execution traces. Although early verification techniques specialized in proving individual correctness properties of interest, such as mutual exclusion or termination, temporal logics soon emerged as a general, unifying framework for expressing and verifying trace properties. Practical model checking tools [12, 18, 31] based on those logics now enable automated verification of program correctness.

Verification of security is not directly possible with such tools, because some important security policies cannot be characterized as properties of individual execution traces [42]. Rather, they are properties of sets of execution traces, also known as *hyperproperties* [16]. Specialized verification techniques have been developed for particular hyperproperties [5, 30, 45, 47], as well as for *2-safety* properties [58], which are properties of pairs of execution traces. But a unifying program logic for expressing and verifying hyperproperties could enable automated verification of a wide range of security policies.

In this paper, we propose two such logics. Both are based, like hyperproperties, on examining more than one execution trace at a time. Our first logic, *HyperLTL*, generalizes linear-time temporal logic (LTL) [48]. LTL implicitly quantifies over only a single execution trace of a system, but HyperLTL allows explicit quantification over multiple execution traces simultaneously, as well as propositions that stipulate relationships among those traces. For example, HyperLTL can express information-flow policies such as *observational determinism* [41, 50, 68], which requires programs to behave as (deterministic) functions

from low-security inputs to low-security outputs. The following two programs do not satisfy observational determinism, because they leak the value of high-security variable h through low-security variable l, thus making the program behave nondeterministically from a low-security user's perspective:

$$(1) \ \ l := h \qquad\qquad (2) \ \ \textbf{if } h = 0 \textbf{ then } l := 1 \textbf{ else } l := 0$$

Other program logics could already express observational determinism or closely related policies [7,33,45]. Milushev and Clarke [44–46] have even proposed other logics for hyperproperties, which we discuss in Section 8. But HyperLTL provides a simple and unifying logic in which many information-flow security policies can be directly expressed.

Information-flow policies are not one-size-fits-all. Different policies might be needed depending on the power of the adversary. For example, the following program does not satisfy observational determinism, but the program might be acceptable if nondeterministic choices, denoted ⫿, are resolved such that the probability distribution on output value l is uniform:

$$(3) \ \ l := h \ \ \| \ \ l := 0 \ \ \| \ \ l := 1$$

On the other hand, if the adversary can influence the resolution of nondeterministic choices, program (3) could be exploited to leak information. Similarly, the following program does satisfy observational determinism, but the program might be unacceptable if adversaries can monitor execution time:

$$(4) \ \ \textbf{while } h > 0 \textbf{ do } \{h := h - 1\}$$

In Section 3, we show how policies appropriate for the above programs, as well as other security policies, can be formalized in HyperLTL.

Our second logic, *HyperCTL**, generalizes a branching-time temporal logic, CTL* [21]. Although CTL* already has explicit trace quantifiers, only one trace is ever in scope at a given point in a formula (see Section 5.1), so CTL* cannot directly express hyperproperties. But HyperCTL* can, because it permits quantification over multiple execution traces simultaneously. HyperLTL and HyperCTL* enjoy a similar relationship to that of LTL and CTL*: HyperLTL is the syntactic fragment of HyperCTL* containing only formulas in *prenex* form—that is, formulas that begin exclusively with quantifiers and end with a quantifier-free formula. HyperCTL* is thus a strict generalization of HyperLTL. HyperCTL* also generalizes a related temporal logic, SecLTL [20], and subsumes epistemic temporal logic [22,60] (see Section 5).

Having defined logics for hyperproperties, we investigate model checking of those logics. In Section 6, we show that for HyperCTL* the model checking problem is decidable by reducing it to the satisfiability problem for quantified propositional temporal logic (QPTL) [56]. Since HyperCTL* generalizes HyperLTL, we immediately obtain that the HyperLTL model checking problem is also decidable. We present a hierarchy of fragments, which allows us to precisely characterize the complexity of the model checking problem in the number quantifier

alternations. The lowest fragment, which disallows any quantifier alternation, can be checked by a space-efficient polynomial-time algorithm (NLOGSPACE in the number of states of the program).

We also prototype a model checker that can handle an important fragment of HyperLTL, including all the examples from Section 3. The prototype implements a new model checking algorithm based on a well-known LTL algorithm [65, 66] and on a self-composition construction [7, 58]. The complexity of our algorithm is exponential in the size of the program and doubly exponential in the size of the formula—impractical for real-world programs, but at least a demonstration that model checking of hyperproperties formulated in our logic is possible.

This paper contributes to theoretical and foundational aspects of security by:
- defining two new program logics for expressing hyperproperties,
- demonstrating that those logics are expressive enough to formulate important information-flow policies,
- proving that the model checking problem is decidable, and
- prototyping a new model checking algorithm and using it to verify security policies.

The rest of the paper is structured as follows. Section 2 defines the syntax and semantics of HyperLTL. Section 3 provides several example formulations of information-flow policies. Section 4 defines the syntax and semantics of HyperCTL*. Section 5 compares our two logics with other temporal and epistemic logics. Section 6 obtains a model checking algorithm for HyperCTL*. Section 7 describes our prototype model checker. Section 8 reviews related work, and Section 9 concludes.

2 HyperLTL

HyperLTL extends propositional linear-time temporal logic (LTL) [48] with explicit quantification over *traces*. A *trace* is an infinite sequence of sets of *atomic propositions*. Let AP denote the set of all atomic propositions. The set TR of all traces is therefore $(2^{\mathsf{AP}})^\omega$.

We first define some notation for manipulating traces. Let $t \in \mathsf{TR}$ be a trace. We use $t[i]$ to denote element i of t, where $i \in \mathbb{N}$. Hence, $t[0]$ is the first element of t. We write $t[0, i]$ to denote the prefix of t up to and including element i, and $t[i, \infty]$ to denote the infinite suffix of t beginning with element i.

Syntax. Let π be a *trace variable* from an infinite supply $\mathcal{V}$ of trace variables. Formulas of HyperLTL are defined by the following grammar:

$$
\begin{aligned}
\psi &::= \ \exists \pi. \, \psi \mid \ \forall \pi. \, \psi \mid \ \varphi \\
\varphi &::= \ a_\pi \quad \mid \ \neg \varphi \quad \mid \ \varphi \vee \varphi \quad \mid \quad \mathrm{X} \, \varphi \quad \mid \ \varphi \, \mathrm{U} \, \varphi
\end{aligned}
$$

Connectives $\exists$ and $\forall$ are universal and existential trace quantifiers, read as "along some traces" and "along all traces." For example, $\forall \pi_1. \, \forall \pi_2. \, \exists \pi_3. \, \psi$ means that for all traces π_1 and π_2, there exists another trace π_3, such that ψ holds on those three traces. (Since branching-time logics also have explicit path quantifiers, it

is natural to wonder why one of them does not suffice to formulate hyperproperties. Section 5.1 addresses that question.) A HyperLTL formula is *closed* if all occurrences of trace variables are bound by a trace quantifier.

An atomic proposition a, where $a \in \mathsf{AP}$, expresses some fact about states. Since formulas may refer to multiple traces, we need to disambiguate which trace the proposition refers to. So we annotate each occurrence of an atomic proposition with a trace variable π. Boolean connectives $\neg$ and $\vee$ have the usual classical meanings. Implication, conjunction, and bi-implication are defined as syntactic sugar: $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$, and $\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, and $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$. True and false, written true and false, are defined as $a_\pi \vee \neg a_\pi$ and $\neg\mathsf{true}$.

Temporal connective $\mathrm{X}\,\varphi$ means that φ holds on the next state of every quantified trace. Likewise, $\varphi_1 \,\mathrm{U}\, \varphi_2$ means that φ_2 will eventually hold of the states of all quantified traces that appear at the same index, and until then φ_1 holds. The other standard temporal connectives are defined as syntactic sugar: $\mathrm{F}\,\varphi \equiv \mathsf{true}\,\mathrm{U}\,\varphi$, and $\mathrm{G}\,\varphi \equiv \neg\,\mathrm{F}\,\neg\varphi$, and $\varphi_1 \,\mathrm{W}\, \varphi_2 \equiv (\varphi_1 \,\mathrm{U}\, \varphi_2) \vee \mathrm{G}\,\varphi_1$, and $\varphi_1 \,\mathrm{R}\, \varphi_2 \equiv \neg(\neg\varphi_1 \,\mathrm{U}\, \neg\varphi_2)$.

We also introduce syntactic sugar for comparing traces. Given a set P of atomic propositions, $\pi[0] =_P \pi'[0] \equiv \bigwedge_{a \in P} a_\pi \leftrightarrow a_{\pi'}$. That is, $\pi[0] =_P \pi'[0]$ holds whenever the first state in both π and π' agree on all the propositions in P. And $\pi =_P \pi' \equiv \mathrm{G}(\pi[0] =_P \pi'[0])$, that is, all the positions of π and π' agree on P. The analogous definitions hold for $\neq$.

Semantics. The validity judgment for HyperLTL formulas is written $\Pi \models_T \psi$, where T is a set of traces, and $\Pi : \mathcal{V} \rightarrow \mathsf{TR}$ is a *trace assignment* (i.e., a *valuation*), which is a partial function mapping trace variables to traces. Let $\Pi[\pi \mapsto t]$ denote the same function as Π, except that π is mapped to t. We write trace set T as a subscript on $\models$, because T propagates unchanged through the semantics; we omit T when it is clear from context. Validity is defined as follows:

$$
\begin{array}{lll}
\Pi \models_T \exists\pi.\,\psi & \text{iff} & \text{there exists } t \in T : \Pi[\pi \mapsto t] \models_T \psi \\
\Pi \models_T \forall\pi.\,\psi & \text{iff} & \text{for all } t \in T : \Pi[\pi \mapsto t] \models_T \psi \\
\Pi \models_T a_\pi & \text{iff} & a \in \Pi(\pi)[0] \\
\Pi \models_T \neg\varphi & \text{iff} & \Pi \not\models_T \varphi \\
\Pi \models_T \varphi_1 \vee \varphi_2 & \text{iff} & \Pi \models_T \varphi_1 \text{ or } \Pi \models_T \varphi_2 \\
\Pi \models_T \mathrm{X}\,\varphi & \text{iff} & \Pi[1, \infty] \models_T \varphi \\
\Pi \models_T \varphi_1 \,\mathrm{U}\, \varphi_2 & \text{iff} & \text{there exists } i \geq 0 : \Pi[i, \infty] \models_T \varphi_2 \\
& & \quad \text{and for all } 0 \leq j < i \text{ we have } \Pi[j, \infty] \models_T \varphi_1
\end{array}
$$

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

We are interested in whether programs satisfy formulas, so we first derive a set T of traces from a program, first using *Kripke structures* as a unified representation of programs. A Kripke structure K is a tuple $(S, s_0, \delta, \mathsf{AP}, L)$ comprising a set of *states* S, an *initial state* $s_0 \in S$, a transition function $\delta : S \rightarrow 2^S$, a set of *atomic propositions* AP, and a labeling function $L : S \rightarrow 2^{\mathsf{AP}}$.

To ensure that all traces are infinite, we require that $\delta(s)$ is nonempty for every state s.

The set $\mathsf{Traces}(K)$ of traces of K is the set of all sequences of labels produced by the state transitions of K starting from initial state. Formally, $\mathsf{Traces}(K)$ contains trace t iff there exists a sequence $s_0 s_1 \ldots$ of states, such that s_0 is the initial state, and for all $i \geq 0$, it holds that $s_{i+1} \in \delta(s_i)$; and $t[i] = L(s_i)$. A Kripke structure K *satisfies* φ, denoted by $K \models \varphi$, if $\mathsf{Traces}(K)$ satisfies φ.

It will later be technically convenient to consider enlarging the set AP of atomic propositions permitted by a Kripke structure to a set AP', such that $\mathsf{AP} \subset \mathsf{AP}'$. We extend $\mathsf{Traces}(K)$ into the set of traces $\mathsf{Traces}(K, \mathsf{AP}')$ that is agnostic about whether each new proposition holds at each state. A trace $(P_0 \cup P_0')(P_1 \cup P_1') \ldots \in \mathsf{Traces}(K, \mathsf{AP}')$ whenever $P_0 P_1 \ldots \in \mathsf{Traces}(K)$, and for all $i \geq 0$: $P_i' \subseteq \mathsf{AP}' \setminus \mathsf{AP}$. The final conjunct requires every possible set of new atomic propositions to be included in the traces.

3   Security Policies in HyperLTL

We now put HyperLTL into action by formulating several *information-flow security policies*, which stipulate how information may propagate from inputs to outputs. Information-flow is a very active field in security; see [23,53] for surveys.

Noninterference. A program satisfies *noninterference* [26] when the outputs observed by low-security users are the same as they would be in the absence of inputs submitted by high-security users. Since its original definition, many variants with different execution models have been named "noninterference." For clarity of our examples, we choose a simple *state-based* synchronous execution model in which atomic propositions of the traces contain the values of program variables, and in which progress of time corresponds to execution steps in the model. We also assume that the variables are partitioned into input and output variables, and into two security levels, *high* and *low*. (We could handle lattices of security levels by conjoining several formulas that stipulate noninterference between elements of the lattice.)

Noninference [42] is a variant of noninterference that can be stated in our simple system model. Noninference stipulates that, for all traces, the low-observable behavior must not change when all high inputs are replaced by a dummy input λ, that is, when the high input is removed. Noninference, a *liveness hyperproperty* [16], can be expressed in HyperLTL as follows:

$$\forall \pi. \exists \pi'. \ (\mathrm{G}\, \lambda_{\pi'}) \ \wedge \ \pi =_L \pi' \tag{5}$$

where $\lambda_{\pi'}$ expresses that all of the high inputs in the current state of π' are λ, and $\pi =_L \pi'$ expresses that all low variables in π and π' have the same values.

Nondeterminism. Noninterference was introduced for use with deterministic programs. Nonetheless, nondeterminism naturally arises when program specifications abstract from implementation details, so many variants of noninterference

have been developed for nondeterministic programs. We formalize two variants here.

A (nondeterministic) program satisfies *observational determinism* [68] if every pair of traces with the same initial low observation remain indistinguishable for low users. That is, the program appears to be deterministic to low users. Programs that satisfy observational determinism are immune to *refinement attacks* [68], because observational determinism is preserved under refinement. Observational determinism, a *safety hyperproperty* [16], can be expressed in HyperLTL as follows:

$$\forall \pi. \forall \pi'. \ \pi[0] =_{L,\text{in}} \pi'[0] \ \rightarrow \ \pi =_{L,\text{out}} \pi' \tag{6}$$

where $\pi =_{L,\text{in}} \pi'$ and $\pi =_{L,\text{out}} \pi'$ express that both traces agree on the low input and low output variables, respectively.

Generalized noninterference (GNI) [39] permits nondeterminism in the low-observable behavior, but stipulates that low-security outputs may not be altered by the injection of high-security inputs. Like noninterference, GNI was original formulated for event-based systems, but it can also be formulated for state-based systems [42]. GNI is a liveness hyperproperty and can be expressed as follows:

$$\forall \pi. \forall \pi'. \exists \pi''. \ \pi =_{H,\text{in}} \pi'' \ \wedge \ \pi' =_L \pi'' \tag{7}$$

The trace π'' in (7) is an *interleaving* of the high inputs of the first trace and the low inputs and outputs of the second trace. Other security policies based on interleavings, such as *restrictiveness* [40], *separability* [42], and *forward correctability* [43] can similarly be expressed in HyperLTL.

Declassification. Some programs need to reveal secret information to fulfill functional requirements. For example, a password checker must reveal whether the entered password is correct or not. The noninterference policies we have examined so far prohibit such behavior. More flexible security policies have been designed to permit *declassification* of information; see [54] for a survey.

With HyperLTL, we easily specify customized declassification policies. For example, suppose that a system inputs a password in its initial state, then declassifies whether that password is correct in the next state. The following policy (a safety hyperproperty) stipulates that leaking the correctness of the password is permitted, but that otherwise observational determinism must hold:

$$\forall \pi. \forall \pi'. (\pi[0] =_{L,\text{in}} \pi'[0] \ \wedge \ \text{X}(pw_\pi \leftrightarrow pw_{\pi'})) \rightarrow \pi =_{L,\text{out}} \pi' \tag{8}$$

where atomic proposition pw expresses that the entered password is correct.

Quantitative noninterference. Quantitative information-flow policies [13, 15, 27, 34] permit leakage of information at restricted rates. One way to measure leakage is with *min-entropy* [57], which quantifies the amount of information an attacker can gain given the answer to a single guess about the secret. The *bounding problem* [67] for min-entropy is to determine whether that amount is bounded

from above by a constant n. Assume that the program whose leakage is being quantified is deterministic, and assume that the secret input to that program is uniformly distributed. The bounding problem then reduces to determining that there is no tuple of $2^n + 1$ low-distinguishable traces [57,67] (a safety hyperproperty). We can express that as follows:

$$\neg \exists \pi_0. \ \ldots \ . \ \exists \pi_{2^n}. \ \left(\bigwedge_i \pi_i =_{L,\mathsf{in}} \pi_0 \right) \ \wedge \ \bigwedge_{i \neq j} \pi_i \neq_{L,\mathsf{out}} \pi_j \tag{9}$$

The initial negation can pushed inside to obtained a proper HyperLTL formula.

Quantitative flow and entropy naturally bring to mind probabilistic systems. We haven't yet explored extending our logics to enable specification of policies that involve probabilities. Perhaps techniques previously used with epistemic logic [28] could be adapted; we leave this as future work.

Event-based systems. Our examples above use a synchronous state-based execution model. Many formulations of security policies, including the original formulation of noninterference [26], instead use an *event-based* system model, in which input and output events are not synchronized and have no relation to time. HyperLTL can express policies for asynchronous execution models, too. For example, HyperLTL can express the original definition of noninterference [26] and observational determinism; Appendix A shows how. The key idea is to allow the system to stutter and to quantify over all stuttered versions of the executions. We characterize the correct *synchronization* of a pair of traces as having updates to low variables only at the same positions. We then add an additional antecedent to the policy formula to require that only those pairs of traces that are synchronized correctly need to fulfill the security condition.

4 HyperCTL*

HyperLTL was derived from LTL by extending the models of formulas from single traces to sets of traces. However, like LTL, HyperLTL is restricted to linear time and cannot express branching-time properties (e.g., all states that succeed the current state satisfy some proposition). We show now that a branching-time logic for hyperproperties could be derived from a branching-time logic for trace properties, such as CTL* [21]. We call this logic HyperCTL*. The key idea is again to use sets instead of singletons as the models of formulas.

Syntax. HyperCTL* generalizes HyperLTL by allowing quantifiers to appear anywhere within a formula. Quantification in HyperCTL* is over *paths* through a Kripke structure. A path p is an infinite sequence of pairs of a state and a set of atomic propositions. Hence, a path differs from a trace by including a state of the Kripke structure in each element. Formally, $p \in (S \times 2^{\mathsf{AP}})^\omega$, where S is the states of the Kripke structure. As with traces, $p[i]$ denotes the element i of p, and $p[i, \infty]$ denotes the suffix of p beginning with element i. We also define a new notation: let $p(i)$ be the state in element i of p.

In HyperCTL*, π is a *path variable* and $\exists\pi$ is a *path quantifier*. Formulas of HyperCTL* are defined by the following grammar:

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

We introduce all the syntactic sugar for derived logical operators, as for HyperLTL. The universal quantifier can now be defined as syntactic sugar, too: $\forall\pi.\ \varphi \equiv \neg\exists\pi.\ \neg\varphi$. A HyperCTL* formula is *closed* if all occurrences of some path variable π are in the scope of a path quantifier. A HyperCTL* *specification* is a Boolean combination of closed HyperCTL* formulas each beginning with a quantifier (or its negation).

Semantics. The validity judgment for HyperCTL* formulas is written $\Pi \models_K \varphi$, where K is a Kripke structure, and $\Pi : \mathcal{V} \to (S \times 2^{\mathsf{AP}})^\omega$ is a *path assignment*, which is a partial function mapping path variables to paths. We write K as a subscript on $\models$, because K propagates unchanged through the semantics; we omit K when it is clear from context. Validity is defined as follows:

$$
\begin{aligned}
\Pi \models_K a_\pi \quad &\text{iff} \quad a \in L\big(\Pi(\pi)(0)\big) \\
\Pi \models_K \neg\varphi \quad &\text{iff} \quad \Pi \not\models_K \varphi \\
\Pi \models_K \varphi_1 \vee \varphi_2 \quad &\text{iff} \quad \Pi \models_K \varphi_1 \text{ or } \Pi \models \varphi_2 \\
\Pi \models_K \mathrm{X}\,\varphi \quad &\text{iff} \quad \Pi[1,\infty] \models_K \varphi \\
\Pi \models_K \varphi_1 \,\mathrm{U}\, \varphi_2 \quad &\text{iff} \quad \text{there exists } i \geq 0 : \Pi[i,\infty] \models_K \varphi_2 \\
&\qquad\quad \text{and for all } 0 \leq j < i \text{ we have } \Pi[j,\infty] \models_K \varphi_1 \\
\Pi \models_K \exists\pi.\ \varphi \quad &\text{iff} \quad \text{there exists } p \in \mathsf{Paths}(K, \Pi(\pi')(0)) : \Pi[\pi \mapsto p] \models_K \varphi
\end{aligned}
$$

In the clause for existential quantification, π' denotes the path variable most recently added to Π (i.e., closest in scope to π). If Π is empty, let $\Pi(\pi')(0)$ be the initial state of K. It would be straightforward but tedious to further formalize this notation, so we omit the details. That clause uses another new notation, $\mathsf{Paths}(K, s)$, which is the set of paths produced by Kripke structure K beginning from state s. Formally, $\mathsf{Paths}(K, s)$ contains path p, where $p = (s_0, P_0)(s_1, P_1)\ldots$ and $P_i \in 2^{\mathsf{AP}}$, iff there exists a sequence $s_0 s_1 \ldots$ of states, such that s_0 is s, and for all $i \geq 0$, it holds that $s_{i+1} \in \delta(s_i)$ and $P_i = L(s_i)$.

Like with Traces in HyperLTL, we define $\mathsf{Paths}(K, s, \mathsf{AP}')$ as follows: We have $(s_0, P_0 \cup P_0')(s_1, P_1 \cup P_1')\ldots \in \mathsf{Paths}(K, s, \mathsf{AP}')$ iff $(s_0, P_0)(s_1, P_1)\ldots \in \mathsf{Paths}(K, s)$, and for all $i \geq 0$, it holds that $P_i' \subseteq \mathsf{AP}' \setminus \mathsf{AP}$.

We say that a Kripke structure K *satisfies* a HyperCTL* specification φ, denoted by $K \models \varphi$, if $\Pi \models_K \varphi$ holds true for the empty assignment. The *model checking problem* for HyperCTL* is to decide whether a given Kripke structure satisfies a given HyperCTL* specification.

HyperCTL* vs. HyperLTL. LTL can be characterized as the fragment of CTL* containing formulas of the form $\mathrm{A}\,\varphi$, where A is the CTL* universal path quantifier and φ contains no quantifiers. Formula $\mathrm{A}\varphi$ is satisfied in CTL* by a Kripke structure iff φ is satisfied in LTL by the traces of the Kripke structure.

A similar relationship holds between HyperLTL and HyperCTL*: HyperLTL can be characterized as the fragment of HyperCTL* containing formulas in *prenex* form—that is, a series of quantifiers followed by a quantifier-free formula. A formula φ in prenex form is satisfied in HyperCTL* by a Kripke structure iff φ is satisfied in HyperLTL by the traces of the Kripke structure. HyperCTL* is a strict generalization of HyperLTL, which extends HyperLTL with the capability to use quantified formulas as subformulas in the scope of temporal operators. For example, consider the program $(l := 0 \; [] \; l := 0) \; [] \; (l := 1 \; [] \; l := 1)$. A low-observer can infer which branch of the center-most nondeterministic choice is taken, but not which branch is taken next. This is expressed by HyperCTL* formula $\forall \pi. \; X \; \forall \pi'. \; X \, (l_\pi \leftrightarrow l_{\pi'})$. There is no equivalent HyperLTL formula.

As we show in Subsection 5.3, the temporal logic SecLTL [20] can be encoded in HyperCTL*, but not in HyperLTL. This provides further examples that distinguish HyperLTL and HyperCTL*.

5 Related Logics

We now examine the expressiveness of HyperLTL and HyperCTL* compared to several existing temporal logics: LTL, CTL*, QPTL, ETL, and SecLTL. There are many other logics that we could compare to in future work; some of those are discussed in Section 8.

5.1 Temporal Logics

HyperCTL* is an extension of CTL* and therefore subsumes LTL, CTL, and CTL*. Likewise, HyperLTL subsumes LTL. But temporal logics LTL, CTL, and CTL* cannot express information-flow policies. LTL formulas express properties of individual execution paths. All of the noninterference properties of Section 3 are properties of sets of execution paths [16, 42]. Explicit path quantification does enable their formulation in HyperLTL.

Even though CTL and CTL* have explicit path quantifiers, information-flow security policies, such as observational determinism (6), cannot be expressed with them. Consider the following fragment of CTL* semantics:

$$s \models A \, \varphi \;\; \text{iff for all } p \in \mathsf{Paths}(K, s) : \; p \models \varphi$$
$$p \models \Phi \;\;\;\; \text{iff } p(0) \models \Phi$$

Path formulas φ are modeled by paths p, and *state* formulas Φ are modeled by states s. State formula $A \, \varphi$ holds at state s when all paths proceeding from s satisfy φ. Any state formula Φ can be treated as a path formula, in which case Φ holds of the path iff Φ holds in the first state on that path. Using this semantics, consider the meaning of $AA \, \varphi$, which is the form of observational determinism (6):

$$s \models AA \, \varphi$$
$$= \text{for all } p \in \mathsf{Paths}(K, s) : \; p \models A\varphi$$
$$= \text{for all } p \in \mathsf{Paths}(K, s) \text{ and } p' \in \mathsf{Paths}(K, s) : \; p' \models \varphi$$

Note how the meaning of $AA\,\varphi$ is ultimately determined by the meaning of φ, where φ is modeled by the single path p'. Path p is ignored in determining the meaning of φ; the second universal path quantifier causes p to "leave scope." Hence φ cannot express correlations between p and p', as observational determinism requires. So CTL^* path quantifiers do not suffice to express information-flow policies. Neither do CTL path quantifiers, because CTL is a sub-logic of CTL^*. In fact, even the modal μ-calculus does not suffice to express some information-flow properties [2].

By using the self-composition construction [7,58], it is possible to express relational noninterference in CTL [7] and observational determinism in CTL^* [33]. Those approaches resemble HyperCTL*, but HyperCTL* formulas express policies directly over the original system, rather than over a self-composed system. Furthermore, the self-composition approach does not seem capable of expressing policies that require both universal and existential quantifiers over infinite executions, like noninference (5) and generalized noninterference (7). It is straightforward to express such policies in our logics.

QPTL. Quantified propositional temporal logic (QPTL) [56] extends LTL with quantification over propositions, whereas HyperLTL extends LTL with quantification over traces. Quantification over traces is more powerful than quantification over propositions, as we now show.

QPTL formulas are generated by the following grammar, where $a \in \mathsf{AP}$:

$$\psi \quad ::= \quad a \ \mid \ \neg\psi \ \mid \ \psi \vee \psi \ \mid \ X\,\psi \ \mid \ F\,\psi \ \mid \ \exists a.\ \psi$$

All QPTL connectives have the same semantics as in LTL, except for propositional quantification:

$$p \models \exists a.\psi \quad \text{iff} \quad \text{there exists } p' \in (2^{\mathsf{AP}})^\omega : \ p =_{\mathsf{AP}\setminus a} p' \text{ and } p' \models \psi \ .$$

Theorem 1. *HyperLTL subsumes QPTL, but QPTL does not subsume Hyper-LTL.*

Proof sketch. To express a QPTL formula in HyperLTL, rewrite the formula to prenex form, and rename all bound propositions with unique fresh names from a set AP'. These propositions act as free variables, which are unconstrained because they do not occur in the Kripke structure. Replace each propositional quantification $\exists a$ in the QPTL formula by a path quantification $\exists \pi_a$ in the HyperLTL formula. And replace each occurrence of a by a_{π_a}. The result is a HyperLTL formula that holds iff the original QPTL formula holds.

But not all HyperLTL formulas can be expressed in QPTL. For example, QPTL cannot express properties that require the existence of paths, such as $\exists \pi.\, X\, a_\pi$. $\qquad\square$

In Section 6, we exploit the relationship between HyperLTL and QPTL to obtain a model checking algorithm for HyperLTL.

5.2 Epistemic Logics

HyperLTL and HyperCTL* express information-flow policies by explicit quantification over multiple traces or paths. Epistemic temporal logic has also been used to express such policies [4, 11, 29, 62] by implicit quantification over traces or paths with the *knowledge* connective K of epistemic logic [22]. We do not yet know which is more powerful, particularly for information-flow policies. But we do know that HyperLTL subsumes a common epistemic temporal logic.

Define ETL (epistemic temporal logic) to be LTL with the addition of K under its perfect recall semantics [4, 22, 60]. The model of an ETL formula is a pair (K, Agts) of a Kripke structure K and a set Agts of equivalence relations on AP, called the *agents*; each relation models the knowledge of an agent. (*Interpreted systems*, rather than Kripke structures, are often used to model ETL formulas [22, 60]. Interpreted systems differ in style but can be translated to our formulation.) In the *asynchronous* semantics of ETL, $\mathsf{K}_A \psi$ holds on state i of trace $t \in \mathsf{Traces}(K)$, denoted $t, i \models \mathsf{K}_A \varphi$, iff

$$\text{for all } t' \in \mathsf{Traces}(K) : \ t[0, i] \approx_A t'[0, i] \text{ implies } t', i \models \varphi,$$

where $\approx_A$ denotes stutter-equivalence on finite traces with respect to A. In the *synchronous* semantics of ETL, stutter-equivalence is replaced by stepwise-equivalence.

The following two theorems show that HyperLTL subsumes ETL:

Theorem 2. *In the synchronous semantics, for every ETL formula ψ and every set Agts of agents, there exists a HyperLTL formula φ such that for all Kripke structures K, we have $(K, \mathsf{Agts}) \models \psi$ iff $K \models \varphi$.*

Theorem 3. *In the asynchronous semantics, for every ETL formula ψ and every set Agts of agents, there exists a HyperLTL formula φ such that for all asynchronous Kripke structures K, we have $(K, \mathsf{Agts}) \models \psi$ iff $K \models \varphi$.*

Proofs of both theorems appear in Appendix B. Theorem 3 requires an additional assumption that K is an *asynchronous* Kripke structure, i.e. that it can always stutter in its current state and that it is indicated in an atomic proposition whether the last state was a stuttering step.

HyperLTL and ETL have the same worst-case complexity for model checking, which is non-elementary. But, as we show in Section 6, the complexity of our model checking algorithm on the information-flow policies of Section 3 is much better—only NLOGSPACE (for observational determinism, declassification, and quantitative noninterference for a fixed number of bits) or PSPACE (for noninference and generalized noninterference) in the size of the system. For those policies in NLOGSPACE, that complexity, unsurprisingly, is as good as algorithms based on self-composition [7]. This ability to use a general-purpose, efficient HyperLTL model checking algorithm for information flow seems to be an improvement over encodings of information flow in ETL.

5.3 SecLTL

SecLTL [20] extends temporal logic with the *hide* modality $\mathcal{H}$, which allows to express information flow properties such as noninterference [26]. The semantics of SecLTL is defined in terms of labeled transition system, where the edges are labeled with valuations of the set of variables. The formula $\mathcal{H}_{H,O}\varphi$ specifies that the current valuations of a subset H of the input variables I are kept secret from an attacker who may observe variables in O until the release condition φ becomes true. The semantics is formalized in terms of a set of *alternative paths* to which the *main path* is compared:

$$\mathsf{AltPaths}(p, H) = \{p' \in \mathsf{Paths}(K_M, p[0]) \mid p[1] =_{I\setminus H} p'[1] \text{ and } p[2,\infty] =_I p'[2,\infty]\}$$

where K_M is the equivalent Kripke structure for the labeled transition system M (we will explain the translation later in this section.) A path p satisfies the SecLTL formula $\mathcal{H}_{H,O}\varphi$, denoted by $p \models \mathcal{H}_{H,O}\varphi$, iff

$$\forall p' \in \mathsf{AltPaths}(p, H). \left(p =_O p', \text{ or there exists } i \geq 0 : \atop p[i,\infty] \models_K \varphi \text{ and } p[1,i-1] =_O p'[1,i-1]\right)$$

A labeled transition system M satisfies a SecLTL formula ψ, denoted by $M \models \psi$, if every path p starting in the initial state satisfies ψ.

SecLTL can express properties like the dynamic creation of secrets discussed in Section 4, which cannot be expressed by HyperLTL. However, SecLTL is subsumed by HyperCTL*. To encode the hide modality in HyperCTL*, we first translate M into a Kripke structure K_M, whose states are labeled with the valuation of the variables on the edge leading into the state. The initial state is labeled with the empty set. In the modified system, $L(p[1])$ corresponds to the current labels. We encode $\mathcal{H}_{H,O}\varphi$ as the following HyperCTL* formula:

$$\forall \pi'. \pi[1] =_{I\setminus H} \pi'[1] \wedge \mathrm{X}\left(\pi[1] =_O \pi'[1] \ \mathrm{W} \ (\pi[1] \neq_I \pi'[1] \vee \varphi)\right)$$

Theorem 4. *For every SecLTL formula ψ and transition system M, there is a HyperCTL* formula φ such that $M \models \psi$ iff $K_M \models \varphi$.*

The model checking problem for SecLTL is PSPACE-hard in the size of the Kripke structure [20]. The encoding of SecLTL specifications in HyperCTL* implies that the model checking problem for HyperCTL* is also PSPACE-hard (for a fixed specification of alternation depth ≥ 1), as claimed in Theorem 6.

6 Model Checking and Satisfiability

In this section we exploit the connection between HyperCTL* and QPTL to obtain a model checking algorithm for HyperCTL* and study its complexity. We identify a hierarchy of fragments of HyperCTL* characterized by the number of quantifier alternations. This hierarchy allows us to give a precise characterization of the complexity of the model checking problem. The fragment of formulas with quantifier alternation depth 0 includes already many formulas of interest and our result provides an NLOGSPACE algorithm in the size of the Kripke structure.

Definition 1 (Alternation Depth). *A HyperCTL* formula φ in NNF has alternation depth 0 plus the highest number of alternations from existential to universal and universal to existential quantifiers along any of the paths of the formula's syntax tree starting in the root. Occurrences of U and R count as an additional alternation.*

Theorem 5. *The model checking problem for HyperCTL* specifications φ with alternation depth k on a Kripke structure K is complete for NSPACE$(g_c(k, |\varphi|))$ and it is in NSPACE$(g_c(k-1, |K|))$ for some $c > 0$.*

The function $g_c(x, y)$ denotes a tower of exponentials of height x with argument y: $g_c(0, y) = y$ and $g_c(x, y) = c^{g(x-1,y)}$. NSPACE$(g_c(x, y))$ denotes the class of languages accepted by a Turing machine bounded in space by $O(g_c(x, y))$. Abusing notation, we define $g_c(-1, y) = \log y$ and NSPACE$(\log y) =$ NLOGSPACE in y.

Proof. Both directions, the lower bound and the upper bound, are based on the complexity of the satisfiability problem for QPTL formulas φ in prenex normal form and with alternation depth k, which is complete for NSPACE$(g(k, |\varphi|))$ [56].

For the upper bound on the HyperCTL* model checking complexity, we first translate until operators $\psi \,U\, \psi'$ as $\exists t.\ t \wedge G(t \to \psi' \vee (\psi \wedge X\,t)) \wedge \neg G\,t$. Let $\psi(K, \mathsf{AP}')$ encode a Kripke structure K, where $K = (S, s_0, \delta, \mathsf{AP}, L)$, as a QPTL formula (cf. [37]) using the set of atomic propositions AP', which must contain atomic propositions replacing those of AP and additional atomic propositions to describe the states S. The formula $\psi(K, \mathsf{AP}')$ is linear in $|K|$ and does not require additional quantifiers.

HyperCTL* path quantifiers $\exists \pi.\varphi$ and $\forall \pi.\varphi$ are then encoded as $\exists \mathsf{AP}_\pi.$ $\psi(K, \mathsf{AP}_\pi) \wedge \varphi_{\mathsf{AP}_\pi}$ and $\forall \mathsf{AP}_\pi.\psi(K, \mathsf{AP}_\pi) \to \varphi_{\mathsf{AP}_\pi}$, where AP_π is a set of *fresh* atomic propositions including a copy of AP and additional atomic propositions to describe the states S. The formula $\varphi_{\mathsf{AP}_\pi}$ is obtained from φ by replacing all atomic propositions referring to path π by their copies in AP_π. Atomic propositions in the formula that are not in AP (i.e. their interpretation is not fixed in K) need to be added to the sets AP_π accordingly.

For the lower bound, we reduce the satisfiability problem for a given QPTL formula φ in prenex normal form to a model checking problem $K \models \varphi'$ of HyperCTL*. We assume, without loss of generality, that φ is closed (if a free proposition occurs in φ, we bind it with an existential quantifier) and each quantifier in φ introduces a different proposition.

The Kripke structure K consists of two states $S = \{s_0, s_1\}$, is fully connected $\delta(s) = S$ for all $s \in S$, and has a single atomic proposition $\mathsf{AP} = \{p\}$. The states are labeled as follows: $L(s_0) = \emptyset$ and $L(s_1) = \{p\}$. Essentially, paths in K can encode all sequences of valuations of a variable in QPTL. To obtain the HyperCTL* formula, we now simply replace every quantifier in the QPTL formula with a path quantifier. The only technical problem left is that quantification in QPTL allows to choose freely the value of p in the current state, while path quantification in HyperCTL* only allows the path to differ in the next state. We solve the issue by shifting the propositions using a next operator. $\square$

Lower bounds in $|K|$. An NLOGSPACE *lower* bound in the size of the Kripke structure for fixed specifications with alternation depth 0 follows from the non-emptiness problem of non-deterministic Büchi automata. For alternation depth 1 and more we can derive PSPACE hardness in the size of the Kripke structure from the encoding of the logic SecLTL into HyperCTL* (see Subsection 5.3).

The result can easily be transferred to HyperLTL, since in the SecLTL formula that is used to prove PSPACE hardness, the Hide operator does not occur in the scope of temporal operators and hence the translation yields a HyperLTL formula.

Theorem 6. *For HyperLTL formulas the model checking problem is hard for PSPACE in the size of the system.*

A Remark on Efficiency The use of the standard encoding of the until operator in QPTL with an additional quantifier shown above is, in certain cases, wasteful. The satisfiability of QPTL formulas can be checked with an automata-theoretic construction, where we first transform the formula into prenex normal form, then generate a nondeterministic Büchi automaton for the quantifier-free part of the formula, and finally apply projection and complementation to handle the existential and universal quantifiers. In this way, each quantifier alternation, including the alternation introduced by the encoding of the until operators, causes an exponential blow-up. However, if an until operator occurs in the quantifier-free part, the standard transformation of LTL formulas to nondeterministic Büchi automata handle this until operator without requiring a quantifier elimination, resulting in an exponential speedup.

Using this insight, the model checking complexity for many of the formulas presented above and in Section 3 can be reduced by one exponent. Additionally, the complexity with respect to the size of the system reduces to NLOGSPACE for HyperCTL* formulas where the leading quantifiers are all of the same type and are followed by some quantifier-free formula which may contain until operators without restriction. Observational determinism and the declassification policy discussed in Section 3 are examples for specifications in this fragment. This insight was used for the prototype implementation described in Section 7 and it avoids an additional complementation step for noninference (5).

Satisfiability. The positive result regarding the model checking problem for HyperCTL* does not carry over to the satisfiability problem. The *finite-state satisfiability problem* consists of the existence of a finite model, while the *general satisfiability problem* asks for the existence of a possibly infinite model.

Theorem 7. *For HyperCTL*, finite-state satisfiability is hard for Σ_1^0 and general satisfiability is hard for Σ_1^1.*

In the proof, located in Appendix C, we reduce the LTL synthesis problem of distributed systems to the satisfiability problem of HyperCTL*.

7 Prototype Model Checker

The results of the previous section yield a model checking algorithm for all of HyperCTL*. But most of our information-flow policy examples do not require the full expressiveness of HyperCTL*. In fact, we have been able implement a prototype model checker for an expressive fragment of the logic mostly using off-the-shelf components.

Define *HyperLTL$_2$* as the fragment of HyperLTL (and of HyperCTL*) in which the series of quantifiers at the beginning of a formula may involve at most one alternation. Every formula in HyperLTL$_2$ thus may begin with at most two (whence the name) kinds of quantifiers—a sequence of $\forall$'s followed by a sequence of $\exists$'s, or vice-versa. For example, $\exists\pi.\psi$ and $\forall\pi_1.\forall\pi_2.\exists\pi_3.\psi$ are allowed, but $\forall\pi_1.\exists\pi_2.\forall\pi_3.\psi$ is not. HyperLTL$_2$ suffices to express all the security policies formulated in Section 3. (Another logic for hyperproperties, $\mathcal{IL}_\mu^k$ [45], similarly restricts fixpoint operator alternations with no apparent loss in expressivity for security policies.)

Our model checking algorithm for HyperLTL$_2$, detailed in Appendices D and E, is based on algorithms for LTL model checking [24, 25, 64]. Those LTL algorithms determine whether a Kripke structure satisfies an LTL formula by performing various automata constructions and by checking language containment. Our algorithm likewise uses automata constructions and language containment, as well as self composition [7, 58] and a new *projection* construction.

We prototyped this algorithm in about 3,000 lines of OCaml code. Our prototype accepts as input a Kripke structure and a HyperLTL$_2$ formula, then constructs the automata required by our algorithm, and outputs a countermodel if the formula does not hold of the structure. For automata complementation, our prototype outsources to GOAL [59], an interactive tool for manipulating Büchi automata. We have used the prototype to verify noninference (5), observational determinism (6), and generalized noninterference (7) for small Kripke structures (up to 10 states); running times were about 10 seconds or less.

Since our algorithm uses automata complementation, the worst-case running time is exponential in the size of the Kripke structure's state space and doubly exponential in the formula size. So as one might expect, our prototype currently does not scale to medium-sized Kripke structures (up to 1,000 states). But our purpose in building this prototype was to demonstrate a proof-of-concept for model checking of hyperproperties. We conjecture that practical symbolic model checking algorithms, such as BMC and IC3, could be used to scale up our approach to real-world systems.

8 Related Work

McLean [42] formalizes security policies as closure with respect to *selective interleaving functions*. He shows that trace properties cannot express security policies such as noninterference and average response time, because those are not properties of single execution traces. Mantel [38] formalizes security policies with *basic security predicates*, which stipulate *closure conditions* for trace sets.

Clarkson and Schneider [16] introduce *hyperproperties*, a framework for expressing security policies. Hyperproperties are sets of trace sets, and are able to formalize security properties such as noninterference, generalized noninterference, observational determinism and average response time. Clarkson and Schneider use second-order logic to formulate hyperproperties. That logic isn't verifiable, in general, because it cannot be effectively and completely axiomatized. Fragments of it, such as HyperLTL and HyperCTL*, can be verified.

Alur et al. [2] show that modal μ-calculus is insufficient to express all *opacity* policies [9], which prohibit observers from discerning the truth of a predicate. (Alur et al. [2] actually write "secrecy" rather than "opacity.") Simplifying definitions slightly, a trace property P is *opaque* iff for all paths p of a system, there exists another path p' of that system, such that p and p' are low-equivalent, and exactly one of p and p' satisfies P. Noninference (5) is an opacity policy [52] that HyperLTL can express.

Huisman et al. reduce observational-determinism properties to properties in CTL* [33] and in modal μ-calculus [32] on a self-composed system. Barthe et al. use self composition to verify observational determinism [7] and noninterference [6] on terminating programs. Van der Meyden and Zhang [63] reduce a broader class of information-flow policies to safety properties on a self-composed system expressible in standard linear and branching time logics, and use model checking to verify noninterference policies. Their methodology requires customized model checking algorithms for each security policy, whereas this work proposes a single algorithm for all policies.

Balliu et al. [4] use a linear-time temporal epistemic logic to specify many declassification policies derived from noninterference. Their definition of noninterference, however, seems to be that of observational determinism (6). They do not consider any information-flow policies involving existential quantification, such as noninference. They also do not consider systems that accept inputs after execution has begun. Halpern and O'Neill [29] use a similar temporal epistemic logic to specify *secrecy* policies, which subsume many definitions of noninterference; they do not pursue model checking algorithms.

Alur et al. [1] discuss branching-time logics with *path equivalences* that are also able to express certain security properties. The authors introduce operators that resemble the knowledge operator of epistemic logics. As the logics build on branching-time logics they are not subsumed by HyperLTL. The relationship to HyperCTL* is still open.

Milushev and Clarke [44–46] propose three logics for hyperproperties:

– *Holistic hyperproperty logic* $\mathcal{HL}$, which is based on coinductive predicates over streams. Holistic hyperproperties "talk about whole traces at once; their specifications tend to be straightforward, but they are difficult to reason about, exemplified by the fact that no general approach to verifying such hyperproperties exists" [44]. HyperLTL and HyperCTL* are logics that talk about whole traces at once, too; and they have straightforward specifications as well as a general approach to verification.

- *Incremental hyperproperty logic $\mathcal{IL}$* is a fragment of *least fixed-point logic* [8]. There is a manual verification methodology for $\mathcal{IL}$ [44], but no automated decision procedure.
- Another incremental hyperproperty logic $\mathcal{IL}_\mu^k$, a fragment of polyadic modal μ-calculus [3] that permits at most one quantifier alternation (a greatest fixed-point followed by a least fixed-point). There is an automated model checking technique [45] for $\mathcal{IL}_\mu^k$ based on *parity games*. That technique has been prototyped and applied to a few programs.

All these logics suffice to express security policies such as noninterference and generalized noninterference. Like our logics, the exact expressive limitation is still an open problem.

As the preceding discussion makes clear, the expressiveness of HyperLTL and HyperCTL* versus several other logics is an open question. It's possible that some of those logics will turn out to be more expressive or more efficiently verifiable than HyperLTL or HyperCTL*. It's also possible that it will turn out to be simply a matter of taste which style of logic is more suitable for hyperproperties. The purpose of this paper was to explore one design option: a familiar syntax, based on widely-used temporal logics, that can straightforwardly express well-known hyperproperties.

9   Concluding Remarks

In designing a logic for hyperproperties, starting with HyperLTL was natural, because hyperproperties are sets of trace sets, and LTL uses trace sets to model programs. From HyperLTL, the extension to HyperCTL* was also natural: we simply removed the restrictions on where quantifiers could appear. The curtailment to HyperLTL$_2$ was also natural, because it was the fragment needed to express information-flow security policies. HyperLTL$_2$ permits up to one quantifier alternation, but what about hyperproperties with more? We do not yet know of any security policies that are examples. As Rogers [49] writes, "The human mind seems limited in its ability to understand and visualize beyond four or five alternations of quantifier. Indeed, it can be argued that the inventions... of mathematics are devices for assisting the mind in dealing with one or two additional alternations of quantifier." For practical purposes, we might not need to go much higher than one quantifier alternation.

References

1. R. Alur, P. Cerný, and S. Chaudhuri. Model checking on trees with path equivalences. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 664–678, Mar. 2007.

2. R. Alur, P. Černý, and S. Zdancewic. Preserving secrecy under refinement. In *Proc. International Colloquium on Automata, Languages and Programming*, pages 107–118, 2006.

3. H. R. Andersen. A polyadic modal mu-calculus. Technical Report 1994-145, Technical University of Denmark (DTU), 1994.

4. M. Balliu, M. Dam, and G. L. Guernic. Epistemic temporal logic for information flow security. In *Proc. Workshop on Programming Languages and Analysis for Security*, June 2011.

5. A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

6. G. Barthe, J. M. Crespo, and C. Kunz. Beyond 2-safety: asymmetric product programs for relational program verification. In *Proc. Logical Foundations of Computer Science*, pages 29–43, Jan. 2013.

7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proc. IEEE Computer Security Foundations Workshop*, pages 100–114, June 2004.

8. J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, pages 721–756. Elsevier, Amsterdam, The Netherlands, 2007.

9. J. Bryans, M. Koutny, L. Mazaré, and P. Y. A. Ryan. Opacity generalised to transition systems. In *Proc. Formal Aspects in Security and Trust*, pages 81–95, July 2005.

10. J. R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Logic, Methodology, and Philosophy of Science*, pages 1–12, 1962.

11. R. Chadha, S. Delaune, and S. Kremer. Epistemic logic for the applied pi calculus. In *Proc. Formal Techniques for Distributed Systems*, pages 182–197, June 2009.

12. A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proc. Computer Aided Verification*, pages 495–499, July 1999.

13. D. Clark, S. Hunt, and P. Malacaria. Quantified interference for a while language. *Electronic Notes in Theoretical Computer Science*, 112:149–166, Jan. 2005.

14. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, Dec. 1999.

15. M. R. Clarkson, A. C. Myers, and F. B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, Oct. 2009.

16. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

17. M. Cohen and A. Lomuscio. Non-elementary speed up for model checking synchronous perfect recall. In *Proc. European Conference on Artificial Intelligence*, pages 1077–1078, Aug. 2010.

18. B. Cook, E. Koskinen, and M. Vardi. Temporal property verification as a program analysis task. In *Proc. Computer Aided Verification*, pages 333–348, July 2011.

19. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1(2/3):275–288, Oct. 1992.

20. R. Dimitrova, B. Finkbeiner, M. Kovács, M. N. Rabe, and H. Seidl. Model checking information flow in reactive systems. In *Proc. International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 169–185, Jan. 2012.

21. E. A. Emerson and J. Y. Halpern. "Sometimes" and "not never" revisited: On branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, Jan. 1986.

22. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. MIT Press, Cambridge, 1995.

23. R. Focardi and R. Gorrieri. Classification of security properties (Part I: Information flow). In *International School on Foundations of Security Analysis and Design: Tutorial Lectures*, pages 331–396, 2000.

24. P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In *Proc. Computer Aided Verification*, pages 53–65, July 2001.

25. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. Protocol Specification, Testing and Verification*, pages 3–18, June 1995.

26. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, Apr. 1982.

27. J. W. Gray, III. Toward a mathematical foundation for information flow security. In *Proc. IEEE Symposium on Security and Privacy*, pages 210–34, May 1991.

28. J. W. Gray, III and P. F. Syverson. A logical approach to multilevel security of probabilistic systems. *Distributed Computing*, 11(2):73–90, 1998.

29. J. Y. Halpern and K. R. O'Neill. Secrecy in multiagent systems. *ACM Transactions on Information and System Security*, 12(1):5:1–47, Oct. 2008.

30. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, Dec. 2009.

31. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

32. M. Huisman and H. Blondeel. Model-checking secure information flow for multi-threaded programs. In *Proc. Theory of Security and Applications*, pages 148–165, Mar. 2011.

33. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. IEEE Computer Security Foundations Workshop*, pages 3–15, July 2006.

34. B. Köpf and D. Basin. An information-theoretic model for adaptive side-channel attacks. In *Proc. ACM Conference on Computer and Communications Security*, pages 286–296, Oct. 2007.

35. L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

36. P. Li and S. Zdancewic. Downgrading policies and relaxed noninterference. In *Proc. Principles of Programming Languages*, pages 158–170, Jan. 2005.

37. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.

38. H. Mantel. Possibilistic definitions of security—an assembly kit. In *Proc. IEEE Computer Security Foundations Workshop*, pages 185–199, July 2000.

39. D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symposium on Security and Privacy*, pages 177–186, Apr. 1988.

40. D. McCullough. A hookup theorem for multilevel security. *Proc. IEEE Transactions on Software Engineering*, 16(6):563–568, June 1990.

41. J. McLean. Proving noninterference and functional correctness using traces. *Journal of Computer Security*, 1(1):37–58, 1992.

42. J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Security and Privacy*, pages 79–93, Apr. 1994.

43. J. K. Millen. Unwinding forward correctability. In *Proc. IEEE Computer Security Foundations Workshop*, pages 2–10, June 1994.

44. D. Milushev and D. Clarke. Towards incrementalization of holistic hyperproperties. In *Proc. Conference on Principles of Security and Trust*, pages 329–348, 2012.

45. D. Milushev and D. Clarke. Incremental hyperproperty model checking via games. In *Proc. Nordic Conference on Secure IT Systems*, pages 247–262, Oct. 2013.

46. D. V. Milushev. *Reasoning about Hyperproperties*. PhD thesis, Katholieke Universiteit Leuven, June 2013.

47. A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

48. A. Pnueli. The temporal logic of programs. In *Proc. Foundations of Computer Science*, pages 46–57, Sept. 1977.

49. H. Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, Massachusetts, 1987.

50. A. W. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–127, May 1995.

51. R. Rosner. *Modular synthesis of reactive systems*. PhD thesis, The Weizmann Institue of Science, 1992.

52. P. Y. A. Ryan and T. Peacock. Opacity—further insights on an information flow property. Technical Report CS-TR-958, Newcastle University, Apr. 2006.

53. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

54. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 255–269, 2005.

55. S. Safra. *Complexity of Automata on Infinite Objects*. PhD thesis, The Weizmann Institue of Science, 1989.

56. A. P. Sistla, M. Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with appplications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

57. G. Smith. On the foundations of quantitative information flow. In *Proc. Conference on Foundations of Software Science and Computation Structures*, pages 288–302, Mar. 2009.

58. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Proc. Static Analysis*, pages 352–367, Sept. 2005.

59. Y. Tsay, Y. Chen, M. Tsai, K. Wu, and W. Chan. GOAL: a graphical tool for manipulating Büchi automata and temporal formulae. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, pages 466–471, Mar. 2007.

60. R. van der Meyden. Axioms for knowledge and time in distributed systems with perfect recall. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 448–457, 1993.

61. R. van der Meyden and N. V. Shilov. Model checking knowledge and time in systems with perfect recall. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, pages 432–445, Dec. 1999.

62. R. van der Meyden and T. Wilke. Preservation of epistemic properties in security protocol implementations. In *Proc. ACM Conference on Theoretical Aspects of Rationality and Knowledge*, pages 212–221, 2007.

63. R. van der Meyden and C. Zhang. Algorithmic verification of noninterference properties. *Electronic Notes in Theoretical Computer Science*, 168:61–75, Feb. 2007.
64. M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proc. Banff Higher Order Workshop*, pages 238–266, Aug. 1996.
65. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.
66. P. Wolper. Constructing automata from temporal logic formulas: A tutorial. In *Lectures on Formal Methods and Performance Analysis, First EEF/Euro Summer School on Trends in Computer Science*, pages 261–277. Springer, 2000.
67. H. Yasuoka and T. Terauchi. On bounding problems of quantitative information flow. In *Proc. European Symposium on Research in Computer Security*, pages 357–372, Sept. 2010.
68. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

A   Event-based Execution Model

A.1   Goguen and Meseguer's noninterference

Noninterference. A point of reference for most of the literature on information flow is the definition of *noninterference* that was introduced by Goguen and Meseguer in 1982 [26]. In this subsection, we show how to express noninterference in a simple HyperLTL formula.

The system model used in [26], which we will refer to as *deterministic state machines*, operates on commands $c \in C$ that are issued by different users $u \in U$. The evolution of a deterministic state machine is governed by the transition function $\mathsf{do} : S \times U \times C \to S$ and there is a separate observation function $\mathsf{out} : S \times U \to Out$ that for each user indicates what he can observe.

We define standard notions on sequences of users and events. For $w \in (U \times C)^*$ and $G \subseteq U$ let $|w|_G$ denote the projection of w to the commands issued by the users in G. Further, we extend the transition function do to sequences, $\mathsf{do}(s, (u, c).w) = \mathsf{do}(\mathsf{do}(s, u, c), w)$, where the dot indicates concatenation. Finally, we extend the observation function out to sequences w, indicating the observation *after* w: $\mathsf{out}(w, G) = \mathsf{out}(\mathsf{do}(s_0, w), G)$. *Noninterference* is then defined as a property on systems M. A set of users $G_H \subseteq U$ does not interfere with a second group of users $G_L \subseteq U$, if

$$\forall w \in (U \times C)^*. \ \mathsf{out}(w, G_L) = \mathsf{out}(|w|_{G_H}, G_L)$$

That is, we ask whether the same output would be produced by the system, if all actions issued by any user in G_H were removed.

Encoding GM's System Model First, we need to map their system model into Kripke structures as used for the formulation of HyperCTL* (which, obviously, must not solve problem by itself). We choose an intuitive encoding of state machines in Kripke structures that indicates in every state (via atomic propositions)

which observations can be made for the different users, and also which action was issued last, including the responsible user.

We choose a simple translation that maps a state machine $M = (S, U, C, Out,$ out, do, $s_0)$ to the Kripke structure $K = (S', s_0', \delta, \mathsf{AP}, L)$, where $S' = \{s_0\} \cup S \times U \times C$, $s_0' = s_0$, $\mathsf{AP} = U \times C \cup U \times Out$, and the labeling function is defined as $L(s_0) = \{(u, \mathsf{out}(s_0, u)) \mid u \in U\}$ for the initial state and $L((s, u, c)) = \{(u, c)\} \cup \{(u', \mathsf{out}(s, u')) \mid u' \in U\}$ for all other states.

The transition function is defined as

$$\delta(s, u, c) = \{(s', u', c') \mid \mathsf{do}(s, u', c') = s', \ u' \in U, \ c' \in C\}$$

Each state (except for the initial state) has labels indicating the command that was issued last, and the user that issued the command. The remaining labels denote the observations that the individual users can make in this state.

To access these two separate pieces of information, we introduce functions $In : S \to U \times C$, which is not defined for s_0, and $\mathsf{out} : S \times U \to Out$ (by abusing notation slightly), with their obvious meanings.

In this system model let $s =_{U \setminus G_H} s'$ mean that if we entered one of the states s and s' with a user not in G_H the users and commands must be identical. If both states are entered by users in G_H then the commands may be different. The output equality on states $s =_{O, G_L} s'$, shall refer to the observations the users G_L can make at this state. Then, noninterference can be expressed as follows:

$$\forall \pi. \forall \pi'. \ \pi[0] =_{U \setminus G_H} \pi'[0] \quad \text{W} \quad \Big(\pi[0] \neq_{U \setminus G_H} \pi'[0] \ \wedge$$
$$\big(\ \pi[0] \in H \ \wedge \ \mathrm{G}(\mathrm{X} \pi[0] =_{U \setminus G_H} \pi'[0]) \ \to \ \mathrm{G}(\mathrm{X} \pi[0] =_{O, G_L} \pi'[0]) \ \big) \Big)$$

Theorem 8. *There is a HyperLTL formula and an encoding $K(M)$ of state machines into Kripke structures such that for every state machine M, and groups of users G_H and G_L it holds $K(M) \models \varphi_{NI}(G_H, G_L)$ iff G_H does not interfere with G_L in M.*

Proof. The formula pattern $\pi[0] =_{U \setminus G_H} \pi'[0] \ \text{W} \ \big(\pi[0] \neq_{U \setminus G_H} \pi'[0] \wedge \varphi \big)$ implies that φ is applied exactly at the first position at which π and π' differ in their input (except, possibly, on the input of users in G_H). As, for a fixed path π, we quantify over all paths π' the subformula φ is hence applied to every position of every path π.

The subformula φ then requires that, if from that position on π's input differs from the input of π' only in that it has an additional (secret) action by some user in G_H, both paths must look equivalent from the view point of the users in G_L. The interesting part here is the use of the next operator, as it enables the comparison of different positions of the traces. Thus, we compare path π to all other paths that have one secret action less than itself. Hence, by transitivity of equivalence, we compare all paths π to a version of itself that is stripped of all secret actions. $\qquad\square$

A.2 Observational determinism

Theorem 9. *There is an encoding of programs as defined in [33] into Kripke structures, such that HyperCTL* can express observational determinism.*

Proof. Huisman et al. [33] define observational determinism over programs in a simple while language, which is very similar to the execution model of [68]. First, they define a special version of self-composition on their programs that allows both sides to move independently by introduces stuttering steps. Then, they encode programs into Kripke structures that do not only maintain the current state, but also remember the last state. The CTL* formula is then proved to precisely express observational determinism. Since HyperCTL* subsumes CTL*, we can make use of the same encoding of programs into Kripke structures, but we leave out the self-composition operation on programs. In order to re-enable the correct synchronization of paths, however, we have to introduce stutter steps for single, not self-composed that is, programs.

We prepend the CTL* formula with two path quantifiers over paths π_1 and π_2 and we let those propositions of the formula that referred to the two copies of the program now refer to the two paths, respectively. □

A.3 Declassification

Likewise, HyperLTL can also express declassification properties in a programming language setting.

Corollary 1. *HyperLTL can express the declassification properties discussed in [4].*

The result follows from the encoding [4] of declassification properties into epistemic temporal logic [22] and the fact that HyperLTL subsumes epistemic temporal logic (see Section 5).

B Epistemic Logics

We start with the proof that HyperCTL* subsumes epistemic temporal logics under the assumption of synchronous time. That is, we assume the Kripke structure to have an atomic proposition that does nothing but switching its value in every step and that is observable by every agent. By this, the stutter-equivalent comparison collapses to a step-wise comparison.

Theorem 10. *Under synchronous time semantics, for every epistemic temporal logic formula ψ and every set of agents Agts, there is a HyperLTL formula φ such that for all Kripke structures K it holds $(K, \mathsf{Agts}) \models \psi$ iff $K \models \varphi$.*

Proof. We start by considering the extension of HyperLTL with the *knowledge* operator with the semantics above. We prepend a universal path quantifier to the formula ψ and apply a stepwise transformation to eliminate all knowledge

operators. (Note that the semantics relations $\pi, i \models \psi$ of linear time epistemic logics and and $\Pi \models_K \varphi$ of HyperCTL* have different parameters, but it is plain how to join them into a single relation $\Pi, i \models_K \varphi$.)

Let φ be a HyperLTL formula in NNF that possibly has knowledge operators. For brevity, we summarize the leading quantifiers of φ with $\mathbf{Q}$, such that $\varphi = \mathbf{Q}.\varphi'$ where φ' is quantifier-free. Let t and u be propositions that φ does not refer to and that are not in the alphabet of the Kripke structure. In case a knowledge operator $\mathsf{K}_A\psi'$ occurs in φ' with positive polarity, translate φ into the following HyperLTL formula:

$$\mathbf{Q}.\exists\pi.\ \forall\pi'.\ \forall\pi''.\ \varphi'|_{\mathsf{K}_A\psi' \to u_\pi}\ \wedge\ \big((t_{\pi'}\ \mathrm{U}\ \mathrm{G}\,\neg t_{\pi'})\ \to$$
$$\mathrm{G}(t_{\pi'} \to (\pi^*[0] =_A \pi''[0]))\ \to\ \mathrm{G}(t_{\pi'} \wedge u_\pi \to [\psi']_{\pi''})\big)$$

and, if the knowledge operator occurs negatively,

$$\mathbf{Q}.\exists\pi.\ \forall\pi'.\ \exists\pi''.\ \varphi'|_{\neg\mathsf{K}_A\psi' \to u_\pi}\ \wedge\ \big((t_{\pi'}\ \mathrm{U}\ \mathrm{G}\,\neg t_{\pi'}) \to$$
$$\mathrm{G}(t_{\pi'} \to (\pi^*[0] =_A \pi''[0]))\ \wedge\ \mathrm{F}(t_{\pi'} \wedge u_\pi \wedge \neg[\psi']_{\pi''})\big)$$

where $\varphi'|_{\mathsf{K}_P\psi \to u_\pi}$ denotes that in φ' one positive or one negative occurrence, respectively, of the knowledge operator $\mathsf{K}_P\psi$ is replaced by proposition u, and where π^* is the path on which the knowledge operator that we currently eliminate was applied on (which could be a different path to the first one, in case of nested knowledge operators). We repeat this transformation until no knowledge operators remain.

Paths π and π' are only used to carry the information about proposition u and t, respectively. That is, we use these paths to quantify over a sequence of atomic propositions, the same way the QPTL quantification works. The sequence of the atomic proposition u indicates at which positions of a given trace the knowledge operator needs to be evaluated. There can be multiple such sequences (e.g. for a formula $\mathrm{F}(\mathsf{K}_A\varphi'')$) that would make the formula satisfied, so the sequence is existentially quantified ($\exists\pi$). We need to fix one particular of these sequences of applications of the knowledge operator in advance, because we cannot go back in time to the initial state where the quantification knowledge operator happens.

The sequence of atomic propositions t is restricted to be true initially until it is globally false. In this way, we select a point in time until which the paths π_n and π'' are required to be equal w.r.t. the observations of agent A ($\mathrm{G}(t_{\pi'} \to (\pi_n[0] =_A \pi''[0]))$). At this selected point in time, if also the knowledge operator is required to hold (indicated by u) we need to check that the subformula ψ' of the knowledge operator is checked on the alternative path. We relaxed this condition to simplify the formula. Instead of the point in the sequence where t switches from true to false, we apply subformula ψ' on *all* positions before the that point ($\mathrm{G}(t_{\pi'} \wedge u_\pi \to [\psi']_{\pi''})$). Since the sequences of t are all-quantified, this does not affect the meaning of the formula. $\qquad\square$

Note that using past operator would simplify the encoding substantially, since we could more easily refer to the initial state where all paths in epistemic temporal logics branch.

We now discuss the adaptations necessary to extend the encoding from the proof of Theorem 10 to non-synchronous systems. For asynchronous systems it is natural to assume that they may stutter in their current state. In this discussion, we also assume that it is visible whether an asynchronous system has last performed a stuttering step, or whether it performed an action. (It may also be possible for a Kripke structure to perform an action that stays in a state.) We call a Kripke structure with these properties an *asynchronous Kripke structure.*

The following proof of Theorem 3 is also a good example for the general treatment of asynchronous executions in HyperCTL*.

Theorem 11. *For every epistemic temporal logic formula ψ and every set of agents* Agts*, there is a HyperLTL formula φ such that for all asynchronous Kripke structures K it holds $(K, \mathsf{Agts}) \models \psi$ iff $K \models \varphi$.*

Proof. We assume that the atomic proposition stutter describes that no action was performed in the last step.

Since there always is the option to stutter in a stat, the number of paths we quantify over is larger. The idea of the proof is to restrict this enlarged set of paths to those that do not stutter forever ($\mathsf{progress}(\pi) = \mathrm{G\,F}\,\neg\mathrm{stutter}_\pi$) and that synchronize correctly $\big(\mathsf{synch}(\pi, \pi') = \mathrm{G}\left(\pi[0] \neq_P \mathrm{X}\,\pi[0] \ \Leftrightarrow \ \pi'[0] \neq_P \mathrm{X}\,\pi'[0]\right)\big)$. By requiring the correct synchronization, we choose an alignment where changes in the observations happen in both paths at the same points in the sequences. That is, the remaining positions must be filled with stuttering steps.

The encoding from the proof of Theorem 10 is now modified as follows for positively occurring knowledge operators:

$$\mathbf{Q}.\exists\pi.\ \varphi'|_{\mathsf{K}_A\psi' \to u_\pi} \ \wedge \ \big(\forall\pi'.\ (t_{\pi'}\ \mathrm{U}\ \mathrm{G}\,\neg t_{\pi'})\ \to$$
$$\forall\pi''.\ \mathsf{progress}(\pi'')\ \wedge\ \mathsf{synch}(\pi,\pi'')\ \wedge$$
$$\mathrm{G}(t_{\pi'} \to (\pi^*[0] =_A \pi''[0]))\ \to\ \mathrm{G}(t_{\pi'} \wedge u_\pi \to [\psi']_{\pi''})\big)$$

and, for negatively occurring knowledge operators:

$$\mathbf{Q}.\exists\pi.\ \varphi'|_{\mathsf{K}_A\psi' \to u_\pi} \ \wedge \ \big(\forall\pi'.\ (t_{\pi'}\ \mathrm{U}\ \mathrm{G}\,\neg t_{\pi'})\ \to$$
$$\exists\pi''.\ \mathsf{progress}(\pi'')\ \wedge\ \mathsf{synch}(\pi,\pi'')\ \wedge$$
$$\mathrm{G}(t_{\pi'} \to (\pi^*[0] =_A \pi''[0]))\ \wedge\ \mathrm{F}(t_{\pi'} \wedge u_\pi \wedge \neg[\psi']_{\pi''})\big)$$

For a positively occurring knowledge operator $\mathcal{K}_A\psi'$, we prove that given two executions π_1 and π_2, if the sub-formula ψ' must hold on π_2 at some position i (by the semantics of the knowledge operator), then the formula above requires that on a stuttered version of π_1 the sub-formula is applied at state $\pi_2[i]$. We consider the prefixes of the two executions, $\pi_1[0, i]$ and $\pi_2[0, j]$ and assume they have the same traces with respect to agent A. For these, there are two stuttered versions π and π'' (named to match the path variables in the formula above) that synchronize the positions at which they change their observations with respect to A and pad the prefixes to the same length k (without changing their final states, that is $\pi_2[i] = \pi''[k]$). There is also a labeling with $t_{\pi'}$ that ensures that the knowledge operator is evaluated until position k. Hence, ψ' is applied on π'' at state $\pi_2[i]$.

The other direction is straightforward, and the case of a negatively occurring knowledge operator follows similarly. The quantifiers do not occur inside the scope of temporal operators, and can thus easily be pulled to the front of the formula, resulting in a HyperLTL formula. □

C   Satisfiability

Theorem 12. *For HyperCTL***, finite-state satisfiability is hard for Σ_1^0 and general satisfiability is hard for Σ_1^1.*

Proof. We give a reduction from the synthesis problem for LTL specifications in a distributed architecture consisting of two processes with disjoint sets of variables. The synthesis problem consists on deciding whether there exist transition systems for the two processes with input variables I_1 and I_2, respectively, and output variables O_1 and O_2, respectively, such that the synchronous product of the two transition systems satisfies a given LTL formula φ. This problem is hard for Σ_1^0 if the transition systems are required to be finite, and hard for Σ_1^1 if infinite transition systems are allowed (Theorems 5.1.8 and 5.1.11 in [51]).

To reduce the synthesis problem to HyperCTL* satisfiability, we construct a HyperCTL* formula ψ as a conjunction $\psi = \psi_1 \wedge \psi_2 \wedge \psi_3$. The first conjunct ensures that φ holds on all paths: $\psi_1 = \forall\pi.[\varphi]_\pi$, where $[\varphi]_\pi$ indicates that the atomic propositions in φ get the index π. The second conjunct ensures that every state of the model has a successor for every possible input: $\forall\pi.\,\mathrm{G}\bigwedge_{I\subseteq I_1\cup I_2}\exists\pi'\,\mathrm{X}\bigwedge_{i\in I} i\ \bigwedge_{i\notin I}\neg i$. The third conjunct ensures that the output in O_1 does not depend on I_2 and the output in O_2 does not depend on I_1: $\psi_3 = \forall\pi.\forall\pi'.\left(\pi =_{I_1} \pi' \to \pi =_{O_1} \pi'\right) \wedge \left(\pi =_{I_2} \pi' \to \pi =_{O_2} \pi'\right)$.

The distributed synthesis problem has a (finite) solution iff the HyperCTL* formula ψ has a (finite) model. □

D   Model-checking Constructions for HyperLTL$_2$

D.1   Self-composition construction

Self-composition is the technique that Barthe et al. [7] adopt to verify non-interference policies. It was generalized by Terauchi and Aiken [58] to verify observational determinism policies [36, 68], and by Clarkson and Schneider [16] to verify k-safety hyperproperties. We extend this technique to model-checking of HyperLTL$_2$.

Büchi automata. Büchi automata [65] are finite-state automata that accept strings of infinite length. A Büchi automaton is a tuple $(\Sigma, S, \Delta, S_0, F)$ where Σ is an alphabet, S is the set of states, Δ is the transition relation such that $\Delta \subseteq S \times \Sigma \times S$, S_0 is the set of initial states, and F is the set of accepting states, where both $S_0 \subseteq S$ and $F \subseteq S$. A *string* is a sequence of letters in Σ. A path $s_0s_1\ldots$ of a Büchi automaton is *over* a string $\alpha_1\alpha_2\ldots$ if, for all $i \geq 0$,

it holds that $(s_i, \alpha_{i+1}, s_{i+1}) \in \Delta$. A string is *recognized* by a Büchi automaton if there exists a path π over the string with some accepting states occurring infinitely often, in which case π is an *accepting path*. The *language* $\mathcal{L}(A)$ of an automaton A is the set of strings that automaton accepts. A Büchi automaton can be derived [14] from a Kripke structure, which is a common mathematical model of interactive, state-based systems.

Self composition. The *n-fold self-composition* A^n of Büchi automaton A is essentially the product of A with itself, n times. This construction is defined as follows:

Definition 2. *Büchi automaton A^n is the n-fold self-composition of Büchi automaton A, where $A = (\Sigma, S, \Delta, S_0, F)$, if $A^n = (\Sigma^n, S^n, \Delta', S_0^n, F^n)$ and for all $s_1, s_2 \in S^n$ and $\alpha \in \Sigma^n$ we have $(s_1, \alpha, s_2) \in \Delta'$ iff for all $1 \le i \le n$, it holds that $(prj_i(s), prj_i(\alpha), prj_i(s')) \in \Delta$.*

Let *zip* denote the usual function that maps an n-tuple of sequences to a single sequence of n-tuples—for example, $zip([1, 2, 3], [4, 5, 6]) = [(1, 4), (2, 5), (3, 6)]$—and let *unzip* denote its inverse. A^n recognizes $zip(\pi_1, \ldots, \pi_n)$ if A recognizes each of $\pi_1, \ldots, \pi_n$:

Proposition 1. $\mathcal{L}(A^n) = \{zip(\pi_1, \ldots, \pi_n) \mid \pi_1, \ldots, \pi_n \in \mathcal{L}(A)\}$

Proof. By the construction of A^n. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \square$

D.2 Formula-to-automaton construction

Given a HyperLTL$_2$ formula $\forall \pi_1 \ldots \forall \pi_k \exists \pi_{k+1} \ldots \exists \pi_{k+j} \psi$, we now show how to construct an automaton that accepts exactly the strings w for which $unzip(w) \models_\emptyset \psi$. Our construction extends standard methodologies for LTL automata construction [24, 25, 64].

1. Negation normal form. We begin by preprocessing ψ to put it in a form more amenable to model checking. The formula is rewritten to be in *negation normal form* (NNF), meaning (i) negation connectives are applied only to atomic propositions in ψ, (ii) the only connectives used in ψ are X, U, R, $\neg$, $\vee$ and $\wedge$. We identify $\neg\neg\psi$ with ψ.

2. Construction. We now construct a *generalized Büchi automaton* [19] A_ψ for ψ. A generalized Büchi automaton is the same as a Büchi automaton except that it has multiple sets of accepting states. That is, a generalized Büchi automaton is a tuple $(\Sigma, S, \Delta, S_0, F)$ where Σ, S, Δ and S_0 are defined as for Büchi automata, and $F = \{F_i \mid 1 \le i \le m$ and $F_i \subseteq S\}$. Each of the F_i is an *accepting set*. A string is recognized by a generalized Büchi automaton if there is a path over the string with at least one of the states in every accepting set occurring infinitely often.

To construct the states of A_ψ, we need some additional definitions. Define *closure $cl(\psi)$* of ψ to be the least set of subformulas of ψ that is closed under the following rules:

- if $\psi' \in cl(\psi)$, then $\neg\psi' \in cl(\psi)$.
- if $\psi_1 \wedge \psi_2 \in cl(\psi)$ or $\psi_1 \vee \psi_2 \in cl(\psi)$, then $\{\psi_1, \psi_2\} \subseteq cl(\psi)$.
- if $X\,\psi' \in cl(\psi)$, then $\psi' \in cl(\psi)$.
- if $\psi_1 \, U \, \psi_2 \in cl(\psi)$ or $\psi_1 \, R \, \psi_2 \in cl(\psi)$, then $\{\psi_1, \psi_2\} \subseteq cl(\psi)$.

And define Mcs to be a *maximal consistent set* with respect to $cl(\psi)$ if $Mcs \subseteq cl(\psi)$ and the following conditions hold:

- $\psi' \in Mcs$ iff $\neg\psi' \notin Mcs$.
- if $\psi_1 \wedge \psi_2 \in cl(\psi)$, then ($\psi_1 \wedge \psi_2 \in Mcs$ iff $\{\psi_1, \psi_2\} \subseteq Mcs$).
- if $\psi_1 \vee \psi_2 \in cl(\psi)$, then ($\psi_1 \vee \psi_2 \in Mcs$ iff $\psi_1 \in Mcs$ or $\psi_2 \in Mcs$).
- if $\psi_1 \, U \, \psi_2 \in Mcs$ then $\psi_1 \in Mcs$ or $\psi_2 \in Mcs$.
- if $\psi_1 \, R \, \psi_2 \in Mcs$ then $\psi_2 \in Mcs$.

Define $ms(\psi)$ to be the set of all maximal consistent sets with respect to ψ. The elements of $ms(\psi)$ will be the states of A_ψ; hence each state is a set of formulas. Intuitively, a state s describes a set of trace tuples where each tuple satisfies all the formulas in s. There will be a transition from a state s_1 to a state s_2 iff every trace tuple described by s_2 is an *immediate suffix* of some tuple described by s_1. (Tuple Π is an immediate suffix of Π' iff $\Pi = \Pi'[1, \infty]$.)

Automaton $A_\psi = (\Sigma_\psi, S_\psi, \Delta_\psi, \{\iota_\psi\}, F_\psi)$ is defined as follows:

- The alphabet Σ_ψ is $\mathcal{P}(\mathsf{AP})^n$ where AP is the set of atomic propositions. Each letter of the alphabet is, therefore, an n-tuple of sets of atomic propositions.
- The set S_ψ of states is $ms(\psi) \cup \{\iota_\psi\}$, where $ms(\psi)$ is defined above and ι_ψ is a distinct initial state.
- The transition relation Δ_ψ contains (s_1, α, s_2), where $\{s_1, s_2\} \subseteq S_\psi \setminus \{\iota_\psi\}$ and $\alpha \in \Sigma_\psi$, iff
    - If $a_{\pi_i} \in s_2$ for some $1 \le i \le n$, then $a \in prj_i(\alpha)$. Likewise, if $\neg a_{\pi_i} \in s_2$, then $a \notin prj_i(\alpha)$.
    - If $X\,\psi' \in s_1$ then $\psi' \in s_2$.
    - If $\psi_1 \, U \, \psi_2 \in s_1$ and $\psi_2 \notin s_1$ then $\psi_1 \, U \, \psi_2 \in s_2$.
    - If $\psi_1 \, R \, \psi_2 \in s_1$ and $\neg\psi_1 \in s_1$ then $\psi_1 \, R \, \psi_2 \in s_2$.
  And Δ_ψ contains $(\iota_\psi, \alpha, s_2)$ iff $\psi \in s_2$ and $(\iota_\psi, \alpha, s_2)$ is a transition permitted by the above rules for atomic propositions and their negations.
- The set of initial states contains only ι_ψ.
- The set F_ψ of sets of accepting states contains one set $\{s \in (S_\psi \setminus \{\iota_\psi\}) \mid \neg(\psi_1 U \psi_2) \in s$ or $\psi_2 \in s\}$ for each until formula $\psi_1 \, U \, \psi_2$ in $cl(\psi)$.

The definition of F_ψ guarantees that, for every until formula $\psi_1 \, U \, \psi_2$, eventually ψ_2 will hold. That is because the transition rules do not allow a transition from a state containing $\psi_1 \, U \, \psi_2$ to a state containing $\neg(\psi_1 \, U \, \psi_2)$ unless ψ_2 is already satisfied.

3. Degeneralization of Büchi automata. Finally, convert generalized Büchi automaton A_ψ to a "plain" Büchi automaton. This conversion is entirely standard [25], so we do not repeat it here.

Correctness of the construction. Again without loss of generality, assume that the names of the traces are natural numbers; then Π is isomorphic to an n-tuple of traces. Henceforth, we treat Π as that tuple. The following proposition states that A_ψ is constructed such that it recognizes computation tuples that model ψ:

Proposition 2. $\Pi \models_\emptyset \psi$ *iff* $zip(\Pi) \in \mathcal{L}(A_\psi)$.

Proof. ($\Leftarrow$) By the construction of A_ψ, the states with a transition from ι_ψ contain ψ. Hence by Lemma 1 below, for all the strings w such that $w = zip(\Pi)$ in $\mathcal{L}(A_\psi)$, it holds that $\Pi \models_\emptyset \psi$.

($\Rightarrow$) Let $s_i = \{\psi' \in cl(\psi) \mid \Pi[i, \infty] \models_\emptyset \psi'\}$ for all $i \geq 0$. Then by the definition, $s_i \in ms(\psi)$. We show that $\iota_\psi s_0 s_1 \ldots$ is an accepting path in A_ψ. By $\Pi \models_\emptyset \psi$ we have $\psi \in s_0$. By the construction of A_ψ, $(i_\psi, \alpha_0, s_0) \in \Delta_\psi$ where $\alpha_0 = zip(\Pi)[0]$. The construction of the path inductively follows the construction of A_ψ, which respects the semantics of HyperLTL. $\qquad \square$

Lemma 1. *Let $\iota_\psi s_0 \ldots$ be an accepting path in A_ψ over the string $w = \alpha_0 \alpha_1 \ldots$. Let $\Pi = unzip(w)$. Then for all $i \geq 0$, it holds that $\psi' \in s_i$ iff $\Pi[i, \infty] \models_\emptyset \psi'$.*

Proof. The proof proceeds by induction on the structure of ψ':

Base cases:

1. $\psi' = a_{\pi_r}$
   ($\Rightarrow$) Assume that $a_{\pi_r} \in s_i$. By the construction of A_ψ, if $a_{\pi_r} \in s_i$ then $a \in prj_r(\alpha_i)$ or equivalently $p \in prj_r(\Pi)[i]$. By the semantics of HyperLTL, we have $\Pi[i, \infty] \models_\emptyset a_{\pi_r}$.
   ($\Leftarrow$) Assume that $\Pi[i, \infty] \models_\emptyset a_{\pi_r}$. Then $a \in prj_r(\Pi)[i]$, which is equivalent to $a \in prj_r(\alpha_i)$. By the fact that states are maximal consistent sets, one of a_{π_r} or $\neg a_{\pi_r}$ must appear in s_i. By the construction of A_ψ and the fact that $a \in prj_r(\alpha_i)$, we have $a_{\pi_r} \in s_i$.

Inductive cases:

1. $\psi' = \neg \psi''$
   ($\Rightarrow$) Assume that $\neg \psi'' \in s_i$, Then $\psi'' \notin s_i$. By induction hypothesis, $\Pi[i, \infty] \not\models_\emptyset \psi''$, or equivalently, $\Pi[i, \infty] \models_\emptyset \neg \psi''$. Hence, $\Pi[i, \infty] \models_\emptyset \psi'$.
   ($\Leftarrow$) Similar to $\Rightarrow$.
2. $\psi' = \psi_1 \vee \psi_2$
   ($\Rightarrow$) By the construction of A_ψ, if $\psi_1 \vee \psi_2 \in s_i$ then $\psi_1 \in s_i$ or $\psi_2 \in s_i$. By induction hypothesis, $\Pi[i, \infty] \models_\emptyset \psi_1$ or $\Pi[i, \infty] \models_\emptyset \psi_2$, which concludes $\Pi[i, \infty] \models_\emptyset \psi_1 \vee \psi_2$.
   ($\Leftarrow$) Similar to $\Rightarrow$.
3. $\psi' = X \psi''$
   ($\Rightarrow$) Assume that $\psi' \in s_i$. By the construction of A_ψ, $\psi'' \in s_{i+1}$. By induction hypothesis, $\Pi[i + 1, \infty] \models_\emptyset \psi''$, which concludes $\Pi[i, \infty] \models_\emptyset X \psi''$.
   ($\Leftarrow$) Similar to $\Rightarrow$ and the fact that always one of $X \psi''$ or $\neg X \psi''$ appears in a state.

4. $\psi' = \psi_1 \ U \ \psi_2$

($\Rightarrow$) Assume that $\psi_1 \ U \ \psi_2 \in s_i$. By the construction of A_ψ and the fact that the path is accepting, there is some $j \geq i$ such that $\psi_2 \in s_j$. Let j be the smallest index. By induction hypothesis, $\Pi[i, \infty] \models_\emptyset \psi_2$. By the construction of A_ψ, for all $i \leq k < j$, $\psi_1 \in s_k$. Therefore by induction hypothesis, $\Pi[k, \infty] \models_\emptyset \psi_1$. which concludes $\Pi[k, \infty] \models_\emptyset \psi_1 \ U \ \psi_2$.
($\Leftarrow$) Similar to $\Rightarrow$.

$\square$

E Prototype Model Checker for HyperLTL$_2$

The model-checking procedure for LTL works as follows:

1. Transform Kripke structure K into its corresponding Büchi automaton [10], A_K. This construction is standard [14]. The language of A_K is $\mathsf{Traces}(K)$.
2. Construct Büchi automaton $A_{\neg\phi}$, whose language is the set of all traces that do not satisfy ϕ.
3. Intersect A_K and $A_{\neg\phi}$, yielding automaton $A_K \cap A_{\neg\phi}$. Its language contains all traces in $\mathsf{Traces}(K)$ that do not satisfy ϕ. This construction is standard [14].
4. Check whether the language of $A_K \cap A_{\neg\phi}$ is empty. If so, all traces $\mathsf{Traces}(K)$ satisfy ϕ, hence we say K satisfies ϕ. If not, then any element of the language is a counterexample showing that K doesn't satisfy ϕ.

Our algorithm for model-checking HyperLTL$_2$ adapts that LTL algorithm. Without loss of generality, assume that the HyperLTL$_2$ formula to be verified has the form $\forall \pi_{1..k} \exists \pi'_{1..j} \psi$, where $\forall \pi_{1..k}$ means $\forall \pi_1 \ldots \forall \pi_k$, and $\exists \pi'_{1..j}$ means $\exists \pi'_1 \ldots \exists \pi'_j$. (Formulas of the form $\exists \pi_{1..k} \forall \pi'_{1..j} \psi$ can be verified by rewriting them as $\forall \pi_{1..k} \exists \pi'_{1..j} \neg\psi$.) Let n equal $k + j$. Semantically, a model Π of ψ must be a set of named traces, where $|\Pi| = n$. To determine whether a Kripke structure K satisfies HyperLTL$_2$ formula $\forall \pi_{1..k} \exists \pi'_{1..j} \psi$, our algorithm follows the same basic steps as the LTL algorithm:

1. Represent K as a Büchi automaton, A_K. Construct the n-fold product of A_K with itself—that is, $A_K \times A_K \times \cdots \times A_K$, where "$A_K$" occurs n times. This construction is straightforward and formalized in Appendix D. Denote the resulting automaton as A_K^n. If $\pi_1, \ldots \pi_n$ are all traces of $\mathsf{Traces}(K)$, then $zip(\pi_1, \ldots \pi_n)$ is a word in the language of A_K^n.
2. Construct Büchi automaton A_ψ. Its language is the set of all words w such that $unzip(w) = \Pi$ and $\Pi \models_\emptyset \psi$—that is, the tuples Π of traces that satisfy ψ. This construction is a generalization of the corresponding LTL construction. It is formalized in Appendix D.
3. Intersect A_K^n and A_ψ, yielding automaton $A_K^n \cap A_\psi$. Its language is essentially the tuples of traces in $\mathsf{Traces}(K)$ that satisfy ψ.

4. Check whether $\mathcal{L}(((A_K^n \cap A_\psi)|_k)^C \cap A_K^k)$ is empty, where (i) A^C denotes the *complement* of an automaton A, (complement constructions are well-known—e.g., [64]—so we do not formalize one here), and (ii) $A|_k$ denotes the same automaton as A, but with every transition label (which is an n-tuple of propositions) *projected* to only its first k elements. That is, if $\mathcal{L}(A)$ contains words of the form $zip(\pi_1, \dots \pi_n)$, then $\mathcal{L}(A|_k)$ contains words of the form $zip(\pi_1, \dots \pi_k)$. Projection erases the final j traces from each letter of a word, leaving only the initial k traces. Thus a word is in the projected language iff there exists some extension of the word in the original language. If $\mathcal{L}(((A_K^n \cap A_\psi)|_k)^C \cap A_K^k)$ is empty, then it holds that

$$\emptyset \models_{\mathsf{Traces}(K)} \forall \pi_{1..k} \exists \pi'_{1..j} \psi.$$

If not, then any element of the language is a counterexample showing that $\emptyset \not\models_{\mathsf{Traces}(K)} \forall \pi_{1..k} \exists \pi'_{1..j} \psi$.

The final step of the above algorithm is a significant departure from the LTL algorithm. Intuitively, it works because projection introduces an existential quantifier, thus enabling verification of formulas with a quantifier alternation. The following theorem states the correctness of our algorithm:

Theorem 13. *Let ϕ be HyperLTL$_2$ formula $\forall \pi_{1..k} \exists \pi'_{1..j} \psi$, and let $n = k + j$. Let K be a Kripke structure. Then ϕ holds of K iff $\mathcal{L}(((A_K^n \cap A_\psi)|_k)^C \cap A_K^k)$ is empty.*

Proof. ($\Rightarrow$, by contrapositive) We seek a countermodel showing that $\forall \pi_{1..k} \exists \pi'_{1..j} \psi$ doesn't hold of K. For that countermodel to exist,

$$\begin{aligned} &\text{there must exist a } k\text{-tuple } \Pi_k : \text{for all } j\text{-tuples } \Pi_j : \\ &\text{if } set(\Pi_k \cdot \Pi_j) \subseteq \mathsf{Traces}(K) \text{ then } \Pi_k \cdot \Pi_j \models_\emptyset \neg\psi, \end{aligned} \tag{10}$$

where $set(\Pi)$ denotes the set containing the same elements as tuple Π. To find that countermodel Π_k, consider $\mathcal{L}(A_K^n \cap A_\psi)$. If that language is empty, then

$$\begin{aligned} &\text{for all } k\text{-tuples } \Pi_k \text{ and for all } j\text{-tuples } \Pi_j : \\ &\text{if } set(\Pi_k \cdot \Pi_j) \subseteq \mathsf{Traces}(K) \text{ then } \Pi_k \cdot \Pi_j \models_\emptyset \neg\psi. \end{aligned} \tag{11}$$

That's almost what we want, except that Π_k is universally quantified in (11) rather than existentially quantified as in (10). So we introduce projection and complementation to relax the universal quantification to existential. First, note that language $\mathcal{L}((A_K^n \cap A_\psi)|_k)$ contains all $zip(\Pi_k)$ for which there exists a Π_j such that $set(\Pi_k \cdot \Pi_j) \subseteq \mathsf{Traces}(K)$ and $\Pi_k \cdot \Pi_j \models_\emptyset \psi$. So if there exists a Π_k^* such that $zip(\Pi_k^*) \notin \mathcal{L}((A_K^n \cap A_\psi)|_k)$, then for all Π_j, if $set(\Pi_k \cdot \Pi_j) \subseteq \mathsf{Traces}(K)$ then $\Pi_k \cdot \Pi_j \models \neg\psi$. That Π_k^* would be exactly the countermodel we seek according to (10). To find such a Π_k^*, it suffices to determine whether $\mathcal{L}((A_K^n \cap A_\psi)|_k) \subset \mathcal{L}(A_K^k)$, because any element that strictly separates those sets would satisfy the requirements to be a Π_k^*. By simple set theory, $X \subset Y$ iff $X^C \cap Y$ is not empty.

Therefore, if $\mathcal{L}(((A_K^n \cap A_\psi)|_k)^C \cap A_K^k)$ is not empty, then a countermodel Π_k^* exists.

($\Leftarrow$) The same argument suffices: if $\mathcal{L}(((A_K^n \cap A_\psi)|_k)^C \cap A_K^k)$ is empty, then no countermodel can exist. □

Formulas without quantifier alternation. Define HyperLTL$_1$ to be the fragment of HyperLTL$_2$ that contains formulas with no alternation of quantifiers. HyperLTL$_1$ can be verified more efficiently than HyperLTL$_2$. To verify $\forall \pi_{1..n}\psi$, it suffices to check whether $A_M^n \cap A_{\neg\psi}$ is non-empty. This is essentially the self-composition construction, as used in previous work [7, 16, 58].

Complexity. The most expensive computation in model checking HyperLTL$_2$ is the Büchi automaton complementation. Safra's construction [55], which can be used to implement complementation, has complexity of $2^{O(m \log m)}$, where m is the number of the states of the original automaton. In step 4 of our algorithm, the automaton being complemented has $O(|K|^n \cdot 2^{O(|\psi|)})$ states, where $|K|$ is the number of states of K, and $|\psi|$ is the length of ψ. Combined with the final intersection and non-emptiness check, the complexity of model checking is $O\left(|K|^n \cdot 2^{O(|K|^{n+1} \cdot O(|\psi|) \cdot 2^{O(|\psi|)})}\right)$. Our prototype is therefore exponential in the size of the program (i.e., Kripke structure) and doubly exponential in the size of the formula. That complexity is worse than the complexity of model-checking LTL, which is polynomial in the size of the program and exponential in the size of the formula [64]. Perhaps the complexity of model-checking HyperLTL$_2$ could be reduced in future work, or perhaps it's simply the price we pay to have a general-purpose logic of hyperproperties.

The worst case time complexity of epistemic logic with perfect recall is similar to HyperLTL$_2$, as it is exponential to the size of the system, and doubly exponential in the size of the formula [17]. The worst-case space complexity of single-quantifier-alternation HyperLTL formulas (i.e., many of the security policies of interest) is NLOGSPACE, whereas epistemic with perfect recall is worse: PSPACE-hard [61]. So there is hardly any theoretical advantage of epistemic over HyperLTL.