Lightweight Verification of Hyperproperties*

Oyendrila Dobe[1], Stefan Schupp[2], Ezio Bartocci[2], Borzoo Bonakdarpour[1], Axel Legay[3], Miroslav Pajic[4], and Yu Wang[5]

[1] Michigan State University, USA,
[2] Technische Universität Wien, Austria.
[3] UCLouvain, Belgium.
[4] Duke University, USA.
[5] University of Florida, USA.

Abstract. Hyperproperties have been widely used to express system properties like noninterference, observational determinism, conformance, robustness, etc. However, the model checking problem for hyperproperties is challenging due to its inherent complexity of verifying properties across sets of traces and suffers from scalability issues. Previously, statistical approaches have proven effective in tackling the scalability of model checking for temporal logic. In this work, we have attempted to combine these two concepts to propose a tractable solution to model checking of hyperproperties expressed as HyperLTL on models involving nondeterminism. We have implemented our approach in PLASMA and experimented with a range of case studies to showcase its effectiveness.

Keywords: Hyperproperties, Statistical Model Checking, Nondeterminism

1 Introduction

Model checking [7] is a well-established method to verify the correctness of a system. Typically, it exhaustively checks if all possible *individual* execution traces of the system satisfy a property of interest. However, several security and privacy policies such as noninterference [36, 42, 49], differential privacy [25], observational determinism [57] are system-wide properties that require to reason across multiple independent system's executions simultaneously. These properties are referred to as *hyperproperties* [18].

In the last decade, researchers have proposed several adaptations of classical temporal logics to specify hyperproperties in a formal and systematic way. Examples in the non-probabilistic setting are HyperLTL [17] and its asynchronous variant A-HLTL [8]. HyperLTL extends LTL [50] with explicit quantification over paths that allows to express relations among execution traces from independent system's runs. Recent works in [33, 39, 41] provide exhaustive and bounded model-checking algorithms for HyperLTL. For probabilistic hyperproperties, there are

* This project was partially funded by the United States NSF Award CCF-2133160, and SaTC Awards 2245114 and 2100989, FWF-project ZK-35, FNRS PDR - T013721, and by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT19018].

two main specification languages: HyperPCTL [2,23], which quantifies over schedulers and argues over computation trees, and Probabilistic HyperLogic (PHL) [21] which adds quantifiers for schedulers and reasons about traces. In both contexts, these approaches face two main challenges: scalability and the need for an explicit model. Scalability is, in particular, critical: (1) HyperLTL model checking is EXSPACE-complete [11], (2) HyperPCTL and A-HLTL model checking are in general undecidable with decidable fragments in EXSPACE [8, 23], (3) PHL model checking is in general undecidable with decidable fragments (reduce to HyperCTL*) in NSPACE [21, 33]. This complexity obstacle has been a major motivation for the development of alternative approaches to handle the problem. One possible approach is to provide an approximate result with certain statistical guarantees, termed statistical model checking (SMC). SMC is an approximate model checking method that is subject to a small probability of drawing a wrong conclusion [45–47]. The main idea is to simulate finitely many traces of a model and conduct *hypothesis testing* to conclude if there is enough evidence that the model satisfies or violates the property, subjecting to a small probability of drawing a wrong conclusion. Such simulation-based approaches have two main advantages: first, we can use them to approximate the probability of satisfying the desired property in a model of considerable size, which we would be otherwise unable to verify exhaustively; second, we can apply them to black-box systems for which we are unable to access the inner model. This approach is also intuitive as it can terminate early for cases where it has already found enough evidence for violation. Consider, a case where a property is required to hold for all traces. In this case, we should not be able to see a violation even if we simulate just one trace. Given these advantages, we want to study its application to verify hyperproperties. Another challenge, in terms of verification, is the handling of nondeterminism. When modeling systems, we have to take into consideration the uncertainty that can arise due to incomplete details, involvement of unknown agents, or noise, in general. From a verification perspective, we need to be able to argue that a property holds under any such possibility of nondeterministic uncertainty. Both HyperPCTL and A-HLTL model checking has the capability of reasoning over nondeterminism, however, the high complexity in their model checking solutions basically stems from the need for "scheduler" synthesis.

Our contribution In this work, we chose to model systems as Markov decision processes (MDPs) to effectively express nondeterminism in terms of possible actions available in a state, as well as randomization is represented as probabilistic distributions of how the system can evolve once an action is executed. PLASMA [48] is a model checker that uses a memory-efficient sampling of schedulers [26] to conduct simulation-based statistical analysis. In this work, we extend PLASMA's capability to include the verification of linear, bounded hyperproperties over systems modeled as MDPs. Our method orchestrates well-established methods from the SMC community for the analysis of an expressive model class in light of bounded HyperLTL properties. The result is a scalable, lightweight verification approach which is the first of its kind to handle this combination of model class and property. We have added and experimented with an extension that supports

using recorded traces or requesting simulation of black-box components on-the-fly for hyperproperty verification. This opens the door to utilizing our approach for applications in cases where explicit modeling is not possible or error-prone. For evaluation, we present a diverse set of scaling benchmarks that raises the demand for this expressive model class and property type. We have selected systems that allow for verification of properties such as *noninterference*, *side-channel information leak*, *opacity*, and *anonymity*. The systems under inspection range from classical examples including dining cryptographers, to examples taken from robotics path planning and real code snippets. The state space of the resulting models varies in the order of magnitude from tens to hundreds of billions involving tens to thousands of nondeterministic actions. Our experimental evaluation indicates good performance on systems, unperturbed by the size of the state space. To summarize, our *main contributions* are:

1. To the best of our knowledge, we provide the first statistical model checking approach for the verification of unquantified and bounded HyperLTL properties involving nondeterminism.
2. We extend the model checker PLASMA by this class of properties. Furthermore, we add capabilities to execute black-box verification.
3. We showcase the general applicability with an extensive evaluation of our method on various scalable case studies taken from different domains.

Paper Organization: In the rest of the paper, we elaborate on the related works in Section 2, describe the model and specification language in Section 3, with the problem formulation in Section 4, the algorithm and implementation details in Section 5, and a range of case studies in Section 6. In Section 7 we describe our experimental and convergence results and in Section 8 we provide conclusions and future work suggestions.

2 Related work

HyperLTL [17] was introduced to express system properties that require simultaneous quantification over multiple paths. The authors provided a model checking algorithm for a fragment of the logic based on self-composition. In [33], the authors presented the first direct automata-theoretic model checking algorithm that converts the model checking problem for HyperLTL to automaton-based problems like checking for emptiness. In recent years, there has been considerable research on HyperLTL verification [19, 31, 32], and monitoring [5, 10, 11, 29, 37, 51]. From a tools perspective, MCHyper [19, 33] has been developed for model checking, EAHyper [28] and MGHyper [27] for satisfiability checking, and RVHyper [30] for run time monitoring. A tractable bounded sublogic of HyperLTL has been proposed in [41] where the authors have suggested a QBF-based algorithm to model-check the logic. HyperQB is a model checker specifically for bounded HyperLTL [40]. However, all of the above approaches suffer from the challenges of scalability, inability to handle probabilistic systems, or lack of support for nondeterminism.

To verify hyperproperties in probabilistic systems there are two main families of approaches proposed in the literature: exact methods [2, 3, 22, 23] and approximated ones [20, 53]. Note that the specification language used in these works differs from our specification language. Exact methods exploit the underlying Markov chain structure of the probabilistic system to be verified for computing precisely (numerically) and for comparing the probabilities of satisfying temporal logic formulas of multiple and independent sequences of sets of states. HyperPCTL [2] was the first logic proposed to reason exhaustively about hyperproperties in probabilistic systems. This logic was later extended to allow reasoning over systems involving nondeterminism [3, 23] and rewards [24]. A verification algorithm was implemented in the model checker HyperProb [22]. The main shortcoming of this approach is scalability. Among the approximated approaches, in [21] the authors propose an over-approximate and another under-approximate automata-based model checking algorithms for the alternation-free n-safety fragment of their logic PHL on n self-composed systems. The scheduler synthesis step is the main challenge in this work.

SMC has been explored to solve problems across different domains for analyzing dynamic software architectures [14], performing security risk assessments using attack-defense trees [34], verifying cyber-physical systems [16], validation of biochemical reaction models [58], etc. Verification of bounded LTL for MDPs has been proposed using SMC [38] and has shown promising results. Extensive tool support exists for SMC on trace properties with respect to discrete-event modeling [12], priced timed automata [13], probabilistic model checking [43,44,56], black-box systems [35]. Statistical verification of probabilistic hyperlogics has been proposed for HyperPCTL* [20, 53], for continuous Markov chains [55], and for real-valued signals [6]. However, none of these works reason about models involving nondeterminism.

3 Preliminaries

We denote the set of natural and real numbers by $\mathbb{N}$ and $\mathbb{R}$, respectively. For $n \in \mathbb{N}$, let $[n] = \{1, \ldots, n\}$. The cardinality of a set is denoted by $|\cdot|$. We denote the set of all finite, non-empty finite, and infinite sequences, taken from S by S^*, S^+, and S^ω, respectively.

3.1 Model Structures

Markov Decision Process A labeled Markov decision process (MDP) [7] is a tuple $\mathtt{M} = (\mathcal{S}, A, s_0, \mathsf{AP}, L, P)$, where (1) $\mathcal{S}$ is a finite set of states, (2) A is the finite set of actions, and $A(s)$ is the set of *enabled* actions that can be taken at state $s \in \mathcal{S}$, (3) s_0 is the initial state, (4) AP is the finite set of atomic propositions, (5) $L : \mathcal{S} \to 2^{\mathsf{AP}}$ is the state labeling function, and (6) $P : \mathcal{S} \times A \times \mathcal{S} \to [0, 1]$ is the transition probability function such that,

$$\sum_{s' \in \mathcal{S}} P(s, a, s') = \begin{cases} 1, & \text{if } a \in A(s) \\ 0, & \text{if } a \notin A(s) \end{cases} \tag{1}$$

A path of the MDP is an infinite sequence of states $\pi = s_0 s_1 s_2 \ldots$ with $s_i \in \mathcal{S}$ such that $\forall i \geq 0$, there exists $a_i \in A$ with $P(s_i, a_i, s_{i+1}) > 0$. A trace $trace\{\pi\} = L(s_0)L(s_1)L(s_2)\ldots$ is the sequence of sets of atomic propositions corresponding to a path. We use $\pi[i] = s_i$ to denote the ith state and $\pi[: i]$ and $\pi[i+1 :]$ to denote the prefix $s_0 s_1 \ldots s_i$, and the suffix $s_{i+1} s_{i+2} \ldots$, respectively.

Discrete-Time Markov Chain A labeled discrete-time Markov chain (DTMC) [7] is a tuple $\mathcal{D} = (\mathcal{S}, s_0, \mathsf{AP}, L, P)$, where (1) $\mathcal{S}$ is the finite set of states, (2) s_0 is the initial state, (3) AP is the finite set of atomic propositions, (4) $L : \mathcal{S} \to 2^{\mathsf{AP}}$ is the state labeling function, (5) $P : \mathcal{S} \times \mathcal{S} \to [0, 1]$ is the transition probability function such that, for all $s \in \mathcal{S}$, $\sum_{s' \in \mathcal{S}} P(s, s') = 1$. A DTMC is an MDP with each state being associated with a single action.

Scheduler A scheduler σ is a function $\sigma : \mathcal{S}^+ \to A$ that resolves the nondeterminism at each state of an MDP. It reduces an MDP to a DTMC. Different scheduler types are distinguished depending on what information is used to resolve the nondeterminism in the current state: a *history-dependent* scheduler $\sigma(s[: n]) \in A(s[n])$ would utilize the history of action and state choices to resolve which action is executed at the current state whereas a *memoryless* scheduler $\sigma(s[n]) \in A(s[n])$, bases its decision only on the current state. In this work, we consider history-dependent schedulers (which include the class of memoryless schedulers) whose memory size is bounded by the length of the paths we generate from the model. We use $\pi_{\mathtt{M}^\sigma}$ to denote a random path drawn from the DTMC that is induced by the scheduler σ on the MDP $\mathtt{M}$.

3.2 HyperLTL

HyperLTL [17] is the extension of linear-time temporal logic (LTL) [50] that allows the expression of temporal specifications involving relations between multiple paths. Each state in the path is observed as a set of atomic propositions that hold true in that state. HyperLTL involves the evaluation of specifications over these propositions. An arbitrary path variable π is used to refer to individual paths that can be generated by the model. Contrary to LTL, each proposition a^π is associated with a path variable π denoting the path on which it should be evaluated.

Syntax We focus on unquantified and bounded HyperLTL defined by the grammar below.

$$\varphi ::= \mathsf{a}^\pi \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \, \mathcal{U}^{\leq k} \, \varphi \tag{2}$$

- $\mathsf{a} \in \mathsf{AP}$ is an atomic proposition that evaluates to *true* or *false* in a state;
- π is a *random path variable* from an infinite supply of such variables Π;
- $\bigcirc$, $\Diamond^{\leq k}$, $\Box^{\leq k}$, and $\mathcal{U}^{\leq k}$ are the 'next', 'finally', 'global', and 'until' temporal operators, respectively,
- $k \in \mathbb{N}$ is the path length within which the operator has to be evaluated.

Following are the connectives defined as syntactic sugar:
$\texttt{true} \equiv \mathsf{a}^{\pi} \vee \neg\mathsf{a}^{\pi}$, $\varphi \vee \varphi' \equiv \neg(\neg\varphi \wedge \neg\varphi')$, $\varphi \Rightarrow \varphi' \equiv \neg\varphi \vee \varphi'$, $\Diamond^{\leq k}\,\varphi \equiv \texttt{true}\,\mathcal{U}^{\leq k}\,\varphi$, $\Box^{\leq k}\,\varphi \equiv \neg\Diamond^{\leq k}\neg\varphi$. We denote $\mathcal{U}^{\leq\infty}$, $\Diamond^{\leq\infty}$, and $\Box^{\leq\infty}$ or the unbounded temporal operators by $\mathcal{U}$, $\Diamond$, and $\Box$, respectively. In our work, we consider only the bounded fragment of HyperLTL such that for all temporal operators (except $\bigcirc$), we evaluate the result on finite fragments of the simulated traces.

Semantics The path evaluation function $V : \Pi \to \mathcal{S}^{\omega}$ assigns each path variable π, a concrete path of the labeled DTMC. Below we consider the semantics of HyperLTL,

$$
\begin{aligned}
V &\models \mathsf{a}^{\pi} && \text{iff } \mathsf{a} \in L\big(V(\pi)[0]\big)\\
V &\models \neg\varphi && \text{iff } V \not\models \varphi\\
V &\models \varphi_1 \wedge \varphi_2 && \text{iff } V \models \varphi_1 \text{ and } V \models \varphi_2\\
V &\models \bigcirc\varphi && \text{iff } V^{(1)} \models \varphi\\
V &\models \varphi_1\,\mathcal{U}^{\leq k}\,\varphi_2 && \text{iff there exists } i \in [0,k], V^{(i)} \models \varphi_2\\
& && \quad \text{and for all } j \in [0,i),\ V^{(j)} \models \varphi_1
\end{aligned}
$$

where $V^{(i)}$ is the i-shift of path assignment V defined by $V^{(i)}(\pi) = (V(\pi))^{(i)}$. For example, the formula $V \models \mathsf{a}_1^{\pi_1}\,\mathcal{U}^{k}\,\mathsf{a}_2^{\pi_2}$ means that a_1 holds on the path $V(\pi_1)$ until a_2 holds on the path $V(\pi_2)$ in k steps.

3.3 Sequential Probability Ratio Test

We use Wald's sequential probability ratio test (SPRT) [52]. The idea is to continue sampling until we are either able to reach a conclusion or we have exhausted a user-provided sampling budget. Assuming we want to verify if a property φ holds on our model $\mathcal{D}$ with probability greater than and equal to θ, i.e., $\Pr_{\mathcal{D}}(\varphi) \geq \theta$. To use SPRT in this case, we add an indifference region around our bound to create two distinct and flexible hypothesis tests [4]. For a given indifference region ε, we define $p_0 = \theta + \varepsilon$ and $p_1 = \theta - \varepsilon$. Our resultant hypotheses are,

$$H_0 : \Pr_{\mathcal{D}}(\varphi) \geq p_0 \quad H_1 : \Pr_{\mathcal{D}}(\varphi) \leq p_1 \tag{3}$$

Using these newly created bounds, we define the following probability ratios,

$$ratio_t = \frac{p_1}{p_0} \quad ratio_f = \frac{1-p_1}{1-p_0} \tag{4}$$

We define an indicator function $\mathbf{1}(T \models \varphi) \in \{0,1\}$ that returns 1 if the trace T satisfies the property φ, and returns 0 otherwise. When evaluating φ on a set of sampled traces $\{T_1, \ldots, T_n\}$, we accumulate $ratio_t$ if $\mathbf{1}(T \models \varphi) = 1$ and $ratio_f$ otherwise. Assuming, we have sampled N traces, the final product of the truth value corresponds to

$$p_{ratio} = \prod_{i=1}^{N} \frac{(p_1)^{\mathbf{1}(T_i \models \varphi)}(1-p_1)^{\mathbf{1}(T_i \models \neg\varphi)}}{(p_0)^{\mathbf{1}(T_i \models \phi)}(1-p_0)^{\mathbf{1}(T_i \models \neg\phi)}} \tag{5}$$

We iteratively calculate this ratio until the exit condition is met. To restrict the error in the estimation of the probability θ, we specify error probabilities α as the maximum acceptable probability of incorrectly rejecting a true H_0, and β as the maximum acceptable probability of accepting a false H_0. The boundary error ratios can be defined as $A = \beta/(1-\alpha)$ and $B = (1-\beta)/\alpha$. To reach a conclusion, we accept H_0 if $p_{ratio} \leq A$, and accept H_1 if $p_{ratio} \geq B$. The case for specifications with $\Pr_{\mathcal{D}}(\varphi) \leq \theta$ is similar except we use the reciprocals of $ratio_t$ and $ratio_f$.

4 Problem Formulation

HyperLTL allows explicit quantification over traces, allowing the user to express whether they want their specification to hold across all paths associated with a path variable or in at least one of those paths. Along with the added expressiveness, this formulation distends existing challenges - (1) While checking a specification across all possible sets of paths provides a robust verification result, it is considerably expensive, making it impractical as we scale to models with larger state spaces. (2) Most real-life systems involve uncertainties in the form of randomization, nondeterminism, or partial observability. Consequently, this raises a need to express that, for instance, a fraction of the paths of the system satisfy the specification.

To handle the above challenges, we propose a practical formulation for expressing unquantified and bounded HyperLTL formulas for models that involve both probabilistic choices and nondeterminism. We quantify over the path variables by associating a probabilistic bound denoting the proportion of the set of traces that should satisfy a given specification. We can express that a specification is *almost always likely* or *highly unlikely* by adjusting the bound of the probability p to $p \geq 1$ or $p \leq 0$, respectively. Intuitively, *almost always likely* can be considered as a weaker counterpart of $\forall$ (forall) quantification, and *highly unlikely* can be considered as a weaker counterpart of $\neg\exists$ (existential) quantification over path variables. Note that these limits our expression of HyperLTL formulas with quantifier alternation in any capacity, and we leave that as an aspect worth exploring in future works.

Consider an MDP $\mathtt{M}$ and an unquantified, bounded HyperLTL formula φ that contains path variables $(\pi_1, \ldots, \pi_m)$. We consider tuples of m schedulers to simulate m traces assigned to these path variables, i.e., we have a one-to-one correspondence between schedulers and path variables. We are interested in checking if there *exists* a combination of schedulers that can satisfy the HyperLTL specification φ on our model within a given probability bound. Formally, this can be expressed as,

$$\exists\sigma_1 \exists\sigma_2 \ldots \exists\sigma_m \ \Pr_{\mathtt{M}}(V \models \varphi) \sim \theta \tag{6}$$

where $\theta \in [\varepsilon, 1-\varepsilon]$ to allow an indifference region for hypothesis testing (see Section 3.3), σ_i are schedulers of $\mathtt{M}$, $V(\pi_i)$ is the path drawn from the DTMC $\mathtt{M}_{\sigma_i}$ for $i \in [n]$ which is induced by σ_i on MDP $\mathtt{M}$, and $\sim \in \{\geq, \leq\}$. Note that we can involve multiple models to yield paths for each scheduler σ_i. For properties where

we want to check if a given specification holds across all scheduler combinations, we negate our specification to re-formulate the problem as in Eq. 6. Since we adopt a statistical model checking algorithm, it is worth noting that we cannot directly observe if a specification holds *for all* cases, thus, we utilize this approach to check if we can satisfy its negation. We elaborate on this in Section 5.

5 Approach

We want to utilize the advantages of SMC to verify hyperproperties by answering our model checking problem using hypothesis testing, specifically SPRT, as described in Sec. 3.3. The overall approach involves the sampling of schedulers and traces from one (or more) given MDPs, monitoring the satisfaction of the property on these traces, and determining if we have gathered enough evidence to reach a concrete verdict for the property. In this section, we explain the concepts and parameters involved in finding the result of this test such that we can directly use it to answer our model checking question.

5.1 Scheduler sampling

One of the main challenges when verifying MDPs is the generation of schedulers for verification. It stems from the complexity involved in the storage of history to resolve nondeterminism in the current state. We utilize the lightweight scheduler sampling feature of PLASMA [26]. This approach avoids the explicit storage of schedulers by using *uniform* pseudo-random number generators (PRNGs) to resolve non-determinism and hashes as seeds for the PRNGs. In the following, we will give an intuition of the approach inbuilt in PLASMA and how we have extended it to argue about hyperproperties.

PRNGs form the core of the smart sampling algorithm of PLASMA. Given a set of possible action choices and sufficient runs of the number generator, they allow the generation of statistically independent numbers that are uniformly distributed across a specified range. They are uniquely mapped to their seed values int_{sch}, ensuring the reproduction of the same value when the generator is provided with the same seed. Note that we can use PRNGs to identify individual schedulers but cannot identify specific schedulers. Furthermore, since the seeds only initialize the PRNGs, using problem-specific information (e.g., about the property) during the generation of the seed does not allow to relate schedulers.

Each state of the MDP is internally represented as a concatenation of the bits representing the values of the atomic propositions that are true at that state. A sequence of states can be represented by concatenating their individual bit sequences. The sum of the bits of such a sequence of numbers int_t, which is an integer, represents a trace. Concatenation of int_{sch} and int_t can be then used to uniquely identify both a scheduler and a trace. PLASMA generates a hash with this concatenated value which represents the history of both the scheduler and the trace and is used as a seed to resolve the next nondeterministic choice. PLASMA uses an efficient iterative hash using modular arithmetic that ensures

efficient storage of the possible schedulers mapping the comparatively large set of schedulers to a smaller set of integers with a low probability of collision. For more details on this, we refer the reader to [26]. Once the nondeterministic choice is resolved at a state, PLASMA uses an independent PRNG to uniformly choose a successor state from the ones available under the chosen action. This is concatenated with the trace before generating the next hash for the nondeterministic resolution.

When working with hyperproperties, we would need to consider a tuple of schedulers and traces. In this aspect, we can either simulate traces from the same MDP using different schedulers, use different schedulers for each MDP, or a combination of both. We define a *scheduler tuple* $\underline{\sigma} \subseteq \Sigma^m$ as a tuple of schedulers sampled from the set of possible schedulers allowed by our MDPs and m is the number of scheduler quantifiers in our specification as shown in Eq. 6. We define a *trace tuple* as a tuple of traces sampled from our model based on the tuple of schedulers. Thus, $\omega^{\underline{\sigma}}$ represents the trace tuple ω, sampled from the DTMCs induced by the scheduler tuple $\underline{\sigma}$. For simplicity, we consider a one-to-one correspondence between our schedulers and MDPs. We define an indicator function $\mathbf{1}(\omega^{\underline{\sigma}} \models \varphi) \in \{0, 1\}$ that returns 1 if the trace tuple $\omega^{\underline{\sigma}}$ satisfies the hyperproperty φ, and returns 0 otherwise.

The aim is to verify the satisfaction of the given specification under all or some combination of nondeterministic choices in our system. Since a scheduler represents a concrete resolution of nondeterminism across the system, our problem is transformed to that of finding a scheduler tuple that satisfies our specification in the form of the described hypothesis in Eq. 6. Intuitively, SMC considers the proportion of the sampled trace tuples that individually satisfy the property to estimate the true satisfaction probability in the overall model. To bind the errors in the estimation, the algorithm uses precision and user-provided error margins.

For the case where we want to conclude all scheduler tuples satisfy the property, we negate the property and try to find a scheduler tuple that satisfies this negated property. The falsity of this property makes our original property true. For the case where we want to search if there exists a scheduler tuple, we pose the hypothesis directly. However, in this case, a false result does not necessarily guarantee the absence of a witness to the specification; it suggests that our algorithm was unable to find such a scheduler tuple within the given budget, error, and precision bound. Note that we cannot derive the exact scheduler tuple (we get the traces generated but not the reduced DTMC) due to the black-box nature of our sampling. We can only reason about its existence or absence within the given budget.

5.2 Implementation

In this section, we discuss the handling of the hypothesis testing of H_0 as shown in Eq. 3 in detail. The case for H_1 is similar except we use the reciprocals of $ratio_t$ and $ratio_f$. As shown in Algorithm 1, we begin by initializing the necessary parameters (line 1). For conducting sequential hypothesis testing on large systems, we need an additional bound to represent the maximum limit of resources we

want to spend on this verification. To this end, PLASMA utilizes the concept of a user-provided *budget*. Following the idea described in [26], the algorithm automatically distributes the budget to determine the number of scheduler and trace tuples the verification should consider as described in the previous section. We generate a set of scheduler tuples $\underline{\Sigma}$ and create a mapping to store which scheduler should be used to produce which trace (deriving this information from the input specification). In lines 2-3, for each scheduler tuple, we use the internal simulator to simulate the traces as specified by the mapping. In the case of multiple initial states, we allow the choice of traces with the same or different initial states. This reduces extra subformulas on the property to decide on initial states and allows us to verify the property only on relevant trace samples. In line 4, we use a custom model checker that we have implemented in PLASMA to verify linear, bounded HyperLTL properties on sets of traces sent as input. We further allow n-ary boolean operations by extending the general idea of AND, OR, XOR, etc, to reduce the length of input property the user has to provide.

Algorithm 1 Hypothesis testing on Hyperproperties

Input: MDP model: $\mathtt{M}$, spec: φ, Hypothesis $H_0 : \Pr_{\mathtt{M}}(V \models \varphi) \geq \theta$
α, β: desired type I, type II errors,
$\mathcal{N}_{max}$: simulation budget, ε: indifference region.
Output: Success, No success, Inconclusive.
1: initialize()// Initializes $\mathcal{N}, \mathcal{M}, p_0, p_1, A, B, k, ratio_t, ratio_f$
2: $\underline{\Sigma} \leftarrow \{\mathcal{M}$ tuples of k randomly chosen seeds$\}$
3: $\forall \underline{\sigma} \in \underline{\Sigma}, \forall i \in \{1, ..., \mathcal{N}\} : \omega_i^{\underline{\sigma}} \leftarrow$ simulate$(\mathtt{M}, \varphi, \underline{\sigma})$
4: $R \leftarrow \{(\underline{\sigma}, n) | \underline{\sigma} \in \underline{\Sigma} \wedge \mathbb{N} \ni n = \sum_{i=1}^{\mathcal{N}} \mathbf{1}(\omega_i^{\underline{\sigma}} \models \varphi)\}$
5: **if** canEarlyAccept(R) **then**
6: Accept H_0 and exit
7: $\underline{\Sigma} \leftarrow \{\underline{\sigma} \in \underline{\Sigma} | R(\underline{\sigma}) > 0\}, \mathcal{M} \leftarrow |\underline{\Sigma}| + 1$ // Remove null schedulers
8: **if** $|\underline{\Sigma}| = 0$ **then**
9: Quit: No suitable scheduler-tuple found
10: **while** $|\underline{\Sigma}| > 1$ **do**
11: initializeSchedulerBasedBounds() // Initializes $\alpha_M, \beta_M, A_M, B_M$
12: **for** $\underline{\sigma} \in \underline{\Sigma},\ i \in \{1, \ldots, |\underline{\Sigma}|\}$ **do**
13: $ratio_i \leftarrow 1$
14: **for** $j \in \{1, \ldots, \mathcal{N}\}$ **do**
15: **if** simulate$(\mathtt{M}, \varphi, \underline{\sigma}) \models \varphi$ **then**
16: $ratio \leftarrow ratio \cdot ratio_T$; $ratio_i \leftarrow ratio \cdot ratio_T$
17: **else**
18: $ratio \leftarrow ratio \cdot ratio_F$; $ratio_i \leftarrow ratio \cdot ratio_F$
19: **if** $ratio_i \leq A_M$ or $ratio \leq A$ **then**
20: Accept H_0 and quit: scheduler found
21: **else if** $ratio_i \geq B_M$ **then**
22: Quit iteration for $\underline{\sigma}$: Scheduler tuple rejected
23: **if** All schedulers were rejected **then**
24: Quit: No scheduler found in given budget
25: $\underline{\Sigma} \leftarrow$ filter$(\underline{\Sigma})$// Keep only the best-performing scheduler tuples
26: Inconclusive: There exists a scheduler that was neither accepted nor rejected.

In line 5 of the algorithm, we compare the ratio generated using Eq. 5 against error boundary A to check if we have already found enough witnesses to accept our null hypothesis H_0. We do not check against boundary B because the absence of a satisfying scheduler in this initial phase does not ensure that the possibility of finding such a scheduler is zero. It hints at the need for further sampling. In

line 7, we filter out the null schedulers, i.e., for which none of the trace tuples satisfied the property. Since we are looking for a scheduler tuple to satisfy the property with positive probability, null schedulers cannot definitely be our best search options. For each scheduler tuple in this filtered set, we again sample $\mathcal{N}$ trace tuples. We essentially conduct multiple independent hypothesis tests, one for each scheduler tuple. Hence, similar to [26], we modify the error for each scheduler to $\alpha_M = 1 - \sqrt[\mathcal{M}]{1-\alpha}$, $\beta_M = 1 - \sqrt[\mathcal{M}]{1-\beta}$ to account for the error correction needed. In the initial phase (lines 2-9), the idea was to check if we can satisfy the boundaries A, B using the truth value of all trace tuples sampled, irrespective of its scheduler. In the rest if the algorithm, we check if we can individually accept or reject any scheduler tuple, alongside the global check for satisfaction across all sampled trace tuples. Since our trace tuples return an overall true/false for the whole tuple, the error bound for each scheduler tuple would not change when we are working with alternation-free hyperproperties instead of trace properties.

In lines 16 and 18, we re-calculate p_{ratio} (as in Eq. 5), both for each scheduler tuple and for all the sampled trace tuples. As we encounter a satisfying tuple of traces, our overall p_{ratio} decreases as $ratio_t$ is a value less than one in this case and with each non-satisfying trace tuple, it increases. If the ratio obtained over all sampled traces across all schedulers is reduced below A or its scheduler counterpart is below A_M, we either have found a scheduler tuple that satisfies the property or over all the sampled trace tuples, we have found enough evidence to reach a conclusion that our hypothesis H_0 is satisfied.

At the end of the iteration over the scheduler tuples, we can quit the test if all our scheduler tuples are rejected, or proceed to the next iteration with only the *best* scheduler tuples. We rearrange our scheduler tuples in an ascending order based on the number of trace tuples that satisfied φ. Since we are aiming to find a scheduler tuple that satisfies φ with a probability greater than θ, we only keep the first half of rearranged scheduler tuples, ensuring that we are looking only at the schedulers that have a higher chance of exceeding the bound. If our evaluation reaches line 26, the set Σ would contain one scheduler which we were neither able to accept nor reject, reaching an inconclusive decision about H_0 within the given budget and precision margins. This inconclusive result would indicate we have to retry the experiment with a higher simulation budget and/or different precision and error margins for further scrutiny.

Convergence The algorithm will always terminate in a finite number of iterations as we eliminate half of our candidate scheduler tuples at each iteration. However, it may not have found a satisfying scheduler tuple within that boundary. For an MDP `M` and property φ, we want to find a *good* scheduler tuple, i.e., one that satisfies φ with probability $p \geq \theta - \varepsilon$. Assuming we have $|\S|$ possible scheduler tuples, and $|\S_g|$ *good* schedulers, we use $\mathbb{P}\colon \S \to [0, 1]$ to denote the probability with which a scheduler tuple satisfies φ. If we sample $\mathcal{M}$ scheduler tuples and $\mathcal{N}$ trace tuples per scheduler tuple, the probability of sampling a trace tuple from a *good* scheduler tuple that satisfies φ is,

$$\underbrace{\left(1-\left(1-\frac{|\S_g|}{|\S|}\right)^{\mathcal{M}}\right)}_{\text{good scheduler tuple}}\underbrace{\left(1-\left(1-\frac{\sum_{\sigma\in\S_g}\mathbb{P}_\sigma}{|\S_g|}\right)^{\mathcal{N}}\right)}_{\text{trace satisfies }\varphi}$$

Our aim is to maximize the value of this probability by optimizing the values of $\mathcal{M}$ and $\mathcal{N}$, across which the budget $\mathcal{N}_{max}$ is the total number of sampling we want to permit. Since we need to find schedulers that satisfy the property with probability at least θ, we set $\mathcal{N} = \lceil \frac{1}{\theta} \rceil$. This ensures that we spend our sampling budget verifying scheduler tuples that have a higher probability of satisfying our property. For example, if our θ is 0.25, $\mathcal{N} = 4$. If we want to check for our specification to be $\geq \theta$, any scheduler that satisfies at least 1 of the 4 sampled traces should be a good candidate for a *good* scheduler. In case we want to check for our specification to be $\leq \theta$, finding such *good* schedulers would help us reject the hypothesis easily. We allocate the rest of the budget (such that $\mathcal{N} \cdot \mathcal{M} \sim \mathcal{N}_{max}$) to sample scheduler tuples, thus, we set $\mathcal{M} = \lceil \theta \mathcal{N}_{max} \rceil$. We have experimented with various values of budget, adjusting them based on the expected accuracy of our results.

6 Case Studies

In this section, we discuss case studies to show the applicability and scalability of our approach. One of the main advantages of statistical model checking lies in the fact that we do not necessarily need access to the underlying model to verify a system. This allowed us to utilize our approach on sets of traces generated from black-box sources. We have separated our case studies into two sections elaborating on the models of the grey-box (where we have access to the underlying model) and black-box (where we just have access to a set of traces generated by different schedulers) examples.

6.1 Grey-box verification

Group Anonymity in Dining Cryptographers (DC) We explored the dining cryptographers problem [15] from the perspective of how it is designed to maintain anonymity. In this model, three cryptographers go out for dinner and at the end, want to figure out who paid the bill (their manager or one of them) while respecting each other's privacy. The protocol proceeds in two stages:(1) each cryptographer flips a coin and only informs the cryptographer on their right of the outcome (head or tail), (2) the cryptographers consider both the coin tosses that they know of, to declare *agree* in case the tosses were the same, or *disagree* otherwise. However, in the case of the cryptographer that actually paid, they would declare the opposite conclusion.

Given an odd number of cryptographers, we should have an odd number of agrees if the manager pays the bill, an even number of agrees if one of the cryptographers paid, and vice versa for an even number of cryptographers. We want to verify if there is any information leakage depending on which cryptographer pays.

In the model, we nondeterministically choose who pays the bill and the order in which the cryptographers toss their coin. In case one of the cryptographers pay in both traces, we expect the parity of coins at the end to be the same. As described in [9], the order of coin toss should not affect the anonymity in the protocol. This good behavior can be expressed as a hyperproperty,

$$\varphi_{DC} = \Big(\bigvee_{i \in (1,2,3)} Cpay_{i\pi_1} \wedge \bigvee_{i \in (1,2,3)} Cpay_{i\pi_2} \Big) \implies$$
$$\Diamond(done \wedge (c1 \oplus c2 \oplus c3))_{\pi_1} \bigwedge \Diamond(done \wedge (c1 \oplus c2 \oplus c3))_{\pi_2}$$

For the correctness of the model, we should not be able to find a scheduler that satisfies the bad behavior $\neg\varphi_{DC}$ with positive probability, thus, we design the hypothesis as,

$$\exists \sigma_1 . \exists \sigma_2 .\ \Pr_{\mathtt{M}}(V \not\models \varphi_{DC}) > 0$$

We expect this property to be false for an odd number of cryptographers and true for even, as our model should ensure anonymity. We experimented with both unbiased and biased coins in the model to check if that affects the parity of agreement. The main challenge for this study was the size of the models as shown in Table 1. Existing exhaustive approaches would take considerable memory and execution time to verify this model. Hence, an approximate approach like SMC has its utility here.

Noninterference in path planning (RNI) Consider the grid in Fig. 1. We have two robots moving across a two-dimensional plane subdivided (discretized) into $n \times n$ cells. The robots can nondeterministically choose to move to their neighboring cells unrestricted (up, down, left, or right) unless it is blocked by the grid boundaries. However, with a certain error probability, the chosen target cell is not reached and instead, the robot stays in its current cell. The grid can hence, be modeled as an MDP where each state models a grid cell. Note that we do not restrict or force any specific strategy for the movement of these robots. Thus, each scheduler corresponds to a specific strategy that defines how the robot moves across the grid. We consider the case where two robots (R1 and R2) are placed in opposite corners of the grid and aim to reach the goal state at the center of the grid. Assume $R1$ is our robot of interest and $R2$ is an intruder. Motivated by the idea in [21], a specification of interest would be to check if the plan of $R1$ to reach the goal is affected by the plans of $R2$. We design the hypothesis as the negation of the required property. Hence, we want to determine if there exists any such scheduler tuple where the movement of $R1$ would be similar but $R1$ fails to reach the goal in one of them. The unquantified HyperLTL formula is as follows,

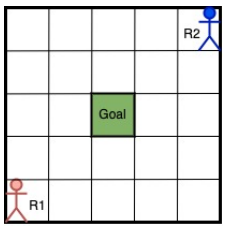


Fig. 1: Two robots attempting to reach the same goal.

$$\varphi_{RNI} = \Box\big(actR1_{\pi 1} = actR1_{\pi 2}\big) \bigwedge (\neg goalR1_{\pi 1}\, \mathcal{U}\, goalR2_{\pi 1}) \oplus (\neg goalR1_{\pi 2}\, \mathcal{U}\, goalR2_{\pi 2})$$

For any arbitrary probability p, we design our specification as,

$$\exists\sigma_1.\exists\sigma_2.\ \Pr_{\mathtt{M}}(V \vDash \varphi_{RNI}) > p$$

Current state opacity (CSO) Consider the grid in Fig. 2 where we use only one robot on the grid, which starts from any of the starting states labeled S and aims to reach the opposite corner labeled G. The gray boxes represent obstacles. Instead of analyzing reachability, we are interested in analyzing opacity similar to [54]. Opacity requires that an unauthorized user should not be able to realize the current state of the system. In the context of a robot, opacity ensures privacy is preserved as the robot moves across the grid. An observer gets an observation corresponding to each movement of the robot. Note that we have divided the grid into three regions (blue: *near initial*, red: *between obstacles*, green: *near goal*) which would generate the same observation even when the robot is in a different position. Current state opacity specifically states that while starting from the same initial state (here: either of the lower left corners marked in blue), it is still feasible to move across the grid using different paths that can produce the same observation. By different paths, we refer to cases where the actual positions of the robot are different due to the execution of different actions (up, down, left, right). This would mean that an intruder should not be able to guess the exact location by merely gathering observations about the movement of the robot. We can express this formally as,

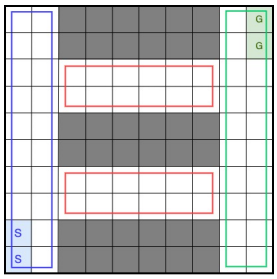


Fig. 2: Grid divided into regions to ensure opacity.

$$\varphi_{CSO} = \big(start_{\pi_1} \wedge start_{\pi_2}\big) \bigwedge \neg\Box_{\leq k}(act_{\pi_1} = act_{\pi_2}) \bigwedge \Box_{\leq k}(region_{\pi_1} = region_{\pi_2})$$

where *act* encodes the action taken by the robot on the grid and *region* denotes the corresponding region observed. We want to check if any such combination of schedulers exists that satisfies the current state opacity with respect to a given threshold. This is expressed as,

$$\exists\sigma_1.\exists\sigma_2.\ \Pr_{\mathtt{M}}(V \models \varphi_{CSO}) > p$$

6.2 Black-box verification

We use the example of a side-channel timing attack on a password checker as a black-box case study. We consider several password checkers that vary in the expected amount of information leaked by observing the execution time, resulting from different input guesses. We ran our password checkers on a microcontroller and considered numerical passwords of length 10 as input.

Following the approach described in Section 5.1, a scheduler is represented as a seed for a pseudo-random number generator. For a black-box model, the model checker calls a `python` script with one parameter (the scheduler seed) as an input. This seed is used by the model to resolve nondeterminism internally via

a pseudo-random number generator. Internally, the script uses the scheduler seed to create a random password guess. Here, we assume the password guess and the actual password is of the same length to simplify code run on the microcontroller. The number of correct digits of the generated password guess is saved and the password is forwarded to the microcontroller via the serial interface over USB. The execution time of the microcontroller together with the number of correct bits are returned by the Python script and the out-stream is parsed and interpreted by the model checker.

We convert numerical return values (rounded to a predefined level of precision), e.g., the number of correct digits (cd) or the execution time (et) to traces whose length of consecutive symbols of a type reflects those values. For instance, the returned pair of values `execution_time=4, correct_digits=1` would be converted into the trace

$$\{et, cd\} \to \{et\} \to \{et\} \to \{et\} \to \{\} \to \dots$$

Leakage of information from an unsafe password checker can be obtained by observing the execution times for several inputs. Intuitively, if the checking of a password with more consecutive correct digits (in the front) takes longer than a password with fewer correct digits, observing the execution time for multiple guesses should allow guessing the correct password. To formalize this, we use the specification of unwanted behavior

$$\varphi_{TAM} = (\Diamond(cd_{\pi_1} \wedge \neg cd_{\pi_2}) \wedge \Diamond(et_{\pi_1} \wedge \neg et_{\pi_2})) \oplus (\Diamond(cd_{\pi_2} \wedge \neg cd_{\pi_1}) \wedge \Diamond(et_{\pi_2} \wedge \neg et_{\pi_1}))$$

Consider the example of a password checker that leaks information (BB-L) in Listing 1.1. In contrast, a simple, safe approach (BB-S) checks the whole password without the option of an early return as in Line 6 and thus always produces the same execution time regardless of the correctness of the guess g. Additionally, we can also add padding to obfuscate actual execution timing. In our experiments, we consider a random delay (BB-*R) between 0 and 10 microseconds or a fixed delay (BB-*F) of 2 microseconds. We want to check, for an arbitrary probability p, whether a combination of schedulers exists, such that bad behavior, i.e., information leakage can be derived. This is expressed as,

```
1 bool checkPassword(String g){
2   int i;
3   for(i=0; i < g.length(); ++i)
4   {
5     if(g[i]!=secret[i])
6       return false;
7   }
8   return true;
9 }
```

Listing 1.1: Possible leaky password checker (BB-L).

$$\exists\sigma_1.\exists\sigma_2.\ \Pr_{\mathtt{M}}(V \models \varphi_{TAM}) > p$$

7 Experimentation/Evaluation

The model details of our grey-box case studies have been reported in Table 1. Experimental results for our case studies have been summarized in Table 2. The parameters in Table 2 refer to the number of schedulers ($\#sch$) and traces ($\#tr$)

that were sampled as determined by our algorithm, and the length of the trace (k) as determined by the user based on knowledge about the model. We separately report the time required for the sampling of the scheduler (Sim) and trace tuples and the time required to verify (Ver) the hyperproperty on them. Reported timing data is the average over 10 runs. Note that in our evaluation we do not compare our results to the existing model checkers for linear hyperproperties as they cannot handle probabilistic models with non-determinism.

7.1 Black-box verification

Experiments were run on an Intel® Core™ i7 (6x3.30 GHz) with 32Gb RAM, the password checkers ran on an `esp32` micro-controller to alleviate variance in timing due to process scheduling. To obtain results with higher precision, we execute using multiple parameter configurations - size of the indifference region (ε), the satisfying probability (θ), and sampling budget ($\mathcal{N}_{max}$). The error probabilities α, β were kept at 0.01 for the whole experiment. Results and running times for the most accurate runs are shown in Table 2, where different variants of password checkers (see also Section 6) are referenced by their labels.

In total, we have run over 480 combinations of parameters to synthesize accurate results. Table 2 lists results of parameter configurations that are maximizing the probability of satisfying the property without being inconclusive to give an estimate on a worst-case scenario. In case the property could be satisfied in the majority of the 10 runs, we show results for two configurations: one leading to a large observed probability and a second one that used a higher budget and smaller indifference region which, thus, can be expected to be more precise.

From the results, we can observe that for safe password checking the tested variants with no padding (S), fixed padding (SF), and random padding (SR) do not allow information leakage about the correctness of the password guess via correlation of the observed execution time. In contrast to this, the experiments with a leaky password generator with a fixed or no padding scheme (LF, L) allow correlating execution time and correctness of passwords. Note that in most cases the created guesses had only zero to one correct digit, as we did not implement adversarial strategies to guess larger parts of the password.

7.2 Grey-box verification

Experiments were run on an Intel® Core™ i7 (4x2.3 GHz) with 32Gb RAM. We ran experiments on each of the described case studies by scaling them across the different parameters involved. However, due to space constraints, we report cases that are sufficient to show the scalability and robustness of our approach.

The DC component in the table corresponds to the verification of the dining cryptographers protocol described in Section 6.1. Our specification φ_{DC} should not hold for an odd number of cryptographers and should hold for even ones. We have scaled the model over $\#c = \{4, 7, 8, 15\}$ and witnessed the expected results. We used a constant budget of 5000 for all the cases reported. We used models directly from PRISM [1] and were able to verify them without alterations.

We experimented with both biased and unbiased coins. The result produced was the same proving that the biases of the coins do not affect the outcome of the protocol. The experiment for the biased coin scenario used the exact same parameters as reported and yielded similar execution times for both scenarios.

Table 1: Model details of grey-box case studies

CS	Param.	#States	#Transitions	#Actions
DC	$c = 4$	2598	6864	5448
	$c = 7$	328760	1499472	1186040
	$c = 8$	1687113	8790480	6952248
	$c = 15$	10^{11}	10^{12}	9×10^{11}
RB Grid Fig.1	$n = 3$	1034	4888	2444
	$n = 5$	12346	77152	38576
	$n = 10$	256926	1852972	926486
RB Grid Fig.2	$n = 10$	200	1440	720
	$n = 20$	800	6080	3040
	$n = 30$	1800	13920	6960

The RNI section in the table corresponds to noninterference case study. We have scaled our grid for $N = \{3, 5, 10\}$. We verify the existence of scheduler tuples that fails to satisfy noninterference with probability bounds of $\{0.1, 0.2, 0.5\}$ with a budget of 2000. The trace lengths have been increased in proportion to the grid sizes. We have experimented on arbitrary trace lengths which have been adjusted as we increase the grid size. As we do not specify any smart movement strategy for the robots, these results are based on possible random walks the robots can make on the grid. The interesting observation here is the difference in execution time based on the parameters. The cases for $\theta \leq 0.1$ seem to be challenging, given the current grid and budget, resulting in an inconclusive result; for $n = 10$ our experiment ran for more than 2 days hinting at an inconclusive result within the given budget. This is expected as we are challenging the algorithm to find a scheduler with a very low probability (between 0 and 0.1) given the large search space. For $\theta \geq 0.2$, we are often able to find our target scheduler tuples in the initial sampling phase, leading to short execution time due to early exit. This is mainly because we are looking for a scheduler across a wider range of probability (between 0.2 and 1). Using $\theta \geq 0.5$ becomes challenging when scaling the model (with the same budget for comparison) due to the growing number of possible scheduler tuples, and the lack of any specific strategy that finds traces where both the robots are aiming to reach the goal. Thus, finding a scheduler with probability on the higher end (between 0.5 - 1) is not always possible in the given budget and indifference regions.

During our experiments on the opacity case study (CSO), we added a subformula to φ_{CSO} to check if the robot reaches the goal in both traces. Given that we do not enforce any smart movement strategy on the movement of the robot, it usually makes a random walk in the grid often looping in a few states for a long time. Consequently, the probability

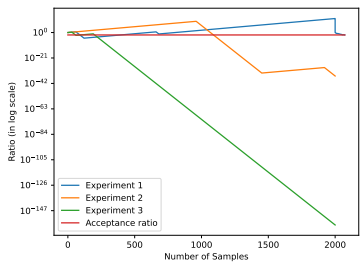


Fig. 3: DC with $n = 4$ ($\Pr \geq 0.1 \pm 0.01$)

Table 2: Data from experimentation. #sch: number of scheduler-tuples sampled, #tr: number of trace tuples sampled per scheduler tuple, k: length of traces sampled. $\alpha = \beta = 0.01$.

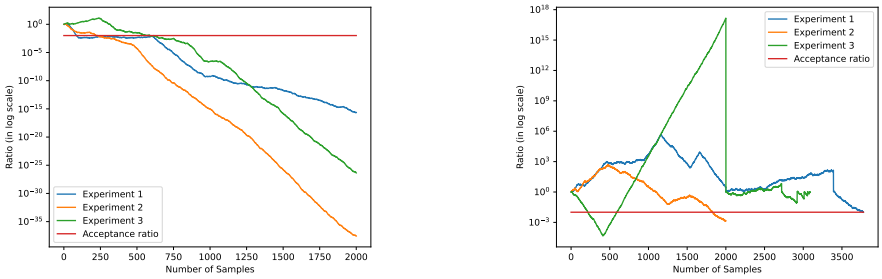
Case Study	Specification	Result	Parameters			Time [sec]	
			#sch	*#tr*	*k*	Sim.	Ver.
BB	$\Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$ # S	False	400	10	80	108	0.09
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$ # SF	False	400	10	80	93.1	0.09
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$ # SR	False	400	10	80	92.8	0.07
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.3 \pm 0.1$ # L	True	1201	4	80	102	0.1
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.25 \pm 0.01$ # L	True	1001	4	80	85.5	0.01
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.15 \pm 0.1$ # LF	True	601	7	80	92	0.09
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$ # LF	Undec	400	10	80	90	0.08
	$\Pr(V \vDash \varphi_{TAM}) \geq 0.1 \pm 0.01$ # LR	False	400	10	80	88.7	0.08
DC	$\Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$ $(\#c = 4)$	True	500	10	20	1.6	0.5
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$ $(\#c = 4)$	True	50	100	20	1.4	0.4
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$ $(\#c = 7)$	False	500	10	25	2.7	0.3
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$ $(\#c = 7)$	False	50	100	25	2.6	0.6
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$ $(\#c = 8)$	True	500	10	30	2.6	0.8
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$ $(\#c = 8)$	True	50	100	30	2.7	0.7
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.1 \pm 0.01$ $(\#c = 15)$	False	500	10	65	4.5	1.8
	$\Pr(V \nvDash \varphi_{DC}) \geq 0.01 \pm 0.001$ $(\#c = 15)$	False	50	100	65	5.1	1.9
RNI	$\Pr(V \vDash \varphi_{RNI}) \leq 0.1 \pm 0.01$ $(n = 3)$	Undec	200	10	10	385	0.3
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$ $(n = 3)$	True	400	5	10	3.8	0.2
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$ $(n = 3)$	True	1000	2	10	210	0.15
	$\Pr(V \vDash \varphi_{RNI}) \leq 0.1 \pm 0.01$ $(n = 5)$	Undec	200	10	26	2999	0.194
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$ $(n = 5)$	True	400	5	26	38.17	0.33
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$ $(n = 5)$	Undec	1000	2	26	1243	0.67
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.2 \pm 0.01$ $(n = 10)$	True	400	5	80	173.65	0.87
	$\Pr(V \vDash \varphi_{RNI}) \geq 0.5 \pm 0.01$ $(n = 10)$	Undec	1000	2	80	10.4k	1.38
CSO	$\Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$ $(n = 10)$	Undec	150	20	30	84	0.82
	$\Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$ $(n = 10)$	True	900	4	30	0.7	0.17
	$\Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$ $(n = 10)$	True	2100	2	30	0.93	0.25
	$\Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$ $(n = 20)$	Undec	150	20	45	376	0.41
	$\Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$ $(n = 20)$	True	900	4	45	2.41	0.34
	$\Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$ $(n = 20)$	True	2100	2	45	1.74	0.41
	$\Pr(V \vDash \varphi_{CSO}) \leq 0.05 \pm 0.001$ $(n = 30)$	True	150	20	55	511	0.35
	$\Pr(V \vDash \varphi_{CSO}) \geq 0.3 \pm 0.01$ $(n = 30)$	True	900	4	55	7.97	0.29
	$\Pr(V \vDash \varphi_{CSO}) \leq 0.7 \pm 0.01$ $(n = 30)$	True	2100	2	55	2.45	0.32

of the robot actually reaching the goal is highly unlikely. We checked the probability of satisfying our specification against $\{0.05, 0.3, 0.7\}$. We used a budget of 3000 for all versions of this experiment and increased the trace length in proportion to the increase in the grid size.

The plots in Fig. 3,4 depict convergence results, where each line in a graph shows how the value of *ratio* changes across a single algorithm run. In each of the plots, the red line represents the ratio A in Algorithm.1 which serves as our exit condition. In Fig. 3, we use the sampling budget of 2000 to calculate the *ratio*.

At the end of this phase, if the *ratio* is below A, we can declare that we have found enough evidence for a concrete result of the specification being satisfied as shown in lines labeled experiment 2 and 3. For the case of experiment 1, we could not reach a concrete conclusion in the initial round, as the line can be seen to be well above A. We were required to enter the main algorithm loop and required a few more samples (~ 75) to reach the same concrete conclusion.

The robotics case plotted in Fig. 4a shows that we were able to get a concrete result in the initial sampling for all three cases. We plotted an undecidable case in Fig. 4b. Note that in experiment 2 we were able to get a concrete result in the initial sampling round; in experiment 1, we were able to reach a concrete result in the main algorithm loop after intensive sampling within the chosen schedulers; and in experiment 3, we could neither find an accepting scheduler nor reject all schedulers, leading to an inconclusive result. This supports the results of undecidability that the algorithm returned. The main reason can be traced back to the fact that we did not specify any strategy for the robots, thus, sampling across random walks of the robot.



(a) RNI with $n = 4(\Pr \geq 0.2 \pm 0.01)$ (b) RNI with $n = 4$ $(\Pr \geq 0.5 \pm 0.01)$

Fig. 4: Plots showing the change in ratio based on sampling across schedulers.

8 Conclusion

We presented a probabilistic formulation of bounded, unquantified HyperLTL and provided a SMC approach to verify them over MDPs. To handle nondeterminism, our approach leverages the smart sampling algorithm presented in [26], extending it to reason about hyperproperties. We have implemented our approach as an extension of PLASMA [48] adding new capabilities to perform black-box verification and demonstrating the scalability of our approach in several case studies with large state spaces. This work aimed to showcase that SMC is a feasible solution for cases where exhaustive or bounded model checking is unable to provide us with any insight. In future directions, we would like to extend support for quantifier alternations for paths (as in HyperLTL) and scheduler tuples, as the current approach can only handle existential scheduler tuples and limits our applicability to a wider variety of security properties.

References

1. PRISM: Dining cryptographers' problem. https://www.prismmodelchecker.org/casestudies/dining_crypt.php
2. Ábrahám, E., Bonakdarpour, B.: HyperPCTL: A temporal logic for probabilistic hyperproperties. In: Proceedings of the 15th International Conference on Quantitative Evaluation of Systems (QEST). pp. 20–35 (2018)
3. Ábrahám, E., Bartocci, E., Bonakdarpour, B., Dobe, O.: Probabilistic hyperproperties with nondeterminism. In: Hung, D.V., Sokolsky, O. (eds.) Automated Technology for Verification and Analysis. pp. 518–534. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-59152-6_29
4. Agha, G., Palmskog, K.: A survey of statistical model checking. ACM Trans. Model. Comput. Simul. **28**(1) (jan 2018). https://doi.org/10.1145/3158668, https://doi.org/10.1145/3158668
5. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: 2016 IEEE 29th Computer Security Foundations Symposium (CSF). pp. 239–252. IEEE, Lisbon (Jun 2016). https://doi.org/10.1109/CSF.2016.24
6. Arora, S., Hansen, R.R., Larsen, K.G., Legay, A., Poulsen, D.B.: Statistical model checking for probabilistic hyperproperties of real-valued signals. In: Legunsen, O., Rosu, G. (eds.) Model Checking Software. p. 61–78. Springer International Publishing, Cham (2022)
7. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
8. Baumeister, J., Coenen, N., Bonakdarpour, B., Sánchez, B.F.C.: A temporal logic for asynchronous hyperproperties. In: Proceedings of the 33rd International Conference on Computer-Aided Verification (CAV). pp. 694–717 (2021)
9. Beauxis, R., Palamidessi, C.: Probabilistic and nondeterministic aspects of anonymity. Theoretical Computer Science **410**(41), 4006–4025 (2009). https://doi.org/https://doi.org/10.1016/j.tcs.2009.06.008
10. Bonakdarpour, B., Sánchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Proc. of ISoLA'18. pp. 8–27 (2018)
11. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: 2018 IEEE 31st Computer Security Foundations Symposium (CSF). pp. 162–174 (2018). https://doi.org/10.1109/CSF.2018.00019
12. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: Quantitative Evaluation of Systems: 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings 10. pp. 160–164. Springer (2013)
13. Bulychev, P., David, A., Larsen, K.G., Mikučionis, M., Poulsen, D.B., Legay, A., Wang, Z.: Uppaal-smc: Statistical model checking for priced timed automata. arXiv preprint arXiv:1207.1272 (2012)
14. Cavalcante, E., Quilbeuf, J., Traonouez, L.M., Oquendo, F., Batista, T., Legay, A.: Statistical model checking of dynamic software architectures. In: Software Architecture: 10th European Conference, ECSA 2016, Copenhagen, Denmark, November 28–December 2, 2016, Proceedings 10. pp. 185–200. Springer (2016)
15. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology **1**(1), 65–75 (Jan 1988). https://doi.org/10.1007/BF00206326

16. Clarke, E.M., Zuliani, P.: Statistical model checking for cyber-physical systems. In: Bultan, T., Hsiung, P.A. (eds.) Automated Technology for Verification and Analysis. p. 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
17. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Abadi, M., Kremer, S. (eds.) Principles of Security and Trust, vol. 8414, pp. 265–284. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
18. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: 2008 21st IEEE Computer Security Foundations Symposium. pp. 51–65. IEEE, Pittsburgh, PA, USA (2008). https://doi.org/10.1109/CSF.2008.7
19. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Proc. of CAV'19. pp. 121–139 (2019)
20. Das, S., Prabhakar, P.: Bayesian statistical model checking for multi-agent systems using hyperpctl* (2022)
21. Dimitrova, R., Finkbeiner, B., Torfah, H.: Probabilistic hyperproperties of markov decision processes. In: Proc. of ATVA 2020: the 18th International Symposium on Automated Technology for Verification and Analysis. LNCS, vol. 12302, pp. 484–500. Springer (2020). https://doi.org/10.1007/978-3-030-59152-6_27
22. Dobe, O., Abraham, E., Bartocci, E., Bonakdarpour, B.: Hyperprob: a model checker for probabilistic hyperproperties. In: Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24. pp. 657–666. Springer (2021)
23. Dobe, O., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: Model checking hyperproperties for markov decision processes. Information and Computation **289**, 104978 (2022)
24. Dobe, O., Wilke, L., Ábrahám, E., Bartocci, E., Bonakdarpour, B.: Probabilistic hyperproperties with rewards. In: NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings. pp. 656–673. Springer (2022)
25. Dwork, C.: Differential privacy. In: ICALP (2). Lecture Notes in Computer Science, vol. 4052, pp. 1–12. Springer (2006)
26. D'Argenio, P., Legay, A., Sedwards, S., Traonouez, L.M.: Smart sampling for lightweight verification of markov decision processes. International Journal on Software Tools for Technology Transfer **17**(4), 469–484 (2015)
27. Finkbeiner, B., Hahn, C., Hans, T.: MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In: Proc. of ATVA'18. pp. 521–527 (2018)
28. Finkbeiner, B., Hahn, C., Stenger, M.: Eahyper: Satisfiability, implication, and equivalence checking of hyperproperties. In: Proc. of CAV'17. pp. 564–570 (2017)
29. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Proc. of the 17th Int. Conf. on Runtime Verification. pp. 190–207 (2017)
30. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Proc. of TACAS'18. pp. 194–200 (2018)
31. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Proc. of CAV'18. pp. 144–163 (2018)
32. Finkbeiner, B., Müller, C., Seidl, H., Zalinescu, E.: Verifying Security Policies in Multi-agent Workflows with Loops. In: Proc. of CCS'17 (2017)

33. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 30–48. Lecture Notes in Computer Science, Springer International Publishing (2015)
34. Gadyatskaya, O., Hansen, R.R., Larsen, K.G., Legay, A., Olesen, M.C., Poulsen, D.B.: Modelling attack-defense trees using timed automata. In: Formal Modeling and Analysis of Timed Systems: 14th International Conference, FORMATS 2016, Quebec, QC, Canada, August 24-26, 2016, Proceedings 14. pp. 35–50. Springer (2016)
35. Gilbert, D.R., Donaldson, R.: A monte carlo model checker for probabilistic ltl with numerical constraints (2008)
36. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symp. on Security and Privacy. pp. 11–20 (1982)
37. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Vojnar, T., Zhang, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. p. 115–131. Springer International Publishing, Cham (2019)
38. Henriques, D., Martins, J.G., Zuliani, P., Platzer, A., Clarke, E.M.: Statistical model checking for markov decision processes. In: 2012 Ninth International Conference on Quantitative Evaluation of Systems. pp. 84–93 (2012). https://doi.org/10.1109/QEST.2012.19
39. Hsu, T., Bonakdarpour, B., Finkbeiner, B., Sánchez, C.: Bounded model checking for asynchronous hyperproperties. CoRR **abs/2301.07208** (2023). https://doi.org/10.48550/arXiv.2301.07208, https://doi.org/10.48550/arXiv.2301.07208
40. Hsu, T., Bonakdarpour, B., Sánchez, C.: Hyperqube: A qbf-based bounded model checker for hyperproperties. CoRR **abs/2109.12989** (2021), https://arxiv.org/abs/2109.12989
41. Hsu, T.H., Sánchez, C., Bonakdarpour, B.: Bounded model checking for hyperproperties. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. p. 94–112. Springer International Publishing, Cham (2021)
42. III, J.W.G., Syverson, P.F.: A logical approach to multilevel security of probabilistic systems. Distributed Comput. **11**(2), 73–90 (1998)
43. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker mrmc. Performance evaluation **68**(2), 90–104 (2011)
44. Kwiatkowska, M., Norman, G., Parker, D.: Prism 4.0: Verification of probabilistic real-time systems. In: Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23. pp. 585–591. Springer (2011)
45. Larsen, K.G., Legay, A.: 30 years of statistical model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12476, pp. 325–330. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_18, https://doi.org/10.1007/978-3-030-61362-4_18
46. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) Runtime Verification. p. 122–135. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)

47. Legay, A., Lukina, A., Traonouez, L.M., Yang, J., Smolka, S.A., Grosu, R.: Statistical model checking. In: Computing and software science: state of the art and perspectives, pp. 478–504. Springer (2019)
48. Legay, A., Sedwards, S.: On statistical model checking with plasma. In: The 8th International Symposium on Theoretical Aspects of Software Engineering (2014)
49. O'Neill, K.R., Clarkson, M.R., Chong, S.: Information-flow security for interactive programs. In: CSFW. pp. 190–201. IEEE Computer Society (2006)
50. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. ieee (1977)
51. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties. In: ter Beek, M.H., McIver, A., Oliveira, J.N. (eds.) Formal Methods – The Next 30 Years. p. 406–424. Springer International Publishing, Cham (2019)
52. Wald, A.: Sequential Tests of Statistical Hypotheses. The Annals of Mathematical Statistics **16**(2), 117 – 186 (1945). https://doi.org/10.1214/aoms/1177731118, https://doi.org/10.1214/aoms/1177731118
53. Wang, Y., Nalluri, S., Bonakdarpour, B., Pajic, M.: Statistical model checking for hyperproperties. In: IEEE Computer Security Foundations Symposium. pp. 1–16. Dubrovnik, Croatia (2021)
54. Wang, Y., Nalluri, S., Pajic, M.: Hyperproperties for robotics: Planning via hyperltl. In: 2020 IEEE International Conference on Robotics and Automation (ICRA). pp. 8462–8468 (2020). https://doi.org/10.1109/ICRA40945.2020.9196874
55. Wang, Y., Zarei, M., Bonakdarpour, B., Pajic, M.: Statistical verification of hyperproperties for cyber-physical systems. ACM Transactions in Embedded Computing Systems **18**(5), 92– (2019). https://doi.org/10.1145/3358232
56. Younes, H.L.: Ymer: A statistical model checker. In: Computer Aided Verification: 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings 17. pp. 429–433. Springer (2005)
57. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: 16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA. p. 29. IEEE Computer Society (2003). https://doi.org/10.1109/CSFW.2003.1212703
58. Zuliani, P.: Statistical model checking for biological applications. International Journal on Software Tools for Technology Transfer **17**, 527–536 (2015)