Stack-Aware Hyperproperties*

Ali Bajwa[2](✉), Minjian Zhang[1], Rohit Chadha[2](✉), and Mahesh Viswanathan[1]

[1] University of Illinois, Urbana-Champaign, USA
{minjian2,vmahesh}@illinois.edu
[2] University of Missouri, Columbia, USA
{azb9q8,chadhar}@missouri.edu

Abstract. A hyperproperty relates executions of a program and is used to formalize security objectives such as confidentiality, non-interference, privacy, and anonymity. Formally, a hyperproperty is a collection of allowable sets of executions. A program violates a hyperproperty if the set of its executions is not in the collection specified by the hyperproperty. The logic HyperCTL* has been proposed in the literature to formally specify and verify hyperproperties. The problem of checking whether a finite-state program satisfies a HyperCTL* formula is known to be decidable. However, the problem turns out to be undecidable for procedural (recursive) programs. Surprisingly, we show that decidability can be restored if we consider restricted classes of hyperproperties, namely those that relate only those executions of a program which have the same call-stack access pattern. We call such hyperproperties, *stack-aware hyperproperties*. Our decision procedure can be used as a proof method for establishing security objectives such as noninference for recursive programs, and also for refuting security objectives such as observational determinism. Further, if the call stack size is observable to the attacker, the decision procedure provides exact verification.

Keywords: Hyperproperties · Temporal Logic · Recursive Programs · Model Checking · Pushdown Systems · Visibly Pushdown Automata.

1 Introduction

Temporal logics HyperLTL and HyperCTL* [5] were designed to express and reason about security guarantees that are *hyperproperties* [6]. A hyperproperty [6] is a security guarantee that does not depend solely on individual executions. Instead, a hyperproperty relates multiple executions. For example, non-interference, a confidentiality property, states that any *two* executions of a program that differ only in high-level security inputs must have the same *low*-security observations. As pointed out in [6], several security guarantees are hyperproperties. The logic HyperCTL* subsumes HyperLTL, and the problem of checking a finite-state system against a HyperCTL* formula is decidable [5].

* Ali Bajwa was partially supported by NSF CNS 1553548. Rohit Chadha was partially supported by NSF CNS 1553548 and NSF SHF 1900924. Mahesh Viswanathan and Minjian Zhang were partially supported by NSF SHF 1901069 and NSF SHF 2007428.

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13993, pp. 308–325, 2023.
https://doi.org/10.1007/978-3-031-30823-9_16

In this paper, we consider the problem of model checking procedural (recursive) programs against security hyperproperties. Recall recursive programs are naturally modeled as a pushdown system. Unlike the case of finite-state transition systems, the problem of checking whether a pushdown system satisfies a HyperCTL* formula is undecidable [16]. In contrast, CTL* model checking is decidable for pushdown systems [3,18].

Our contributions. We consider restricted classes of hyperproperties for recursive programs, namely those that relate only those executions that have the same *call-stack access pattern*. Intuitively, two executions have the same stack access pattern if they access the call stack in the same manner at each step, i.e., if in one execution there is a push (pop) at a point, then there is a push (pop) at the same point in the other execution. Observe that if two executions have the same stack access pattern, then their stack sizes are the same at all times. We call such hyperproperties, *stack-aware hyperproperties.*

In order to specify stack-aware hyperproperties, we extend HyperCTL* to sHCTL*. The logic sHCTL* has a two level syntax. At the first level, the syntax is identical to HyperCTL* formulas, and is interpreted over executions of the pushdown system with the same stack access pattern. At the top-level, we quantify over different stack access patterns. The formula $E\psi$ is true if for some stack access pattern ρ of the system, the pushdown system restricted to executions with stack access pattern ρ satisfies the HyperCTL* formula ψ. The formula $A\psi$ is true if for each stack access pattern ρ of the system, the pushdown system restricted to executions with stack access pattern ρ satisfies the HyperCTL* formula ψ. See Figure 1 on Page 8 for a side-by-side comparison of the syntax for HyperCTL* and sHCTL*. HyperLTL is extended to sHLTL similarly. Please note that sHCTL* subsumes sHLTL, and that sHCTL* (sHLTL) coincides with HyperCTL* (HyperLTL) for finite state systems as all executions of finite state systems have the same stack access pattern.

We show that the model checking problem for sHCTL* is decidable. We demonstrate three different ways this result can aid in verifying recursive programs. First, for security guarantees such as noninference [14], which are expressible in the $\forall\exists^*$ fragment of HyperLTL, we can use the model checking algorithm to establish that a recursive program satisfies the HyperLTL property. Secondly, for the $\forall^*$ fragment of HyperLTL, the model checking algorithm can be used to detect security flaws by establishing that a recursive program does not satisfy security guarantees. Observational determinism [13,19] is an example of such a property. Finally, when the attacker can observe stack access patterns (or, equivalently, stack sizes), we can get exact verification for several properties. The assumption of the attacker observing stack access patterns holds, for example, in the program counter security model [15] in which the attacker has access to program counters at each step. As argued in [15], the program security model is appropriate to capture control-flow side channels such as those arising from timing behavior and/or disclosure of errors.

The decision procedure uses an automata-theoretic approach inspired by the model checking algorithm for finite state systems and HyperCTL* given in [10]. Since stack-aware hyperproperties relate only executions with the same stack access-pattern, a set of executions with the same stack access pattern can be encoded as a word over a *pushdown* alphabet, [3] and the problem of model checking a sHCTL* formula can be reduced to the problem of checking emptiness of a *non-deterministic visibly pushdown automaton (NVPA)* over infinite words [1]. The reduction of the model checking problem to the emptiness problem is based on a compositional construction of an automaton for each sub-formula which accepts exactly the set of assignments to path variables that satisfy the sub-formula. For this construction to be optimal, we carefully leverage two equi-expressive classes of automata on infinite words, namely NVPAs and *1-way alternating jump automata (1-AJA)* [4]. The model checking algorithm for sHCTL* against procedural programs has a complexity that is very close to the complexity of model checking finite state systems against HyperCTL*. If $g(k,n)$ denotes a tower of exponentials of height k, where the top most exponent is $\mathsf{poly}(n)$, then for a formula with formula complexity r, [4] and a system and formula whose size is bounded by n, our algorithm is in $\mathsf{DTIME}(g(\lceil \frac{r}{2} \rceil, n))$. In comparison, model checking finite state systems against HyperCTL* is in $\mathsf{NSPACE}(g(\lceil \frac{r}{2} \rceil - 1, n))$. This slight difference in complexity is consistent with checking other properties like invariants for finite state systems (NL) versus procedural programs (P).

We also prove that our model checking algorithm is optimal by proving a matching lower bound. Our proof showing $\mathsf{DTIME}(g(\lceil \frac{r}{2} \rceil, n)$-hardness of the model checking problem for formulas with (formula) complexity r, relies on reducing the membership problem for $g(\lceil \frac{r}{2} \rceil - 1, n)$ space bounded *alternating Turing machines* (ATM) to the model checking problem. The reduction requires identifying an encoding of computations of ATMs, which are trees, as strings that can be guessed and generated by pushdown systems. The pushdown system we construct for the model checking problem guesses potential computations of the ATM, while the sHCTL* formula we construct checks if the guessed computation is a valid accepting computation.

Related work. Clarkson and Schneider introduced *hyperproperties* [6] and demonstrated their need to capture complex security properties. Temporal logics HyperLTL and HyperCTL*, that describe hyperproperties, were introduced by Clarkson et al. [5]. They also characterized the complexity of model checking finite state transition systems against HyperCTL* specifications by a reduction to the satisfiability problem of QPTL [17]. Subsequently, other model checking algorithms for verifying finite state systems against HyperCTL* properties have been proposed [10,7]. Tools that check satisfiability [8] and runtime verification [9] for HyperLTL formulas have also been developed. Finkbeiner et al. introduced the automata-theoretic approach to model checking HyperCTL* for finite-state systems [10].

[3] A pushdown alphabet is an alphabet that is partitioned into three sets: a set of call symbols, a set of internal symbols, and a set of return symbols. See Section 4.1.

[4] Our definition of formula complexity is roughly double the usual notion of quantifier alternation. For a precise definition, see Definition 4.

The model checking problem for HyperLTL, and consequently HyperCTL*, was shown to be undecidable for pushdown systems in [16]. For restricted fragments of HyperLTL, Pommellet and Tayssir [16] introduced over-approximations and under-approximations to establish/refute that a pushdown system satisfies a HyperLTL formula in those fragments. Gutsfeld et al. introduced stuttering H_μ, a *linear* time logic for checking asynchronous hyperproperties for recursive programs in [12]. The authors present complexity results for the model checking problem under an assumption of *fairness*, and a restriction of *well-alignment*. While the restriction to paths with the same *stack access pattern* is similar to the well-alignment restriction, we do not assume any fairness condition to establish decidability. However, as sHCTL* is a branching time logic and only considers synchronous hyperproperties, the two logics are not directly comparable. It is also worth mentioning that the branching nature of sHCTL* requires us to "copy" a potentially unbounded stack, from the most recently quantified path variable, when assigning a path to the "current" quantified path variable. In contrast, all path assignments in [12] start with an empty stack.

For lack of space reasons, some proofs are omitted and can be located in [2].

2 Motivation

Clarkson and Schneider [6] argue that many important *security* guarantees are expressible only as *hyperproperties*. We discuss two examples of security hyperproperties, and the logics HyperLTL and HyperCTL* used to specify them.

Hyperproperties and temporal logics. We discuss two variants of non-interference [11] that model confidentiality requirements. In non-interference, the inputs of a system are partitioned into *low*-level input security variables and *high*-level input security variables. The attacker is assumed to know the values of low-level security inputs. During an execution, the attacker can observe parts of the system configuration such as system outputs, or the memory usage. A system satisfies *non-interference* if the attacker cannot deduce the values of high-level inputs from the low-level observations. To formally specify the variants, we use the logic HyperLTL [5], a fragment of the logic HyperCTL* [5]. The precise syntax of HyperLTL and HyperCTL* is shown in Fig. 1. In the syntax, π is a path variable and the formula a_π is true if the proposition a is true along the path "π". Intuitively, the formula $\exists\pi.\,\psi$ is existential quantification over paths, and is true if there is a path that can be assigned to π such that ψ is true. Universal quantification ($\forall\pi.\,\psi$), and other logical connectives such as conjunction ($\wedge$), implication ($\rightarrow$), equivalence ($\leftrightarrow$) and the temporal operators G and F can be defined in the standard way. By having explicit path variables, HyperLTL and HyperCTL* allow quantification over multiple paths simultaneously.

Example 1. The first variant, noninference [14], states that for each execution σ of a program, there is another execution σ' such that (a) σ' is obtained from σ by replacing the high-level security inputs by a dummy input, and (b) σ and σ' have the same low-level observations. Noninference is a hyperliveness property [5,6].

Let us assume that the low-level observations of a configuration are determined by the values of the propositions in $L = \{\ell_1, \cdots \ell_m\}$. As shown in [5], noninference is expressible by the HyperLTL formula: $\mathsf{NI} \overset{\text{def}}{=} \forall\pi.\,\exists\pi'.(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi'$. Here $\mathsf{G}\,\lambda_{\pi'}$ expresses that G*lobally* (or in each configuration of the execution) the high input of π' is the dummy input λ, and $\pi \equiv_L \pi' \overset{\text{def}}{=} \mathsf{G}(\wedge_{\ell \in L}(\ell_\pi \leftrightarrow \ell_{\pi'}))$ expresses that π and π' have the same low-level observations.

Example 2. The second variant, observational determinism [13,19], states that any two executions that have the same low-level initial inputs, must have the same low-level output observations. Observational determinism is a hypersafety property [5,6], and is also expressible in HyperLTL using the formula [5]: $\mathsf{OD} \overset{\text{def}}{=} \forall\pi.\,\forall\pi'.(\pi[0] \equiv_{L,in} \pi'[0]) \rightarrow \pi \equiv_{L,out} \pi'$. Here $\equiv_{L,in}$ and $\equiv_{L,out}$ express the fact that π and π' have the same low-security inputs and outputs respectively.

Procedural (recursive) programs and Stack-aware hyperproperties. Pushdown systems model procedural programs that do not dynamically allocate memory, and whose program variables take values in finite domains. Unlike finite-state transition systems, the problem of checking whether a pushdown system satisfies a HyperCTL* formula is undecidable [16]. However, we identify a natural class of hyperproperties for which the model checking problem becomes decidable. As we shall shortly see, this class of hyperproperties not only enjoys decidability, but is also useful in reasoning about security hyperproperies such as noninference and observational determinism.

We consider a restricted class of hyperproperties for recursive programs, which relate only executions that access the call stack in the same manner, i.e., push or pop at the same time. An execution of a pushdown system $\mathcal{P}$ is a sequence of configurations (control state + stack) $\sigma = c_1 c_2 \cdots$, such that the stacks of consecutive configurations c_i and c_{i+1} differ only due to the possible presence of an additional element at the top of the stack of either c_i or c_{i+1}. For such a sequence, we can associate a sequence $\mathsf{pr}(\sigma) = \mathsf{o}_1\mathsf{o}_2 \cdots$ such that $\mathsf{o}_i \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$ such that $\mathsf{o}_i = \mathsf{call}$ (ret respectively) if and only if the stack in c_{i+1} has one more (less respectively) element than c_i. The sequence $\mathsf{pr}(\sigma)$ is said to be the *stack access pattern* of σ. Observe that the stack sizes of two executions with the same stack access pattern evolve in a similar fashion. Thus, equivalently, we can consider this class of hyperproperties to be the hyperproperties that relate executions with identical memory usage.

To specify these hyperproperties, we propose the logic sHCTL* which extends HyperCTL*. sHCTL* has a two level syntax. At the innermost level, the syntax is identical to that of HyperCTL* formulas, but is interpreted over executions of the pushdown system with the same stack access pattern. At the outer level, we quantify over different stack access patterns. Intuitively, the formula $E\psi$ is true if there is a stack access pattern ρ exhibited by the system such that the set of executions with access pattern ρ satisfy the hyperproperty ψ. The dual formula $A\psi$, defined as $\neg E \neg \psi$, is true if for each stack access pattern ρ exhibited by the system, the set of all executions with stack access pattern ρ

satisfy ψ. The syntax of sHLTL is obtained from HyperLTL in a similar fashion. Please see Fig. 1 on Page 8 for a side-by-side comparison of the syntax of HyperCTL* (HyperLTL) and sHCTL* (sHLTL). Unlike HyperCTL*, we show that the problem of checking sHCTL* is decidable for pushdown systems (Theorem 3). Formal definitions of stack access patterns, syntax and semantics of sHCTL* are in Section 3.

For the rest of the paper, hyperproperties expressible in sHCTL* will be called *stack-aware hyperproperties.* Restricting to stack-aware hyperproperties is useful in verifying security guarantees of recursive programs as discussed below.

Proving $\forall\exists^*$ hyperproperties. The *noninference* property (Example 1) can be expressed in HyperLTL as $\mathsf{NI} \overset{\text{def}}{=} \forall\pi.\,\exists\pi.'(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi'$. Consider the sHLTL formula $A(\mathsf{NI})$ obtained by putting an A in front NI. A pushdown system satisfies $A(\mathsf{NI})$ only if for each execution σ of the system, there is another execution σ' with *the same stack access pattern as* σ such that σ, σ' together satisfy $(\mathsf{G}\,\lambda_{\sigma'}) \wedge \sigma \equiv_L \sigma'$. Thus, if the pushdown system satisfies the sHLTL formula $A(\mathsf{NI})$, then it also satisfies noninference. Thus, a decision procedure for sHLTL can be used to prove that a recursive program satisfies noninference.

The above observation generalizes to HyperLTL formulas of the form $\psi = \forall\pi.\exists\pi_1.\ldots\exists\pi_k.\psi'$ — if a system satisfies the sHLTL formula $A\psi$ then it must also satisfy the HyperLTL formula ψ. Though the model checking problem is undecidable for pushdown systems even when restricted to such HyperLTL formulas, we gain decidability by restricting the search space for $\pi, \pi_1, \ldots, \pi_k$.

Refuting $\forall^*$ hyperproperties. *Observational determinism* (Example 2) can be written in HyperLTL as $\mathsf{OD} \overset{\text{def}}{=} \forall\pi.\,\forall\pi'.(\pi[0] \equiv_{L,in} \pi'[0]) \to \pi \equiv_{L,out} \pi'$. Consider the sHLTL formula $A(\mathsf{OD})$. A pushdown system *fails* to satisfy the sHLTL formula $A(\mathsf{OD})$ only if there is a stack access pattern ρ and executions σ_1 and σ_2 with stack access pattern ρ such that the pushdown system does not satisfy $(\sigma[0] \equiv_{L,in} \sigma'[0]) \to \sigma \equiv_{L,out} \sigma'$.

This observation generalizes to HyperLTL formulas of the form $\psi = \forall\pi_1.\ldots\forall\pi_k.\psi'$ — if a pushdown system fails to satisfy the sHLTL formula $A\psi$ then it does not satisfy ψ. Even though model checking pushdown systems against such restricted specifications is undecidable, our decision procedure can be used to show that a recursive program does not meet such properties.

Exact verification when stack access pattern is observable. Often, it is reasonable to assume that the attacker can observe the stack access pattern. For example, in the program counter security model [15], the attacker has access to the program counter transcript, i.e., the sequence of program counters during an execution. Access to the program counter transcript implies that the attacker can observe stack access pattern. The assumption that the program counter transcript is observable helps model control flow side channel attacks which include timing attacks and error disclosure attacks [15]. sHCTL* can be used to verify security guarantees in this security model. For example, the sHCTL* formula $A(\mathsf{NI})$ models noninference faithfully by introducing a unique proposition for each control state. Observational determinism can also be verified in this model by suitably transforming the pushdown automaton.

Another scenario in which stack access patterns are observable is when the attacker can observe the memory usage of a program in terms of stack size. As observing stack size may lead to information leakage, stack size should be considered a low-level observation. Since the stack size can be unbounded, it cannot be modeled as a proposition. sHCTL*, however, can still be used to verify security guarantees in this case. For example, $A(\mathsf{NI}) = A(\forall\pi.\,\exists\pi.'(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi')$ faithfully models non-inference as semantics of sHCTL* forces π and π' to have the same call-stack size in addition to other low-level observations. Once again, observational determinism can also be verified in this model by suitably transforming the pushdown automaton.

3 Stack-aware Hyper Computation Tree Logic (sHCTL*)

Stack-aware Hyper Computation Tree Logic (sHCTL*), and its sub-logic Stack-aware Hyper Linear Temporal Logic (sHLTL) are formally presented. We begin by establishing some conventions over strings.

Strings. A *string/word* w over a finite alphabet Σ is a sequence $w = a_0a_1\cdots$ of finite or infinitely many symbols from Σ, i.e., $a_i \in \Sigma$ for all i. The *length* of a string w, denoted $|w|$, is the number of symbols appearing in it — if $w = a_0a_1\cdots a_{n-1}$ is finite then $|w| = n$, and if $w = a_0a_1\cdots$ is infinite then $|w| = \omega$. The *unique* string of length 0, the *empty string*, is denoted ε. For a string $w = a_0a_1\cdots a_i\cdots$, $w(i) = a_i$ denotes the ith symbol, $w[\,:i] = a_0a_1\cdots a_{i-1}$ denotes the prefix of length i, $w[i:\,] = a_ia_{i+1}\cdots$ denotes the suffix of w starting at position i, and $w[i:j] = a_ia_{i+1}\cdots a_{j-1}$ denotes the substring from position i (included) to position j (not included). Thus $w[0:\,] = w$. By convention, when $i \leq 0$, we take $w[\,:i] = \varepsilon$. Over Σ, the set of all finite strings is denoted Σ^*, and the set of all infinite strings is denoted Σ^ω. For a finite string u and a (finite or infinite) string v, uv denotes the *concatenation* of u and v.

3.1 Pushdown Systems

Pushdown systems naturally model for sequential recursive programs. Formally, an AP-*labeled pushdown system* is a tuple $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$, where S is a finite set of *control states*, Γ is a finite set of *stack symbols*, $s_{\mathsf{in}} \in S$ is the *initial control state*, $L : S \to 2^{\mathsf{AP}}$ is the *labeling function*, and Δ is the transition relation. The transition relation $\Delta = \Delta_{\mathsf{int}} \uplus \Delta_{\mathsf{call}} \uplus \Delta_{\mathsf{ret}}$ is the disjoint union of *internal transitions* $\Delta_{\mathsf{int}} \subseteq S \times S$ where the stack is unchanged, *call transitions* $\Delta_{\mathsf{call}} \subseteq S \times (S \times \Gamma)$ where a single symbol is *pushed* onto the stack, and *return transitions* $\Delta_{\mathsf{ret}} \subseteq (S \times \Gamma) \times S$ where a single symbol is *popped* from the stack. When AP is clear from the context, we simply refer to them as pushdown systems.

Transition System Semantics. We recall the standard semantics of a pushdown system as a transition system. Let us fix a pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$. A *configuration* c of $\mathcal{P}$ is a pair (s, α) where $s \in S$ and $\alpha \in \Gamma^*$.

$$a \in \mathsf{AP}, \pi \in \mathcal{V}$$

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

(a) HyperCTL*

$$\begin{aligned} \theta &::= E\psi \mid \neg\theta \mid \theta \vee \theta \\ \psi &::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathsf{X}\psi \mid \psi \,\mathsf{U}\, \psi \mid \exists\pi.\,\psi \end{aligned}$$

(b) sHCTL*

Fig. 1: BNF for HyperCTL* and sHCTL*. Let $\forall$ denote $\neg\exists\neg$ and A denote $\neg E\neg\psi$. HyperLTL is the set of HyperCTL* formulas $Q_1\pi_1.\cdots Q_r\pi_r.\psi$ where $Q_i \in \{\exists, \forall\}$ and ψ is quantifier-free. sHLTL is the set of sHCTL* formulas $\mathbb{E}\varphi$, where $\mathbb{E} \in \{A, E\}$ and φ is in HyperLTL.

The set of all configurations of $\mathcal{P}$ will be denoted $\mathsf{Conf}_\mathcal{P} = S \times \Gamma^*$. The *labeled transition system* associated with $\mathcal{P}$ is $[\![\mathcal{P}]\!] := (\mathsf{Conf}_\mathcal{P}, c_{\mathsf{in}}, \longrightarrow, \mathsf{AP}, L)$ where $c_{\mathsf{in}} = (s_{\mathsf{in}}, \varepsilon)$ is the *initial configuration*, $\longrightarrow \subseteq \mathsf{Conf}_\mathcal{P} \times (\{\mathsf{call}, \mathsf{ret}, \mathsf{int}\} \times S \times (\Gamma \cup \{\varepsilon\}) \times S) \times \mathsf{Conf}_\mathcal{P}$ is the *transition relation*, and L is the *labeling function* that extends the labeling function of $\mathcal{P}$ to configurations as follows: $L(s, \alpha) = L(s)$. The transition relation $\longrightarrow$ is defined to capture the informal semantics of internal, call, and return transitions — for any $\alpha \in \Gamma^*$, (int) $(s, \alpha) \xrightarrow{(\mathsf{int}, s, \varepsilon, s')} (s', \alpha)$ iff $(s, s') \in \Delta_{\mathsf{int}}$; ($\mathsf{call}$) $(s, \alpha) \xrightarrow{(\mathsf{call}, s, a, s')} (s', a\alpha)$ iff $(s, (s', a)) \in \Delta_{\mathsf{call}}$; and ($\mathsf{ret}$) $(s, a\alpha) \xrightarrow{(\mathsf{ret}, s, a, s')} (s', \alpha)$ iff $((s, a), s') \in \Delta_{\mathsf{ret}}$.

A *path* of $[\![\mathcal{P}]\!]$ is an infinite sequence of configurations $\sigma = c_0, c_1, \ldots$ such that for each i, $c_i \xrightarrow{(\mathsf{o}, s, a, s')} c_{i+1}$ for some $\mathsf{o} \in \{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$, $s, s' \in S$ and $a \in \Gamma \cup \{\varepsilon\}$. The path σ is said to *start* in configuration c_0 (the first configuration in the sequence). We will use $\mathsf{Paths}([\![\mathcal{P}]\!], c)$ to denote the set of paths of $[\![\mathcal{P}]\!]$ starting in the configuration c and $\mathsf{Paths}([\![\mathcal{P}]\!])$ to denote all paths of $[\![\mathcal{P}]\!]$.

We conclude this section by introducing some notation on configurations. For $c = (s, \alpha)$, its *stack height* is $|\alpha|$, its *control state* is $\mathsf{state}(c) = s$, and its *top of stack symbol* is $\mathsf{top}(c) = a \in \Gamma$ if $\alpha = a\alpha'$ and is undefined if $\alpha = \varepsilon$.

3.2 Syntax of sHCTL*

Let us fix a set of atomic propositions AP, and a set of path variables, $\mathcal{V}$. The BNF grammar for sHCTL* formulas is given in Figure 1(b). In the BNF grammar, $a \in \mathsf{AP}$ is an *atomic proposition*, π is a *path variable*, ψ is a *cognate formula*, and θ is a sHCTL* formula. The syntax has two levels, with the inner level identical to HyperCTL* formulas, while the outer level allows quantification over different stack access patterns (see Section 3.3). Also, following [5,10], we assume that the until operator U only occurs within the scope of a path quantifier.

Remark 1. We have chosen to not have A, the dual of E, and conjunction as explicit logical operators to keep our exposition simple. This choice does makes the automata constructions presented here less efficient for formulas involving conjunction. Adding them explicitly does not pose a technical challenge to our setup and our automata constructions can be extended to handle them explicitly. In addition, we will sometimes use other quantifiers and logical operators to write formulas. Some standard examples include: $\theta_1 \wedge \theta_2 = \neg(\neg\theta_1 \vee \neg\theta_2)$, where θ_i ($i \in \{1,2\}$) is either a sHCTL* or cognate formula; $\forall\pi.\psi = \neg\,\exists\pi.\,\neg\psi$; $\mathsf{F}\,\psi = \mathsf{true}\,\mathsf{U}\,\psi$, where $\mathsf{true} = a_\pi \vee \neg a_\pi$; $\mathsf{G}\,\psi = \neg\,\mathsf{F}\,\neg\psi$.

We call formulas of the form $\mathbb{E}\psi$ (where $\mathbb{E} \in \{A, E\}$ and ψ is a cognate formula) *basic formulas*. Observe that any sHCTL* formula is a Boolean combination of basic formulas. A sHCTL* formula θ is a *sentence* if in each basic sub-formula $\mathbb{E}\psi$, ψ is a sentence, i.e., every path variable appearing in ψ is quantified. Without loss of generality, we assume that in any cognate formula ψ, all bound variables in ψ are renamed to ensure that any path variable is quantified at most once. We will only consider sHCTL* sentences in this paper. The logic sHLTL is the sub-logic of sHCTL* consisting of all formulas of the form $\mathbb{E}Q_1\pi_1.\cdots Q_r\pi_r.\psi$ where $\mathbb{E} \in \{A, E\}$, $Q_i \in \{\exists, \forall\}$ and ψ is quantifier free.

3.3 Semantics of sHCTL*

The syntax of cognate formulas is identical to that HyperCTL* formulas. Their semantics will be described in a similar manner, in a context where free path variables in the formula are interpreted as executions of a system. However, we will require that the interpretations of every path variable share a *common* stack access pattern — hence the term *cognate*. Thus, before defining the semantics, we will define what we mean by the *stack access pattern* of a path and a *path environment* that assigns an interpretation to path variables.

For the rest of this section let us fix a pushdown system $\mathcal{P} = (S, \mathit{\Gamma}, s_{\mathsf{in}}, \mathit{\Delta}, L)$. A string $w \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ is said to be *well matched* if either $w = \varepsilon$ or $w = \mathsf{int}$ or $w = \mathsf{call}\; u\; \mathsf{ret}$ or $w = uv$, where $u, v \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ are (recursively) well matched. In a string $\rho \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$, $\rho(i)$ is an *unmatched return*, if $\rho[\,:i+1] = w\,\mathsf{ret}$, where w is well matched. We are now ready to present the definition of a stack access pattern.

Definition 1 (Stack access pattern). *A string $\rho \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ is a* stack access pattern *if the set $\{i \in \mathbb{N} \mid \rho(i)$ is an unmatched return$\}$ is finite.*

A path $\sigma = c_0c_1c_2\cdots \in \mathsf{Paths}([\![\mathcal{P}]\!])$ is said to have a stack access pattern $\rho = o_0o_1\cdots$ (denoted $\mathsf{pr}(\sigma) = \rho$) if for every i: (a) $o_i = \mathsf{call}$ if and only if $\mathsf{stack}(c_{i+1}) = \mathsf{top}(c_{i+1})\,\mathsf{stack}(c_i)$, (b) $o_i = \mathsf{int}$ if and only if $\mathsf{stack}(c_{i+1}) = \mathsf{stack}(c_i)$, and (c) $o_i = \mathsf{ret}$ if and only if $\mathsf{stack}(c_i) = \mathsf{top}(c_i)\,\mathsf{stack}(c_{i+1})$.

We now present the definition of *path environment* that interprets the free path variables in a cognate formula as paths of $[\![\mathcal{P}]\!]$ such that they share a common stack access pattern. This plays a key role in defining the semantics of sHCTL*. For a set of path variables $\mathcal{V}$, let $\mathcal{V}^\dagger$ be defined as the set $\mathcal{V} \uplus \{\dagger\}$.

Definition 2 (Path Environment). *A* path environment *for pushdown system $\mathcal{P}$ over variables $\mathcal{V}$ is function $\mathit{\Pi} : \mathcal{V}^\dagger \to \mathsf{Paths}([\![\mathcal{P}]\!]) \uplus \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ such* *that $\Pi(\dagger)$ is a stack access pattern , and for every $\pi \in \mathcal{V}$, $\Pi(\pi) \in \mathsf{Paths}([\![\mathcal{P}]\!])$ with $\mathsf{pr}(\Pi(\pi)) = \Pi(\dagger)$. When the pushdown system is clear from the context, we will simply refer to it as a path environment over $\mathcal{V}$.*

When $\mathcal{V} = \emptyset$, we additionally require that there is a path $\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!], c_{\mathsf{in}})$ (where c_{in} is the initial configuration of $[\![\mathcal{P}]\!]$) such that $\mathsf{pr}(\sigma) = \Pi(\dagger)$.

We introduce some notation related to path environments. Let us fix a path environment Π over variables $\mathcal{V}$. Given a path $\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!])$, $\Pi[\pi \mapsto \sigma]$ denotes the path environment over $\mathcal{V} \cup \{\pi\}$ such that $\Pi[\pi \mapsto \sigma](\pi) = \sigma$, and $\Pi[\pi \mapsto \sigma](\pi') = \Pi(\pi')$, for any $\pi' \in \mathcal{V}^{\dagger}$ with $\pi' \neq \pi$. Finally, for $i \in \mathbb{N}$, $\Pi[i:\,]$ denotes the *suffix* path environment, where every variable is mapped to the suffix of the path starting at position i. More formally, for every $\pi' \in \mathcal{V}^{\dagger}$, $\Pi[i:\,](\pi') = \Pi(\pi')[i:\,]$.

We now define when a pushdown system $\mathcal{P}$ satisfies a sHCTL* sentence θ, denoted $\mathcal{P} \models \theta$. The definition of satisfaction of θ relies on a definition of satisfaction for cognate formulas. To inductively to define the semantics of cognate formulas, we will interpret free path variables using a path environment. Finally, as in HyperCTL*, it is important to track the most recently quantified path variable because that influences the semantics of $\exists\pi(\cdot)$. Thus satisfaction of cognate formulas takes the form $\mathcal{P}, \Pi, \pi' \models \psi$, where π' is the most recently quantified path variable, and Π is a path environment over the free variables of ψ. Finally, by convention, we will take $\mathsf{Paths}([\![\mathcal{P}]\!], \Pi(\dagger)(0))$ to mean $\mathsf{Paths}([\![\mathcal{P}]\!], c_{\mathsf{in}})$, where c_{in} is the initial configuration of $[\![\mathcal{P}]\!]$ [5]. Below, θ, θ_1, and θ_2 are sHCTL* sentences, while ψ, ψ_1, ψ_2 are cognate formulas.

$$
\begin{aligned}
&\mathcal{P} \models \neg\theta \text{ iff } \mathcal{P} \not\models \theta\\
&\mathcal{P} \models \theta_1 \vee \theta_2 \text{ iff } \mathcal{P} \models \theta_1 \text{ or } \mathcal{P} \models \theta_2\\
&\mathcal{P} \models E\psi \text{ iff for some path environment } \Pi \text{ over } \emptyset, \mathcal{P}, \Pi, \dagger \models \psi\\
&\mathcal{P}, \Pi, \pi' \models a_\pi \text{ iff } a \in L(\Pi(\pi)(0))\\
&\mathcal{P}, \Pi, \pi' \models \neg\psi \text{ iff } \mathcal{P}, \Pi, \pi' \not\models \psi\\
&\mathcal{P}, \Pi, \pi' \models \psi_1 \vee \psi_2 \text{ iff } \mathcal{P}, \Pi, \pi' \models \psi_1 \text{ or } \mathcal{P}, \Pi, \pi' \models \psi_2\\
&\mathcal{P}, \Pi, \pi' \models \mathsf{X}\psi \text{ iff } \mathcal{P}, \Pi[1:\,], \pi' \models \psi\\
&\mathcal{P}, \Pi, \pi' \models \psi_1 \,\mathsf{U}\, \psi_2 \text{ iff } \exists i \geq 0 : \mathcal{P}, \Pi[i:\,], \pi' \models \psi_2 \text{ and } \forall j, 0 \leq j < i,\\
&\qquad\qquad\qquad\qquad \mathcal{P}, \Pi[j:\,], \pi' \models \psi_1\\
&\mathcal{P}, \Pi, \pi' \models \exists\pi.\,\psi \text{ iff } \exists\sigma \in \mathsf{Paths}([\![\mathcal{P}]\!], \Pi(\pi')(0)) \text{ with } \mathsf{pr}(\sigma) = \Pi(\dagger),\\
&\qquad\qquad\qquad\qquad \text{such that } \mathcal{P}, \Pi[\pi \mapsto \sigma], \pi \models \psi
\end{aligned}
$$

4 A Decision Procedure for sHCTL*

Given a pushdown system $\mathcal{P}$ and a sHCTL* sentence θ, we present an algorithm that determines if $\mathcal{P} \models \theta$. Our approach is similar to the one in [10]. Given a finite state transition system $\mathcal{K}$ and a HyperCTL* formula φ, Finkbeiner et. al. [10], construct an alternating (finite state) Büchi automaton $\mathcal{A}_{\mathcal{K},\varphi}$, by induction on φ, such that an input word σ is accepted by $\mathcal{A}_{\mathcal{K},\varphi}$ if and only if σ is the encoding

[5] The convention is needed because $\Pi(\dagger)(0)$ is not a configuration but an element of the set $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$.

of a path environment Π such that $\mathcal{K}, \Pi \models \varphi$. Determining if $\mathcal{K} \models \varphi$ then reduces to checking if $\mathcal{A}_{\mathcal{K},\varphi}$ accepts any string.

Extending these ideas to sHCTL* and pushdown systems, requires one to answer two questions: (a) What is an encoding of path environments for cognate formulas where path variables are mapped to sequences of configurations (control state + stack)?; (b) Which automata models can capture the collection of path environments satisfying a cognate formula with respect to a pushdown system? We encode path environments for cognate formulas using strings over a *pushdown alphabet* — pushdown tags on symbols adds structure that helps encode sequences of configurations. And for automata, we consider automata that process such strings and accept *visibly pushdown languages*. A natural generalization of the approach outlined in [10] would suggest the use of alternating visibly pushdown automata (AVPA) on infinite strings [4]. However, using AVPAs results in an inefficient algorithm. To get a more efficient algorithm, we instead rely on a careful use of *nondeterministic visibly pushdown automata (NVPA)* [1] and *1-way alternating jump automata (1-AJA)* [4]. The advantage of using NVPA and 1-AJA can be seen in the case of existential quantification ($\exists\pi$.) which requires converting an alternating automaton to a nondeterministic one [10]: Converting from 1-AJA to NVPA leads to exponential blowup while converting AVPA to NVPA leads to a doubly exponential blowup [4].

The rest of this section is organized as follows. We begin by introducing the automata models on pushdown alphabets (Section 4.1). Next we present our encoding of path environments, and finally our automata constructions that establish the decidability result (Section 4.2).

4.1 Automata on Pushdown Alphabets

A *pushdown alphabet* is a finite set Σ that is partitioned into three sets $\Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$, where Σ_{call} is the set of *call symbols*, Σ_{int} is the set of *internal symbols*, and Σ_{ret} is the set of *return symbols*. Automata models processing strings over a pushdown alphabet are restricted to perform certain types of transitions based on whether the read symbol is a call, internal, or return symbol. We introduce, informally, two such automata models next. Precise definition and its semantics can be found in the detailed version of this paper [2].

Nondeterministic Visibly Pushdown Büchi Automata. A *nondeterministic visibly pushdown automaton (NVPA)* [1] is like a pushdown system. It has finitely many control states and uses an unbounded stack for storage. However, unlike a pushdown system, it is an automaton that processes an infinite sequence of input symbols from a pushdown alphabet $\Sigma = \Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$. Transitions are constrained to conform to pushdown alphabet — whenever a Σ_{call} symbol is read, a symbol onto the stack, whenever a Σ_{ret} symbol is read, the top stack symbol is popped, and whenever Σ_{int} symbol is read, the stack is unchanged.

1-way Alternating Jump Automata. Our second automaton model is *1-way Alternating Parity Jump Automata (1-AJA)* [4]. 1-AJA are computationally equivalent to NVPAs (i.e., accept the same class of languages) but provide greater flexibility in describing algorithms. 1-AJAs are alternating automata, which means that they can define acceptance based on multiple runs of the machine on an input word. Though they are finite state machines with no auxiliary storage, their ability to spawn a computation thread that jumps to a future portion of the input string on reading a symbol, allows them to have the same computational power as a more conventional machine with storage (like NVPAs).

We present some useful properties of NVPA and 1-AJA. The two models are equi-expressive with the size of automata constructed by the translation known.

Theorem 1 ([4]). *For any NVPA N of size n, there is a 1-AJA $\mathcal{A}_N$ of size $O(n^2)$, such that $\mathcal{L}(\mathcal{A}_N) = \mathcal{L}(N)$. Conversely, for any 1-AJA $\mathcal{A}$ of size n, there is a NVPA $N_{\mathcal{A}}$ of size $2^{O(n)}$, such that $\mathcal{L}(N_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$. Constructions can be carried out in time proportional to the size of the resulting automaton.*

Both 1-AJA and NVPAs are closed for language operations like complementation, union and prefixing. We also recall the following result.

Theorem 2 ([1]). *For NVPAs, the emptiness problem is* PTIME*-complete.*

4.2 Algorithm for sHCTL*

Let us fix a pushdown system $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ and a sHCTL* sentence θ. Our goal is to decide if $\mathcal{P} \models \theta$. We will reduce this problem to checking the emptiness of multiple NVPAs (Theorem 2). Our approach is similar to [10] — for each cognate sub-formula ψ (not necessarily sentence) of θ, we will compositionally construct an automaton that accepts the path environments satisfying ψ. Path environments will be encoded by strings over pushdown alphabets as follows.

For a path $\sigma = c_0 c_1 c_2 \cdots$ of $[\![\mathcal{P}]\!]$, the *trace* of σ, denoted $\mathsf{tr}(\sigma)$, is the (unique) sequence $(\mathsf{o}_0, q_0, a_0, q_1)(\mathsf{o}_1, q_1, a_1, q_2) \cdots$ such that for every $i \in \mathbb{N}$, $c_i \xrightarrow{(\mathsf{o}_i, q_i, a_i, q_{i+1})} c_{i+1}$ where $\mathsf{o}_i \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$, $q_i, q_{i+1} \in Q$, and $a_i \in \Gamma \cup \{\varepsilon\}$ [6].

While $\mathsf{tr}(\sigma)$ is uniquely determined by the path σ, the converse is not true — different paths may have the same trace. To see this, consider the following example. For configuration c and $\gamma \in \Gamma^*$, let $\gamma(c)$ denote the configuration $(\mathsf{state}(c), \mathsf{stack}(c)\gamma)$, i.e., the configuration with the same control state, but with stack containing the symbols in γ at the bottom. Observe that, for any $\gamma \in \Gamma^*$, if $\sigma = c_0 c_1 c_2 \cdot$ is a path then so is $\gamma(\sigma) = \gamma(c_0)\gamma(c_1)\gamma(c_2) \cdots$. Additionally, $\mathsf{tr}(\sigma) = \mathsf{tr}(\gamma(\sigma))$. Two paths σ_1 and σ_2 of $[\![\mathcal{P}]\!]$ will be said to be equivalent if $\mathsf{tr}(\sigma_1) = \mathsf{tr}(\sigma_2)$ and will be denoted as $\sigma_1 \simeq \sigma_2$. Observe that equivalent paths have the same stack access pattern , i.e. if $\sigma_1 \simeq \sigma_2$ then $\mathsf{pr}(\sigma_1) = \mathsf{pr}(\sigma_2)$. The semantics of sHCTL* doesn't distinguish between equivalent paths.

[6] Observe that even when σ is not a path in $[\![\mathcal{P}]\!]$ (i.e., corresponds to an actual sequence of transitions of $\mathcal{P}$), the trace of σ is uniquely defined as long as stacks of successive configurations of σ can be obtained by leaving the stack unchanged, or pushing/popping one symbol.

Proposition 1. *Let φ be a cognate formula with $\mathcal{V}$ as the set of free path variables. Let Π_1 and Π_2 be two path environments such that for every $\pi \in \mathcal{V}$, $\Pi_1(\pi) \simeq \Pi_2(\pi)$. Then, $\mathcal{P}, \Pi_1, \pi \models \varphi$ if and only if $\mathcal{P}, \Pi_2, \pi \models \varphi$.*

The proof of Proposition 1 follows by induction on cognate formulas. Proposition 1 establishes that the set of path environments satisfying a cognate formula is a union of equivalence classes with respect to path equivalence. Thus, instead of constructing automata that accept path environments, we will construct automata that accept mappings from path variables to traces of paths. For $m \in \mathbb{N}$, let $\Sigma[m] = \Sigma[m]_{\mathsf{call}} \uplus \Sigma[m]_{\mathsf{int}} \uplus \Sigma[m]_{\mathsf{ret}}$ be the pushdown alphabet where $\Sigma[m]_{\mathsf{call}} = \{\mathsf{call}\} \times S^m \times \Gamma^m$, $\Sigma[m]_{\mathsf{int}} = \{\mathsf{int}\} \times S^m \times \{\varepsilon\}^m$, and $\Sigma[m]_{\mathsf{ret}} = \{\mathsf{ret}\} \times S^m \times \Gamma^m$. Observe $\Sigma[0]$ is (essentially) the set $\{\mathsf{int}, \mathsf{call}, \mathsf{ret}\}$.

Definition 3 (Encoding Path Environments). *Consider a set of m path variables $\mathcal{V} = \{\pi_1, \pi_2, \ldots \pi_m\}$. A string $w \in \Sigma[m]^\omega$ where for any $j \in \mathbb{N}$, $w(j) = (\mathsf{o}_j, (s_1^j, s_2^j, \ldots s_m^j), (a_1^j, a_2^j, \ldots a_m^j))$ encodes all path environments Π such that*

$$\Pi(\dagger) = \mathsf{o}_0\mathsf{o}_1\mathsf{o}_2 \cdots \mathsf{o}_j \cdots$$
$$\mathsf{tr}(\Pi(\pi_i)) = (\mathsf{o}_0, s_i^0, a_i^0, s_i^1)(\mathsf{o}_1, s_i^1, a_i^1, s_i^2) \cdots$$

for any $i \in \{1, 2, \ldots m\}$. The string encoding a path environment Π is denoted as $\mathsf{enc}(\Pi)$ (= w, in this case).

Based on the definitions, the following observation about traces and encodings can be concluded.

Proposition 2. *For any path $\sigma \in \mathsf{Paths}(\llbracket\mathcal{P}\rrbracket)$ and $i \in \mathbb{N}$, $\mathsf{tr}(\sigma[i:]) = \mathsf{tr}(\sigma)[i:]$. For any path environment Π and $i \in \mathbb{N}$, $\mathsf{enc}(\Pi[i:]) = \mathsf{enc}(\Pi)[i:]$.*

The encoding of path environments as strings over $\Sigma[m]$ (for an appropriate value of m) is used in our decision procedure, which compositionally constructs automata that accept path environments satisfying each cognate formula. The size of our constructed automata, like in [10], will be tower of exponentials that depends on the *formula complexity* of the cognate formula φ.

Definition 4 (Formula Complexity). *The* formula complexity *of a* sHCTL* *formula φ, denoted $\mathsf{fc}(\varphi)$, is inductively defined as follows. Let $\mathsf{odd} : \mathbb{N} \to \mathbb{N}$ be the function that maps a number n to the smallest odd number $\geq n$, i.e., $\mathsf{odd}(n) = n$ if n is odd and $\mathsf{odd}(n) = n + 1$ if n is even. Similarly, $\mathsf{even} : \mathbb{N} \to \mathbb{N}$ maps n to the smallest even number $\geq n$, i.e., $\mathsf{even}(n) = \mathsf{odd}(n + 1) - 1$. Below ψ_1, ψ_2 denote cognate formulas, and θ_1, θ_2 denote* sHCTL* *sentences.*

$$\mathsf{fc}(a_\pi) = 0 \qquad \mathsf{fc}(\neg\psi_1) = \mathsf{even}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(\mathsf{X}\psi_1) = \mathsf{fc}(\psi_1)$$
$$\mathsf{fc}(\psi_1 \vee \psi_2) = \max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)) \qquad \mathsf{fc}(\psi_1 \,\mathsf{U}\, \psi_2) = \mathsf{even}(\max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)))$$
$$\mathsf{fc}(\exists\pi.\, \psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(E\psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1))$$
$$\mathsf{fc}(\neg\theta_1) = \mathsf{fc}(\theta_1) \qquad \mathsf{fc}(\theta_1 \vee \theta_2) = \max(\mathsf{fc}(\theta_1), \mathsf{fc}(\theta_2))$$

Observe the difference in the definition of $\mathsf{fc}(\neg\theta_1)$ and $\mathsf{fc}(\neg\psi_1)$; for $\neg\theta_1$ there is no change in formula complexity, while for $\neg\psi_1$ we move to the next even level.

Our main technical lemma is a compositional construction of an automaton for cognate formulas ψ. Depending on the parity of $\mathsf{fc}(\psi)$, the automaton we construct will either be a 1-AJA or a NVPA. Before presenting this lemma, we define a function that is a tower of exponentials. For $c, k, n \in \mathbb{N}$, the value $g_c(k, n)$ is defined inductively on k as follows: $g_c(0, n) = cn \log n$, and $g_c(k+1, n) = 2^{g_c(k,n)}$. We use $g_{O(1)}(k, n)$ to denote the family of functions $\{g_c(k, n) \mid c \in \mathbb{N}\}$.

Lemma 1. *Consider pushdown system $\mathcal{P} = (S, \mathit{\Gamma}, s_{\mathsf{in}}, \mathit{\Delta}, L)$ and* sHCTL* *sentence θ. Let ψ be a cognate subformula of θ with free path variables in the set $\mathcal{V} = \{\pi_1, \ldots \pi_m\}$ for $m \in \mathbb{N}$. We assume, without loss of generality, that the variables $\pi_1, \ldots \pi_m$ are in the order in which they are quantified in θ with π_m being the first free variable of ψ that will be quantified in the context θ. In addition, we assume that the size of both ψ and $\mathcal{P}$ is bounded by n. There is an automaton $\mathcal{A}_\psi$ over pushdown alphabet $\mathit{\Sigma}[m]$ such that for any path environment $\mathit{\Pi}$ over $\mathcal{V}$,*

$$\mathcal{P}, \mathit{\Pi}, \pi_m \models \psi \text{ if and only if } \mathsf{enc}(\mathit{\Pi}) \in \mathcal{L}(\mathcal{A}_\psi).$$ [7]

The automaton $\mathcal{A}_\psi$ is a NVPA if $\mathsf{fc}(\psi)$ is odd, and a 1-AJA if $\mathsf{fc}(\psi)$ is even. The size of $\mathcal{A}_\psi$ is at most $g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil, n)$[8].

Before presenting the proof of Lemma 1, we would like to highlight a subtlety about its statement. The result guarantees that for *valid* path environments $\mathit{\Pi}$, encoding $\mathsf{enc}(\mathit{\Pi})$ is accepted by $\mathcal{A}_\psi$ if and only if $\mathit{\Pi}$ satisfies ψ. It says nothing about path environments that are not valid. In particular, there may be functions that map path variables to traces that do not correspond to actual paths of $[\![\mathcal{P}]\!]$, but which are nonetheless accepted by $\mathcal{A}_\psi$. Notice, however, when $\psi = \exists\pi.\, \psi_1$ is a cognate sentence, a string over $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}$ will, by conditions guaranteed in Lemma 1, be accepted if and only if it corresponds to a stack access pattern of a path from the initial state that satisfies $\exists\pi.\, \psi_1$.

Proof (Sketch of Lemma 1). Our construction of $\mathcal{A}_\psi$ will proceed inductively. The type of automaton constructed will be consistent with the parity of $\mathsf{fc}(\psi)$, i.e., an NVPA if $\mathsf{fc}(\varphi)$ is odd and a 1-AJA if $\mathsf{fc}(\psi)$ is even. We sketch the main ideas here, with the full proof in [2].

For a_π, $\neg\psi_1$, $\psi_1 \vee \psi_2$, and $\mathsf{X}\psi_1$, the construction essentially proceeds by converting $\mathcal{A}_{\psi_i}$ ($i \in \{1, 2\}$) if needed, into the type (NVPA or 1-AJA) of the target automaton using Theorem 1, and then using standard closure properties to combine them to get the desired automaton. In case of $\psi = \psi_1 \,\mathsf{U}\, \psi_2$, we first convert (if needed) $\mathcal{A}_{\psi_i}$ ($i \in \{1, 2\}$) into a 1-AJA. At each step, the automaton for ψ will choose to either run $\mathcal{A}_{\psi_2}$, or run $\mathcal{A}_{\psi_1}$ *and* restart itself. Correctness relies on the fact that our encoding for path environments satisfies Proposition 2.

The most interesting case is that of $\psi = \exists\pi.\, \psi_1$. We will first convert (if needed) the automaton for ψ_1 into a NVPA $\mathcal{A}_1$. The automaton for ψ will essentially guess the encoding of a path that is consistent with the transitions of

[7] When $m = 0$, we take π_m to be †.

[8] When the size of the specification ψ is considered constant, the size of $\mathcal{A}_\psi$ is at most $g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil - 1, n)$

$\mathcal{P}$, and check if assigning the guessed path to variable π satisfies ψ_1 by running the automaton $\mathcal{A}_1$. The additional requirement we have is that the guessed path start at the *same configuration* as the current configuration of the path assigned to variable π_m which introduces some subtle challenges. In order to be able to guess a path, $\mathcal{A}_\psi$ will keep track of $\mathcal{P}$'s control state in its control state, and use its stack to track $\mathcal{P}$'s stack operations along the guessed path. Since the stacks of all paths are synchronized, it makes it possible for $\mathcal{A}_\psi$ to use its (single stack) to track the stack of both $\mathcal{P}$ and the stack of $\mathcal{A}_1$. □

Using Lemma 1, we can establish the main result of this section.

Theorem 3. *Given a* $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ *and a* sHCTL* *sentence* θ*, the problem of determining if* $\mathcal{P} \models \theta$ *is in* $\cup_c \mathit{DTIME}(g_c(\lceil \frac{\mathsf{fc}(\theta)}{2} \rceil, n))$*, where* n *is a bound on the size of* $\mathcal{P}$ *and* θ*.*

Proof. Recall that a sHCTL* sentence is a Boolean combination of formulas of the form $E\psi$, where ψ is a cognate sentence. Results on whether $\mathcal{P} \models E\psi$ for each such subformula can be combined to determine whether $\mathcal{P} \models \theta$. Given this, the time to determine if $\mathcal{P} \models \theta$ is at most the time to decide if $\mathcal{P}$ satisfies each subformula of the form $E\psi$ plus $O(n)$ (to compute the Boolean combination of these results). Next, recall that the construction in Lemma 1 ensures that for a cognate sentence of the form $\exists\pi.\,\psi$, $\mathcal{L}(\mathcal{A}_{\exists\pi.\,\psi})$ consists exactly of strings in $\{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ that encode a path environment over $\emptyset$ that satisfy $\exists\pi.\,\psi$.

Consider a sHCTL* sentence $E\psi$. Let π be a path variable that does not appear in the sentence ψ. Based on the semantics of sHCTL* the following observation holds: $\mathcal{P} \models E\psi$ if and only if for some path environment Π over $\emptyset$, $\mathcal{P}, \Pi, \dagger \models \exists\pi.\,\psi$. Which is equivalent to saying that $\mathcal{P} \models E\psi$ if and only if $\mathcal{L}(\mathcal{A}_{\exists\pi.\,\psi}) \neq \emptyset$. Since $\mathsf{fc}(E\psi) = \mathsf{fc}(\exists\pi.\,\psi)$, and the emptiness problem of NVPA can be decided in polynomial time (Theorem 2), our theorem follows. □

5 Lower Bound

In this section, we establish a lower bound for the problem of model checking sHCTL* sentences against pushdown systems. Our proof establishes a hardness result for the sHLTL sub-fragment of sHCTL*. Before presenting this lower bound, we introduce the function $h_c(\cdot,\cdot)$, which is another tower of exponentials, inductively defined as follows: $h_c(0, n) = n$, and $h_c(k+1, n) = h_c(k, n) \cdot c^{h_c(k,n)}$.

Theorem 4. *Let* $\mathcal{P}$ *be a pushdown system and* θ *be a* sHLTL *sentence such that the sizes of both* $\mathcal{P}$ *and* θ *is bounded by* n *and* $\mathsf{fc}(\theta) = 2k - 1$ *for some* $k \in \mathbb{N}$*. The problem of checking if* $\mathcal{P} \models \theta$ *is* $\mathit{DTIME}(h_c(k, n))$*-hard, for every* $c \in \mathbb{N}$*.*

Proof (Sketch). We sketch the main intuitions behind the proof. To highlight the novelties of this proof, it is useful to recall how $\mathsf{NSPACE}(h_c(k-1, n))$-hardness for HyperLTL model checking is proved [5]. The idea is to reduce the language of a nondeterministic $h_c(k-1, n)$ space bounded machine M to the model checking problem by constructing a finite state transition system that guesses a run of M, and a HyperLTL formula that checks if the path is a valid accepting run.

To get the stricter bound of $\mathsf{DTIME}(h_c(k, n))$, we use the fact that we are checking pushdown systems. The stack of the pushdown system can be used to guess a *tree*, as opposed to a simple trace. Therefore, we reduce a $h_c(k - 1, n)$ space bounded *alternating* Turing machine, instead of a nondeterministic machine. Since $\mathsf{ASPACE}(f(n)) = \mathsf{DTIME}(2^{O(f(n))})$ for $f(n) \geq \log n$, the theorem will follow if the reduction succeeds.

Recall that a run of an alternating Turing machine M is a rooted, labeled tree, where vertices are labeled by configurations of M in a manner that is consistent with the transition function of M. To faithfully encode a tree as a sequence of symbols, we record the DFS traversal of the tree, making explicit the stack operations performed during such a traversal. Consider a labeled, rooted tree T with root r whose label is $\ell(r)$ with T_1 as a the left sub-tree and T_2 as the right sub-tree. The DFS traversal of T will push $\ell(r)$, traverse T_1 recursively, pop $\ell(r)$, push $\ell(r)$, traverse T_2, and then pop $\ell(r)$. We will use such a DFS traversal to guess and encode runs of M. Popping and pushing $\ell(r)$ between the traversals of T_1 and T_2 may seem redundant. Why not simply do nothing between the traversals of T_1 and T_2? For T to be a valid run of M, the configuration labeling of the root of T_2 must be the result of taking one step from $\ell(r)$. Such checks will be encoded in our sHLTL sentence, and for that to be possible, we need successive configurations of M to be consecutive in the string encoding.

To highlight some additional consistency checks, let us continue with our example tree T from the previous paragraph. For a string to be a correct encoding of T, it is necessary that the string pushed before the traversal of T_i $(i \in \{1, 2\})$ be the same as the string popped after the traversal. This can be ensured by the pushdown system by actually pushing and popping those symbols. In addition, the string popped after T_1's traversal must be the same as the string pushed before T_2's traversal. Neither the stack nor the finite control of the pushdown system can be used to ensure this. Instead this must be checked by the sHLTL sentence we construct. But the symbols while popping $\ell(r)$ will be in reverse order of the symbols being pushed, and it is challenging to perform this check in the formula. To overcome this, we push/pop the label *and its reverse* at the same time. This ensures that if we want to check if a string pushed is the same as a string that was just popped, then we can check for string *equality*, and this check is easier to do using formulas in sHLTL. Additional checks to ensure that the tree encodes a valid accepting run are performed by the sHLTL sentence using ideas from [17]. Full details can be found in [2]. □

6 Conclusions

In this paper, we introduced a branching time temporal logic sHCTL* that can be used to specify synchronous hyperproperties for recursive programs modeled as pushdown systems. The primary difference from the standard branching time logic HyperCTL* for synchronous hyperproperties is that sHCTL* considers a restricted class of hyperproperties, namely, those that relate only executions that the same stack access pattern. We call such hyperproperties stack-aware hyperproperties. We showed that the problem of model checking pushdown systems sHCTL* specifications is decidable, and characterized its complexity. We also showed how this result can potentially be used to aid security verification.

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the 36th Annual ACM Symposium on Theory of Computing. pp. 202–211. ACM (2004)
2. Bajwa, A., Zhang, M., Chadha, R., Viswanathan, M.: Stack-aware hyperproperties. https://arxiv.org/abs/2301.11521 (2023)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Concurrency Theory, 8th International Conference. pp. 135–150. Springer (1997)
4. Bozzelli, L.: Alternating automata and a temporal fixpoint calculus for visibly pushdown languages. In: Concurrency Theory, 18th International Conference. pp. 476–491. Springer (2007)
5. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Principles of Security and Trust - Third International Conference. pp. 265–284. Springer (2014)
6. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium. pp. 51–65. IEEE Computer Society (2008)
7. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Computer Aided Verification - 31st International Conference. pp. 121–139. Springer (2019)
8. Finkbeiner, B., Hahn, C., Stenger, M.: EAHyper: Satisfiability, implication, and equivalence checking of hyperproperties. In: Computer Aided Verification - 29th International Conference. pp. 564–570. Springer (2017)
9. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference. pp. 194–200. Springer (2018)
10. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Computer Aided Verification - 27th International Conference. pp. 30–48. Springer (2015)
11. Goguen, J.A., Meseguer, J.: Security policies and security models. In: IEEE Symposium on Security and Privacy. pp. 11–20. IEEE Computer Society (1982)
12. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Deciding asynchronous hyperproperties for recursive programs. CoRR **abs/2201.12859** (2022)
13. McLean, J.: Proving noninterference and functional correctness using traces. Journal of Computer Security **1**(1), 37–58 (1992)
14. McLean, J.: A general theory of composition for trace sets closed under selective interleaving functions. In: IEEE Computer Society Symposium on Research in Security and Privacy. pp. 79–93. IEEE Computer Society (1994)
15. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: Proceedings of the 8th international conference on Information Security and Cryptology. p. 156–168. Springer-Verlag (2005)

16. Pommellet, A., Touili, T.: Model-checking HyperLTL for pushdown systems. In: Model Checking Software - 25th International Symposium. pp. 133–152. Springer (2018)
17. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for büchi automata with appplications to temporal logic. Theoretical Computer Science **49**, 217–237 (1987)
18. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Computer Aided Verification, 8th International Conference. pp. 62–74. Springer (1996)
19. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of the 16th IEEE Computer Security Foundations Workshop. p. 29. IEEE Computer Society (2003)