Mining Hyperproperties from Behavioral Traces

Mayank Rawat Sujit Kumar Muduli Pramod Subramanyan*
Indian Institute of Technology, Kanpur
{*mayankr, smuduli, spramod*}@cse.iitk.ac.in

This paper is dedicated to the loving memory of Pramod Subramanyan (1984 - 2020), our academic advisor who guided and motivated us for this work.*

***Abstract*—Many important specifications of hardware and software systems, such as secure information flow and determinism are expressible only as *hyperproperties*. In contrast to the well-studied class of trace properties, which specify sets of valid runs (aka traces) of a system, hyperproperties can specify relations that must hold between the traces of a system. While hyperproperties have many applications, primarily in security verification, coming up with hyperproperties for SoC validation is challenging. In this paper, we work toward addressing this challenge by introducing a framework for mining hyperproperties from execution traces of SoC designs. We introduce novel algorithms based on coverage-guided fuzzing that enable the generation of good input traces for the hyperproperty miner. We also present novel optimistic and pessimistic semantics for Hyper Linear Temporal Logic (HyperLTL) that enable principled evaluation of HyperLTL formulas over finite traces. Finally, we propose algorithms for scalably evaluating non-trivial satisfaction of candidate hyperproperties on sets of traces. Experiments on a small but realistic SoC design show the framework is effective in identifying useful hyperproperties.**

I. Introduction

An important methodology for the verification and validation of modern systems-on-chip (SoC) platforms has been assertion-based verification (ABV). ABV enables SoC designers to declaratively specify properties that the design must satisfy. These properties can be verified/validated using formal techniques such as bounded and unbounded model checking, semi-formal techniques such as concolic execution as well as simulation-based validation. Commonly used property specification languages include SystemVerilog Assertions (SVA) [4] and the Property Specification Language (PSL) [12]. These specification languages are based on the underlying formalism of linear temporal logic (LTL) [19].[1] Unfortunately, several important classes of requirements, including information flow properties like confidentiality and integrity *cannot* be specified in formalisms based on LTL/CTL [18]. This exacerbates the critical challenge of SoC security verification.

Recent efforts, both academic [6, 11, 24] and commercial [1, 23], are tackling the security verification problem by introducing novel specification languages that can express information flow assertions, based on the theory of hyperproperties [7]. However, two important bottlenecks in using these specification languages are: (i) coming up with a meaningful set of security-related assertions, and (ii) keeping this set of assertions updated as the design evolves. In the context of traditional assertions in LTL-based formalisms like SVA/PSL, analogous problems are mitigated via *assertion mining*: techniques for algorithmically identifying likely assertions satisfied by the design by the examination of behavioral simulation traces [8–10, 13–15, 17, 25, 26, 29]. However, existing methods cannot mine hyperproperties and extending them to hyperproperties poses unique challenges (discussed later in this section). In this paper, we address this gap in the literature by proposing novel techniques for mining hyperproperties from behavioral traces of SoC designs.

[1] PSL includes an optional extension based on computation tree logic [5].

A. Hyperproperties and their Applications

LTL-based formalisms can only specify *trace properties* [2]. Roughly speaking, a trace property is a set of good traces and a system satisfies this trace property if the set of traces of the system is a subset of this set of good traces.

1) Beyond trace properties: Consider the property of determinism. Determinism requires that the output of a module be a deterministic function of only its input(s). Assume we have a hardware module whose input is the 32-bit signal x and the output is a 16-bit signal y that is produced one cycle after the input is given. We would like to show that y is a function of only x. If we are given a trace where the input is $x = 1$ and the output is $y = 2$, can we determine whether the module behaved deterministically or not in this trace? We cannot! A counterexample to determinism requires two executions or two traces with the same inputs and different outputs. So, if we had two traces, both with the same input $x = 1$ but the outputs being $y = 2$ in one trace and $y = 3$ in the other, this pair of traces would be a counterexample to determinism.

The above property can be expressed in the logic HyperLTL, introduced by Clarkson et al. [6] as follows: $\forall\pi_1.\forall\pi_2.\ \mathbf{G}\left((x_{\pi_1} = x_{\pi_2}) \rightarrow \mathbf{X}\,(y_{\pi_1} = y_{\pi_2})\right)$. We will provide a more detailed introduction to HyperLTL in § III-B; we informally describe its meaning here. This property is satisfied by a system if for every pair of traces of the system, named here π_1 and π_2, if the value of x in those traces is equal at some cycle i, then the value of the variable y is also equal at cycle $i + 1$. This property is a relation over traces and not a set of traces. It is an example of a *hyperproperty* [7].

2) Information Flow Hyperproperties: A particularly important class of hyperproperties are secure information flow properties which state that information must not flow from a specified source to a specified destination [1, 11, 23, 24].

978-1-7281-5409-1/20/$31.00 © 2020 IEEE

Information flow properties can capture both confidentiality (e.g. secret key must flow to output) and integrity (e.g. firmware input must not influence register).

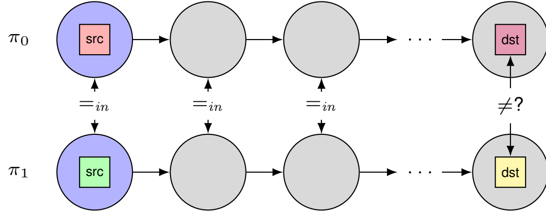


Fig. 1: Information flow property.

To check whether information can flow from a source component to a destination component, it is sufficient to check the following. Consider two almost-identical executions (i.e. traces) of system state where all registers except those of the source component have the same values in the initial state. The two source components have arbitrarily different initial values. Now suppose we ask a model checker whether these two different initial states can result in different values of the registers in the destination component when given the same input in both traces. If the answer is in the affirmative, that means information can flow from the source to the destination component. If the answer is negative, that means that values at the source are effectively indistinguishable at the destination, so no information flow exists. This is also a hyperproperty that corresponds to a symmetric binary relation over traces. It can be expressed in HyperLTL as $\forall\pi_1.\,\forall\pi_2.\,(\sigma_{\pi_1} = \sigma_{\pi_2} \wedge \mathbf{G}\,(\iota_{\pi_1} = \iota_{\pi_2})) \rightarrow \mathbf{G}\,(dst_{\pi_1} = dst_{\pi_2})$, where σ refers to all the variables in the design except for src. This is illustrated in Figure 1. Like determinism, secure information flow is not a trace property.

B. Challenges in Mining Hyperproperties

As we see above, some important specifications of SoCs are only expressible as hyperproperties. Therefore, it would be desirable to have assertion mining techniques for hyperproperties. However, in comparison to traditional trace properties, there are three challenges in mining hyperproperties.

Hyperproperties are almost always implications; i.e. formulas like $\varphi \rightarrow \psi$ where both φ and ψ are relations between states. If traces for behavioral mining are generated randomly, we may end up with no traces that satisfy φ, so formulas of the form $\varphi \rightarrow \psi$ are satisfied vacuously. Such vacuously satisfied formulas are unlikely to be valid or interesting. This is the first problem in hyperproperty mining: *generating good traces in order to drive the property miner towards likely properties.*

We collect traces of the system by simulating it and capturing how values of a set of interesting variables evolve. Traces collected in such a manner must necessarily be of finite length. However, a tuple of traces satisfying a hyperproperty is defined mathematically in terms of infinite-length traces. As we will demonstrate in § III-B1, the traditional semantics for satisfaction poses problems in determining satisfaction over finite traces. Therefore, defining finite trace satisfaction in a principled manner is another important challenge.

The final challenge is in efficient evaluation of the hyperproperties. Since hyperproperties are k-ary relations over traces, the time taken to check that n traces with maximum length L satisfy such a relation is $O(L \times n^k)$. This can quickly blow-up for long traces or for large sets of traces and efficient techniques are required to ensure that we test only promising hyperproperties, and reject invalid hyperproperties early.

C. An Overview of HyperMiner

In this paper, we address each of the above challenges and introduce an new framework called HyperMiner for mining hyperproperties from behavioral traces of SoC designs. Figure 2 shows an overview of the HyperMiner framework. The framework consists of two main parts: (a) the tracer, and (b) the miner. The tracer is responsible for generating a set of traces of SoC execution capturing how the values of the variables evolve over time. This set of traces is fed to the miner which uses a formula generator and satisfaction tester to shortlist hyperproperties satisfied by these traces.[2]

D. Contributions of This Paper

This paper makes the following novel contributions.

- We introduce a framework for mining temporal hyperproperties from behavioral traces of SoC designs. To the best of our knowledge, it is the first framework that mines temporal hyperproperties, and the first to mine any kind of hyperproperty in the context of hardware designs.
- We introduce a novel coverage-guided algorithm that generates traces for hyperproperty mining. Our algorithm modifies coverage-guided fuzzing with three novelties: (i) test sketches instead of "seed inputs", (ii) a test mutator to converts sketches into concrete tests, and (iii) novel coverage metrics to help generate meaningful traces.
- Building on the HyperLTL specification language, we introduce novel optimistic and pessimistic semantics for satisfaction of formulas over finite traces. We show the need for these semantics when finding satisfying and violating traces of a candidate hyperproperty.

We evaluate the HyperMiner framework on a small but realistic SoC consisting of two μ-controller cores, accelerators for AES, SHA, RSA along with an MMU and shared memory. The framework identifies many tens of likely hyperproperties in various modules of the SoC design.

II. Trace Generation in HyperMiner

This section describes the tracer component of HyperMiner, shown in Figure 2(a). Our tracer is based on coverage-guided fuzzing [20, 27] with the following novel modifications: (i) introducing *test sketches* instead of seed inputs, (ii) introducing *test mutators* to concretize sketches, and (iii) system-level *coverage metrics*. A test sketch is essentially a test with "holes". The "filling of holes" is done by the *test mutator*, which takes in a random bitstream from the fuzzer

[2] Currently, HyperMiner only outputs likely hyperproperties. However, it is a straightforward matter to hook-up the output of HyperMiner to hyperproperty model checking tool like MCHyper [11] to check their validity.

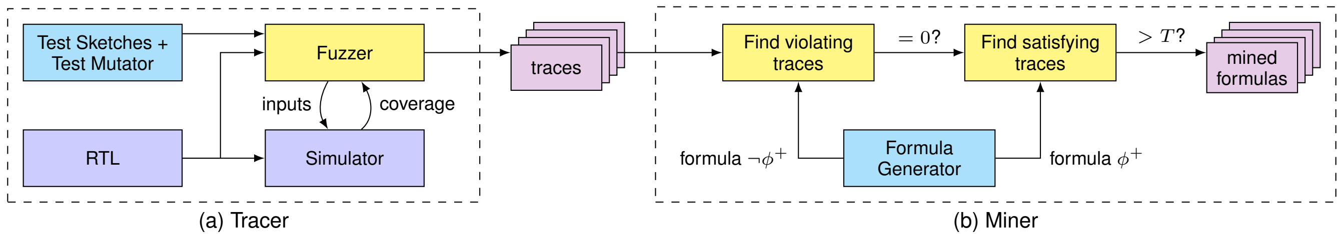


Fig. 2: Overview of the HYPERMINER framework. The color-coding is as follows. Yellow boxes show new reusable components developed as part of this paper. Blue boxes show existing components from SoC designs. Violet components are automatically generated. Cyan boxes show where design-specific inputs are required.

and produces a concrete test without holes. This concrete test is executed using the register-transfer level (RTL) simulator. During execution, the fuzzer measures one or more *coverage* metrics. The coverage metric is an estimate of which parts of the design were exercised by the test. Tests that increase coverage are prioritized for additional mutation.

A. Test Sketches and Test Mutator

We used two kinds of test sketches in our tracing.

a) NOP Sketch: This is a firmware-based test sketch where a firmware program (e.g. one that performs AES encryption) has a sequence of NOP instructions inserted into it. The test mutator concretizes this sketch by replacing the NOPs with an arbitrary sequence of instructions.

b) FSMWriter Sketch: This sketch has finite state machine connected to the SoC interconnect that repeatedly generates write-traffic to a sequence of memory locations (some of which refer to the on-chip memory-mapped I/O). The mutator concretizes this sketch by programming the sequence of address/data accesses made by the FSM.

The NOP Sketch guides the fuzzer towards executing new types of instructions in the microcontroller while the FSMWriter sketch generates new memory and memory-mapped I/O (MMIO) accesses.

B. Coverage Metrics

The coverage metric is used by the fuzzer to evaluate test quality. A good metric is extremely important for effective fuzzing. The default coverage metric used by traditional software fuzzers like `afl-fuzz` and `libFuzzer` is based on basic block coverage, and therefore inapplicable for hardware designs. The hardware fuzzer RFUZZ [16] uses mux-coverage: a measure of how many times select signals were toggled for the multiplexers (muxes) in the design. We found that this metric also did not work well; the fuzzer found very few new paths (i.e. inputs that increased coverage). This is likely because with a good test most muxes in the design are already toggled, so finding a new input that toggles some previously unexercised mux via random mutation is unlikely. Our insight is that system-level coverage metrics that correspond to software-visible events are likely to provide better feedback to the fuzzer. Therefore, we used the following novel coverage metrics in this work.

a) Instruction Bigrams: the number of unique 2-tuples of consecutive instructions executed by the primary μ-controller.

b) Memory Access Bigrams: estimates the number of unique memory access bigrams at the interface to the shared memory space in the SoC (note this includes MMIO).

These metrics were helpful in guiding the fuzzer towards tests that either execute new instructions or make new memory accesses. Results show that this helped the tracer generate traces with more interesting events for mining.

C. Putting it all together

In typical usage, the tracer is run until a certain number of traces are generated. These traces are examined by the miner (shown in Figure 2b) to determine likely hyperproperties that hold on the collected traces. Note both the sketches/mutator and the coverage metric need to work in concert in order to generate good traces. If we have the right sketch but do not have the appropriate coverage metrics, the fuzzer will be unable to distinguish good mutations from bad ones. This will cause it potentially waste a lot of time exploring redundant or meaningless paths. Similarly, a good coverage metric with a poor mutator is also useless because there will not be an easy way for the fuzzer to generate inputs that maximize the coverage metric. Finally, the use of a mutator that modifies only a part of the test, as opposed to one that randomizes the entire test is extremely important for hyperproperty mining. This ensures that most traces are related to one or more other traces – they have the same or similar values for various SoC state variables – thus preventing vacuous satisfaction of the antecedents in conjectured hyperproperties.

III. HYPERPROPERTY MINING

In this section, we first introduce the hyperproperty specification language, its syntax and semantics. We then describe the architecture of the miner component in HYPERMINER.

A. Preliminaries

A transition system M is defined as the tuple $\langle S, S_0, R, L\rangle$ where S is the set of states of the transition system, $S_0 \subseteq S$ is the set of initial states, $R \in S \times S$ is the transition relation. Our definition of the labelling function $L : S^k \rightarrow 2^{\mathsf{AP}}$ is a little non-standard. L maps k-tuples of states to a set of the atomic propositions AP. If $k = 1$, this corresponds to the

standard definition where each state has zero or more atomic propositions associated with it. However, if $k = 2$, then every *pair of states* has zero or more atomic propositions associated with it. The k-tuple labels encode relations between traces.

A trace of the system M is a (finite or infinite) sequence of states $\pi = s_0 s_1 s_2 \ldots s_i \ldots$ such that $s_0 \in S_0$ and for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. We will use the notation π^i to denote the i-th element of a trace. In the above example, $\pi^0 = s_0$, $\pi^2 = s_2$, $\pi^i = s_i$, etc. For a finite-length trace π, its length is given by $len(\pi)$; $len(\pi)$ is undefined for an infinite trace. We will write $\pi[i, \infty]$ for the suffix of the trace π starting from element i. If $len(\pi) = N$, then $len(\pi[i, \infty]) \leq N - i$. The set of all traces of the system M is denoted by Φ_M.

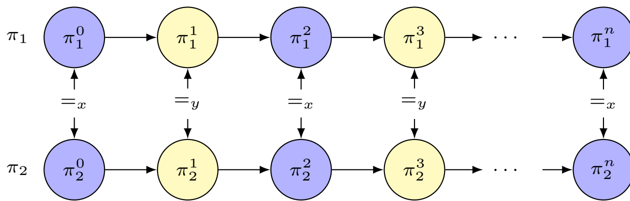
B. HyperLTL

$$
\begin{aligned}
\psi \quad & ::= \forall \pi.\ \psi \mid \exists \pi.\ \psi \mid \varphi \\
\varphi \quad & ::= \mathsf{AP}_{\pi_1,\ldots,\pi_k} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi\, \mathbf{U}^{\pm}\, \varphi \mid \mathbf{X}^{\pm}\, \varphi \\
\pm \quad & ::= + \mid -
\end{aligned}
$$

Fig. 3: HyperLTL Syntax

Figure 3 shows the grammar for the HyperLTL-variant that is considered in this paper. Formulas are required to be in *prenex form* – all quantifiers must appear in the beginning of the formula. We do not support quantifier alternation. For simplicity of presentation, we assume the specification formula is universally quantified. Extending the ideas to existentially-quantified formulas is straightforward. Atomic propositions (APs) apply to k-tuples of traces and the specific traces involved are denoted by the subscript of the AP. The other logical and temporal operators are standard except for the introduction of optimistic (superscript $^+$) and pessimistic (superscript $^-$) versions of each temporal operator.

Given the operators in Figure 3, we can define the usual abbreviations: $\varphi \vee \psi \equiv \neg(\neg\varphi \wedge \neg\psi)$, $\mathbf{F}^{\pm}\, \psi \equiv \mathsf{true}\, \mathbf{U}^{\pm}\, \psi$, etc. Note polarity (optimism/pessimism) is preserved in the above identities. However, negation reverses polarity of optimism/pessimism. For example, $\mathbf{G}^{\pm}\, \psi \equiv \neg \mathbf{F}^{\mp}\, \neg\psi$.



$\forall \pi_1 . \forall \pi_2 .\ \mathbf{G}\ ((x_{\pi_1} = x_{\pi_2}) \to \mathbf{X}\ (y_{\pi_1} = y_{\pi_2}))$

Fig. 4: Example traces for the above property.

1) Need for Optimistic and Pessimistic Variants: To see why optimistic and pessimistic variants are required, consider the formula $\forall \pi_1 . \forall \pi_2 .\ \mathbf{G}\ \big((x_{\pi_1} = x_{\pi_2}) \to \mathbf{X}\ (y_{\pi_1} = y_{\pi_2})\big)$ described in § I-A1. The notation $v_{\pi_1} = v_{\pi_2}$ is syntactic sugar for an atomic proposition that holds for all pairs of states in which the valuation of the variable v are equal. Recall the hyperproperty means that y is a deterministic function of x with a delay of one cycle. Figure 4 shows two traces of length $n + 1$ that appear to satisfy the above formula, but with a catch. The AP $x_{\pi_1} = x_{\pi_2}$ holds at steps 0, 2 and n of the two traces while the AP $y_{\pi_1} = y_{\pi_2}$ holds at steps 1 and 3 of the traces. In the final states of the traces, at step n, we have $x_{\pi_1} = x_{\pi_2}$. Since the trace has been truncated at this point, we cannot determine the value of y at step $n + 1$. Therefore, we cannot determine satisfaction with traditional (Hyper-)LTL. The optimistic operator $\mathbf{G}^+$ allows us to conclude that the formula is indeed satisfied in such scenarios if some extension to the trace exists that will satisfy the formula.

One may wonder why we need pessimistic variants in addition to the optimistic variants. Consider the negation of the above formula: $\exists \pi_1 . \exists \pi_2 .\ \mathbf{F}\ \big((x_{\pi_1} = x_{\pi_2}) \wedge \mathbf{X}\, \neg(y_{\pi_1} = y_{\pi_2})\big)$. If we use optimistic operators, then the same pair of traces in Figure 4 satisfies the negated formula as well! This contradiction must be avoided. *Therefore, we use optimistic operators when searching for satisfying traces and pessimistic variants when searching for counterexamples/violations.*

2) Formal Satisfaction Semantics: As in standard HyperLTL [11] without the optimistic/pessimistic variants, the validity judgement of a property ψ by system $M = \langle S, S_0, R, L \rangle$ is defined with respect to a trace assignment $\Pi : \mathcal{V} \to \Phi_M$. Here, $\mathcal{V}$ is a trace variable; recall that Φ_M is the set of traces of the system M. The partial function Π is a mapping from trace variables to traces. We use the notation $\Pi[\pi \mapsto \rho]$ to refer to a trace assignment that is identical to Π except that the variable π maps to trace ρ. We write $\Pi \models_M \psi$ if the system M satisfies the property ψ under the trace assignment Π. We use the notation $\Pi[i, \infty]$ as an abbreviation for the new trace assignment obtained by taking the suffix starting from index i of every trace in Π: $\Pi'(\pi) = \Pi(\pi)[i, \infty]$ for every trace $\pi \in dom(\Pi)$, where $dom(\Pi)$ is the domain of Π. We write $\Pi \not\models_M \psi$ when $\Pi \models_M \psi$ is not satisfied. We write $len(\Pi)$ to refer to the length of the minimum length trace in Π: $len(\Pi) = \min\{len(\pi) \mid \pi \in dom(\Pi)\}$. Given the above definitions, satisfaction semantics are shown in Figure 5. We say that system M satisfies the property ψ, denoted by $M \models \psi$ if $\Pi_\emptyset \models_M \psi$ for the empty trace assignment $\Pi_\emptyset$.

Lemma 1. *Equivalence of optimistic/pessimistic variants: The optimistic and pessimistic variants of the operators coincide in satisfaction for infinite-length traces.*

The proof is by induction on the structure of the formula and relies on the fact that the additional satisfaction introduced by the optimistic variants only applies to finite-length traces. (Recall $len(\pi)$ is undefined for infinite-length traces.)

C. Overview of the Miner in HYPERMINER

We now describe the miner component in Figure 2. It consists of a formula generator which outputs candidate HyperLTL formulas. These are checked on the set of collected traces to: (i) ensure that no violations exist, and (ii) the number of satisfying traces is more than the threshold T.

$$\begin{array}{ll}
\Pi \models_M \forall\pi.\ \psi & \text{iff for all } \rho \in \Phi_M : \Pi[\pi \mapsto \rho] \models_M \psi \\
\Pi \models_M \exists\pi.\ \psi & \text{iff exists } \rho \in \Phi_M : \Pi[\pi \mapsto \rho] \models_M \psi \\
\Pi \models_M \psi\, \mathbf{U}^-\, \varphi & \text{iff there exists } i \geq 0 : \Pi[i,\infty] \models_M \varphi \\
 & \quad \text{and for all } 0 \leq j < i : \Pi[j,\infty] \models_M \psi \\
\Pi \models_M \psi\, \mathbf{U}^+\, \varphi & \text{iff } \Pi \models_M \psi\, \mathbf{U}^-\, \varphi \text{, or} \\
 & \quad \text{for all } 0 \leq j < len(\Pi) : \Pi[j,\infty] \models_M \psi \\
\Pi \models_M \mathbf{X}^-\, \psi & \text{iff } \Pi[1,\infty] \models \psi \\
\Pi \models_M \mathbf{X}^+\, \psi & \text{iff } \Pi \models \mathbf{X}^-\, \psi \text{ or } len(\Pi[1,\infty]) = 0 \\
\Pi \models_M a_{\pi_1,\ldots\pi_k} & \text{iff } a \in L(\Pi(\pi_1)^0, \ldots, \Pi(\pi_k)^0) \\
\Pi \models_M \neg\psi & \text{iff } \Pi \not\models_M \psi \\
\Pi \models_M \psi \wedge \varphi & \text{iff } \Pi \models_M \psi \text{ and } \Pi \models_M \varphi
\end{array}$$

Fig. 5: Satisfaction semantics for our HyperLTL-variant.

1) Formula Generator: The formula generator produces candidate formulas for further examination. Our current implementation uses template-based enumeration for generating these candidate formulas. We use two templates. The first is of the form $\forall\pi_1.\forall\pi_2.\ \mathbf{G}\,(u_{\pi_1} = u_{\pi_2}) \rightarrow \mathbf{G}\,(v_{\pi_1} = v_{\pi_2})$ where u and v are variables in the design. This is a form of observational determinism, a secure information flow property [28]. The second is a template of the form $\forall\pi_1.\forall\pi_2.\ \mathbf{G}\,(u_{\pi_1} = u_{\pi_2} \rightarrow v_{\pi_1} = v_{\pi_2})$. As before, u and v are variables in the design. This captures a deterministic dependency between two variables in the design. We also enumerated random formulas including the temporal operators $\mathbf{G}$ and $\mathbf{X}$ and variable equalities. We note that extension of this component to support additional templates and/or static/dynamic analyses as proposed in past work (e.g. [9, 17, 25]) is straightforward.

2) Finding Violating and Satisfying Traces: The formula generator outputs a formula in plain HyperLTL without optimistic or pessimistic variants of the operators. The negation of the optimistic variant is checked against the trace set to determine if there are any counterexamples for the formula. If there are none, then the optimistic variant of the original formula is checked to determine if there are more than T traces that non-trivially satisfy it. If so, the formula is output.

Example: Suppose the formula generator outputs $\mathbf{G}\,(x_{\pi_1} = x_{\pi_2}) \rightarrow \mathbf{G}\,(y_{\pi_1} = y_{\pi_2})$. The optimistic version of this formula is: $\mathbf{G}^+\,(x_{\pi_1} = x_{\pi_2}) \rightarrow \mathbf{G}^+\,(y_{\pi_1} = y_{\pi_2})$. This is equivalent to $\mathbf{F}^-\,\neg(x_{\pi_1} = x_{\pi_2}) \vee \mathbf{G}^+\,(y_{\pi_1} = y_{\pi_2})$, and its negation is $\mathbf{G}^+\,(x_{\pi_1} = x_{\pi_2}) \wedge \mathbf{F}^-\,\neg(y_{\pi_1} = y_{\pi_2})$. This negation is checked on the mined traces to determine whether a counterexample exists. If none is found, we count satisfying traces exist for the optimistic variant. If the count is more than the threshold, the formula is shortlisted for output.

IV. Experimental Evaluation

This section presents our experimental evaluation of HyperMiner. We present details of the methodology, an overview of the test SoC design and the experimental results.

A. Implementation and Methodology

We implemented the tracer using `afl-fuzz` [27] and Verilator [22] for RTL simulation. Formula generation was implemented in Python while property satisfaction testing was implemented in C++. In keeping with the focus of VLSI-SoC 2020, the HyperMiner and SoC platform have been open-sourced at https://github.com/c0demag/HyperMiner_SourceCode.

1) SoC Overview: We evaluated HyperMiner by examining various modules in a small SoC design consisting of two microcontroller cores and accelerators for AES, SHA, modular exponentiation, a memory management unit (MMU), shared memory and I/O devices. The accelerators are controlled using memory-mapped I/O. The SoC RTL description consists of about 16000 lines of Verilog. The firmware used for the test sketches is about 1000 lines of C code.

2) Evaluation Methodology: We collected system-level traces for this SoC and then tried to mine hyperproperties within individual modules. We also provide a comparison with Bach [21], a tool for inferring (non-temporal) hyperproperties in a functional programming language. We report the number of hyperproperties found, the time taken to check satisfaction/violation and briefly discuss some mined properties.

B. Results

Table I summarizes the result of our evaluation. We show a comparison between random mutations and the coverage-guided mutation strategy for generating traces described in § II. We show results for the two hyperproperty templates used in the formula generator as well as random formula enumeration (§ III-C1). For each experiment, we report the number of mined likely hyperproperties (#hp) and the time taken for mining these hyperproperties in seconds.

TABLE I: Summary of Assertion Mining Results.

Module	Random Mutations				Coverage-guided Mutations			
	Template		Random		Template		Random	
	#hp	time	#hp	time	#hp	time	#hp	time
μC	239	561	33	318	**310**	558	**41**	96
AES	39	43	12	52	**42**	115	**33**	91
SHA	108	83	37	74	**152**	281	**60**	102
MMU	**178**	59	**57**	59	174	196	25	26

In three out of four cases, the coverage-guided tracing strategy works better than random mutations. In the fourth case, the difference for template-based formula enumeration is minimal (174 vs 178) properties. This supports our claim that coverage-guided tracing is effective. The specific templates chosen by us produce many more likely hyperproperties than random formula enumeration – this is an expected result.

The time taken to evaluate the traces is only a few minutes. While we do not report detailed results due to a lack of space, we compared runtime with Bach [21] and found that our custom-built hyperproperty evaluator was about 200× faster than the Datalog-based approach proposed in that paper.

Interesting Assertions Mined: : We provide a few examples of assertions mined by the tool. In the μcontroller, the tool identified the assertion $\mathbf{G}\,(decoderop_{\pi_1} = decoderop_{\pi_2} \rightarrow decoderrdsel_{\pi_1} = decoderrdsel_{\pi_2})$. This property states that the decoder's read-select signal is determined by the current operation in the decoder. Note the hyperproperty does not say *how* the read-select signal is determined by the decoder operation; this allows it to remain valid even if the encoding of decoder op is changed. Similar assertions capturing the relation between input data length and the byte iterator registers were identified in the AES and SHA modules.

V. Related Work

GoldMine [15, 25] mines assertions automatically using static analysis and decision trees and uses a formal verification engine to check their validity. A-Team [8] introduces a methodology to mine temporal assertions of the form $\mathbf{G}\,(\varphi \rightarrow \psi)$ by combining coverage analysis with data mining. Danese et al. [9] is an earlier effort that proposes a similar methodology but with multiple templates and using static and dynamic techniques. Ghasempouri [14] present a way to measure the relevance or "interestingness" of a specification by the use of contingency tables and support metrics. This provides a way to order/rank mined assertions. Malburg et al. [17] propose an approach to mine temporal properties by building dynamic dependency graphs and deducing the properties via graph traversal. In comparison to our work, none of these efforts can mine hyperproperties. Further, these ideas – the use of static analysis, decision trees, dependency graphs, ranking assertions, etc. are orthogonal to our approach and can potentially be incorporated in the formula generator component of HyperMiner to further improve its performance.

A noteworthy effort that does mine hyperproperties of functions, in the context of a pure functional language as opposed to hardware or SoC designs, is Bach, by Smith et al. [21]. Unlike our work, Bach cannot mine temporal hyperproperties. Further, as shown in our evaluation, extending it support temporal properties results in extremely poor performance.

Three-valued semantics for LTL proposed by Bauer et al. [3] is related to our notion of optimistic and pessimistic operators. However, their semantics was developed for monitoring and yields indeterminate for the corner cases similar to Figure 4. An indeterminate result is not useful for mining, as it does not tell us whether to shortlist a property.

VI. Conclusion

In this paper, we introduced a framework for mining *hyperproperties* from execution traces of SoC designs. Our framework had two components: a tracer that is based on coverage-guided trace generation and a miner that evaluates the collected traces to find likely hyperproperties in those traces. An important theoretical contribution of this paper are the novel optimistic and pessimistic semantics for Hyper Linear Temporal Logic (HyperLTL) that enable principled evaluation of HyperLTL formulas over finite traces. Experiments showed that the framework was effective in finding interesting hyperproperties in an SoC design.

References

[1] JasperGold: Security Path Verification App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html?CMP=SVG_JasGApp_IntDgn, 2020.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.

[4] Doron Bustan, Dmitry Korchemny, Erik Seligman, and Jin Yang. SystemVerilog Assertions: Past, present, and future SVA standardization Experience. *IEEE Design & Test of Computers*, 29(2):23–31, 2012.

[5] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[6] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust*, pages 265–284. Springer, 2014.

[7] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[8] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. A-team: Automatic template-based assertion miner. In *Design Automation Conference*, pages 1–6. IEEE, 2017.

[9] Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. Automatic extraction of assertions from execution traces of behavioural models. In *Design, Automation & Test in Europe*, pages 67–72. IEEE, 2015.

[10] Calvin Deutschbein and Cynthia Sturton. Mining security critical linear temporal logic specifications for processors. In *Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 18–23. IEEE, 2018.

[11] Bernd Finkbeiner, Markus N Rabe, and César Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *Computer Aided Verification*, pages 30–48. Springer, 2015.

[12] Harry Foster, Erisch Marschner, and Yaron Wolfsthal. IEEE 1850 PSL: The Next Generation. In *Design and Verification Conference and Exhibition*, 2005.

[13] Tara Ghasempouri, Jan Malburg, Alessandro Danese, Graziano Pravadelli, Goerschwin Fey, and Jaan Raik. Engineering of an effective automatic dynamic assertion mining platform. In *Very Large Scale Integration (VLSI-SoC)*, pages 111–116. IEEE, 2019.

[14] Tara Ghasempouri and Graziano Pravadelli. On the estimation of assertion interestingness. In *Very Large Scale Integration (VLSI-SoC)*, pages 325–330. IEEE, 2015.

[15] Samuel Hertz, David Sheridan, and Shobha Vasudevan. Mining Hardware Assertions with Guidance from Static Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):952–965, 2013.

[16] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *International Conference on Computer-Aided Design*, pages 1–8. IEEE, 2018.

[17] Jan Malburg, Tino Flenker, and Görschwin Fey. Property mining using dynamic dependency graphs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 244–250. IEEE, 2017.

[18] John Mclean. Proving Noninterference and Functional Correctness Using Traces. *Journal of Computer Security*, 1:37–58, 1992.

[19] A. Pnueli. The Temporal Logic of Programs. In *Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[20] Kostya Serebryany. libFuzzer – a library for coverage-guided fuzz testing. 2015.

[21] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering Relational Specifications. In *Foundations of Software Engineering*, pages 616–626, 2017.

[22] Wilson Snyder. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*, 2004.

[23] P. Subramanyan and D. Arora. Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design. In *Design, Automation & Test in Europe*, 2014.

[24] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung. Verifying Information Flow Properties of Firmware using Symbolic Execution. In *Design Automation & Test in Europe*, 2016.

[25] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. Goldmine: Automatic assertion generation using data mining and static analysis. In *Design, Automation & Test in Europe*, pages 626–629. IEEE, 2010.

[26] Chenguang Wang, Yici Cai, Qiang Zhou, and Haoyi Wang. ASAX: Automatic security assertion extraction for detecting Hardware Trojans. In *Asia and South Pacific Design Automation Conference*, pages 84–89. IEEE, 2018.

[27] Michal Zalewski. Technical whitepaper for afl-fuzz, 2014.

[28] Steve Zdancewic and Andrew C Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43. IEEE, 2003.

[29] Tong Zhang, Daniel Saab, and Jacob A Abraham. Automatic assertion generation for simulation, formal verification and emulation. In *IEEE Computer Society Annual Symposium on VLSI*, pages 471–476. IEEE, 2017.