HyperFuzzing for SoC Security Validation

Sujit Kumar Muduli
IIT Kanpur
smuduli@cse.iitk.ac.in

Gourav Takhar
IIT Kanpur
tgourav@cse.iitk.ac.in

Pramod Subramanyan*
IIT Kanpur
spramod@cse.iitk.ac.in

ABSTRACT

Automated validation of security properties in modern systems-on-chip (SoC) designs is challenging due to three reasons: (i) specification of security in the presence of adversarial behavior, (ii) co-validation of hardware (HW) and firmware (FW) as security bugs may span the HW/FW interface, and (iii) scaling verification to the analysis of large systems-on-chip designs.

In this paper, we address the above concerns via the development of a unified co-validation framework for SoC security property specification. On the specification side, we introduce a new logic, HyperPLTL (Hyper Past-time Linear Temporal Logic), that enables intuitive specification of SoC security concerns. On the validation side, we introduce a framework for mutational coverage-guided fuzzing of hyperproperties. The three novel aspects of our validation framework are a novel methodology for incorporating adversarial behavior through tamper functions, novel coverage-metrics that guide the fuzzer toward generating useful stimuli, and algorithms for efficient evaluation of hyperproperty satisfaction. Experiments on a small but realistic SoC show promising results.

ACM Reference Format:
Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. 2020. HyperFuzzing for SoC Security Validation. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3415709

1 INTRODUCTION

Platform security primitives in modern systems-on-chip (SoC) designs (e.g., secure boot [23, 46], enclave platforms [3, 44], authenticated firmware update [26]) are implemented by a combination of hardware and firmware primitives. The security of system and application software running on these platforms is reliant on the correctness of these primitives. Therefore, verifying that platform security primitives satisfy their claimed security requirements is a crucially important problem. Unfortunately, current techniques for security analysis of modern SoCs, especially for end-to-end verification, are mainly limited to expert review and targeted hacking efforts (aka "hackathons") [11, 36]. These are not scalable, are automatable only to a limited extent and can miss bugs. There is a pressing need for systematic and automated validation techniques to ensure the security of systems-on-chip platforms.

Unfortunately, automated security validation is a challenging problem for three reasons. First, we have a specification problem. Existing property specification languages are often not sufficient to precisely specify the security properties of a design. This is especially true in attempting to capture end-to-end security of protocols (aka "flows"), rather than piecemeal checking of necessary but insufficient conditions that detect known attacks. Second, system specification and modeling must incorporate an adversary model and reason about the state changes introduced by adversarial components in the SoC. An important complication in SoCs is that adversarial behavior appears not just at SoC inputs, but may in fact, it may come from *within* the SoC. This is due to the incorporation of untrusted third-party intellectual property (IP) cores [5], untrusted OEM firmware [42] and untrusted system software [43]. Finally, scalability of verification remains a difficult challenge. System-level security evaluation, by definition, requires analysis at SoC-level but existing techniques for systematic testing and/or formal verification are not scalable beyond the module-level.

In this paper, we address the above problems by building on the success of mutational coverage-guided fuzzing for automated identification of vulnerabilities in the software world and aim to transplant these success to SoC security validation. The use of fuzzing is especially attractive from the perspective of scalability because fuzzing can be accelerated using FPGA/emulation platforms [27], and can be trivially parallelized to a very large number of cores. Therefore, it has the potential to scale to system-level validation of large SoCs. That said, fuzzing is not a magic bullet for SoC security validation. Important research challenges still need to be solved – end-to-end security property specification, modeling and reasoning about adversarial behavior and ensuring the fuzzer generates meaningful input stimuli.

In this paper, we introduce a framework, called HyperFuzzer, to address the above challenges. We make the following contributions.

(1) To solve the specification challenge, we introduce a new logic for the property specification called Hyper Past-time Linear Temporal Logic (HyperPLTL) that is tailored to dynamic verification of security properties in SoC designs. We provide examples of SoC security properties such as confidentiality, integrity, and noninterference expressed in HyperPLTL.
(2) HyperFuzzer introduces new techniques for modeling adversarial *tampering* as part of the fuzzing loop, enabling the incorporation of adversarial behavior in the fuzzing process.
(3) We introduce novel coverage metrics that guide the fuzzer toward inputs that exercise security-critical behaviors of the SoC. These metrics prioritize the execution of diverse instructions, and memory and I/O accesses that could trigger time-of-check to time-of-use (TOCTOU) vulnerabilities.

*In Memoriam - This paper is dedicated to the loving memory of our doctoral advisor and co-author Pramod Subramanyan (1984 - 2020).

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11…$15.00
https://doi.org/10.1145/3400302.3415709

(4) We evaluate the efficacy of HyperFuzzer on a small but realistic SoC which includes an implement of an authenticated boot protocol [26, 34]. We demonstrate how system-level security concerns of this SoC can be effectively validated using our framework. Results are promising and show that the framework is effective in identifying security flaws.

All put together, HyperFuzzer provides a systematic framework for system-level security validation of SoCs.

2 BACKGROUND AND MOTIVATION

This section provides a brief review of coverage-guided fuzzing and security property specification.

2.1 Mutational Coverage-Guided Fuzzing

The term "fuzzing" refers to randomized testing of programs and was first described by the seminal work of Miller, Frederiksen and So [32]. Their work provided randomly generated inputs to commonly used utilities in Unix and found that most utilities crashed on some of these inputs. Modern fuzzers like American Fuzzy Lop (`afl`) [47] or `libFuzzer` [38] have significantly evolved since those days and are best described as *mutational coverage-guided fuzzers*.

These fuzzers differ from Miller's initial work in three ways. First, they are *mutational* which means that they start with user-provided "seed" inputs and modify these to generate new inputs. Modifications can either be rule-based such as inserting or deleting bytes, exchanging words, replacing bytes by special values such as `0xFFFF` and `0`, or they could be randomized. Two important advantages of mutational input generation are: (a) generating inputs that look similar to the real-world inputs, and (b) providing expert-guidance during input generation by starting the fuzzer in interesting locations via carefully-crafted seed inputs. Modern fuzzers are also *coverage-guided*. This means they use a coverage metric to identify which parts of a program (i.e. basic blocks) were executed when given a particular input. Inputs that increase the coverage metric are prioritized for mutation. This feedback loop turns the fuzzer into a genetic algorithm for finding inputs that maximize execution coverage. Finally, besides finding crashes, modern fuzzers can identify other kinds of errors: e.g. memory safety errors using AddressSanitizer [39] in `libFuzzer`, performance problems as in PerfFuzz [28], etc. This is typically done by instrumenting the source program to record events of interest and providing coverage feedback based on this instrumentation.

While mutational coverage-guided fuzzing is a well-studied area that has been widely applied in the context of software security analysis, applying it to SoC designs poses the following challenges.
Adversary Modeling: In the context of SoC designs, adversarial input generation is more challenging than in software fuzzing because SoCs consist of both trusted and untrusted modules. For example, a common scenario is reasoning about the security of hardware primitives in the context of adversarial firmware/software [36]. Another is ensuring that untrusted third-party IPs are unable to exfiltrate secrets or modify sensitive values in trusted components [5, 36]. In these examples, adversarial input is not at the input pins to the SoC. Instead, some (sub-)component is adversarial and reasoning about its interactions with the rest of the design is important. Therefore, enabling the fuzzer to provide meaningful adversarial behavior for SoC security components is an important challenge.
Property Specification: Software security properties are often straightforward to specify: no out-of-bounds accesses, no segmentation faults, and so on. There are no analogous equivalents of such properties in the context of SoC hardware. Therefore, a second important challenge is precise specification of SoC security so that the fuzzer can determine if these specifications have been violated.
Coverage Metrics: Recall that modern fuzzers are coverage-guided, which means that each input is executed and a coverage metric is measured in order to evaluate the quality/fitness of the input. Software fuzzers like `afl` and `libFuzzer` use basic block coverage for this purpose – this is a metric that guides the fuzzer towards inputs that execute new (previously unexecuted) basic blocks. This metric is not applicable to register-transfer level descriptions. As we will show in our evaluation, a good coverage metric plays a very important rule in determining the efficacy of the fuzzer. This leads us to the third challenge in fuzzing for SoC security analysis: determining meaningful coverage metrics for fuzzing.

2.2 Hyperproperties for Security Specification

Traditional property specification languages such as linear time temporal logic (LTL) [35], and formalisms based on LTL such as SystemVerilog Assertions (SVA) [7] can express only *trace properties* [2, 12]. Viewed abstractly, let Ψ be the universe of all traces, then a trace property is a set of traces $\Phi \subseteq \Psi$. A system M satisfies Φ if the set of traces corresponding to M, denoted by Φ_M, is a subset of Φ. Equivalently, M does not satisfy a trace property Φ, iff there exists at least one trace $\tau_{\text{cex}} \in \Phi_M$ such that $\tau_{\text{cex}} \notin \Phi$.

It is important to note that not all specifications can be expressed as trace properties. For a simple example, consider determinism which we will denote by *Det*: the requirement that a system's outputs be a deterministic function of its inputs. A single trace τ_1 cannot be a counterexample to determinism. We need a pair of traces (τ_1, τ_2) such that both traces have the same inputs but different outputs. In fact, determinism is a relation over traces, $Det \subseteq \Psi \times \Psi$. A system satisfies determinism if every pair of traces of the system is related according to the relation *Det*. Determinism, as well as similar notions such as observational determinism [37], which states that a system's adversary-observable outputs must be a deterministic function of its adversarial inputs, and noninterference [21], all belong to the class of hyperproperties [12].

Hyperproperties for Security Validation. In this paper, we build upon the theory of hyperproperties to express security specifications of systems-on-chip designs. However, this poses unique challenges when performing dynamic verification (falsification) because a hyperproperty that is a k-ary relation over traces will require $O(n^k)$ tuples of traces to be evaluated for satisfaction. This can turn into a performance bottleneck. A second challenge is in generating traces that do not lead to vacuous satisfaction. Notice the determinism is expressed as an implication, a violation requires that the inputs to two traces be the same at all time steps but the outputs be different at some time step. Purely random inputs will very likely lead to all traces having different inputs, so the property will be vacuously satisfied. All non-trivial hyperproperties have this implication

structure and therefore input generation becomes an even more important problem for dynamic falsification of hyperproperties.

3 OVERVIEW OF HYPERFUZZER

We now describe the SoC architecture and threat model considered in this paper and then provide an overview of HyperFuzzer.

3.1 SoC Architecture and Threat Model

Figure 2 shows the architecture of the SoC used in the evaluation of this paper. It consists of two microcontroller cores and a number of accelerators for cryptographic operations. We note that this type of architecture where a number of intellectual property cores (IPs) are interconnected via an on-chip fabric is typical of contemporary SoCs [26, 36, 42].

Threat Model. As is typical, some components in the SoC are untrusted. Even some of the trusted components may contain partially trusted sub-components. For example, μC_1 contains a partially-trusted instruction memory because some parts of the instruction memory are user-writeable. SoCs have two main classes of security requirements. Integrity requires that sensitive assets not be affected by the behavior of the untrusted components. Confidentiality requires that private data is not leaked to the untrusted components. Untrusted components can be of multiple types, as discussed below.

Untrusted firmware executes on trusted microcontrollers and may execute arbitrary instructions, and send arbitrary commands to the accelerators in an attempt to break confidentiality or integrity. Note untrusted firmware may attempt to exploit various types of bugs, ranging from weaknesses in hardware protections to misconfigurations in trusted firmware.

Untrusted hardware components will send arbitrary commands over the interconnect to other IPs in order to violate the SoCs security requirements. They may also respond incorrectly to requests for service from trusted components.

There could be **externally-triggered malicious behavior** such as fault injection attacks.

Our framework can model all three of the above threats and we will discuss them in more detail in Section 5.

3.2 HyperFuzzer Overview

Figure 1 provides a bird's-eye view of the HyperFuzzer framework. The framework needs three types of input from the verification engineer: (a) seed tests, (b) definition of adversarial tampering, and (c) the security property specification. Concretely, the fuzzer works by first simulating the seed test without adversarial tampering (steps 1 and 2). Steps 4–7 are repeated for each fuzzer-generated input. Fuzzer input is used to guide adversarial tampering, and different inputs result in different instantiations of tampering. For example, different inputs could correspond to the execution of different instructions in untrusted firmware code. Each instantiation of tampering results in a different trace. Security specification is validated against a pair of traces: one trace with no tampering and the other including tampering. The security specification (step 3) is given in the new logic HyperPLTL which is described in Section 4. Steps 4–7 are the core of the coverage-guided fuzzing process and are described in Section 5.

4 SECURITY PROPERTY SPECIFICATION

This section describes the property specification logic introduced in this paper, describes it semantics and provides several examples of SoC security properties specified in this logic.

4.1 Preliminaries

A transition system M is the tuple $M = \langle S, I, R, L\rangle$. Here S is the set of states of the transition system. $I \subseteq S$ is the set of initial states. $R \in S \times S$ is the transition relation. Our definition of the labelling function $L : S^k \to 2^{\mathsf{AP}}$ is non-standard and maps k-tuples of states to a set of atomic propositions. If $k = 1$, this devolves into the standard definition and each state has zero or more atomic propositions associated with it. However, if $k > 1$, then every k-tuple of traces has zero or more atomic propositions associated with it. These k-tuple labels encode relations over k-tuples of states. A trace of the system M is a finite sequence of states $\pi = s_0 s_1 \ldots s_i \ldots s_{N-1}$ such that $s_0 \in I$ and for every $0 \leq i < N-1$, $(s_i, s_{i+1}) \in R$. We will only consider finite traces in this paper because we are interested in simulation/emulation-based falsification. We will use the notation π^i to refer to the i-th element of the trace π. In the above example, $\pi^0 = s_0$, $\pi^1 = s_1$, and $\pi^i = s_i$. We will denote sets of traces by Φ.

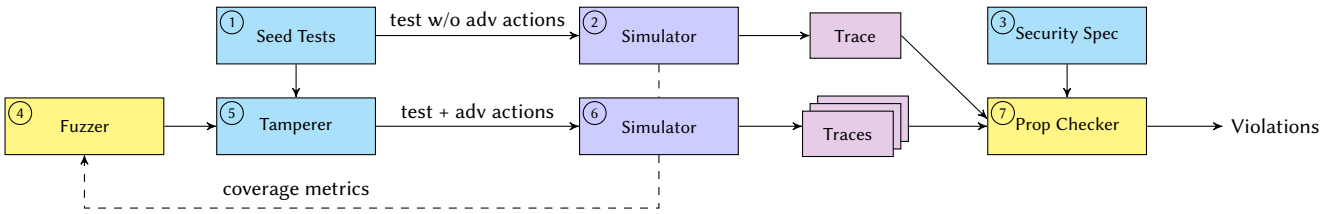
4.2 HyperPLTL

Figure 3 shows the syntax of the security property specification logic introduced in this paper. The logic is based on HyperLTL [17] but makes a couple of significant changes. The first is the exclusive use of past-time operators [6]: **Y** stands for yesterday and is the past-time dual of **X** (next), **O** representing once is the dual of **F** (future), **H** (historically) is the past-time dual of **G** (always) while **S** (since) is the dual of **U** (until). These operators are required because we are monitoring executions at simulation time and cannot and cannot access atomic propositions corresponding to future states.

Observe that HyperPLTL formulas must be in prenex normal form (i.e. all the quantifiers appear at the beginning of the formula) and universally quantified over some number of traces. A formula of the form $\forall \pi_1.\ \forall \pi_2.\ \varphi$ is satisfied if *every pair* of traces satisfies the formula φ. This ability to reason over pairs (or in general k-tuples) of traces simultaneously allows HyperPLTL to encode various hyperproperties [12] such as secure information flow [21, 48], determinism, injectivity and monotonicity [41].

Semantics of HyperPLTL. Satisfaction for HyperPLTL is defined in with respect to a tuple (Π, i) for a set of traces Φ. Π is a partial function that maps trace variables to traces and i is an integer that is less than or equal to the minimum trace length in Φ. We write $(\Pi, i) \models_\Phi \psi$ if the tuple (Π, i) satisfies a HyperPLTL formula for the set of traces Φ. Intuitively, $(\Pi, i) \models_\Phi \psi$ if all of the traces in Φ when truncated to length i satisfy the property ψ for the trace assignment Π. We use the notation $\Pi[\pi \to \tau]$ to refer to the partial function that is identical to Π but maps the variable π to the trace τ.

Satisfaction semantics are shown in Figure 4. Most of the satisfaction rules are straightforward. $\forall \pi.\ \psi$ is satisfied if every trace in Φ satisfies ψ. In HyperPLTL formulas, every atomic proposition is suffixed with the tuple of traces to which it is applied to; an atomic proposition $a_{\pi_1,\ldots,\pi_k}$ is satisfied at step i if the k-tuple of states defined by $(\pi_1^i, \ldots, \pi_k^i)$ is labelled with a. The operator $\mathbf{Y}\,\varphi$,



Note on color coding: 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.

Figure 1: Overview of the HyperFuzzer framework.

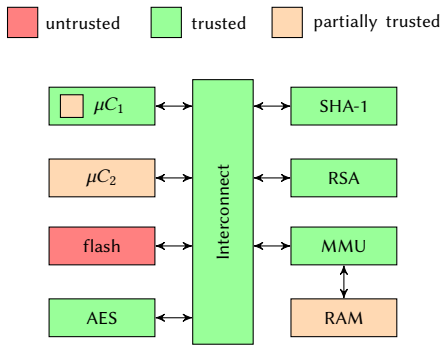


Figure 2: Architecture of the SoC used in this work.

$$
\begin{aligned}
\psi &::= \forall \pi.\ \psi \mid \varphi \\
\varphi &::= \mathrm{AP}_{\pi_1,\ldots,\pi_k} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
&\quad \mid \mathbf{Y}\,\varphi \mid \mathbf{O}\,\varphi \mid \mathbf{H}\,\varphi \mid \varphi\,\mathbf{S}\,\varphi
\end{aligned}
$$

Figure 3: HyperPLTL Syntax

is satisfied at a time step i if the previous time step $i-1$ satisfies φ, in other words if φ was satisfied yesterday. The operator $\mathbf{H}\,\varphi$ is *historically* operator that is satisfied if every state in a trace, or every k-tuple of corresponding states in a k-tuple of traces satisfies formula φ until the step i. The operator $\mathbf{O}\,\varphi$ is satisfied at time step i if φ was satisfied at least *once* sometime before time step i. The propositional operators $\neg$, $\wedge$ and $\vee$ have their usual meanings.

Given a set of traces Φ, we say that a formula ψ is satisfied by these traces, written as $\Phi \models \psi$, iff $(\Pi_\emptyset, len) \models_\Phi \psi$. Here, $\Pi_\emptyset$ is the empty mapping – a partial function whose domain is empty and $len = \min\{len(\tau) \mid \tau \in \Phi\}$ is the minimum trace length in Φ.

An important benefit of using a logic with only past-time operators is that satisfaction of a formula can be decided with only O(1) storage required for traces [4]. This makes it feasible to implement fuzzing and property evaluation on FPGAs.

4.3 Examples of SoC Security Properties

We now give a few examples of SoC security properties and explain how they can be specified in HyperPLTL.

$$
\begin{aligned}
&(\Pi, i) \models_\Phi \forall\pi.\ \psi && \text{iff for all } \tau \in \Phi_M : (\Pi[\pi \mapsto \tau], i) \models_T \psi \\
&(\Pi, i) \models_\Phi a_{\pi_1,\ldots,\pi_k} && \text{iff } a \in L(\pi_1^i, \ldots, \pi_k^i) \\
&(\Pi, i) \models_\Phi \mathbf{Y}\,\psi && \text{iff } (\Pi, i-1) \models \psi \\
&(\Pi, i) \models_\Phi \mathbf{H}\,\psi && \text{iff for all } j : 0 \le j \le i \implies (\Pi, j) \models \psi \\
&(\Pi, i) \models_\Phi \mathbf{O}\,\psi && \text{iff there exists } j \le i : (\Pi, j) \models \psi \\
&(\Pi, i) \models_\Phi \psi\,\mathbf{S}\,\varphi && \text{iff there exists } j \le i : (\Pi, j) \models_\Phi \varphi \text{ and} \\
& && \quad \text{for all } k : j < k \le i \implies (\Pi, k) \models_\Phi \psi \\
&(\Pi, i) \models_\Phi \neg\psi && \text{iff } (\Pi, i) \not\models_\Phi \psi \\
&(\Pi, i) \models_\Phi \psi \wedge \varphi && \text{iff } (\Pi, i) \models_\Phi \psi \text{ and } (\Pi, i) \models_\Phi \varphi \\
&(\Pi, i) \models_M \psi \vee \varphi && \text{iff } (\Pi, i) \models_M \psi \text{ or } (\Pi, i) \models_M \varphi
\end{aligned}
$$

Figure 4: Satisfaction Semantics for HyperPLTL.

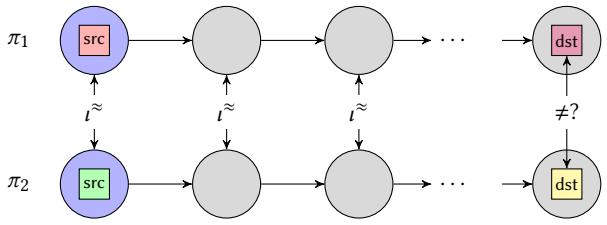


Figure 5: Secure information flow. The system starts in a pair of initial states such that all variables except the source are identical. The same inputs are given to both traces and property is violated if there exists a pair of reachable states in which the destination takes different values.

4.3.1 Secure Information Flow Properties. Secure information flow properties express the requirement information must not "flow" from certain sources in the design to certain sinks. When considering confidentiality, a secure information flow property states that a secret register value must not flow to an untrusted component. In the context of integrity, secure information flow requires that untrusted inputs must not flow to a sensitive register. Both integrity and confidentiality are common security requirements in modern SoCs [42, 43]. Confidentiality is important for ensuring the secret data, such as cryptographic keys, are not output to untrusted components (e.g. firmware-readable registers). Integrity ensures that

untrusted inputs (e.g. from BIOS/system software) cannot overwrite/modify sensitive SoC state registers.

One formulation of secure information flow, stating that information cannot from a *src* to a *dst*, can be expressed in HyperPLTL as $\forall\pi_1.\,\forall\pi_2.\,(notsrc^{\approx}_{\pi_1,\pi_2} \wedge \mathbf{H}\,(\iota^{\approx}_{\pi_1,\pi_2})) \Rightarrow \mathbf{H}\,(dst^{\approx}_{\pi_1,\pi_2})$. In the above, $notsrc^{\approx}$ is an atomic proposition that holds for a pair of states (s_1, s_2) if the valuation of all state variables except *src* is equal in these states. The atomic proposition $\iota^{\approx}$ holds if the attacker-controlled input is identical in a pair of states while $dst^{\approx}$ holds if the value of destination variable is identical in a pair of states. This property is illustrated in Figure 5. Note the property is violated if there exist a pair of initial states where all variables except the source *src* are equal, and there is some sequence of attacker inputs ι that can take us to a pair of states with differing destinations (*dst*). If no such traces exist, then the destination is "independent" of the source, so there is no information flow from source to destination.

4.3.2 Noninterference. Noninterference captures the notion that certain adversarial actions must not interfere with the computation in a system. For example, let us consider fault injection attacks (e.g. RowHammer [25]) where DRAM bits are flipped by the adversary). Suppose the SoC implements protection against fault injection attacks (for instance via error correcting codes), how does one verify that the implemented protection is sufficient? Noninterference can express such requirements. It is satisfied if the behavior of the SoC in presence of adversarial input is identical to the behavior of the SoC in the absence of adversarial input. This is stated as $\forall\pi_1.\,\forall\pi_2.\,(\mathbf{H}\,(\iota^{\approx}_{\pi_1,\pi_2}) \wedge \mathbf{H}\,(\neg fault_{\pi_1})) \Rightarrow \mathbf{H}\,(o^{\approx}_{\pi_1,\pi_2})$. $\iota^{\approx}$ and $o^{\approx}$ are atomic propositions which are satisfied when a pair of states have the same inputs and outputs respectively. $\neg fault$ is satisfied if there is no fault injected in a particular state.

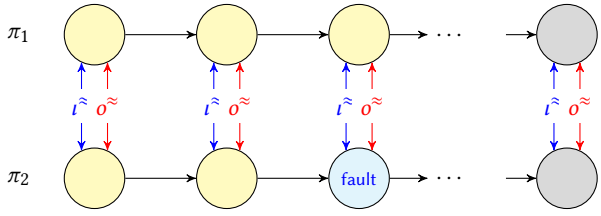


Figure 6: Noninterference. Note assumptions (that inputs are equal at every step) are shown in blue while proof obligations (that outputs must be equal at every step) are in red.

Figure 6 depicts noninterference. The two traces shown start-off identical and receive the same inputs. At some point, a fault is injected into trace π_2 while there are no faults in the trace π_1. The property is satisfied if the system's output in both traces is identical despite the injection of the fault in trace π_2.

4.3.3 Variants of Secure Information Flow and Noninterference. Many other security properties can be expressed in HyperPLTL. While we cannot describe them in detail due to limited space, a few examples are unique program execution checking [15], speculative non-interference [22], secure speculation [9], security of authenticated load [34], enclave measurement injectivity [44] etc. Note that most of the above properties involve only *two* traces, therefore in the rest of the paper the presentation will assume 2-trace properties. Extending the ideas in this paper to more traces is straightforward.

5 FUZZING FOR SOC SECURITY VALIDATION

HyperFuzzer contains three novel ideas in comparison to traditional mutational coverage-guided fuzzers: (a) the incorporation of *tamperers* to model and provide adversarial behavior in SoCs, (b) the use of novel *high-level* coverage metrics to guide the fuzzer towards more meaningful inputs, and (c) using HyperPLTL security specifications to find violations of security requirements. In the rest of this section, we first provide a brief overview of the fuzzing algorithm and then describe the tamperers and coverage metrics in more detail.

5.1 Algorithm Overview

Algorithm 1 shows the HyperFuzzer algorithm. It takes as input a HyperPLTL formula ψ which is the security specification to be checked, a test that exercises some portion of the SoC, an initial pool of inputs for the fuzzer seeded with a few interesting behaviors, the tamperer and coverage evaluation functions. Note that the test need not contain any adversarial behavior, so any functional test can be used here; automated (functional) test generation techniques [8, 10] can also be used.

Algorithm 1 HyperFuzzer Algorithm

```
1:  procedure HypFuzz(ψ, Test, initPool, Tamper, Coverage)
2:      τ0 ← Simulate(Test, ⊥)
3:      Pool ← initPool
4:      covered ← ∅
5:      while time budget for fuzzing has not expired do
6:          inp ← RandMutate(RandomChoice(Pool))
7:          τ1 ← Simulate(Test, Tamper(inp))
8:          cov ← Coverage(τ1)
9:          if cov ∩ covered ≠ ∅ then
10:             Pool ← Pool ∪ {inp}
11:             covered ← covered ∪ cov
12:         end if
13:         if {τ0, τ1} ⊨ ¬ψ then
14:             report violation
15:         end if
16:     end while
17: end procedure
```

First, the SoC is simulated with the input test and a trace of execution is obtained (τ_0). Then the fuzzing loop starts on line 5. Within the loop, the fuzzer repeatedly generates a newly mutated input based on the inputs in the pool. This input is fed to the tamperer, which is used during simulate the test in the presence of adversarial tampering (line 7). This produces the trace τ_1 and the coverage metric is evaluated on this trace (line 8). Viewed abstractly, the coverage metric is a multiset of events that occurred during simulation and increased coverage corresponds to either new events occurring or an existing event occurring more often than previously. If the test results in "new" coverage (line 9), this input is added to the pool of interesting inputs. Finally, we check whether the property is falsified by this pair of traces (line 13), and if so, report violations.

5.2 Adversarial State Tamperers

We model adversarial behavior in the SoC by the incorporation of user-provided state *tamperers* that modify SoC state using fuzzer input. While the tamperer can modify state in arbitrary ways, we implemented three kinds of tamperers that model specific adversaries considered in our threat model. These tamperers are described below. We emphasize that this is not an exhaustive list of possible tamperers, and our framework enables the definition of other arbitrary state tamperers.

Firmware Tamperer: The firmware tamperer models untrusted firmware components. Fuzzer input is used to generate a randomized sequence of instructions that are executed on a specified microcontroller core. These instructions attempt to interfere with operation of the SoC, either by sending arbitrary commands to accelerators, modifying shared memory to conduct time-of-check to time-of-use (TOCTOU) attacks, or modify microcontroller state that might be used later by trusted code.

NoC Txn Tamperer: This models an untrusted hardware component that is connected to the on-chip network. This tamperer uses fuzzer input to send arbitrary (randomized) write transactions to the network. These requests can either modify shared memory (this models TOCTOU attacks) or send commands to other accelerators (this models confused-deputy attacks).

Bitflip Tamperer: This models fault-injection attacks on the on-chip memory components. It periodically selects a random memory address that has previously not been tampered with and flips a single-bit in this location. It is used to evaluate the SoC's resilience to fault-injection attacks.

5.3 High-Level Coverage Metrics

As discussed in sections 2.1 and 5.1, the fuzzer uses the coverage measurement as feedback to determine which inputs are interesting, and therefore should be prioritized for further mutation and execution. The intuition is that inputs for which coverage increases are those which are likely to exercise previously unexplored parts of the design, and therefore are more likely to find violations. Recall that the coverage metric is a multiset of events that occurred during simulation and the coverage metric defines what an event is in this definition.

Past work in SoC fuzzing [27] for functional verification has suggested a MUX coverage metric. In this metric, each event is the toggling of the select signal for particular MUX in the design. We experimented with this and similar metrics and found that it was not helpful in providing useful feedback to the fuzzer as many MUXes are toggled for most inputs anyway, and MUXes can be toggled even if there isn't necessarily any activity in that part of the design.

Our key insight is that the coverage metric must attempt to directly measure the impact of adversarial behavior in security fuzzing. Based on this insight, we propose the following coverage metrics.

(1) **Instruction Bigram Events**: In this metric, each event is 2-tuple of consecutive executed instruction opcode and program counter values. This metric helps the firmware tamperer generate new instruction sequences.
(2) **NoC Access Bigrams**: Each event is a bigram of transactions sent on the on-chip interconnect. This metric helps the NoC Txn Tamperer generate more interesting tampering transactions.
(3) **Bitflip Read Events**: Each event corresponds to when an SoC component reads a memory address that has been tampered by the Bitflip Tamperer.

Our evaluations shows that these coverage metrics perform much better than low-level metrics such as MUX/toggle coverage.

6 EVALUATION

This section contains our evaluation of HyperFuzzer.

6.1 Methodology

HyperFuzzer[1] builds on the `afl` [47] fuzzer. We modify `afl` for SoC security validation in three ways. First, the execution of the SoC RTL is done using `Verilator` [40], an open source tool for Verilog HDL simulation. Verilator output is instrumented with C++ code to implement the tamperers and collect coverage metrics. We implemented a C++ library called `libprop` that evaluates satisfaction of HyperPLTL formulas. `afl` interfaces with the instrumented output of Verilator, and the tampers and `libprop` when fuzzing with the design. This means we use `afl` for its mutation engine and its fork-server while replacing all of its other components.

6.2 SoC Description

The SoC we evaluate is shown in Figure 2. Its architecture and threat model are described in Section 3.1.

Firmware Tests and Verification Objectives. The SoC had a number of firmware tests that we reused as seed tests during the fuzzing. These tests are described below.

`aes_test`. loads input data into RAM, and then encrypts and decrypts it using the AES accelerator. The test passes if the decrypted data is the same as the original input data. The test control flow goes through an untrusted firmware routine during the encryption and decryption operations. We check to ensure a weak form of noninterference between this untrusted firmware routine and the rest of the test using the firmware tamperer. The SoC does not satisfy the noninterference property so expect to (and do) find a violation.

`aes_test_ecc`. This is the same as `aes_test` in terms of functionality. However, we consider a different adversary model of fault injection attacks into the RAM using the bitflip tamperer. We check the same property as in (1) with two versions of this test. In the safe version, the RAM region being used is protected using error correcting codes (ECC). We ensure that fault injection in the RAM does not result in the property being violated. In the unsafe version, RAM is not protected using ECC, so the property is violated.

`sha_test`. This test computes the SHA-1 hash of an input block and checks that the result matches an expected value. We use this test to evaluate a No TOCTOU property [34] – stating that if the hash matches a particular value then the input data must have a particular value. The adversarial tampering in this case comes from the NoC Txn Tamperer. Here too, the test does not actually satisfy the No TOCTOU property so we find a violation.

[1] https://github.com/skmuduli92/HyperFuzzer

Table 1: Experimental Results.

Test	Property	Coverage Metric		master	slave #1	slave #2	slave #3
aes_test	Noninterference	toggle coverage	#execs	748	715	595	609
			#paths	13	18	13	18
			#crash	1	1	1	1
		high-level coverage	#execs	2257	1629	1664	1614
			#paths	231	163	167	164
			#crash	1	1	1	1
aes_test_ecc_unsafe	Noninterference	toggle coverage	#execs	1373	1371	1373	1374
			#paths	1	1	1	1
			#crash	1	1	1	1
		high-level coverage	#execs	18771	18856	18818	18833
			#paths	19	22	22	22
			#crash	1	1	1	1
aes_test_ecc_safe	Noninterference	toggle coverage	#execs	1388	1381	1384	1366
			#paths	1	1	1	1
			#crash	0	0	0	0
		high-level coverage	#execs	18578	18529	18591	18598
			#paths	33	33	33	33
			#crash	0	0	0	0
sha_test	Noninterference + No TOCTOU	toggle coverage	#execs	3900	3956	3919	3913
			#paths	17	23	24	23
			#crash	1	1	1	1
		high-level coverage	#execs	3938	4013	3901	3954
			#paths	80	137	102	73
			#crash	1	1	1	1
secureboot	Header checks	toggle coverage	#execs	544	337	297	344
			#paths	7	7	4	6
			#crash	1	1	1	1
		high-level coverage	#execs	606	424	385	391
			#paths	9	15	16	12
			#crash	1	1	1	1
	Data Block 1 checks	toggle coverage	#execs	442	341	309	309
			#paths	6	3	5	5
			#crash	1	1	1	1
		high-level coverage	#execs	457	345	327	367
			#paths	11	15	12	14
			#crash	1	1	1	1
	Data Block 2 checks	toggle coverage	#execs	378	242	240	265
			#paths	5	5	3	6
			#crash	1	1	1	1
		high-level coverage	#execs	494	341	330	340
			#paths	11	10	10	11
			#crash	1	1	1	1

secureboot. This is an implementation of an authenticated boot protocol [26, 34]. At boot time, the implementation loads a firmware image from an input device into the shared RAM, checks the authenticity of the image by using public key cryptography and if the image is found to be valid, it starts execution of the image. The authenticity check is performed in three stages: first the header is checked using the RSA and SHA-1 accelerators, and then the two data block hashes are checked against values stored in the header using the SHA-1 accelerators. The security properties we check are taken from [34]. They check whether protocol state hijacking, TOCTOU and/or confused deputy attacks is possible. We use the NoC Txn Tamperer in this case study.

6.3 Results

Table 1 shows the results of running HyperFuzzer on the tests described in the previous section. We executed four fuzzer processes in parallel for each test. One process runs in master mode and other three in child mode. The master process performs deterministic mutations initially and then switches to random mutations, while slaves processes perform only random mutations. The processes

periodically synchronize to share interesting inputs. All processes are run with a timeout of 1000 seconds. Important observations from the results are as follows.

Verilator's toggle coverage, which is closely related to the notion of MUX coverage introduced in [27], results in both fewer executions and fewer paths than the high-level coverage metrics introduced in this paper for the same time duration of fuzzing. This is because instrumenting the design to capture MUX coverage introduces significant overhead (because non-trivial designs have a lot MUXes) slowing down the simulations.

Second, even when normalized for the same number of executions, toggle coverage leads to fewer new paths being discovered per execution. A new *path* in `afl` terminology is an execution which led to increased coverage. Many MUXes are always toggled and random mutations are unlikely to toggle a previously unexercised MUX. In contrast, our high-level coverage metrics are easier for the fuzzer to optimize for.

Thirdly, we see that fuzzing is trivially parallelizable. We can increase our chances of finding a bug by throwing more cores at the problem. Finally, the experiments show that HyperFuzzer is effective in finding property violations in the design.

7 RELATED WORK

SoC validation is a richly-studied problem and due to a lack of space we cannot survey all of it. We provide a brief overview of some of the most closely-related work to ours in this section.

RTL Test Generation: Coverage-guided testing of hardware designs is a well-studied area with a rich body of literature. An early effort is that of Tesiran et al. [45] which used a Markov chain analysis to guide input generation towards increasing coverage. Many subsequent efforts have studied various forms of coverage-guided test generation. The line of work in [19] and [49] uses automated test pattern generation (ATPG) to generate RTL inputs that are also used for functional validation. The use of ATPG ensures that structural coverage metrics can be maximized. HYBRO [30] combines dynamic and static analysis to generate inputs that increase coverage. HYBRO itself draws upon the idea of whitebox fuzzing as introduced by SAGE [20] in the context of software security analysis. Similar ideas have been explored by Ahmed et al. [1] and Farahmandi et al. [16]. Li et al. [29], and Gent et al. [18] used ant colony optimization (ACO) to generate inputs that increase coverage. A notable effort that is related to our own is rfuzz, which uses coverage-guided mutational fuzzing via afl to perform FPGA-based validation of RTL design designs [27]. All of the above efforts are complementary and orthogonal to our work. They focus on the problem of *how* to generate inputs given a particular coverage-metric. However, we argue the real problem in SoC security validation is in *what (hyper-)property to validate*, and *how to model adversarial behavior*? Our work introduces techniques for solving these two problems, and ideas such as the use of SAT/SMT solvers for whitebox fuzzing à la SAGE/HYBRO/QUEBS [1, 20, 30], the use of ACO as in [18, 29], or acceleration via FPGAs [27] can all be readily integrated into our framework to further improve its performance.

Blackbox and Whitebox Fuzzing: Seminal work on fuzzing was done by Miller and colleagues [32, 33]. While many subsequent efforts have proposed a number of improvements to fuzzing methodology, we cannot discuss all them due to limited space, and instead direct the reader to McNally et al. [31] who provide an excellent survey of the area. American Fuzzy Lop (afl) [47] and libFuzzer [38] are two commonly used software fuzzers. Our work builds on top of afl but replaces its coverage metrics while adding support for hyperproperty validation while fuzzing. SAGE [20] and KLEE [8] both introduced techniques based on symbolic analysis for test generations. These ideas can be integrated into our framework to further improve coverage and doing so will be the subject of future work. Ideas based on concolic execution inspired by KLEE have been used in the context of firmware security validation [10, 14]. An important difference between these latter efforts and our work is the use of hyperproperties for security specification and an adversary model that allows modeling of various kinds of attacks, ranging from malicious firmware to fault injection attacks.

Information Flow Analysis: The idea of information flow verification dates back to Goguen and Meseguer who introduced the noninterference [21]. Observational determinism, which is another secure information flow property was introduced by Roscoe [37]. Clarkson and Schneider [12] introduced the class of specifications called hyperproperties and observed that noninterference and observational determinism, as well as other variants of secure information flow were all hyperproperties. Clarkson et al. [13] introduced temporal logics for hyperproperty specification, of which HyperLTL is one example. Finkbeiner et al. [17] introduced model checking algorithms for HyperLTL. As discussed earlier, HyperLTL is not suitable for dynamic verification because of its future-time operators. Subramanyan et al. [42, 43] described techniques for information flow property checking using symbolic execution and model checking. In contrast to fuzzing, these techniques are much less scalable and therefore less applicable to analysis at the SoC level. We note that commercial offerings exist that can perform model checking of information flow on hardware designs [24]. We note that the logic introduced in this paper HyperPLTL can express more than just information flow and further our framework is able to perform co-validation of hardware and firmware.

8 CONCLUSION

This paper introduced the HyperFuzzer framework for SoC security validation based on mutational coverage-guided fuzzing. HyperFuzzer includes a novel property specification logic HyperPLTL that enables the succinct specification of system-level security properties in SoCs. We also introduced new methods for modeling adversarial state modifications in SoCs through the use of *tamperers* and demonstrated that these can capture a wide variety of threat models. We introduced new high-level coverage metrics that help the fuzzer generate better stimuli. Experiments on our test SoC showed promising results and identified several vulnerabilities.

ACKNOWLEDGMENTS

This work was supported in part by the Semiconductor Research Corporation (SRC) under task 2854.001. Views expressed in this paper are of the authors alone and do not necessarily represent the views of SRC.

REFERENCES

[1] Alif Ahmed and Prabhat Mishra. Quebs: Qualifying event based search in concolic testing for validation of rtl models. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 185–192. IEEE, 2017.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985. ISSN 0020-0190.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative Technology for CPU based Attestation and Sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[4] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Eagle does space efficient LTL monitoring. *Pre-Print CSPP-25, University of Manchester, Department of Computer Science*, October 2003.

[5] Abhishek Basak, Swarup Bhunia, Thomas Tkacik, and Sandip Ray. Security assurance for system-on-chip designs with untrusted IPs. *IEEE Transactions on Information Forensics and Security*, 12(7):1515–1528, 2017.

[6] Marco Benedetti and Alessandro Cimatti. Bounded model checking for past ltl. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 18–33. Springer, 2003.

[7] 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.

[8] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of Operating Systems Design and Implementation*, 2008.

[9] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815, June 2019. doi: 10.1109/CSF.2019.00027.

[10] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. CRETE: A Versatile Binary-Level Concolic Testing Framework. In Alessandra Russo and Andy Schürr, editors, *Fundamental Approaches to Software Engineering*, pages 281–298, Cham, 2018. Springer International Publishing.

[11] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L. Wang. Challenges and trends in modern soc design verification. *IEEE Design & Test*, 34(5):7–22, 2017.

[12] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, September 2010. ISSN 0926-227X.

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

[14] K. Cong, F. Xie, and L. Lei. Symbolic execution of virtual devices. In *Proceedings of the 13th International Conference on Quality Software*, pages 1–10. IEEE, 2013.

[15] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *2019 Design, Automation & Test in Europe Conference & Exhibition*, pages 994–999. IEEE, 2019.

[16] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. *Automated Test Generation for Detection of Malicious Functionality*, pages 153–171. Springer International Publishing, Cham, 2020. ISBN 978-3-030-30596-3. doi: 10.1007/978-3-030-30596-3_8. URL https://doi.org/10.1007/978-3-030-30596-3_8.

[17] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for Model Checking HyperLTL and HyperCTL*. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV 2015)*, pages 30–48, July 2015.

[18] Kelson Gent and Michael S Hsiao. Fast multi-level test generation at the rtl. In *IEEE Computer Society Annual Symposium on VLSI*, pages 553–558. IEEE, 2016.

[19] Indradeep Ghosh and Srivaths Ravi. On automatic generation of RTL validation test benches using circuit testing techniques. In *Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 289–294, 2003.

[20] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[21] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982. doi: 10.1109/SP.1982.10014. URL http://dx.doi.org/10.1109/SP.1982.10014.

[22] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018. URL http://arxiv.org/abs/1812.08639.

[23] James Hendricks and Leendert van Doorn. Secure Bootstrap is Not Enough: Shoring Up the Trusted Computing Base. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, EW 11, New York, NY, USA, 2004. ACM. doi: 10.1145/1133572.1133600. URL http://doi.acm.org/10.1145/1133572.1133600.

[24] Cadence Inc. 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, 2020.

[25] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, pages 361–372, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-4394-4.

[26] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor. Security of SoC firmware load protocols. In *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust*, pages 70–75, 2014.

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

[28] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[29] Min Li, Kelson Gent, and Michael S Hsiao. Design validation of RTL circuits using evolutionary swarm intelligence. In *IEEE International Test Conference*, pages 1–8. IEEE, 2012.

[30] Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation & Test in Europe*, pages 1–6. IEEE, 2011.

[31] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. Fuzzing: the state of the art. Technical report, DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA), 2012.

[32] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[33] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekandanda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1995.

[34] Sujit Kumar Muduli, Pramod Subramanyan, and Sayak Ray. Verification of authenticated firmware loaders. In *Proceedings of Formal Methods in Computer-Aided Design*. IEEE, 2019.

[35] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[36] Sandip Ray and Yier Jin. Security policy enforcement in modern soc designs. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 345–350. IEEE, 2015.

[37] A. W. Roscoe. CSP and determinism in security modelling. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 8-10, 1995*, pages 114–127, 1995. doi: 10.1109/SECPRI.1995.398927. URL http://dx.doi.org/10.1109/SECPRI.1995.398927.

[38] K Serebryany. libfuzzer a library for coverage-guided fuzz testing. *LLVM project*, 2015.

[39] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

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

[41] Marcelo Sousa and Isil Dillig. Cartesian Hoare Logic for verifying k-safety properties. In *Proc. of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 57–69, 2016. ISBN 978-1-4503-4261-2.

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

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

[44] Pramod Subramanyan, Rohit Sinha, Ilia A. Lebedev, Srinivas Devadas, and Sanjit A. Seshia. A formal foundation for secure remote execution of enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2435–2450, 2017. doi: 10.1145/3133956.3134098. URL http://doi.acm.org/10.1145/3133956.3134098.

[45] Serdar Tasiran, Farzan Fallah, David G Chinnery, Scott J Weber, and Kurt Keutzer. A functional validation technique: biased-random simulation guided by observability-based coverage. In *Proceedings 2001 IEEE International Conference on Computer Design*, pages 82–88. IEEE, 2001.

[46] Richard Wilkins and Brian Richardson. UEFI Secure Boot in Modern Computer Security Solutions. In *UEFI Forum*, 2013.

[47] Michal Zalewski. Technical whitepaper for afl-fuzz. *Accessed April*, 1, 2017.

[48] 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.

[49] Liang Zhang, Indradeep Ghosh, and Michael S Hsiao. A framework for automatic design validation of RTL circuits using ATPG and observability-enhanced tag coverage. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2526–2538, 2006.