Bomberman: Defining and Defeating Hardware Ticking Timebombs at Design-time

Timothy Trippel*, Kang G. Shin
Computer Science & Engineering
University of Michigan
Ann Arbor, MI
{trippel,kgshin}@umich.edu

Kevin B. Bush
Cyber-Physical Systems
MIT Lincoln Laboratory
Lexington, MA
kevin.bush@ll.mit.edu

Matthew Hicks*†
Computer Science
Virginia Tech
Blacksburg, VA
mdhicks2@vt.edu

***Abstract*—To cope with ever-increasing design complexities, integrated circuit designers increase both the size of their design teams and their reliance on third-party intellectual property (IP). Both come at the expense of trust: it is computationally infeasible to exhaustively verify that a design is free of *all possible* malicious modifications (i.e., hardware Trojans). Making matters worse, unlike software, hardware modifications are *permanent*: there is no "patching" mechanism for hardware; and *powerful*: they serve as a foothold for subverting software that sits above.**

To counter this threat, prior work uses both static and dynamic analysis techniques to verify hardware designs are Trojan-free. Unfortunately, researchers continue to reveal weaknesses in these "one-size-fits-all", heuristic-based approaches. Instead of attempting to detect *all* possible hardware Trojans, we take the first step in addressing the hardware Trojan threat in a divide-and-conquer fashion: defining and eliminating Ticking Timebomb Trojans (TTTs), forcing attackers to implement larger Trojan designs detectable via existing verification and side-channel defenses. Like many system-level software defenses (e.g., Address Space Layout Randomization (ASLR) and Data Execution Prevention (DEP)), our goal is to systematically constrict the hardware attacker's design space.

First, we construct a definition of TTTs derived from their functional behavior. Next, we translate this definition into fundamental components required to realize TTT behavior in hardware. Using these components, we expand the set of all known TTTs to a total of six variants—including unseen variants. Leveraging our definition, we design and implement a TTT-specific dynamic verification toolchain extension, called *Bomberman*. Using four real-world hardware designs, we demonstrate Bomberman's ability to detect *all* TTT variants, where previous defenses fail, with <1.2% false positives.

***Index Terms*—Hardware Trojans, Ticking Timebombs, 3rd Party IP, Verification**

I. Introduction

As microelectronic hardware continues to scale, so too have design complexities. To design an Integrated Circuit (IC) of modern complexity targeting a $7\,nm$ process requires 500 engineering years [1], [2]. Because it is impractical to take 500 years to create a chip, semiconductor companies reduce time-to-market by adding engineers: increasing both the size of their design teams and their reliance on 3rd-party Intellectual Property (IP). Namely, they purchase pre-designed blocks for inclusion in their designs, such as CPU cores and cryptographic accelerators (e.g., AES). This year, analysts estimate that a typical System-on-Chip (SoC) will contain over 90 IP blocks [3]. From a security perspective, this reduces trust in the final chip: with an increased number of (both in-house and external) designers molding the design, there is increased opportunity for an attacker to insert a hardware Trojan.

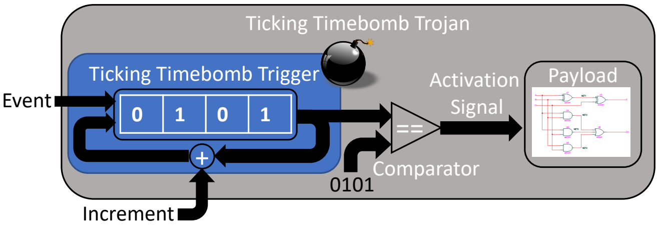

Fig. 1. **Ticking Timebomb Trojan (TTT).** A TTT is a hardware Trojan that implements a ticking timebomb *trigger*. Ticking timebomb triggers monotonically move closer to activating as the system runs longer. In hardware, ticking timebomb triggers maintain a non-repeating sequence counter that increments upon receiving an event signal.

Hardware Trojans inserted during design time are both *permanent* and *powerful*. Unlike software, hardware cannot be patched in a general-purpose manner; repercussions of hardware flaws echo throughout the chip's lifetime. As hardware vulnerabilities like Meltdown [4], Spectre [5], and Foreshadow [6] show, replacement is the only comprehensive mitigation, which is both costly and reputationally damaging. Moreover, vulnerabilities in hardware cripple otherwise secure software that runs on top [7]. Thus, it is vital that hardware designers verify their designs are Trojan-free.

Prior work attempts to detect hardware Trojans at both design and run time. At design time, researchers propose static (FANCI [8]) and dynamic (VeriTrust [9] and UCI [10]) analyses of the Register Transfer Level (RTL) design and gate-level netlists to search for rarely-used circuitry, i.e., potential Trojan circuitry. At run time, researchers: 1) employ hardware-implemented invariant monitors that dynamically verify design behavior matches specification [11], [12], and 2) scramble inputs and outputs between trusted and untrusted components [13] to make integration of a hardware Trojan into an existing design intractable. These attempts to develop general, "one-size-fits-all", approaches inevitably leave chips

* Work completed at MIT Lincoln Laboratory.

† Corresponding faculty author

vulnerable to attack [14]–[16].

Verifying a hardware design is Trojan-free poses two technical challenges. First, hardware Trojan designs use the same digital circuit building blocks as non-malicious circuitry, making it difficult to differentiate Trojan circuitry from non-malicious circuitry. Second, it is infeasible to exhaustively verify, manually or automatically, even small hardware designs [17], let alone designs of moderate complexity. These challenges are the reason why **"one-size-fits-all" approaches are incomplete and akin to proving a design is bug-free**.

Instead of verifying a design is free of *all* Trojan classes, we advocate for a divide-and-conquer approach, breaking down the RTL Trojan design space and systematically ruling out each Trojan class. We begin this journey by eliminating the most pernicious RTL hardware Trojan threat: the TTT. As Waksman *et al.* state [11], [13], when compared with other stealthy design-time Trojans (i.e., data-based Trojans), TTTs provide "the biggest bang for the buck [to the attacker] ... [because] they can be implemented with very little logic, are not dependent on software or instruction sequences, and can run to completion unnoticed by users." Moreover, TTTs are a flexible Trojan design in terms of deployment scenarios. **An attacker looking to deploy a TTT does not require any *a priori* knowledge of how the victim circuit will be deployed at the system level, nor post-deployment (physical or remote) access to the victim circuit [11], [13]**. By eliminating the threat of TTTs, we mimic the attack-specific nature of system-level software defenses like DEP and ASLR in hardware, i.e., we force RTL attackers to implement Trojan designs that require post-deployment attacker interaction. This is the hardware analog to defending against data injection attacks in software, forcing attackers to employ more complex data reuse attacks; a necessary part of a comprehensive, layered defense.

To ensure our defense is systematic and avoids implicit assumptions based on existing TTTs, we first define an abstract TTT based on its behavior. At the heart of any TTT is a trigger that tracks the progression of values that form some arbitrary sequence. The simplest concrete example is a down-counter that releases the attack payload when it reaches `zero`. Thus, we define TTTs as devices that track an arbitrary sequence of values constrained by only two properties:

- **the sequence never repeats a value,**
- **the sequence is incomplete.**

Fig. 1 shows the basic hardware components required to implement such a sequence counter in hardware. It has three building blocks: 1) **State-Saving Components (SSCs)**, 2) an **increment value**, and 3) an **increment event**.

To understand the power our definition gives to attackers, we use it to enumerate the space of all possible TTT triggers. We define a total of six TTT variants, including *distributed* TTTs that couple together SSCs scattered across the design to form a sequence counter and *non-uniform* TTTs that conceal their behavior by incrementing with inconsistent values, i.e., expressing what looks like a random sequence.

We leverage our definition of TTTs to locate SSCs in a design that behave like TTT triggers during functional verification. Specifically, we reduce the Trojan search space of the Design Under Test (DUT) by analyzing *only* the progression of values expressed by SSCs of potential TTT triggers. We design and implement an automated extension to existing functional verification toolchains, called **Bomberman**, for identifying the presence of TTTs in hardware designs. Bomberman computes a Data-Flow Graph (DFG) from a design's Hardware Description Language (HDL) (either pre- or post- synthesis) to identify the set of all combinations of SSCs that could construct a TTT. Initially, Bomberman assumes all SSCs are *suspicious*. As Bomberman analyzes the results obtained from functional verification, it marks any SSCs that violate our definition as *benign*. Bomberman reports any remaining *suspicious* SSCs to designers, who use this information to create a new test case for verification, or manually inspect connected logic for malice.

We demonstrate the effectiveness of Bomberman by implanting all six TTT variants into four different open-source hardware designs: a RISC-V CPU [18], an OR1200 CPU [19], a UART [19] module, and an AES accelerator [20]. Even with verification simulations lasting less than one million cycles,[1] Bomberman detects the presence of *all* TTT variants across *all* circuit designs with a false positive rate of less than 1.2%.

This paper makes the following contributions:

- An abstract definition and component-level breakdown of TTTs (§IV).
- Design of six TTT variants (§IV-C), including new variants that evade existing defenses (§VI-C1).
- Design and implementation of an automated verification extension, Bomberman, that identifies TTTs implanted in RTL hardware designs (§V).
- Evaluation of Bomberman's false positive rate (§VI-B) and a comparative security analysis against a range of both TTT-focused and "one-size-fits-all" design-time hardware Trojan defenses (§VI-C); Bomberman is the only approach capable of detecting all TTT variants, including state-of-the-art pseudo-random [21] and non-deterministic [22] TTTs.
- Algorithmic complexity analysis (§VI-D) of Bomberman's *SSC Enumeration* and *SSC Classification* stages.
- Open-source release of Bomberman and TTTs [23].

II. Background

A. IC Development Process

Developing complex ICs, like the Apple A13 Bionic chip that contains 8.5 billion transistors [24], employs several design phases (Fig. 2) that are heavily augmented with Computer Aided Design (CAD) tools. First, to minimize time-to-market, hardware designers often purchase existing IP blocks from third parties to integrate into their designs. Next, designers integrate all third-party IP, and describe the behavior of any custom circuitry at the RTL, using a Hardware Description Language (HDL) like Verilog. Next, CAD tools synthesize

[1] Typical verification simulations last $\approx$ millions of cycles [11].

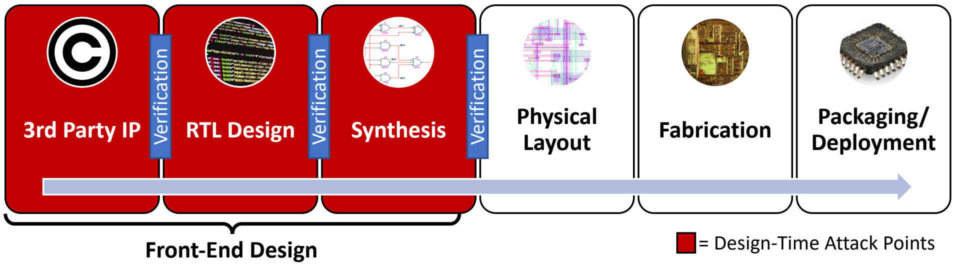

Fig. 2. **IC Development Process.** As ICs have become increasingly complex, both the reuse of 3rd party IP and the size of design teams has increased [3].

the HDL into a gate-level netlist (also described using HDL) targeting a specific process technology, a process analogous to software compilation. After synthesis, designers lay out the circuit components (i.e., logic gates) on a 3-dimensional grid and route wires between them to connect the entire circuit. CAD tools encode the physical layout in a Graphics Database System II (GDSII) format, which is then sent to the fabrication facility. Finally, the foundry fabricates the IC, and returns it to the designers who test and package it for mounting onto a printed circuit board. HDL-level Trojans inserted at design time compromise the final chip—even if the tools, back-end design, and fabrication are secure.

B. Hardware Trojans

Hardware Trojans are malicious modifications to a hardware design for the purpose of modifying the design's behavior. In Fig. 3 we adopt a hardware Trojan taxonomy that makes characterizations according to 1) where in the IC development process (Fig. 2) they are inserted, and 2) their architectures [25], [26]. Specifically, hardware Trojans can be inserted at design time [7], [11], [13], [27], at fabrication time [28]–[30], or during packaging/deployment [31]. In this paper, we focus on design-time Trojans, specifically Trojans inserted during front-end (i.e., HDL) design.

Hardware Trojans are comprised of two main components: a *trigger* and *payload* [32]–[34]. The trigger initiates the delivery of the payload upon reaching an activation state. It enables the Trojan to remain dormant under normal operation, e.g., during functional verification and post-fabrication testing. Conversely, the payload waits for a signal from the trigger to alter the state of the victim circuit. Given the focus of this work is identifying a specific class of Trojans defined by their *trigger*, we further classify Trojans accordingly.

There are two main types of triggers: *always-on* and *initially dormant*. As their names suggest, *always-on* triggers indicate a triggerless Trojan that is always activated, and are thus trivial to detect during testing. Always-on triggers represent an extreme in a trigger design trade-space—not implementing a trigger reduces the overall Trojan footprint at the cost of sacrificing stealth. Alternatively, *initially dormant* triggers activate when a signal within the design, or an input to the design, changes as a function of normal, yet rare, operation, ideally influenced by an attacker. *initially dormant* triggers enable stealthy, controllable, and generalizable hardware Trojans. As prior work shows, it is most advantageous for attackers to be able to construct triggers that hide their Trojan payloads to evade detection during testing [8]–[10], [13], [22], so we focus on *initially dormant* triggers.

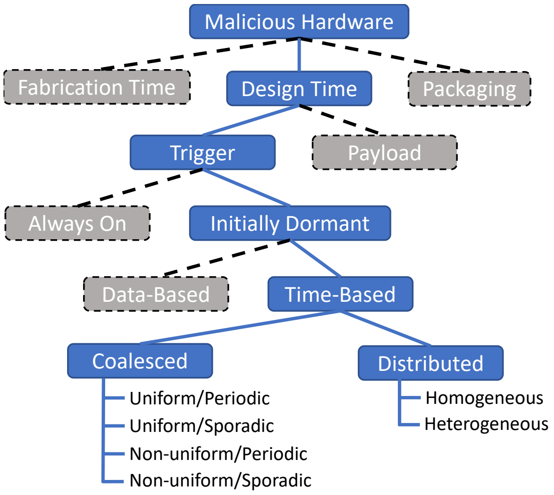

Fig. 3. **Taxonomy of Hardware Trojans.** Hardware Trojans are malicious modifications to a hardware design that alter its functionality. We focus on time-based Trojans (TTTs) and categorize them by design and behavior.

Initially dormant triggers consist of two sub-categories: *data-based* and *time-based* [11], [13], [22]. Data-based triggers, or *cheat codes*, wait to recognize a single data value (single-shot) or a sequence of data values to activate. Alternatively, time-based triggers, or *ticking timebombs*, become increasingly more likely to activate the more time has passed since a system reset. While, ticking timebombs can implement a indirect and probabalistic notion of time (§IV), a simple ticking timebomb trigger is a periodic up-counter, where every clock cycle the counter increments, as shown in Fig. 4A. In this work, we eliminate the threat of TTTs to force attackers to implement data-based Trojans that require post-deployment attacker interaction to trigger [11].

III. Threat Model

Our threat model follows that used by prior work on design-time Trojan attacks and defenses [11], [13], [16], [21], [22], [35], [36]. Specifically, we focus on malicious modifications that are embedded in in-house, 3rd party, or netlist HDL (Fig. 2). Our focus, on design-time attacks is driven by current design trends and economic forces that favor reliance on untrusted 3rd parties and large design teams [3]. Additionally, without a trusted HDL design, any result of back-end design and fabrication cannot be trusted.

We assume that a design-time adversary has the ability to add, remove, and modify the RTL or netlist HDL of the core design in order to implement hardware Trojans. This can be done either by a single rogue employee at a hardware design company, or by entirely rogue design teams. We also assume an attacker only makes modifications that evade detection during design verification. Thus, no part of the design can be trusted until vetted by Bomberman and other heuristics-based tools [8]–[10]. Like prior work [8]–[10], [13], [21], [22], we assume that malicious circuit behavior triggered by Trojan activation is caught via verification testing.

We focus on identifying TTTs as we define them in §IV. In doing so, we force attackers to implement data-based (cheat code) Trojans, which require large state machines to achieve stealth during design verification [14], [15], subsequently making them detectable post-fabrication via side channels [37]–[43]. Moreover, data-based Trojans have limited deployability—e.g., they cannot target air-gapped machines—since they require post-deployment attacker interaction [11]. Our defense can be deployed at any point throughout the front-end design process—i.e., directly verifying 3rd party IP, after RTL design, or after synthesis—after which the design is trusted to be free of TTTs.

IV. Ticking Timebomb Triggers

First, we define TTTs by their behavior. Based on this definition, we synthesize the fundamental components required to implement a TTT in hardware. Finally, using these fundamental components we enumerate six total TTT variants, including previously contrived TTTs that resemble contiguous time counters [11], [13], to more complex, distributed, non-uniform, and sporadic [21], [22] designs.

A. Definition

We define TTTs as the set of hardware Trojans that implement a time-based trigger that monotonically approaches activation as the victim circuit continuously operates without reset. ***More succinctly, we define a ticking timebomb trigger based on two properties of the values it exhibits while still dormant yet monotonically approaching activation***:

Property 1: The TTT does NOT repeat a value without a system reset.

Property 2: The TTT does NOT enumerate all possible values without activating.

Property 1 holds by definition, since, if a TTT trigger repeats a value in its sequence, it is no longer a ticking timebomb, but rather a data-based "cheat code" trigger [11], [13]. Property 2 holds by contradiction in that, if a TTT trigger enumerates all possible values *without* triggering, i.e., no malicious circuit behavior is observed, then the device is not malicious, and therefore not part of a TTT. Upon these two properties, we derive the fundamental hardware building blocks of a TTT.

Figs. 4A–D illustrate example ticking timebomb behaviors that are captured by our definition, in order of increasing complexity. The most naive example of a ticking timebomb trigger is a simple periodic up-counter. While effective, a clever attacker may choose to hide the monotonically increasing behavior of a *periodic* up-counter by either 1) obscuring the relationship between successive counter values (e.g., AES counter mode sequence, Fig. 4B), or 2) *sporadically* incrementing the counter (e.g., a non-deterministic TTTs [22], Fig. 4). Even more sophisticated, the attacker may choose to do both (Fig. 4D).

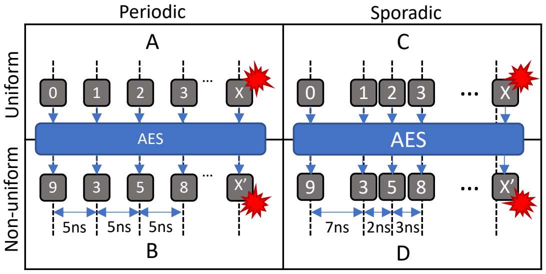

Fig. 4. **Ticking Timebomb Trigger Behaviors.** There are four primitive ticking timebomb trigger counting behaviors, in order of increasing complexity, captured by our definition (Properties 1 & 2 in §IV-A). **A)** The simplest counting behavior is both *periodic* and *uniform*. Alternatively, more sophisticated counting behaviors are achieved by: **B)** encrypting the count to make the sequence *non-uniform*, **C)** incrementing it *sporadically*, or **D)** both.

B. TTT Components

From our definition, we derive the fundamental components required to implement a TTT in hardware. Fig. 1 depicts these components. For TTTs to exhibit the behaviors summarized in Fig. 4, they must implement the notion of an *abstract time counter*. TTT time counters require three components to be realized in hardware: 1) **State-Saving Components (SSCs)**, 2) increment **value**, and 3) increment **event**.

The *SSC* defines how the TTT saves and tracks the triggering state of the time counter. SSCs can be either *coalesced* or *distributed*. Coalesced SSCs are comprised of *one* N-bit register, while distributed SSCs are comprised of M, N-bit registers declared across the design. Distributed SSCs have the advantage of increasing stealth by combining a subset of one or multiple coalesced SSCs whose count behaviors *individually* violate the definition of a TTT trigger (i.e., Properties 1 and 2), but when considered *together* comprise a valid TTT. Distributed SSCs can also reduce hardware overhead through reuse of existing registers.

The TTT *increment value* defines *how* the time counter is incremented upon an increment event. The increment value can be uniform or non-uniform. Uniform increments are hard-coded values in the design that do not change over time, e.g., incrementing by one at every increment event. Non-uniform increments change depending on device state and operation, e.g., incrementing by the least-significant four bits of the program counter at every increment event.

Lastly, the TTT *increment event* determines *when* the time counter's value is incremented. Increment events may be *periodic* or *sporadic*. For example, the rising edge of the clock is periodic, while the rising edge of an interrupt is sporadic.

C. TTT Variants

From the behavior of the fundamental TTT components, we extrapolate six TTT variants that represent the TTT design space as we define. We start by grouping TTTs according to their SSC construction. Depending on their sophistication level, the attacker may choose to implement a simplistic *coalesced* TTT, or construct a larger, more complex, *distributed* TTT. If the attacker chooses to implement a *coalesced* TTT, they have four variants to choose from, with respect to increment uniformity and periodicity. The most naive attacker may choose to implement a coalesced TTT with uniform increment values and periodic increment events. To make the coalesced TTT more difficult to identify, the attacker may choose to implement non-uniform increment values and/or sporadic increment events.

To increase stealth, an attacker may choose to combine two or more coalesced TTTs, that alone violate the definition of being a TTT trigger, but combined construct a valid *distributed* TTT. An attacker has two design choices for distributed TTTs. Seeking to maximize stealth, the attacker may choose to combine several copies of the *same* coalesced TTT with non-uniform increment values and sporadic increment events, thus implementing a *homogeneous* distributed TTT. Alternatively, the attacker may seek integration flexibility, and choose to combine various coalesced TTTs to implement a *heterogeneous* distributed TTT. For homogeneous distributed TTTs, an attacker has the same four design choices as in coalesced TTTs. However, for heterogeneous distributed TTTs, the design space is much larger. Specifically, the number of subcategories of heterogeneous distributed TTTs can be computed using the binomial expansion, $\binom{n}{k}$, with n, the number of coalesced sub-triggers, and k, the number of unique subtrigger types. We summarize all six TTT variants and their behaviors in Figs. 3 and 4, respectively, and provide example implementations in Verilog in Appendix A.

V. Bomberman

Now that we have defined what a TTT is, and how it behaves, how do we automatically locate them within complex RTL designs? To address this question, we design and implement *Bomberman*, a dynamic Trojan verification framework.[2] ***To summarize, Bomberman locates potential TTTs by tracking the sequences expressed by all SSCs in a design, as SSCs are one of the fundamental building blocks of TTTs.*** Initially, Bomberman classifies all SSCs as suspicious. Then, any SSCs whose sequence progressions, recorded during simulation, violate either Properties in §IV-A, are marked benign.

Bomberman takes as input 1) a design's HDL, and 2) verification simulation results, and automatically flags suspicious SSCs that could be part of a TTT. The Bomberman dynamic analysis framework is broken into two phases:

1) **SSC Identification**, and
2) **SSC Classification**.

During the *SSC Identification* phase, Bomberman identifies all *coalesced* and *distributed* SSCs within the design. During the *SSC Classification* phase, Bomberman analyzes the value progressions of all SSCs to identify suspicious SSCs that may comprise a TTT. Fig. 5 illustrates the Bomberman architecture.

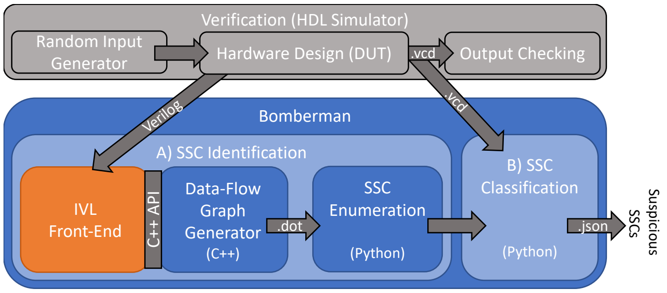

Fig. 5. **Bomberman Architecture.** Bomberman is comprised of two stages: A) *SSC Identification*, and B) *SSC Classification*. The first stage (A) identifies all *coalesced* and *distributed* SSCs in the design. The second stage (B) *starts by assuming all SSCs are **suspicious***, and marks SSCs as ***benign*** as it processes the values expressed by each SSC during verification simulations.

A. SSC Identification

The first step in locating TTTs, is *identifying* all SSCs in the design. Identifying coalesced SSCs is straightforward: any component in the HDL that may be synthesized into a coalesced collection of flip-flops (or latches §VII-2) is considered a coalesced SSC. Enumerating distributed SSCs is more challenging. Since distributed SSCs are comprised of various combinations of coalesced SSCs that are interconnected the host circuit, a naive approach would be to enumerate the power set of all coalesced SSCs in the design. However, this creates an obvious state-explosion problem, and is unnecessary. Instead, *we take advantage of the fact that not every component in a circuit is connected to every other component*. Moreover, the structure of the circuit itself tells us what connections between coalesced SSCs are possible, and thus the distributed SSCs Bomberman must track.

Therefore, we break the *SSC Identification* phase into two sub-stages: 1) Data-Flow Graph (DFG) Generation, and 2) SSC Enumeration (Fig. 5A). First, we generate a DFG from a circuit's HDL, where each node in the graph represents a signal, and each edge represents connections between signals facilitated by intermediate combinational or sequential logic. Then, we systematically traverse the graph to enumerate: 1) the set of all coalesced SSCs, and 2) the set of all *connected* coalesced SSCs, or distributed SSCs.

*1) **DFG Generation**:* We implement the *DFG Generation* stage of the *SSC Identification* phase using the open-source Icarus Verilog (IVL) [46] compiler front-end with a custom back-end written in C++. Our custom IVL back-end traverses the intermediate HDL code representation generated by the IVL front-end, to piece together a bit-level signal dependency, or data-flow, graph. In doing so, it distinguishes between

[2] Unfortunately, no commercial verification tool exists to track complex state that defines TTT invariants, i.e., asserting no repeated values or *distributed* state exhaustion. Moreover, the closest such tools—JasperGold [44] and VC Formal [45]—deploy *bounded static* analysis approaches that suffer from state-explosion when applied to such invariants.

state-saving signals (i.e., signals gated by flip-flops) and intermediate signals output from combinational logic. Continuous assignment expressions are the most straightforward to capture as the IVL front-end already creates an intermediate graph-like representation of such expressions. However, procedural assignments are more challenging. Specifically, at the RTL level, it is up to the compiler to infer what HDL signals will synthesize into SSCs. To address this challenge, we use a similar template-matching technique used by modern HDL compilers [47], [48]. The data-flow graph is expressed using the Graphviz *.dot* format. Fig. 11 in Appendix C shows an example data-flow graph generated by Bomberman.

*2) **SSC Enumeration**:* We implement the *SSC Enumeration* stage of the *SSC Identification* phase using a script written in Python. First, our *SSC Enumeration* script iterates over every node in the circuit DFG, and identifies nodes (signals) that are outputs of registers (flip-flops). The script marks these nodes as coalesced SSCs. Next, the script performs a Depth-First Search (DFS), starting from each non-coalesced SSC signal node, to piece together distributed SSCs. The DFS backtracks when an input or coalesced SSC signal is reached. When piecing together distributed SSCs, Bomberman does *not* take into account word-level orderings between root coalesced SSCs. The order of the words, and thus the bits, of the distributed SSC does not affect whether it satisfies or violates the properties of our definition of a TTT trigger (§IV-A). Our definition does not care about the progression of values expressed by the SSC(s), but only cares if *all values are not expressed* and *individual values are not repeated*. Note, a clever attacker may try to avoid detection by selecting a slice of a single coalesced SSC to construct a ticking timebomb trigger. However, our implementation of Bomberman classifies a single sliced coalesced SSC as a *distributed* SSC with a single root *coalesced* SSC.

Algorithm 1: SSC Classification Algorithm

```
Input: Set, $P$, of all possible SSCs
Output: Set, $S$, of all suspicious SSCs
$S \leftarrow P$;
foreach $p \in P$ do
    $n \leftarrow SizeOf(p)$;
    $V_p \leftarrow \emptyset$; /* previous values of $p$ */
    foreach $t \in T$ do
        $value \leftarrow ValueAtTime(p, t)$;
        if $value \in V_p$ then
            Remove $p$ from $S$;
            Break;
        else
            Add $value$ to $V_p$;
        end
    end
    if $\|V_p\| == 2^n$ then
        Remove $p$ from $S$;
    end
end
```

B. SSC Classification

After all SSCs have been enumerated, Bomberman analyzes the values expressed by every SSC during verification simulations to classify whether each SSC is either suspicious—meaning it *could* construct a TTT—or benign. *Bomberman begins by assuming all SSCs within the design are **suspicious***. At every update time within the simulation, Bomberman checks to see if any SSC expresses a value that causes it to violate either property of our definition (§IV-A). If a property is violated, the SSC no longer meets the specifications to be part of a TTT, and Bomberman classifies it *benign*. ***Bomberman does not care how, when, what, or the amount an SSC's value is incremented; rather, Bomberman only monitors if an SSC repeats a value, or enumerates all possible values***. Lastly, Bomberman reports any remaining suspicious SSCs for manual analysis by verification engineers.

We implement the *SSC Classification* algorithm—Algorithm 1—using Python. Our classification program (Fig. 5B) takes as input a Value Change Dump (VCD) file, encoding the verification simulation results, and cross-references the simulation results with the set of suspicious SSCs, initially generated by the *SSC Identification* stage (Fig. 5A). For coalesced SSCs, this is trivial: our analysis program iterates over the values expressed by each coalesced SSC during simulation, and tests if either property from our definition (§IV-A) is violated. SSCs that break our definition of a TTT are marked *benign*. However, distributed SSCs are more challenging. To optimize file sizes, the VCD format only records signal values when they change, not every clock cycle. This detail is important when analyzing distributed SSCs, whose root coalesced SSCs may update at different times. We address this detail by time-aligning the root coalesced SSC values with respect to each other to ensure the recording of all possible distributed SSC values expressed during simulation. Finally, any remaining suspicious SSCs are compiled into a JSON file, and output for verification engineers to inspect and make a final determination on whether or not the design contains TTTs.

VI. Evaluation

By construction Bomberman cannot produce false negatives since it initially assumes *all* SSCs are suspicious, and only marks SSCs as benign if they express values during simulation that violate the definition of TTT SSC behavior. However, false positives *are* possible. To quantify Bomberman's false positives rate, we evaluate Bomberman against four real-world hardware designs with TTTs implanted in them. To model a production simulation-based verification flow, we use a mix of existing test vectors (from each core's repository), random test vectors (commonly used to improve coverage), and custom test vectors (to fill coverage gaps). To contextualize Bomberman's effectiveness compared to state-of-the-art TTT defenses, we build an end-to-end (E2E) TTT—that uses a pseudorandom sequence to trigger a privilege escalation within a processor—that evades all defenses except Bomberman. Lastly, we provide an asymptotic complexity analysis of the Bomberman framework, and characterize Bomberman's performance in practice.

A. Experimental Setup

*1) **Hardware Designs**:* We evaluate Bomberman against four open-source hardware designs: 1) an AES accelera-

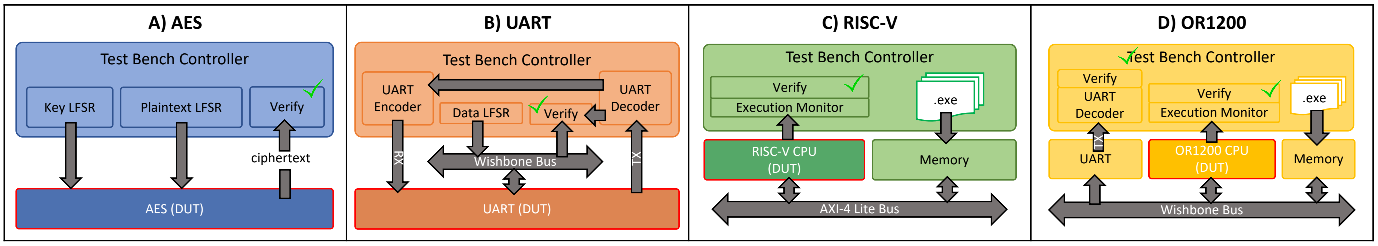

Fig. 6. **Hardware Testbenches.** Testbench architectures for each DUT (outlined in red). For the AES and UART designs, LFSRs generate random inputs for testing. For the RISC-V and OR1200 CPUs, we compile ISA-specific assembly programs [18], [49] into executables to exercise each design.

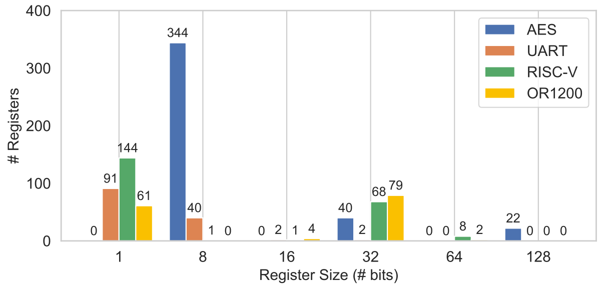

Fig. 7. **Hardware Design Complexities.** Histograms of the (coalesced) registers in each hardware design.

tor [20], 2) a UART module [19], 3) a RISC-V CPU [18], and 4) an OR1200 CPU [19]. Fig. 6 provides details on the testing architectures we deployed to simulate each IP core. We also summarize the size and complexity of each hardware design in terms of the number of registers (i.e., potential SSCs) in Fig. 7. The AES, RISC-V, and OR1200 designs are shown to be the most computationally-intensive designs for Bomberman to analyze, since they have large registers (≥32-bits), i.e., potentially suspicious SSCs that can increment almost indefinitely.

AES Accelerator. The AES core operates solely in 128-bit counter (CTR) mode. It takes a 128-bit key and 128-bits of plaintext (i.e., a counter initialized to a random seed) as input, and 22 clock cycles later produces the ciphertext. Note, the design is pipelined, so only the first encryption takes 22 clock cycles, and subsequent encryptions are ready every following clock cycle. We interface two Linear Feedback Shift Registers (LFSRs) to the DUT to generate random keys and plaintexts to exercise the core (Fig. 6A). Upon testing initialization, the testbench controller resets and initializes both LFSRs (to different random starting values) and the DUT. It then initiates the encryption process, and verifies the functionality of the DUT is correct, i.e., each encryption is valid.

UART Module. The UART module interfaces with a Wishbone bus and contains both a transmit (TX) and receive (RX) FIFO connected to two separate 8-bit TX and RX shift registers. Each FIFO holds a maximum of sixteen 8-bit words. The core also has several CSRs, one of which configures the baud rate, which we set to 3.125 MHz. We instantiate a Wishbone bus arbiter to communicate with the DUT, and an LFSR to generate random data bytes to TX/RX (Fig. 6B). We also instantiate a UART encoder/decoder to receive, and echo back, any bytes transmitted from the DUT. Upon initialization, the testbench controller resets and initializes the Wishbone bus arbiter, LFSR, and DUT, and begins testing.

RISC-V CPU. The RISC-V CPU contains 32 general-purpose registers, a built-in interrupt handler, and interfaces with other on-chip peripherals through a 32-bit AXI-4 Lite or Wishbone bus interface. We instantiate an AXI-4 Lite bus arbiter to connect the DUT with a simulated main memory block to support standard memory-mapped I/O functions (Fig. 6C). The testbench controller has two main jobs after it initializes and resets all components within. First, it initializes main memory with an executable to be run on the bare metal CPU. These programs are in the form of *.hex* files that are compiled and linked from RISC-V assembly or C programs using the RISC-V cross-compiler toolchain [50]. Second, it monitors the progress of each program execution and receives any output from an executing program from specific memory addresses. We configure the testbench controller to run multiple programs sequentially, without resetting the device.

OR1200 CPU. The OR1200 CPU implements the OR1K RISC ISA. It contains a 5-stage pipeline, instruction and data caches, and interfaces with other peripherals through a 32-bit Wishbone bus interface. We instantiate a Wishbone bus arbiter to connect the DUT with a simulated main memory block and a UART module to support standard I/O functions (Fig. 6D). The testbench controller has two jobs after it initializes and resets all components within. First, it initializes main memory with an executable to be run on the bare metal CPU. These programs are in the form of *.vmem* files that are compiled and linked from OR1K assembly or C programs using the OR1K cross-compiler toolchain [51]. Second, it monitors the progress of each program execution and receives any program output from the UART decoder. Like the RISC-V, we configure the OR1200 testbench controller to run multiple programs sequentially, without resets in between.

2) ***System Setup**:* As described in §V, Bomberman interfaces with Icarus Verilog (IVL). IVL is also used to perform all verification simulations of our four hardware designs. In both cases, we use version 10.1 of IVL. Both IVL and Bomberman were compiled with the Clang compiler (version 10.0.1) on a MacBook Pro with a 3.1 GHz Intel Core i7 processor and 16 GB DDR3 RAM. All RTL simulations and Bomberman analyses were also run on the same machine.

B. False Positives

We empirically quantify Bomberman's false positive rate by analyzing four real world hardware designs (§VI-A1). Addi-

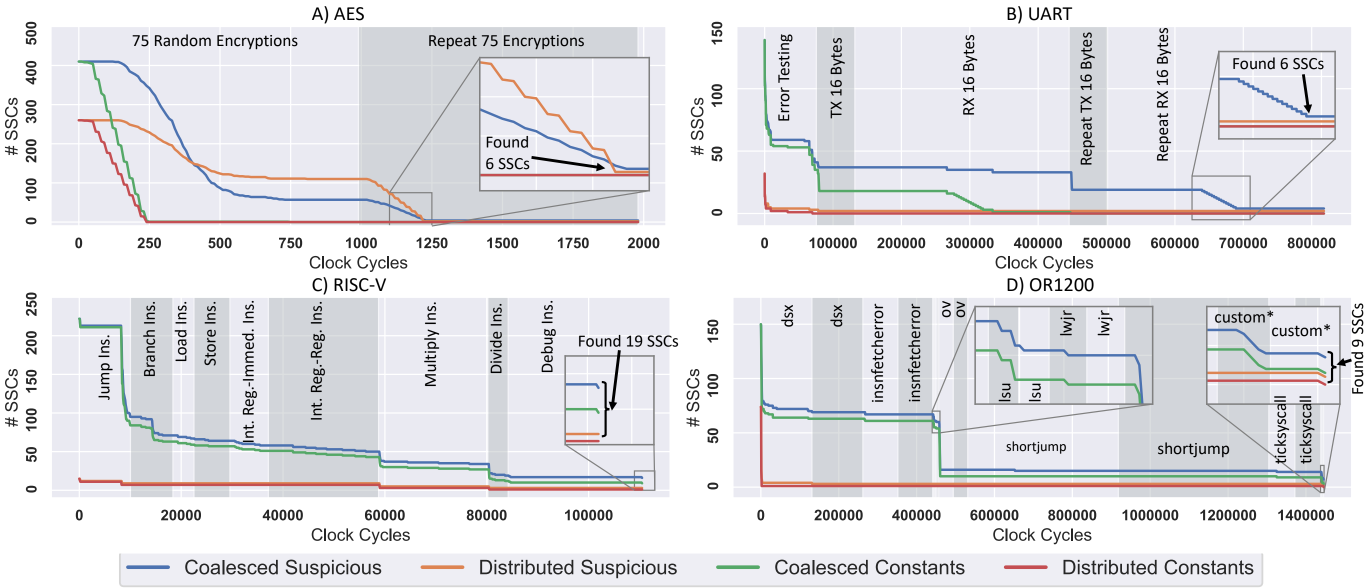

Fig. 8. **False Positives.** Reduction in SSCs classified as suspicious across all four hardware designs over their simulation timelines. **A) AES.** Bomberman identifies the SSCs of all six TTT variants implanted with zero false positives. **B) UART.** (Same as AES). **C) RISC-V.** Bomberman flags 19 SSCs as suspicious, six from implanted TTTs, three from benign performance counters, and ten benign constants resulting from on-chip CSRs. **D) OR1200.** Bomberman flags nine SSCs as suspicious, six from implanted TTTs, and three benign constants.

tionally, we verify our implementation of Bomberman does not produce false negatives—**as this should be impossible**—by implanting all six TTT variants (§A) within each design. For each design, we plot the number of suspicious SSCs flagged by Bomberman over a specific simulation timeline. Based on the TTT trigger definitions provided in §IV-A, we categorize SSCs within each plot as follows:

1) **Suspicious**: a (coalesced or distributed) SSC for which all possible values have not yet been expressed *and* no value has been repeated;
2) **Constant**: a (coalesced or distributed) SSC for which only a *single* value has been expressed.

Note *coalesced* and *distributed* classifications *are* mutually exclusive, as they are SSC design characteristics. However, *suspicious* and *constant* classifications are *not* mutually exclusive. By definition (§IV-A), an SSC that has only expressed a single value during simulation is suspicious. While constants SSCs are also suspicious, we plot both to enable Bomberman users to distinguish between SSCs that store configuration settings (commonly used in larger designs) from SSCs that store sequence progressions (e.g., TTTs or performance counters).

AES Accelerator. We configure the AES testbench to execute 75 random encryptions, i.e., 75 random 128-bit values with 75 (random and different) 128-bit keys, and subsequently repeat the same 75 encryptions. We simulate the AES core at 100 MHz. In Fig. 8A we plot the number of suspicious SSCs tracked by Bomberman over the simulation timeline.

During the first 250 clock cycles of simulation, as registers cycle through more than one value, they are removed from the sets of constants. During the initial 75 random encryptions, after ≈ 750 clock cycles, the 8-bit registers toggle through all 256 possible values, and thus are also eliminated from the sets of suspicious SSCs. However, after the initial 75 encryptions, the number of false positives is still quite high, as the 32- and 128-bit registers have yet to toggle through all possible values, or repeat a value. Since these registers are quite large, toggling through all possible values is infeasible. Driven by the observation that the data-path of a TTT-free design tracks state from test inputs, not since the last system reset, we take an alternative approach to eradicate large SSC false positives. Formally, ***we repeat the same test case(s) without an intermediate system reset to cause only non-suspicious SSCs to repeat values*** (violating Property 1 in §IV-A). We use this insight to efficiently minimize suspicious SSC false positives. Since ***the AES core is a deterministic state machine with no control-path logic***, we simply reset the LFSRs, and repeat the same 75 encryptions. After ≈ 1200 clock cycles, we achieve a false positive rate of 0% while detecting 100% of the TTT variants implanted in the core.

UART Module. We configure the UART testbench to perform configuration, error, and TX/RX testing. During the configuration and error testing phases, configuration registers are toggled between values, and incorrect UART transactions are generated to raise error statuses. During the TX/RX testing, 16 random bytes are transmitted by the DUT, and upon being received by the UART decoder, are immediately echoed back, and received by the DUT. Following our insights from the AES experiments, we transmit and receive the *same* set of 16 bytes again, to induce truly non-suspicious SSCs to repeat values. We plot the number of suspicious SSCs identified by Bomberman over the simulation timeline in Fig. 8B.

During the first ≈ 80k clock cycles (error testing phase), Bomberman eliminates over 50% of all potentially suspicious (coalesced) SSCs, as many of the UART's registers are either single-bit CSRs that, once toggled on and off, both: 1) cycle through all possible values, and 2) repeat a value. Subsequently, during the first TX testing phase, the 16-byte TX FIFO is saturated causing another 50% reduction in the number of coalesced constants. Likewise, once the DUT transmits all 16 bytes to the UART decoder, and the UART encoder echos them all back, the 16-byte RX FIFO is saturated causing another reduction in the number of coalesced constants.

After the initial TX/RX testing phase, we are still left with several (suspicious) false positives. This is because the TX and RX FIFO registers have yet to cycle through all possible values, nor have they repeated a value. While these registers are small (8-bits), and continued random testing would eventually exhaustively exercise them, we leverage our observations from the prior AES simulation: we repeat the previous TX/RX test sequence causing data-path registers to repeat values, eliminating all false positives. Again, Bomberman successfully identifies *all* TTT variants with *zero* false positives.

RISC-V CPU. We configure the RISC-V CPU testbench to run a single RISC-V assembly program that exercises all eight instruction types. The assembly test program was selected from the open-source RISC-V design repository [18]. These instructions include jumps, branches, loads, stores, arithmetic register-immediate and register-register, multiplies, and divides. We simulate the RISC-V core and again plot the number of suspicious SSCs identified by Bomberman (Fig. 8C).

During the execution of the first set of instructions (jumps), Bomberman largely reduces potential constant and suspicious SSCs. This is because, like the UART module, most of the registers within the RISC-V CPU are 1-bit CSRs for which enumerating all (2) possible values is trivial. The remaining 90 suspicious SSCs are slowly eradicated as more instructions execute, causing the remaining control-path signals to enumerate all possible values. Similar to repeating the same encryptions during the AES simulation, the assembly programs were designed to load and store repeated values in the large ($\geq$ 32-bit) registers, causing them to violate Property 1 (§IV-A).

In the end, Bomberman identifies 19 suspicious SSCs: 16 coalesced and three distributed. Upon manual inspection, we identify four of the 16 coalesced SSCs, and two of the three distributed SSCs, as components of the six implanted (malicious) TTTs. Of the 12 remaining coalesced SSCs, we identify three as *benign* timeout and performance counters, and nine as *benign* constants that stem from unused CPU features, the hard-coded zero register, and the interrupt mask register. Lastly, we identify the single remaining distributed SSC as a combination of some of the *benign* coalesced constants. In a real world deployment scenario, we imagine verification engineers using Bomberman's insights to tailor their test cases to maximize threat-specific testing coverage, similar to how verification engineers today use coverage metrics to inform them of gaps in their current test vectors.

Recall, Bomberman only flags SSCs whose value progressions do not violate the properties of a TTT (§IV-A). At most, Bomberman will only flag SSCs as *suspicious*. It is up to the designer or verification engineer to make the final determination on whether or not an SSC is *malicious*. By locating all (malicious) implanted TTTs and (benign) performance counters, we validate Bomberman's correctness.

OR1200 CPU. Lastly, we configure the OR1200 testbench to run eight different OR1K assembly programs. Like the AES and UART simulations, we configure the testbench to perform repeated testing, i.e., execute each program twice, consecutively, without an intermediate device reset. The first seven test programs are selected from the open-source OR1K testing suite [49], while the last program is custom written to exercise specific configuration registers not exercised by the testing suite. We simulate the OR1200 at 50 MHz, and plot the number of suspicious SSCs identified by Bomberman over the simulation timeline in Fig. 8D.

In the end, Bomberman identifies nine suspicious SSCs, seven coalesced and two distributed. Four of the seven coalesced SSCs, and both distributed SSCs, are components of the six implanted TTTs. The remaining three coalesced SSCs are constants, and false positives. We manually identify these false positives as shadow registers only used when an exception is thrown during a multi-operand arithmetic instruction sequence.

C. Comparative Analysis of Prior Work

To demonstrate the need for *Trojan-specific* verification tools like Bomberman, we provide a two-fold comparative analysis between Bomberman and existing design-time Trojan defenses. First, we study the capabilities of each defense in defeating all six TTT variants described in §IV-C. We summarize each defense and its effectiveness in Tab. I, and describing why some defenses fail to defeat all TTT variants below. Armed with this knowledge, we construct an E2E TTT in Verilog—targeting the OR1200 [19] processor—that is capable of bypassing all existing defenses except Bomberman. We describe the fundamental building blocks of our TTT—and the corresponding Verilog components in our implementation—that enable it to defeat prior defenses.

*1) **Security Analysis of Existing Defenses**:* There are two approaches for defending against TTTs: 1) ***Trojan-agnostic***, 2) ***TTT-specific***. *Trojan-agnostic* techniques are primarily verification focused, and include: FANCI [8], UCI [10] and VeriTrust [9]. While these approaches differ in implementation (static vs. dynamic), from above they are similar. All three locate rarely used logic that comprise most generic Trojan circuits. Unfortunately, researchers have demonstrated systematic approaches to transform almost any Trojan circuit to evade these techniques, while maintaining logical equivalence [14], [15]. Alternatively, *TTT-specific* approaches such as WordRev [35], [36] and Waksman *et al.*'s Power Resets [13], attempt to counter only TTTs. While these approaches work against known TTTs at the time of their respective publications, they fail to recognize the behavior of *all* TTT variants presented in this work. In Tab. I, we summarize each defense, and the TTT variants (§IV-C and §A) they can defeat. Below,

TABLE I
COMPARATIVE SECURITY ANALYSIS OF TTT DEFENSES AND BOMBERMAN.

	Defense	UCI [10]	FANCI [8]	VeriTrust [9]	WordRev [35]	Power Resets [13]	Bomberman
	Type	Trojan-Agnostic	Trojan-Agnostic	Trojan-Agnostic	TTT-Specific	TTT-Specific	TTT-Specific
	Analysis	Dynamic	Static	Dynamic	Static	N/A†	Dynamic
	Target	Activation Signals	Comparator Inputs	Activation Signals	Increment Logic	SSCs	SSCs
TTT Type	CUP	✓	✗	✗	✓	✓	✓
	CUS	✓	✗	✗	✓	✗	✓
	CNP	✓	✗	✗	✗	✗	✓
	CNS	✓	✗	✗	✗	✗	✓
	D-HMG	✗	✗	✗	✗	✗*	✓
	D-HTG	✗	✗	✗	✗	✗	✓

XXX: Coalesced or Distributed SSC — XXX: Uniform or Non-uniform Increment Value — XXX: Periodic or Sporadic Increment Event

† *Power Resets* [13] are a runtime mechanism, not a verification technique.

* Power resets ***only*** defend against homogeneous distributed TTTs compromised entirely of CUP sub-components.

we provide a security analysis of each defense, describing how and what TTT variants are defeated.

UCI. UCI [10] is a ***Trojan-agnostic*** dynamic verification tool that searches HDL for intermediate combinational logic that does not affect signal values from source to sink during verification simulations. Since TTT trigger components—SSCs, increment event, increment amount—remain active during simulation, UCI would not flag them as suspicious. However, TTTs also have a comparator that checks if the SSC's count has reached its activation state. Since the output of this comparator—the trigger ***activation signal*** (Fig. 1)—would remain unchanged during simulation, UCI would flag it. Unfortunately, as Sturton *et al.* show [14], having two activation signals—e.g., a distributed TTT—that each express their activation states under simulation, but never simultaneously, would evade UCI. As we show in our E2E TTT below (§VI-C2), this can be achieved using a distributed SSC constructed of *fast* and *slow* (coalesced) counters that wrap around (repeat values individually). Since the overall distributed SSC would not violate TTT properties (§IV-A), it would still be flagged by Bomberman.

FANCI. FANCI [8] is a ***Trojan-agnostic*** static verification framework that locates potential Trojan logic by computing "control values" for inputs to intermediate *combinational* logic in a design. Inputs with low control values are *weakly-affecting* [8], and most likely Trojan ***comparator inputs*** (Fig. 1) that indicate the current state of the trigger, e.g. a specific time counter value. Control values can be approximated by randomly sampling the truth tables of intermediate logic across the design. Unfortunately, Zhang *et al.* construct a systematic framework—*DeTrust* [15]—that distributes trigger comparator inputs across layers of *sequential* logic to increase their control values, hiding them from FANCI. Since any TTT variant can be used with *DeTrust*-transformed comparator logic, FANCI cannot identify any TTTs.

VeriTrust. Similar to UCI, VeriTrust [9] is a ***Trojan-agnostic*** dynamic verification framework that locates (unused) Trojan trigger ***activation signals*** (Fig. 1) in combinational logic cones that drive sequential logic. However, unlike UCI, VeriTrust locates activation signals by locating unused *inputs*—not logic—to the victim logic encapsulating a Trojan's *payload. This semantic difference enables VeriTrust to detect Trojans irrespective of their implementations.* Unfortunately, using their *DeTrust* framework [15], Zhang *et al.* illustrate how splitting the activation signals of any TTT design across multiple combinational logic cones, separated by layers of sequential logic, evades VeriTrust.

WordRev. WordRev [35], [36] is ***TTT-specific*** static analysis tool that identifies SSCs that behave like counters. WordRev leverages the notion that the carry bit propagates from the least-significant position to the most-significant position in counter registers. Thus, the ***increment logic*** connecting SSCs must be configured to allow such propagation. However, this operating assumption causes WordRev to miss distributed TTTs, and TTTs with non-uniform increment values.

Power Resets. Waksman *et al.* [13] suggest intermittent power resets as a ***TTT-specific*** defense. Intermittent power resets prevent potential TTT SSCs from reaching their activation states. This approach requires formally verifying/validating the correct operation of the DUT for a set amount of time, denoted the *validation epoch*. Once they guarantee no TTT is triggered within the validation epoch, the chip can safely operate as long as its power is cycled in time intervals less than the validation epoch. Unfortunately, as Imeson *et al.* [22] point out, this type of defense only works against TTTs with uniform increment values and periodic increment events, as it is impractical to formally verify non-deterministic (sporadic and/or non-uniform) designs.

*2) **End-to-End Supervisor Transition TTT**:*

Using the approaches for defeating each *Trojan-agnostic* and *TTT-specific* defense described above [14], [15], we systematically construct an E2E TTT (List. 2) that evades all defenses, except Bomberman. Our Trojan provides a *supervisor transition foothold* that enables attackers to bypass system authentication mechanisms and obtain root-level privileges.

Attack Target. Our TTT (List. 2) is based on a *supervisor transition foothold* Trojan first described by Sturton *et al.* in [14]. This Trojan targets a microprocessor circuit, and enables an attacker to arbitrarily escalate the privilege mode of the processor to supervisor mode. In List. 1, we provide a simplified version of the un-attacked processor HDL that updates the processor's supervisor mode register. Under non-trigger conditions, the supervisor signal—*super*—is either updated via an input signal—*in.super*—on the following clock edge, if the $holdn$ bit is 1 ($holdn$ is active low), otherwise the $super$ signal holds the same value from the previous clock period. Additionally, the $super$ signal is reset to 1 (supervisor mode) when the processor is reset via the active-low $resetn$ signal.

Listing 1. Unmodified HDL of the processor's supervisor-mode update logic.

```verilog
always @(posedge clk) begin
  super <= ~resetn | (~holdn & super) | (holdn & in.super);
end
```

Listing 2. Verilog HDL of a TTT that evades all existing design-time Trojan detection techniques—including UCI [10], FANCI [8], VeriTrust [9], WordRev [35], [36], and power resets [13]—except ***Bomberman***. This TTT alters logic (List. 1) that updates the supervisor-mode bit register.

```verilog
1  // Distributed TTT SSCs to evade UCI
2  reg [15:0] count_1; // Assume reset to 16'h0000
3  reg [15:0] count_2; // Assume reset to 16'h0000
4
5  // TTT Trigger Deployment Signal
6  reg [6:0] deploy_1; // Assume reset to 7'b0000000
7  reg [6:0] deploy_2; // Assume reset to 7'b0000000
8
9  // Update SSCs non−uniformly and sporadically
10 // to defeat WordRev and Power Resets
11 always @posedge(pageFault) begin
12     count_1 <= count_1 + PC[3:0];
13     count_2 <= count_2 + PC[5:2];
14 end
15
16 // Distribute trigger activation input signal (count_1)
17 // across layers of sequential logic to defeat FANCI.
18 always @(posedge clk) begin
19   if (count_1[3:0] == 'DEPLOY_0)
20     deploy_1[0] <= 1;
21   else
22     deploy_1[0] <= 0;
23        :        :        :
24   if (count_1[15:12] == 'DEPLOY_3)
25     deploy_1[3] <= 1;
26   else
27     deploy_1[3] <= 0;
28 end
29
30 always @(posedge clk) begin
31   if (deploy_1[2:0] == 2'b11)
32     deploy_1[4] <= 1;
33   else deploy_1[4] <= 0;
34   if (deploy_1[3:2] == 2'b11)
35     deploy_1[5] <= 1;
36   else deploy_1[5] <= 0;
37   if (deploy_1[5:4] == 2'b11)
38     deploy_1[6] <= 1;
39   else deploy_1[6] <= 0;
40 end
41
42 // Repeat lines 16−−40, but with count_2 and deploy_2
43
44 // Hide trigger activation signals (deploy_1 and deploy_2)
45 // inside fan−in logic cone of three additional signals
46 // (h_1, h_2, and h_3) to evade VeriTrust. Note, holdn_prev
47 // and in.super_prev are values of holdn and in.super from
48 // previous clock cycles, added to maintain timing.
49 always @(posedge clk) begin
50   holdn <= holdn_prev;
51   in.super <= in.super_prev;
52   h_1 <= deploy_1[6];
53   h_2 <= ~deploy_2[6] & holdn_prev & in.super_prev |
           deploy_2[6];
54   h_3 <= (~deploy_1[6] | deploy_2[6]) & (holdn_prev &
           in.super_prev);
55 end
56
57 always @(posedge clk) begin
58   super <= ~resetn | (~holdn & super) | (h_1 & h_2) | h_3;
59 end
```

Stealth Characteristics. We systematically construct our TTT (shown in List. 2) with several characteristics that enable it to evade all existing Trojan defenses except Bomberman. First, armed with Sturton *et al.*'s insights [14], we deploy a distributed SSC architecture to evade detection by UCI. Distributed SSCs enable the TTT's **activation signals** to bypass UCI since each coalesced SSC sub-component—$count_1$ and $count_2$—can express their individual triggered states during verification testing–defined by the $`DEPLOY_X$ constants—while the overall distributed SSC does *not* express its triggered state. Next, we **increment our TTT's SSCs non-uniformly**, to evade WordRev [35], [36] and power resets [13]. Lastly, we deploy *DeTrust* transformations [15] on the Trojan's: 1) **comparator inputs** ($count_1$ and $count_2$)—splitting them amongst several layers of sequential logic—and 2) trigger **activation signals** ($deploy_1[6]$ and $deploy_2[6]$)—hiding them inside a logic cone of three additional signals: h_1, h_2, and h_3. This hides our TTT from FANCI [8] and VeriTrust [9], respectively. Since Bomberman: 1) is TTT-specific, 2) considers distributed SSC architectures, and 3) is agnostic of how or when SSCs are incremented, it is the *only* defense that can detect this TTT.

D. Run Time and Complexity Analysis

Since Bomberman is a dynamic verification framework, its run time is roughly proportional to the size of the DUT (number of SSCs and wires, see Fig. 7) and simulation time (number of time steps). ***Across all designs we study, the run time of Bomberman did not exceed 11 minutes on a commodity laptop.*** Compared with other Trojan verification frameworks [8]–[10], [35], [36], Bomberman is two orders of magnitude faster when analyzing the same circuits; this is due, in part, to Bomberman's targeted nature. As we show in Tab. II, Bomberman's run time on real-world hardware designs scales proportionally with respect to the number of SSCs and number simulation test cases.

TABLE II
BOMBERMAN SCALABILITY COMPARISON FOR CIRCUIT DFGS WITH n SIGNALS SIMULATED OVER c CLOCK CYCLES.

Framework	Analysis Type	Time Complexity	Space Complexity	Average Run Time
Bomberman	Dynamic	$\mathcal{O}(nc)$	$\mathcal{O}(nc)$	1x Minutes
FANCI [8]	Static	$\mathcal{O}(n)$	$\mathcal{O}(n)$	10x Hours
UCI [10]	Dynamic	$\mathcal{O}(n^2c)$	$\mathcal{O}(nc)$	1x Hours
VeriTrust [9]	Dynamic	$\mathcal{O}(n2^n)$	$\mathcal{O}(nc)$	10x Hours
WordRev [35]	Static	Not Reported	Not Reported	1x Hours

The Bomberman framework consists of two main components that contribute to its overall time and space complexities (Fig. 5): 1) *SSC Enumeration*, and 2) *SSC Classification*.[3] Below, we provide an in-depth complexity analysis for each stage, and Bomberman as a whole.

[3] In our experiments, we did not observe the *DFG Generation* stage to be computationally dominant.

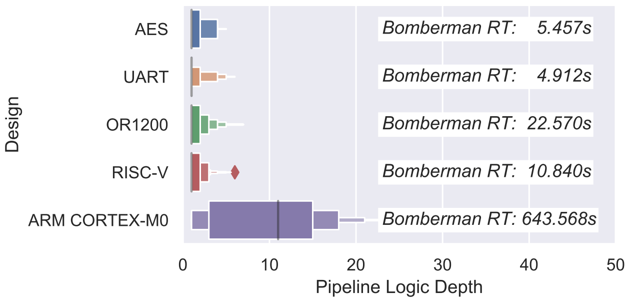

Fig. 9. **Distributions of Logic Depths per Pipeline Stage.** The length of combinational logic chains between any two sequential components in most hardware designs is ***bounded*** to optimize for performance, power, and/or area. High performance designs have the shortest depths (less than 8 [53]), while even the ***flattened and obfuscated*** logic model of the lowest-performance Arm processor available [52] (worst case scenario) has a depth <25. Even in the worst case, Bomberman's run time (overlaid for each core), is <11 min. on a commodity laptop.

*1) **SSC Enumeration**:* During the SSC Enumeration stage, Bomberman locates signals that are the direct outputs of *coalesced* SSCs, and signals that form *distributed* SSCs (§V-A). For a circuit DFG with n nodes (each node representing a signal), a maximum fan-in of f for signal nodes, a maximum logic depth per pipeline stage[4] of d, the asymptotic time complexity for enumerating SSCs is $\mathcal{O}(nf^d)$. Since most hardware designs are optimized for either power, performance (clock speed), and/or area, the maximum logic depth, d, is usually small and bounded. Therefore, the time complexity is *polynomial*. To show this, we plot (Fig. 9) the distributions of logic depths within pipeline stages—and the corresponding Bomberman run time—across the four designs we study, representing both mid-to-high performance and mid-to-large designs. Additionally, ***to stress-test Bomberman, we measure its run time in the worst-case scenario: analyzing the flattened and obfuscated functionally-equivalent logic model of the most low-performant and low-power Arm processor available [52]***. For all designs, the logic depths were less than 25 across all pipeline stages.[5] Additionally, the maximum fan-in for a signal node is often small—less than 10—and bounded [8], further reducing the time complexity to $\mathcal{O}(n)$. By extension, the asymptotic space complexity reduces from $\mathcal{O}(n + nf)$ to $\mathcal{O}(n)$, to store the DFG.

While Bomberman's SSC Enumeration time complexity is bounded by conventional circuit size and performance constraints, from a security perspective it is important to understand how an attacker might manipulate these bounds. Fortunately, while an attacker *can* control the maximum logic depth in a pipeline stage, d, *and* the maximum fan-in of a signal node, f, choosing large values for either in hopes of rendering Bomberman analyses computationally infeasible would reveal them: the victim design would be rendered unusable—either too large or too slow—by its intended customers and the tools would direct the designer to inspect the Trojan logic.

*2) **SSC Classification**:* In the SSC Classification stage, Bomberman analyzes verification simulation traces to determine if an SSC is suspicious—potentially part of a TTT (Algo. 1). For a circuit DFG with n nodes (each node representing a signal), and c simulation clock cycles, the asymptotic time and space complexities are both $\mathcal{O}(nc)$. This accounts for tracking the values expressed by each SSC over each simulation clock cycle. Since the time and space complexities of the SSC Classification stage dominate, compared with the *SSC Enumeration* stage, they represent the time and space complexities for the entire Bomberman framework.

VII. Discussion

*1) **Test Vector Selection**:* During the AES and UART false positive evaluations, we witnessed a plateauing reduction in false positives after executing initial verification tests (Figs. 8A–B). Upon a closer look, we find this *initial* reduction is a result of test vectors exhaustively exercising *small* registers—1- to 16-bit—violating *Property 2* in §IV-A. For *large* registers—32-bit and larger—cycling through all register values is not computationally feasible. Thus, to quickly reduce the number of false positives across both designs, we deploy a **repeat testing strategy** (§VI-B). For most circuit designs, we observe: ***the state of most benign SSCs is a function of design inputs***. By repeating tests, we induce benign SSCs to repeat a value, violating Property 1 (§IV-A).

How do we know which test cases to repeat in order to induce repeated values in benign SSCs? For designs with unstructured, **data-path-only inputs**—like the AES design—repeating any test vector will suffice. Alternatively, for designs that require structured **control-path inputs**, inducing repeated SSC values requires activating the same control-path multiple times while also repeating data-path inputs. Determining which control-paths to activate, i.e., control-paths that influence specific SSCs, is tantamount to crafting test vectors with high SSC coverage. Fortunately, Bomberman provides verification engineers with two channels of information to aid in this process: 1) the circuit DFG (Fig. 11 in Appendix C) illustrates the control-path that exercises a specific SSC, and 2) the SSC Classification output indicates the extent suspicious SSCs have/have-not been exercised. Together, these Bomberman insights guide verification engineers in creating test vectors that achieve high coverage, with respect to Bomberman *invariants* (Properties 1 and 2 in §IV-A), therefore minimizing false positives. For example, in §VI-B, when analyzing the OR1200 processor, we noticed designer-provided test vectors [49] did not exercise several CSRs. By referencing Bomberman's output, we located the (non-)suspicious SSCs and crafted test vectors to exercise them.

*2) **Latches**:* For Bomberman to locate TTTs in a hardware design, it first locates all SSCs by identifying signals in the design's HDL that are inferred as flip-flops during synthesis (§V-A). However, flip-flops are not the only circuit components that store state. SSCs can also be implemented with latches. However, it is typically considered bad practice to include latches in sequential hardware designs as they often

[4] The logic depth in a pipeline stage is the number of stages of combinational logic between layers of sequential logic.

[5] If we could plot the logic depths within commercial x86 processors in Fig. 9, we would expect them to be smaller than the OR1200, RISC-V, and Arm designs, as the maximum depth of logic per pipeline stage of *GHz* processors must be less than eight [53].

induce unwanted timing errors. As a result, HDL compilers in synthesis CAD tools issue warnings when they infer latches in a design—highlighting the TTT. Nonetheless, to support such (bad) design practices, we design Bomberman's data-flow graph generation compiler back-end to also recognize latches.

3) ***TTT Identification in Physical Layouts****:* Bomberman is designed as an extension into existing front-end verification tool-chains that process hardware designs (Fig. 2). Under a different threat model—one encapsulating untrusted back-end designers—it may be necessary to analyze physical layouts for the presence of TTTs. Bomberman can analyze physical layouts for TTTs, provided the layout (GDSII) file is first reverse-engineered into a gate-level netlist. As noted by Yang *et al.* [28], there are several reverse-engineering tools for carrying out this task. Bomberman also requires HDL device models for all devices in the netlist (e.g., NAND gate). This informs Bomberman of a device's input and output signals, which is required to create a DFG. Fortunately, HDL device models are typically provided as a part of the process technology IP portfolio purchased by front-end designers.

4) ***Memories****:* Bomberman is designed to handle memories, or large arrays of SSCs, in the same fashion that it handles flip-flop-based SSCs. Namely, Bomberman creates a DFG of the addressable words within a memory block to curb state-explosion when locating distributed SSCs. For memories that mandate word-aligned accesses, Bomberman generates a coalesced SSC for every word. For memories that allow unaligned accesses—which represent a minority, i.e., part of two adjacent words could be addressed simultaneously, Bomberman generates a coalesced SSC for every word, and multiple word-sized distributed SSCs created by sliding a word-sized window across every adjacent memory word pair. In either case, Bomberman's DFG filtering mechanism greatly reduces the overall set of potentially suspicious SSCs.

5) ***Limitations****:* Bomberman is capable of detecting all TTTs with zero false negatives, within the constraints of our definition (§IV-A). However, these constraints impose limitations. First, if an attacker knows Bomberman is in use, they may alter their Trojan to repeat a value to avoid detection. There are two ways they may do this: 1) add an extra state bit to the SSC(s) that does not repeat a value, or 2) add additional logic that resets the SSC(s) upon recognizing specific circuit behavior. The first design would be detected by Bomberman since, by definition, describes a *distributed* SSC. However, the second scenario describes a Trojan that, by definition, is a data-based (cheat code) Trojan [13] not a TTT. Therefore, it would not be detected by Bomberman. Data-based Trojans [13] are better addressed by techniques that target rarely used *activation signals* [9], [10] or *comparator inputs* [8] (Tab. I). Second, Bomberman is incapable of detecting TTTs that use analog SSCs, like the A2 Trojan [28], as there is no notion of analog SSCs in front-end designs.[6] Detecting Trojans like A2 require knowledge of the physical layout of the circuit, and are best addressed during circuit layout [54].

VIII. Related Work

The implantation, detection, and prevention of hardware Trojans across hardware design phases have been widely studied. Attacks range from design-time attacks [7], [22], [27], [55], to layout-level modifications at fabrication time [28]–[30]. On the defensive side, most work focuses on post-fabrication Trojan detection [37]–[43], [54], [56], [57], given that most hardware design houses are fab-less, and therefore must outsource their designs for fabrication. However, as hardware complexity increases, reliance on 3rd-party IP [3] brings the trustworthiness of the design process into question. Thus, there is active work in both detection [8]–[10], [35], [36] and preventation [11], [13] of design-time Trojans.

On the attack side, King *et al.* [7] demonstrate embedding hardware Trojans in a processor for the purpose of planting footholds for high-level exploitation in software. They demonstrate how small perturbations in a microprocessor's hardware can be exploited to mount wide varieties of software-level attacks. Lin *et al.* [27] propose a different class of hardware Trojans, designed to expose a side-channel for leaking information. Specifically, they add flip-flops to an AES core to create a power side channel large enough to exfiltrate key bytes, but small enough that it resides below the device's power noise margin. While both attacks demonstrate different payloads, they both require triggering mechanisms to remain dormant during verification and post-fabrication testing. Thankfully, our defense is payload-agnostic and trigger-specific. We focus on detecting hardware Trojans by their *trigger*. As a byproduct, we can identify any payloads by inspecting portions of the design that the trigger output influences.

Wang *et al.* [21] propose the first variant of sporadic TTTs, called *Asynchronous Counter Trojans*. *Asynchronous Counter Trojans* increment pseudo-randomly from a non-periodic internal event signal (e.g., Fig. 4C and D). Similarly, Imeson *et al.* [22] propose non-deterministic TTTs. Non-deterministic TTTs are also sporadic, but they differ from pseudo-random TTTs in that their event signals are *not* a function of the state of the victim device, rather, they are a function of a true source of entropy. Unlike, Waksman *et al.*'s power reset defense [13], this nuance is irrelevant to Bomberman, who identifies TTTs by the values expressed by their SSCs, not the source or predictability of their event signals.

On the defensive side, both design- and run-time approaches have been proposed. At design-time, Hicks *et al.* [10] propose a dynamic analysis technique for *Unused Circuit Identification* (UCI) to locate potential trigger logic. After verification testing, they replace all unused logic with logic to raise exceptions at run-time to be handled in software. Similarly, Zhang *et al.* [9] propose *VeriTrust*, a dynamic analysis technique focused on the behavioral functionality, rather than implementation, of the hardware. Conversely, Waksman *et al.* [8] propose *FANCI*, a static analysis technique for locating *rarely used logic* based on computing *control values* between

[6] While the non-deterministic (sporadic) TTTs proposed by Imeson *et al.* [22] do use non-simulatable analog behavior (i.e., phase noise) as an entropy source for the increment event, they do not use analog SSCs. Thus, they **are detectable by Bomberman**.

inputs and outputs. Lastly, Li and Subramanyan *et al.* [35], [36] propose WordRev, a different static analysis approach, whereby they search for counters in a gate-level netlist by identifying groups of latches that toggle when low order bits are 1 (up-counter), or low order bits are 0 (down-counter). As static analysis approaches, FANCI and WordRev have the advantage of not requiring verification simulation results. In §VI-C2 we leverage prior work on defeating such defenses [14]–[16] to construct a TTT that bypasses these defenses—but Bomberman detects. At run-time, Waksman *et al.* [13] thwart TTTs, using intermittent power resets. As shown in §VI-C1, power-resets are also incapable of thwarting all TTT variants.

IX. Conclusion

Bomberman is an effective example of a threat-specific defense against TTTs. Unlike prior work, we do not attempt to provide a panacea against *all* design-time Trojans. Instead, we define the behavioral characteristics of a specific but important threat, TTTs, and develop a complete defense capable of identifying *all* TTT variants as we define them. Across four open-source hardware designs, Bomberman detects all six TTT variants, with less than 1.2% false positives.

Bomberman demonstrates the power of threat-specific verification, and seeks to inspire future threat-specific defenses against hardware Trojans and common hardware bugs. We believe that no one defense will ever provide the level of security achievable by defense-in-depth strategies. Thus, by combining Bomberman with existing design-time Trojan defenses [8]–[10], [13], along with future threat-specific defenses, we aim to create an insurmountable barrier for design-time attackers.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Stefan Katzenbeisser, for their thoughtful feedback that enhanced the quality of this paper.

DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited. This material is based upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Additionally, the work reported in this paper was supported in part by the US National Science Foundation under Grant CNS-1646130 and Graduate Research Fellowship Program under Grant DGE 1256260, as well as the US Army Research Office under Grant W911NF-21-1-0057. Any opinions, findings, conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] M. Lapedus, "10nm versus 7nm," April 2016, https://semiengineering.com/10nm-versus-7nm/.

[2] P. Gupta, "7nm power issues and solutions," November 2016, https://semiengineering.com/7nm-power-issues-and-solutions/.

[3] J. Blyler, "Trends driving ip reuse through 2020," November 2017, http://jbsystech.com/trends-driving-ip-reuse-2020/.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[5] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[6] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *USENIX Security Symposium*, 2018.

[7] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[8] A. Waksman, M. Suozzo, and S. Sethumadhavan, "FANCI: identification of stealthy malicious logic using boolean functional analysis," in *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[9] J. Zhang, F. Yuan, L. Wei, Y. Liu, and Q. Xu, "VeriTrust: Verification for hardware trust," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[10] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[11] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[12] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[13] A. Waksman and S. Sethumadhavan, "Silencing hardware backdoors," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[14] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating UCI: Building stealthy and malicious hardware," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[15] J. Zhang, F. Yuan, and Q. Xu, "DeTrust: Defeating hardware trust verification with stealthy implicitly-triggered hardware trojans," in *ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2014.

[16] A. Waksman, J. Rajendran, M. Suozzo, and S. Sethumadhavan, "A red team/blue team assessment of functional analysis methods for malicious circuit identification," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2014.

[17] V. Patankar, A. Jain, and R. Bryant, "Formal verification of an ARM processor," in *International Conference on VLSI Design (VLSID)*, 1999.

[18] C. Wolf, "Picorv32," https://github.com/cliffordwolf/picorv3#cycles-per-instruction-performance.

[19] OpenCores.org, "Openrisc or1200 processor," https://github.com/openrisc/or1200.

[20] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *IEEE International Conference on Computer Design (ICCD)*, 2013.

[21] X. Wang, S. Narasimhan, A. Krishna, T. Mal-Sarkar, and S. Bhunia, "Sequential hardware trojan: Side-channel aware design and placement," in *IEEE International Conference on Computer Design (ICCD)*, 2011.

[22] F. Imeson, S. Nejati, S. Garg, and M. Tripunitara, "Non-deterministic timers for hardware trojan activation (or how a little randomness can go the wrong way)," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2016.

[23] T. Trippel, "Bomberman," December 2020, https://github.com/timothytrippel/bomberman.

[24] J. Cross, "Inside apple's A13 bionic system-on-chip," October 2019, https://www.macworld.com/article/3442716/inside-apples-a13-bionic-system-on-chip.html.

[25] R. Karri, J. Rajendran, K. Rosenfeld, and M. Tehranipoor, "Trustworthy hardware: Identifying and classifying hardware trojans," *Computer*, 2010.

[26] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, 2010.

[27] L. Lin, M. Kasper, T. Güneysu, C. Paar, and W. Burleson, "Trojan side-channels: Lightweight hardware trojans through side-channel engineering." in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2009.

[28] K. Yang, M. Hicks, Q. Dong, T. Austin, and D. Sylvester, "A2: Analog malicious hardware," in *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[29] R. Kumar, P. Jovanovic, W. Burleson, and I. Polian, "Parametric trojans for fault-injection attacks on cryptographic hardware," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2014.

[30] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson, "Stealthy dopant-level hardware trojans," in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2013.

[31] S. Ghosh, A. Basak, and S. Bhunia, "How secure are printed circuit boards against trojan attacks?" *IEEE Design & Test*, 2014.

[32] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *IEEE International High Level Design Validation and Test Workshop (HLDVT)*. IEEE, 2009.

[33] Y. Jin and Y. Makris, "Hardware trojan detection using path delay fingerprint," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[34] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty, "Towards trojan-free trusted ics: Problem analysis and detection scheme," in *ACM Conference on Design, Automation and Test in Europe (DATE)*, 2008.

[35] W. Li, A. Gascon, P. Subramanyan, W. Y. Tan, A. Tiwari, S. Malik, N. Shankar, and S. Seshia, "WordRev: Finding word-level structures in a sea of bit-level gates," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2013.

[36] P. Subramanyan, N. Tsiskaridze, K. Pasricha, D. Reisman, A. Susnea, and S. Malik, "Reverse engineering digital circuits using functional analysis," in *ACM Conference on Design, Automation and Test in Europe (DATE)*, 2013.

[37] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *IEEE Symposium on Security and Privacy (S&P)*, 2007.

[38] M. Potkonjak, A. Nahapetian, M. Nelson, and T. Massey, "Hardware trojan horse detection using gate-level characterization," in *ACM/IEEE Design Automation Conference (DAC)*, 2009.

[39] S. Narasimhan, X. Wang, D. Du, R. S. Chakraborty, and S. Bhunia, "Tesr: A robust temporal self-referencing approach for hardware trojan detection," in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 2011.

[40] J. Balasch, B. Gierlichs, and I. Verbauwhede, "Electromagnetic circuit fingerprints for hardware trojan detection," in *IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015.

[41] J. Li and J. Lach, "At-speed delay characterization for ic authentication and trojan horse detection," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[42] D. Forte, C. Bao, and A. Srivastava, "Temperature tracking: An innovative run-time approach for hardware trojan detection," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013.

[43] S. Kelly, X. Zhang, M. Tehranipoor, and A. Ferraiuolo, "Detecting hardware trojans using on-chip sensors in an asic design," *Journal of Electronic Testing*, 2015.

[44] Cadence Design Systems, "JasperGold," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[45] Synopsys, "VC Formal," https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html.

[46] S. Williams, "Icarus verilog," http://iverilog.icarus.com/.

[47] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin ii-an open-source verilog hdl synthesis tool for cad research," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2010.

[48] C. H. Kingsley and B. K. Sharma, "Method and apparatus for identifying flip-flops in hdl descriptions of circuits without specific templates," 1998, US Patent 5,854,926.

[49] OpenCores.org, "Openrisc or1k tests," https://github.com/openrisc/or1k-tests/tree/master/native/or1200.

[50] U. of California, "Risc-v gnu compiler toolchain," https://github.com/riscv/riscv-gnu-toolchain.

[51] OpenCores.org, "Or1k-elf toolchain," https://openrisc.io/newlib/.

[52] Arm, "Arm Cortex-M0," https://developer.arm.com/ip-products/processors/cortex-m/cortex-m0.

[53] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar, "The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays," in *IEEE International Symposium on Computer Architecture (ISCA)*, 2002.

[54] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks, "ICAS: an extensible framework for estimating the susceptibility of ic layouts to additive trojans," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[55] E. Biham, Y. Carmeli, and A. Shamir, "Bug attacks," in *Annual International Cryptology Conference*, 2008.

[56] M. Banga and M. S. Hsiao, "A region based approach for the identification of hardware trojans," in *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST)*, 2008.

[57] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao, "Guided test generation for isolation and detection of embedded trojans in ics," in *ACM Great Lakes Symposium on VLSI (GLSVLSI)*, 2008.

[58] Google LLC, "RISCV-DV," https://github.com/google/riscv-dv.

Appendix

A. Ticking Timebomb Trigger Variants

In these Verilog examples of TTT triggers, we use a three letter naming convention to describe their building blocks: SSC type (C or D), increment value (U or N), and increment event (P or S). For example, a CNS TTT indicates a *Coalesced* (C) SSC, with a *Non-uniform* (N) increment value, and a *Sporadic* (S) increment event. For TTTs comprised of *distributed* SSCs we use the "D-<type>" naming convention to indicate the type: homogeneous or heterogeneous. This list is not comprehensive, but rather a representative sampling of the TTT design space. Note, all examples assume a processor victim circuit, with a *pageFault* flag, *overflow* flag, and a 32-bit *program counter* (PC) register.

```
// 1. CUP = Coalesced SSC, Uniform increment, Periodic event
reg [31:0] ssc;
always @posedge(clock) begin
    if ( reset )
        ssc <= 0;
    else
        ssc <= ssc + 1;
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 2. CUS = Coalesced SSC, Uniform increment, Sporadic event
reg [31:0] ssc;
always @posedge(pageFault) begin
    if ( reset )
        ssc <= 0;
    else
        ssc <= ssc + 1;
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 3. CNP = Coalesced SSC, Non−uniform increment, Periodic event
reg [31:0] ssc;
always @posedge(clock) begin
    if ( reset )
        ssc <= 1;
    else
        ssc <= ssc << PC[3:2];
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 4. CNS = Coalesced SSC, Non−uniform increment, Sporadic event
reg [31:0] ssc;
always @posedge(pageFault) begin
    if ( reset )
        ssc <= 0;
    else
        ssc <= ssc + PC[3:0];
end
assign doAttack = (ssc == 32'hDEAD_BEEF);
```

```
// 5. D−Homogeneous = Distributed SSC, same sub−components
wire [31:0] ssc_wire;
reg  [15:0] lower_half_ssc;
reg  [15:0] upper_half_ssc;
assign ssc_wire = {upper_half_ssc, lower_half_ssc};

// Two CUP sub−counters
always @posedge(clock) begin
    if(reset) begin
        lower_half_ssc <= 0;
        upper_half_ssc <= 0;
    end
    else begin
        lower_half_ssc <= lower_half_ssc + 1;
        upper_half_ssc <= upper_half_ssc + 1;
    end
end
assign doAttack = (ssc_wire == 32'hDEAD_BEEF);
```

```
// 6. D−Heterogeneous = Distributed SSC, different
    sub−components
wire [31:0] ssc_wire;
reg  [15:0] lower_half_ssc;
reg  [15:0] upper_half_ssc;
assign ssc_wire = {upper_half_ssc, lower_half_ssc};

// CUS sub−counter
always @posedge(pageFault) begin
    if(reset)
        lower_half_ssc <= 0;
    else
        lower_half_ssc <= lower_half_ssc + 1;
end

// CNP sub−counter
always @posedge(clock) begin
    if(reset)
        upper_half_ssc <= 0;
    else
        upper_half_ssc <= upper_half_ssc + PC[3:0];
end
assign doAttack = (ssc_wire == 32'hDEAD_BEEF);
```

B. Constrained Randomized Verification

Given the size and complexity of modern hardware designs, verification engineers typically use randomly-generated test vectors to maximize verification coverage. Similarly, for two of the four designs we study (AES and UART), we use LFSRs to generate random **data-path** inputs for test vectors. Thus, we ask the question: *does contrained random verification degrade Bomberman's performance?* To demonstrate Bomberman is test-case agnostic, we generate 25 random test sequences for both the AES and UART designs by randomly seeding the LFSR(s) in each design's respective test bench (Fig. 6). Note, we do not experiment with constrained random verification of the RISC-V and OR1200 designs as these require random instruction stream generators, for which (to the best of our knowledge) none exist in the open-source community that are compatible with open-source RTL simulators like IVL or Verilator.[7]

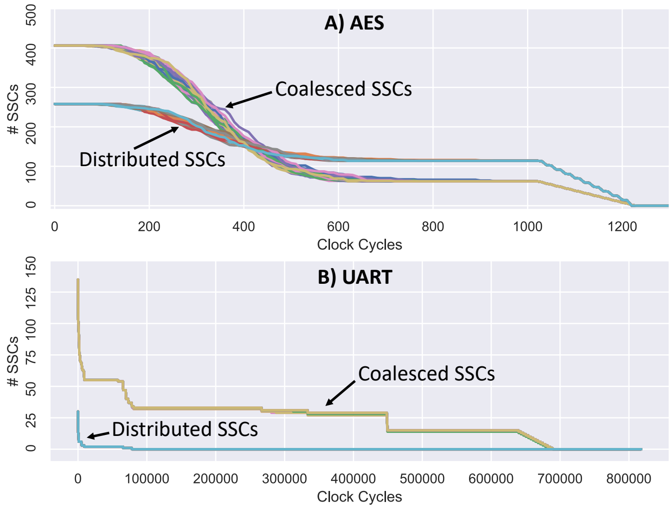

Fig. 10. **Randomized Testing.** Randomly generated verification test vectors do not affect Bomberman's performance. Rather, Bomberman's performance is dependent on verification coverage with respect to SSC Properties 1 & 2 (§IV-A) that define the behavior of a TTT. Namely, tests that cause more SSCs to cycle through all possible values, or repeat a value, reduce false positives.

For the AES design, we generate 25 random sequences of seventy-five 128-bit keys and plaintexts. For the UART design, we generate 25 random sequences of 16 bytes (to TX/RX). Similar to the false positive experiments (§VI-B), each test sequence for each design was repeated twice, without a system reset in between. Given Bomberman's inability to produce false negatives, we only study the effects of randomness on Bomberman's false positive rate. Thus, unlike the false positive experiments, no TTT variants were implanted in either design. In Fig. 10, we plot the suspicious SSC traces produced by Bomberman across all randomly generated test vectors. Across both designs, zero suspicious SSCs (false positives) are observed at the end of all 25 simulations, and each simulation trace is nearly identical. Thus, Bomberman's performance is not test-case specific, rather, it is dependent on verification *coverage*, with respect to TTT invariants,[8] i.e. Properties 1 & 2 in §IV-A.

C. Example Hardware Data Flow Graph

An example hardware DFG generated by Bomberman's *SSC Identification* phase is shown below in Fig. 11.

[7] Google's RISCV-DV open-source random instruction stream generator is not compatible with either IVL or Verilator [58].

[8] Verification coverage with respect to TTT invariants, is **not** to be confused with generic verification coverage such as functional, statement, condition, toggle, branch, and Finite State Machine (FSM) coverage. The former entails exercising SSCs such that they violate TTT invariants (Properties 1–2 in §IV-A).

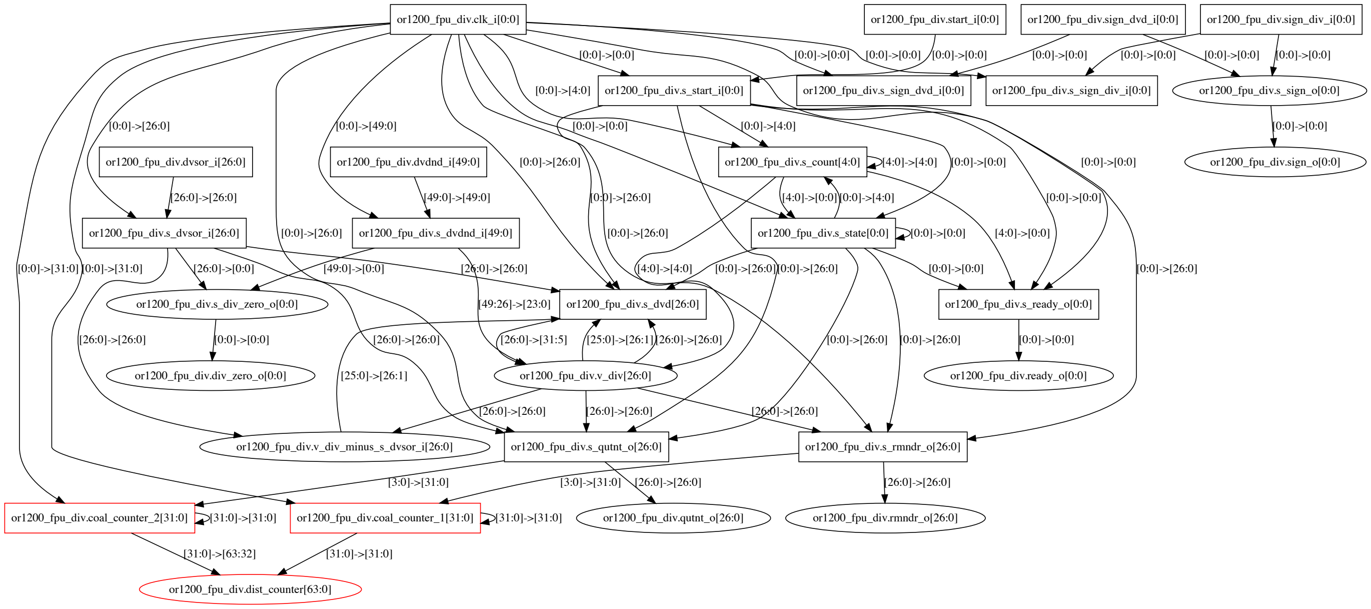
or1200_fpu_div.clk_i[0:0]
or1200_fpu_div.start_i[0:0]
or1200_fpu_div.sign_dvd_i[0:0]
or1200_fpu_div.sign_div_i[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[0:0]
[0:0]->[4:0]
or1200_fpu_div.s_start_i[0:0]
or1200_fpu_div.s_sign_dvd_i[0:0]
or1200_fpu_div.s_sign_div_i[0:0]
or1200_fpu_div.s_sign_o[0:0]
[0:0]->[26:0]
[0:0]->[49:0]
[0:0]->[0:0]
[0:0]->[4:0]
[0:0]->[0:0]
or1200_fpu_div.dvsor_i[26:0]
or1200_fpu_div.dvdnd_i[49:0]
[0:0]->[26:0]
or1200_fpu_div.s_count[4:0]
[4:0]->[4:0]
[0:0]->[0:0]
[0:0]->[0:0]
or1200_fpu_div.sign_o[0:0]
[26:0]->[26:0]
[49:0]->[49:0]
[0:0]->[26:0]
[4:0]->[0:0]
[0:0]->[4:0]
[0:0]->[0:0]
or1200_fpu_div.s_dvsor_i[26:0]
[0:0]->[26:0]
or1200_fpu_div.s_dvdnd_i[49:0]
[0:0]->[26:0]
or1200_fpu_div.s_state[0:0]
[0:0]->[0:0]
[4:0]->[0:0]
[0:0]->[31:0]
[0:0]->[31:0]
[26:0]->[0:0]
[49:0]->[0:0]
[26:0]->[26:0]
[4:0]->[4:0]
[0:0]->[26:0]
[0:0]->[26:0]
[0:0]->[0:0]
[0:0]->[26:0]
or1200_fpu_div.s_div_zero_o[0:0]
[49:26]->[23:0]
or1200_fpu_div.s_dvd[26:0]
or1200_fpu_div.s_ready_o[0:0]
[26:0]->[26:0]
[0:0]->[0:0]
[26:0]->[26:0]
[26:0]->[31:5]
[25:0]->[26:1]
[26:0]->[26:0]
[0:0]->[26:0]
[0:0]->[26:0]
[0:0]->[0:0]
or1200_fpu_div.div_zero_o[0:0]
[25:0]->[26:1]
or1200_fpu_div.v_div[26:0]
or1200_fpu_div.ready_o[0:0]
[26:0]->[26:0]
[26:0]->[26:0]
[26:0]->[26:0]
or1200_fpu_div.v_div_minus_s_dvsor_i[26:0]
or1200_fpu_div.s_qutnt_o[26:0]
or1200_fpu_div.s_rmndr_o[26:0]
[3:0]->[31:0]
[3:0]->[31:0]
[26:0]->[26:0]
[26:0]->[26:0]
or1200_fpu_div.coal_counter_2[31:0]
[31:0]->[31:0]
or1200_fpu_div.coal_counter_1[31:0]
[31:0]->[31:0]
or1200_fpu_div.qutnt_o[26:0]
or1200_fpu_div.rmndr_o[26:0]
[31:0]->[63:32]
[31:0]->[31:0]
or1200_fpu_div.dist_counter[63:0]

Fig. 11. **Hardware Data-Flow Graph.** Example data-flow graph, generated by Bomberman, of an open-source floating-point division unit [19]. Bomberman cross-references this graph with verification simulation results to identify SSCs (red). In the graph, rectangles represent registers, or flip-flops, and ellipses represent intermediate signals, i.e., outputs from combinational logic. Red rectangles indicate *coalesced* SSCs, while red ellipses represent *distributed* SSCs.