Bomberman: Defining and Defeating Hardware Ticking Timebombs

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

Kevin B. Bush
Cyber Systems & Operations
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, semiconductor designers increase both the size of their design teams and their reliance on third-party intellectual property. Both come at the expense of trust: it is computationally infeasible to exhaustively verify 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 general-purpose "patching" mechanism for hardware. Additionally, malicious hardware modifications serve as a foothold for subverting all layers of software that sit 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 expose weaknesses in these "one-size-fits-all", heuristic-based approaches. Instead of attempting to detect *all* possible hardware Trojans, we address the hardware Trojan threat in a divide-and-conquer fashion: formalizing and eliminating Ticking Timebomb Trojans (TTTs), while relying on future defenses to address other Trojan classes. The goal is to systematically constrict the attacker's design space.

The heart of our approach is an abstract formal definition of TTTs based on their functional behavior. We translate this definition into the fundamental components required to realize TTT behavior in hardware. Using these components, we expand the set of known TTTs to a total of six variants, including a novel class of *distributed* TTTs. On the defense side, we leverage our formal definition to design and implement a TTT-specific verification toolchain extension, called *Bomberman*. Across three real-world hardware designs, we demonstrate how verification engineers use Bomberman as both a verification-enhancing coverage metric and as security-analysis tool that detects *all* implanted TTTs with less than 0.3% false positives.

***Index Terms*—Hardware Trojans, Ticking Timebomb Triggers, 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 $7nm$ process requires 500 engineering years [1], [2]. Because it is impractical to take 550 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 and 3DES). By 2020, analysts estimate that a typical System-on-a-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 designers (both in-house and external) molding the design, there is an increased opportunity for an attacker to insert a hardware Trojan during its design.

*Work completed at MIT Lincoln Laboratory.

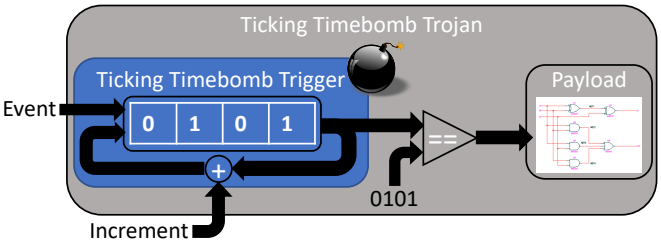

Fig. 1. **Ticking Timebomb Trojan.** A Ticking Timebomb Trojan (TTT) is a hardware Trojan that implements a ticking timebomb *trigger*. Ticking timebomb triggers monotonically move closer to activating as the system runs longer. In hardware, ticking timebomb triggers maintain a non-repeating sequence 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 of that hardware [7]. Thus, it is vital that hardware designers defend against hardware Trojans.

Prior work attempts to detect hardware Trojans at both design and run time. At design time, prior work proposes static (FANCI [8]) and dynamic (VeriTrust [9] and UCI [10]) analyses of the Hardware Description Language (HDL) 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 gaps that leave chips vulnerable to attack [14]–[16].

Verifying a hardware design is Trojan-free poses two main 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 "one-size-fits-all" approaches are akin to proving a design is bug-free—hence incomplete.

Instead of verifying a design is free of *all* Trojan classes, we advocate for a divide-and-conquer approach, breaking down the Trojan design space and systematically ruling out each Trojan class. Thus, we take the first step in moving towards a Trojan free design by tackling Ticking Timebomb Trojans (TTTs). As Waksman *et al.* [11], [13] point out, among hardware 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." Additionally, TTTs are extremely stealthy since "a [time-bounded] formal validation technique that verifies all possible input values cannot prove that a timebomb will never go off." Thus, TTTs are an important class of hardware Trojan, yet challenging to defend against.

To ensure our defense is systematic and avoids implicit assumptions based on existing TTTs, we start with a formal definition of an abstract TTT based on its behavior. We observe that at the heart of any TTT is a trigger that tracks the progression values that form some arbitrary sequence of values. The most simple concrete example being a down-counter that releases the attack payload when it reaches `0`. Thus, we define TTTs as an arbitrary sequence of values constrained by only two properties:

1) the sequence never repeats a value
2) the sequence is incomplete

Fig. 1, shows the fundamental hardware components required to implement such a sequence in hardware. It has three components:

1) State-Saving Components (SSCs)
2) an increment value
3) an increment event

To understand the power our formal definition gives to attackers, we use it to design novel TTTs. We define a total of six TTT variants, several new to the literature, including a composite TTT we call a *distributed* TTT. Distributed TTTs piece together SSCs across the design to reduce hardware overhead and increase stealth.

On the defensive front, we leverage our formal definition of TTTs to identify 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 identifying *only* the 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 from a design's HDL[1] 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 formal definition as *not-suspicious*. Bomberman reports any SSCs remaining after functional verification as *suspicious*; designer use this information to create a new test case for verificaton or manually inspect connected logic for malice.

We demonstrate the effectiveness of Bomberman by implanting all six TTT variants into three different open-source hardware designs: an OR1200 processor [18], a UART [18], and an AES accelerator [19]. Even with verification simulations lasting less than 1.5 million cycles[2], Bomberman detects the presence of *all* TTT variants across *all* circuit designs with a false positive rate of less than 0.3%.

This paper makes the following contributions:

- A formal abstract definition and component-level breakdown of TTTs.
- Design and implementation of six variants of TTTs, including probabilistic counters and a class of distributed TTTs that bypass existing defenses.
- Design of a formally directed approach to verifying a hardware design is free of TTTs.
- Implementation of an automated verification extension, Bomberman, for identifying potential TTTs implanted in a hardware design. Bomberman provides engineers with a threat-specific coverage metric to enable them to direct testing to make their designs more secure.
- We open-source Bomberman and submit our TTT implementations to an existing hardware Trojan repository [19].

II. Background

A. Hardware Design Process

In order to design complex ICs, like the Apple A12 Bionic chip that contains 6.9 billion transistors [20], the design process is broken down into several phases (Fig. 2) and heavily augmented with Computer-Aided Design (CAD) tools. To design complex ICs, while minimizing time-to-market, hardware designers often first purchase existing IP blocks from third parties to integrate into their designs. Next, designers integrate all third-party IP blocks and describe the behavior of any custom circuitry at the Register Transfer Level (RTL), using Hardware Description Languages (HDL) like VHDL or Verilog. Next, CAD tools synthesize 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

[1]Bomberman can analyze either pre- or post- synthesis HDL.

[2]Typical verification simulations run on the order of millions of cycles [11].

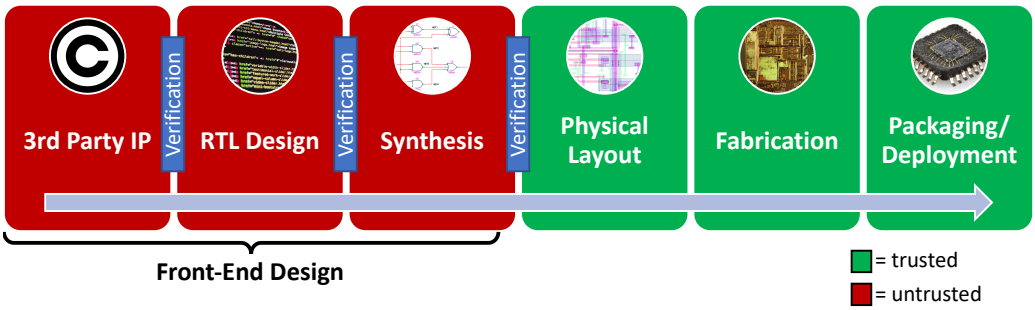

Fig. 2. **IC Design Process.** As ICs have gotten increasingly complex, the reuse of 3rd party IPs has increased, and design teams have gotten larger [3]. Like prior work on design-time attacks [8]–[11], [13], we assume all stages in the design process are trusted except the front-end design.

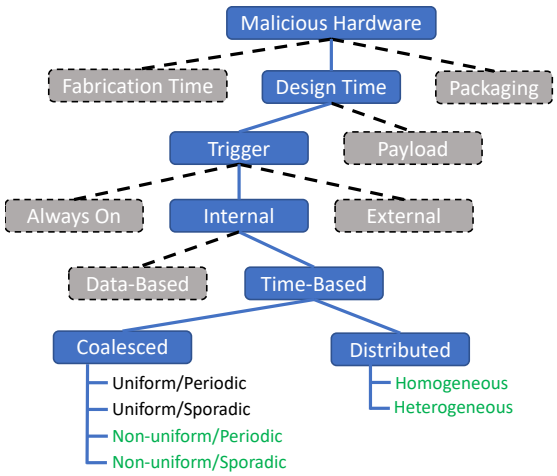

Fig. 3. **Taxonomy of Hardware Trojans.** Hardware Trojans are malicious modifications to a hardware design for the purpose of altering its intended functionality. Trojans can be inserted into a design anywhere in the design process, and can be characterized by their two main components: *trigger* and *payload*. We focus on TTTs, and categorize them according to their design and behavioral characteristics. In green are unexplored TTT variants.

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 the file format that is sent to the fabrication facility. Finally, the foundry fabricates the IC, and returns it to the designers who then test and package it for mounting onto a printed circuit board.

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 design process (Fig. 2) they are inserted, and 2) their architectures [21], [22]. Specifically, hardware Trojans can be inserted at design time [7], [11], [13], [23], at fabrication time [24]–[26], or during packaging/deployment [27], [28]. In this paper, we focus on design-time Trojans, specifically Trojans inserted during front-end design.

Hardware Trojans are comprised of two main components: a *trigger* and *payload* [29]–[31]. 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 three main types of triggers: *always-on*, *internal*, and *external*. As their name suggests, *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. *External* triggers activate when a signal external to the chip (i.e., an input) changes. They require an attacker to have knowledge of how the circuit will be deployed, and thus are not generalizable. Lastly, *internal* triggers activate when a signal within the design changes as a function of normal, yet rare, operation. They are both stealthy and generalizable. 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], thus we focus on internal triggers.

Internal triggers can further be broken into two categories: *data-based* and *time-based* [11], [13]. Date-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. The most naive ticking timebomb triggers resemble a simple periodic up-counter, where every clock cycle the counter increments, as shown in Fig. 4A. For the purposes of this work, we focus on ticking timebomb triggers as data-based triggers often require more logic to implement and require an attacker to interact with the design post-fabrication [11]. Our goal is to eliminate the general threat of TTTs, thus forcing attackers to implement larger, more disruptive trigger designs that existing [8]–[10], [13] and future defenses can detect.

III. Threat Model

In line with prior work [11], [13], [16], [32], [33], our threat model focuses on design-time attacks. More specifically, we focus on malicious modifications that are embedded in 3rd party IPs, in the RTL HDL, or in the gate-level netlist HDL (Fig. 2). Our focus, on design-time attacks inserted by hardware designers, is driven by current design trends and economic forces that favor reliance on untrusted 3rd parties and large design teams [3].

We assume that a design-time adversary has ability to add, remove, and modify the RTL or netlist HDL of the core design or IP block 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 by design-level functional 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], we assume that any malicious circuit behavior induced by Trojan trigger activation is caught via verification testing. Lastly, analog TTTs, like the A2 Trojan [24], are not in the scope of this work as there is no notion of analog components in front-end design.

The focus of this work is TTTs. We focus on identifying TTTs, as we define them (§IV), and leave the identification of other Trojan types to existing heuristics-based [8]–[10], and future design-specific defenses. 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 formally define TTTs by their behavior. From our 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 traditional TTTs that resemble simple time counters [11], [13], to more complex designs. Our abstract definition of TTTs enables our verification tool, Bomberman, to identify a broader range of TTTs than traditionally conceived.

A. Formal Definition

We define TTTs as the set of all hardware Trojans that implement a time-based trigger that monotonically approaches activation as the victim circuit continuously operates without reset (Fig. 3) [11], [13]. Concisely, over discrete trigger update times $t \in T = \{0, 1, \ldots, m\}$, an n-bit ticking timebomb trigger expresses values, v_t, such that:

$$\text{Given: } \theta(v) = \{v_t\}_{t \in T}, \text{where } v = [v_0, v_1, \ldots, v_m], \quad (1)$$

$$|\theta(v)| = |T| \text{ and} \quad (2)$$

$$\theta(v) \subset \{0, 1, \ldots, 2^n - 1\} \quad (3)$$

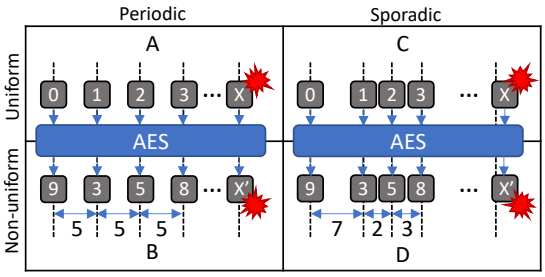

Fig. 4. **Ticking Timebomb Trigger Behaviors.** There are 4 primitive ticking timebomb trigger counting behaviors, in order of increasing complexity, captured by our formal definition in Eqs. (1)–(3). (A) The naive attacker may choose to implement a simplistic counting behavior that is both periodic and uniform. However, a more sophisticated attacker may choose to conceal the count by encrypting it, e.g., AES CTR mode, (B), or update the count sporadically (C), or both (D).

In other words, a ticking timebomb trigger exhibits two fundamental properties while still dormant yet monotonically approaching activation:

Property 1: The TTT does NOT repeat a value without a system reset (Eqs. (1) & (2)).

Property 2: The TTT does NOT enumerate all possible values without activating (Eq. (3)).

Property 1 holds by definition, since, if a TTT trigger repeats a value in the sequence, it is no longer a ticking timebomb, but rather a data-based "cheat code" trigger [11], [13]. Thus, in Eqs. (1) and (2), the size of the set of all distinct trigger values, $\theta(v)$, is equal to the size of the set of all trigger update times, $t \in T$. 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. Thus, in Eqs. (1) and (3), the set of all distinct trigger values, $\theta(v)$, is a proper subset of all possible values of an n-bit trigger value. It is upon these two properties that we derive the fundamental hardware building blocks of a TTT.

Figs. 4A–D illustrate example ticking timebomb behaviors that are captured by our formal 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., encrypting using AES CTR mode in Fig. 4B), or 2) *sporadically* incrementing the counter (Fig. 4C). Even more sophisticated, the attacker may choose to do both (Fig. 4D).

B. TTT Components

From our formal 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 component(s) (SSC)**

2) **increment value**
3) **increment event**

The *SSC* defines how the TTT saves and tracks the triggering state of the time counter, e.g., registers. The SSC 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 formal 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 can be either *periodic* or *sporadic*. For example, the rising edge of the clock is a periodic event, while the rising edge of the system status overflow bit 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 formal 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 sub-categories 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 sub-trigger types.

We summarize all six TTT variants and their behaviors in Figs. 3 and 4, respectively, and provide example implementations in Verilog code below. In these figures and code, we utilize a three letter naming convention for each *coalesced* TTT type, specified in the order of SSC type (C or D), increment value (U or N), and increment event (P or S). For example, a CNS ticking timebomb indicates a *Coalesced* (C) SSC, with a *Non-uniform* (N) increment value, and a *Sporadic* (S) increment event. For *distributed* TTTs we use the "D-<type>" naming convention to indicate the type, either homogeneous or heterogeneous. Due to space constraints, the list of TTT Verilog examples is *not* comprehensive, but rather a representative sampling of the TTT design space. Note that all implementation examples assume a processor victim circuit, with notions of signals such as a *pageFault* flag, *overflow* flag, and a 32-bit *program counter* (PC) register.

```
// 1. CUP = Coalesced SSC with Uniform increment and 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 with Uniform increment and 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 with Non-uniform increment and 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 with Non-uniform increment and 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 with Homogeneous increments
      and events
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 with Heterogeneous
      increments and events
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);
```

V. Bomberman

Next, we discuss the implementation of our verification extension, Bomberman. Bomberman takes as input a design's HDL and verification simulation results, and automatically flags suspicious TTTs in a design. Bomberman breaks the process of TTT identification into two phases:

1) **SSC Indentification**
2) **Simulation Analysis**

During the *SSC Indentification* phase, Bomberman enumerates all *coalesced* and *distributed* SSCs within the HDL. During the *Simulation Analysis* phase, Bomberman analyzes the value progressions of all SSCs to identify suspicious SSCs that may be components of a TTT. Bomberman starts by assuming all SSCs are suspicious, and eliminates candidates, according to the rule set in Section IV-A, as more simulation results are processed. Lastly, Bomberman flags any remaining SSCs as suspicious TTT components, and marks them for further analysis. The Bomberman architecture is shown in Fig. 5.

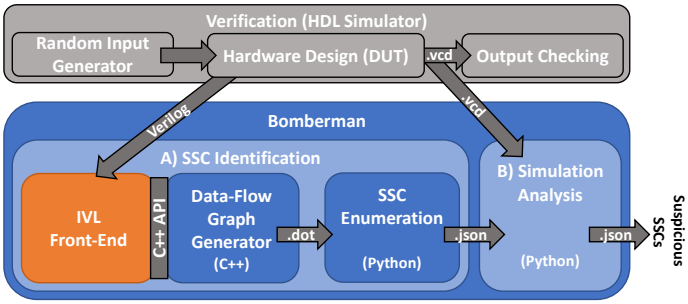

Fig. 5. **Bomberman Architecture.** Bomberman is comprised of two main stages: A) the *SSC Identification* stage, and B) the *Simulation Analysis* stage. The first stage (A) identifies all possible—coalesced and distributed—SSCs in the design. The second stage (B) starts by assuming all SSCs are *suspicious*, and marks SSCs as *non-suspicious* as it analyzes the values expressed by each SSC during verification simulations. An SSC is *non-suspicious* if it violates the formal definition (§IV-A) of a TTT trigger.

A. SSC Identification

The first step in identifying TTTs, is locating SSCs within the design. For coalesced SSCs, this is straightforward: any component in the HDL that can be synthesized into a coalesced collection of flip-flops (or latches §VII-A) is considered a coalesced SSC. For distributed SSCs, the task is more challenging. A naive approach would be to enumerate the power set of all coalesced SSCs. However, this creates an obvious state-explosion problem. 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 those that are not.

Therefore, we break the *SSC Identification* phase into two sub-stages. First, we generate the data-flow graph from the HDL for a given circuit. Then, we systematically traverse the graph to enumerate: 1) the set of all coalesced SSCs, and 2) the set of all *connected* coalesced SSCs, i.e., distributed SSCs. We implement the first stage—data-flow graph generation—using the open-source Icarus Verilog (IVL) [34] compiler front-end with a custom back-end written in C++, and the second stage—SSC enumeration—using a depth-first graph traversal algorithm written in Python (Fig. 5A).

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 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

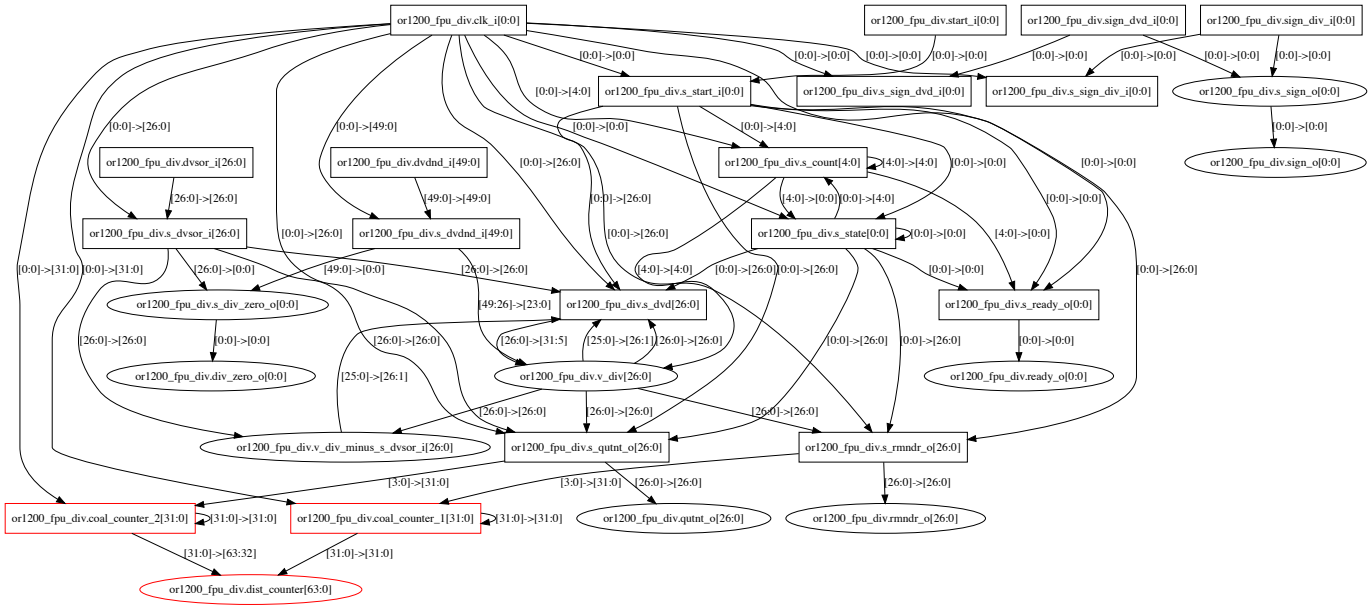

Fig. 6. **Hardware Data-Flow Graph.** Example data-flow graph, generated by Bomberman, of a floating-point division unit within the open-source OR1200 CPU [18]. Bomberman cross-references this graph with verification simulation results to identify SSCs (red). In the graph, rectangles represent registers, or flip-flops, and ellipses represent intermediate signals, i.e., outputs from combinational logic. Red rectangles indicate *coalseced* SSCs, while red ellipses represent *distributed* SSCs.

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 [35], [36]. The final data-flow graph is expressed using the Graphviz .dot format. An example data-flow graph generated from an open-source circuit [18] is shown in Fig. 6.

Next, our SSC enumeration script iterates over every node in the circuit data-flow graph, and identifies nodes (signals) that are outputs of registers (flip-flops). The script marks these nodes as coalesced SSCs. Finally, the script performs a depth-first-search (DFS) over every signal to piece together distributed SSCs. Our DFS reaches the base case when an input or register 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 formal definition of a TTT trigger. Our formal definition does not care about the progression of values expressed by the SSC(s), but cares only 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: Suspicious 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. Simulation Analysis

Bomberman begins by assuming *all* SSCs within the design are suspicious. Bomberman takes as input the simulation results from verification testing, and analyzes the values expressed by each suspicious SSC. At every update time within the simulation, Bomberman checks to see if any SSC expresses a value that causes it to violate any properties from our

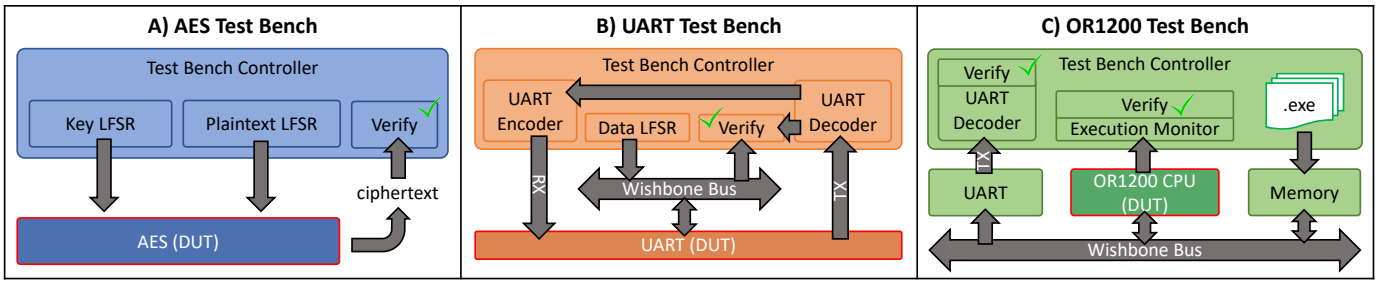

Fig. 7. **Hardware Test Benches.** Test bench architectures for each hardware design. The device under test (DUT) in each setup is outlined in red. For the AES and UART designs, linear feedback shift registers (LFSRs) generate random inputs for testing. For the OR1200 CPU, we compile OR1K assembly programs [37] into executables to exercise the design.

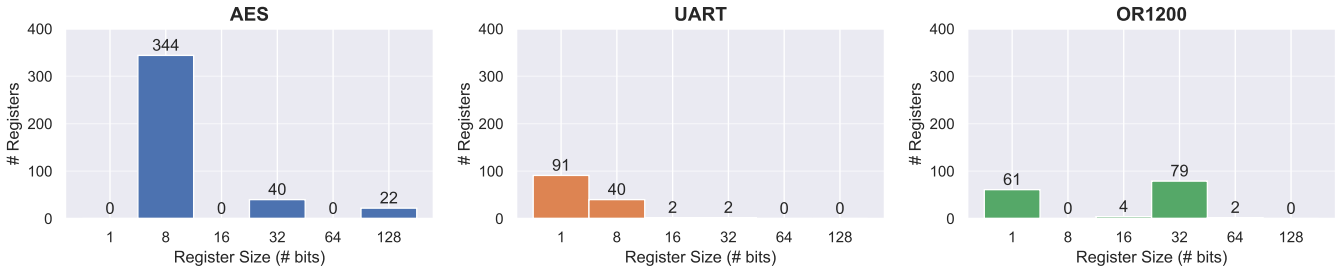

Fig. 8. **Design Characteristics.** Histograms illustrating the size and number of (coalesced) registers in each hardware design. Bomberman searches for registers (SSCs) in each design, and cross-references verification simulations to rule out if each SSC is suspicious, i.e., part of a TTT.

formal definition. If a property is violated, the SSC no longer meets the specifications to be part of a TTT, and Bomberman marks it *not suspicious*. Bomberman outputs any remaining suspicious SSCs for further analysis by verification engineers.

We implement the *Simulation Analysis* algorithm, as shown in Algorithm 1, using Python. Our analysis 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 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 formal definition (§IV-A) is violated. SSCs that break our formal definition of a TTT are marked *not suspicious*. 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 coalseced SSC values with respect to each other to ensure the recording of all possible distributed SSC values. 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

The primary goal of our evaluation is to quantify the effectiveness of our TTT identification approach with regards to false negatives and false positives. Thus, we evaluate Bomberman against three real-world hardware designs with real TTTs implanted in them. Our secondary goal is to evaluate Bomberman's performance when integrated with modern verification workflows. When verifying hardware designs, it is common practice to randomly generate test vectors in order to improve testing coverage. Therefore, we evaluate the effects of randomized testing on Bomberman's performance. Lastly, we highlight our improvement in TTT detection over prior work.

A. Hardware Designs

We evaluate Bomberman against three open-source hardware designs:

1) an AES accelerator,
2) a UART module,
3) and an OR1200 CPU.

We utilize a commonly studied AES accelerator, found in the TrustHub repository [19], and UART module and OR1200 CPU, from the OpenCores ORPSoC [18]. Fig. 7 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 Table I, and characterize the number of registers (i.e., potential SSCs) in each design in Fig. 8.

As shown in Table I, in terms of lines of HDL code and number of signals, the OR1200 CPU is the most complex while the UART module is the least complex. However, as Fig. 8 indicates, the AES design is the most computationally-intensive design for Bomberman to analyze, since it has the most registers, i.e., potentially suspicious SSCs.

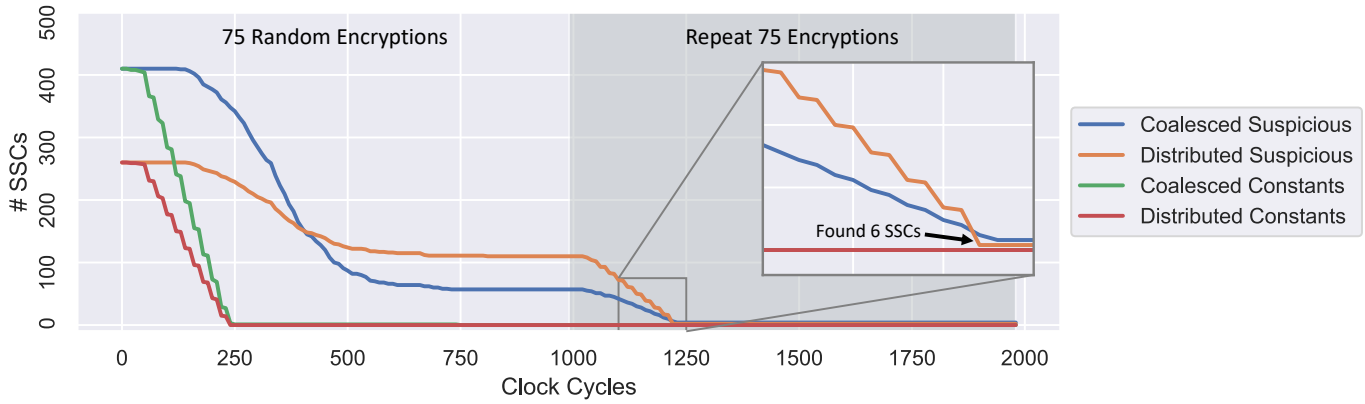

Fig. 9. **AES False Positives.** Reduction in number of suspicious SSC false positives over verification simulation timeline. Bomberman is able to identify the SSCs of all six TTT variants implanted in the design with zero false positives.

TABLE I
HARDWARE DESIGNS EVALUATED WITH BOMBERMAN.

Hardware Design	LOC	Total # Signals	# Registers
AES Accelerator	1045	2476	406
UART Module	2819	759	135
OR1200 CPU	25570	3716	146

*1) **AES Accelerator**:* The AES core is designed to operate solely in 128-bit counter (CTR) mode. The core takes as input a 128-bit key and 128-bits of a plaintext (i.e., a counter initialized to a random seed), and 22 clock cycles later produces the ciphertext. Note that the design is pipelined, thus 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 device-under-test (DUT) to generate random keys and plaintexts to exercise the core (Fig. 7A). Upon testing initialization, the test bench controller resets and initializes both LFSRs (to different 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.

*2) **UART Module**:* The UART core 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. Additionally, the UART core has several configuration and status registers. The core was configured to operate at a baud rate of 3.125 MHz.

We instantiate a Wishbone bus arbiter to interface to the DUT, and an LFSR to generate random data bytes to exercise the DUT (Fig. 7B). We also instantiate a UART encoder/decoder to receive, and echo back, any bytes transmitted from the DUT. Upon testing initialization, the tench bench controller resets and initializes the Wishbone bus arbiter, LFSR, and DUT, and launches the verification simulations.

*3) **OR1200 CPU**:* The OR1200 CPU is an implementation of the OR1K RISC instruction set architecture. It contains a 5-stage pipeline, instruction and data caches, and interfaces with other on-chip peripherals through a 32-bit Wishbone bus interface.

We instantiate a Wishbone bus arbiter to interface the DUT with a simulated main memory block, and a UART module to support standard I/O functions (Fig. 7C). The test bench controller has two main jobs after it initializes and resets all components contained 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 [38]. Second, it monitors the progress of each program execution and receives any output from an executing program from the UART decoder. We configure the test bench controller to run multiple programs sequentially, without resetting the device.

B. System Setup

As described in §V, Bomberman interfaces with Icarus Verilog (IVL). We also use IVL to perform all verification simulations of our three hardware designs. In both cases, we utilize version 10.1 of IVL. Both IVL and Bomberman were compiled with the Apple LLVM 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.

C. False Positives

Bomberman initially assumes all SSCs in a design are suspicious, and only eliminates suspicious SSCs that violate our formal definition of a TTT trigger (Algorithm 1). Thus, while it is impossible for Bomberman to generate false negatives, it *may* generate false positives. We empirically quantify Bomberman's false positive rate by exploring three real world hardware designs (§VI-A). Additionally we verify our implantation of Bomberman does not produce false negatives by implanting all six TTT variants (§IV) within each design. For each design, we plot the number of suspicious SSCs flagged by Bomberman over a specific verification simulation timeline.

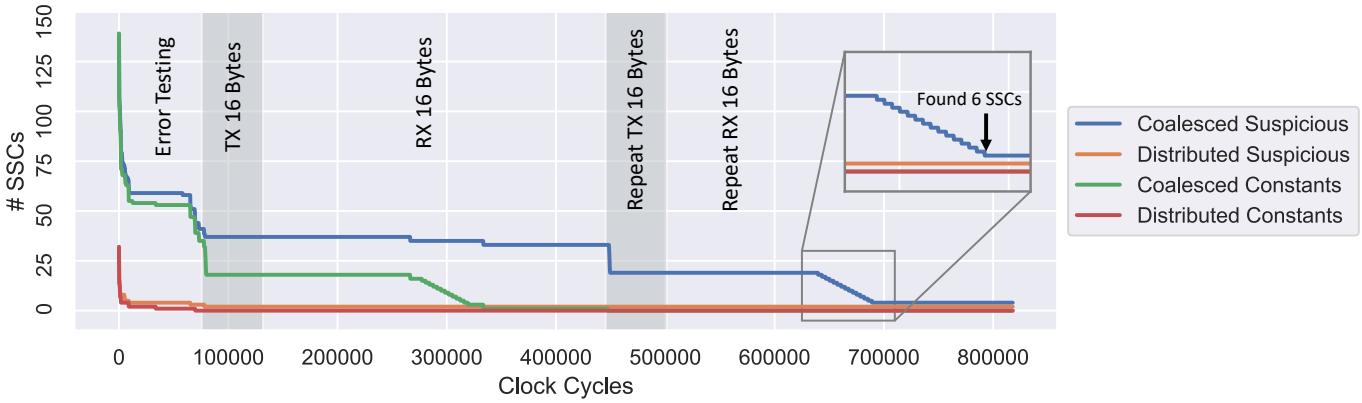

Fig. 10. **UART False Positives.** Similar to Fig. 9, but for the UART module. Bomberman is able to identify the SSCs of all six TTT variants implanted in the design with zero false positives.

Based on the TTT trigger definitions provided in Section IV, 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* a value has not 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 these are SSC design characteristics. However, *suspicious* and *constant* classifications are *not* mutually exclusive. In other words, an *constant* SSC is also *suspicious*, by our formal definition of a TTT trigger (§IV-A).

*1) **AES Accelerator**:* We configure the AES test bench to execute 75 random encryptions, i.e., 75 random 128-bit values with 75 (random and different) 128-bit keys, and subsequently repeat those same 75 encryptions (note: we explore the effects of random testing in §VI-D). We simulate the AES core at a clock frequency of 100 MHz. In Fig. 9 we plot the number of suspicious SSCs identified by Bomberman over the test bench simulation timeline.

During the first 250 clock cycles of the simulation, as registers cycle through more than one value, they are removed from the sets of coalesced and distributed constants. During the initial 75 random encryptions, after $\approx$750 clock cycles, the 8-bit registers toggle through all 256 possible values, and thus are also eliminated from the set 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 a TTT-free design only tracks state within tests, not since the last system reset, we take an alternative approach to eradicate large SSC false positives. Formally, ***repeating the same test case(s) without an intermediate system reset cause only non-suspicious SSCs to repeat values*** (violating Property 1 in §IV-A). We utilize this insight to efficiently minimize suspicious SSC false positives. Since the AES core is a deterministic state machine, we simply reset the LFSRs, and repeat the same 75 encryptions. After $\approx$ 1200 clock cycles, we achieve a false positive rate of 0% while detecting 100% of the TTT variants implanted in the AES core.

*2) **UART Module**:* We configure the UART test bench 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* 16 bytes again, to induce truly non-suspicious SSCs to repeat values. We simulate the UART core at a clock frequency of 100 MHz, and plot the number of suspicious SSCs identified by Bomberman over the simulation timeline in Fig. 10.

During the error testing phase, i.e., the first $\approx$80k clock cycles, Bomberman dramatically reduces the number of false positive constant and suspicious SSCs, as many of the registers in the UART module are either single bit status or configuration registers 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 a large reduction in the number of coalesced constants. Likewise, once the DUT transmits all bytes to the UART decoder, and the UART encoder echos all 16 bytes back to the DUT, 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 still not cycled 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 repeated values that

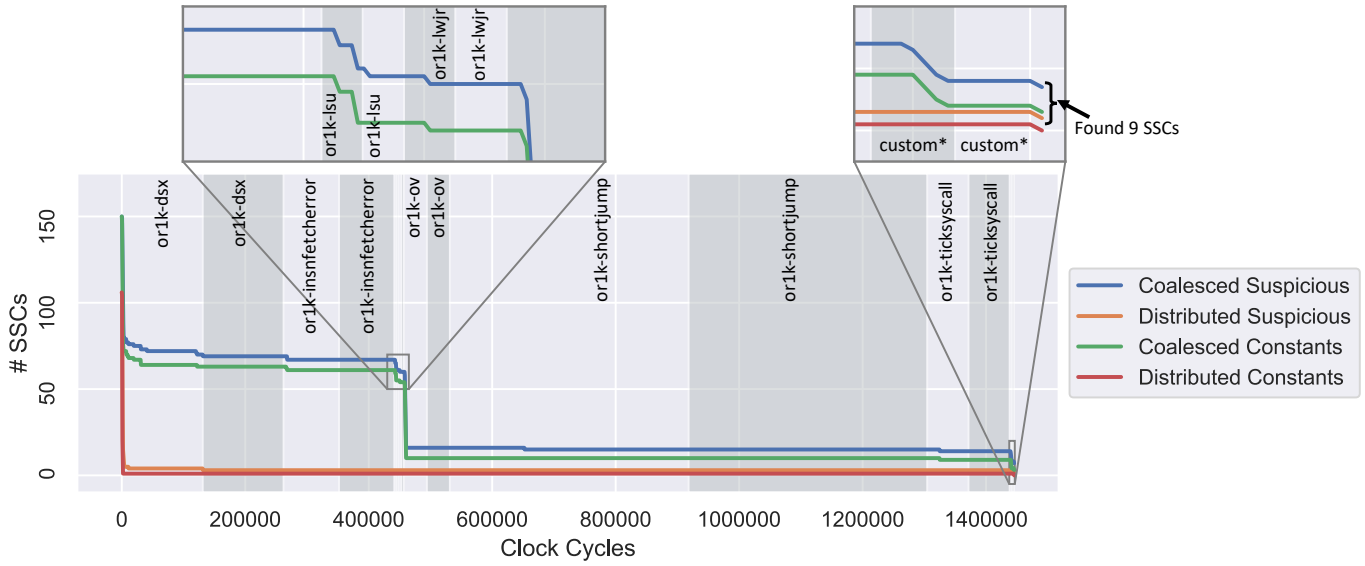

Fig. 11. **OR1200 False Positives.** Similar to Fig. 9, but for the OR1200 CPU. Bomberman flags nine SSCs as suspicious, six from implanted TTTs, and three benign, false positive constants. Verification engineers can use Bomberman results to direct testing to exercise the three miss-classified SSCs, and confirm they are benign.

eradicate the remaining false positives. Again, Bomberman successfully identifies 100% of TTT variants with a false positive rate of 0%.

*3) **OR1200 CPU**:* Lastly, we configure the OR1200 CPU test bench to run eight different OR1K assembly programs. Like the AES and UART simulations, we configure the test bench 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 [37], while the last program is custom written to exercise specific configuration registers not exercised by the testing suite.

During the execution of the first program (or1k-dsx), Bomberman largely reduces the number of false positive constant and suspicious SSCs. This is because, like the UART module, most of the registers within the OR1200 CPU are 1-bit configuration and status registers for which enumerating all (2) possible values is trivial. The next large reduction in suspicious SSCs is observed when the *or1k-ov* test is executed. This program exercises the register file and ALU causing the 32-bit registers to repeat values. The remaining tests, including the final custom test, yield minor reductions in suspicious SSCs, as they only exercise rarely used status and/or configuration registers.

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 utilized when an exception is thrown during a multi-operand arithmetic instruction sequence. In a real world deployment scenario, we imagine verification engineers utilizing Bomberman's insights to tailor their test cases to maximize their threat-specific testing coverage.

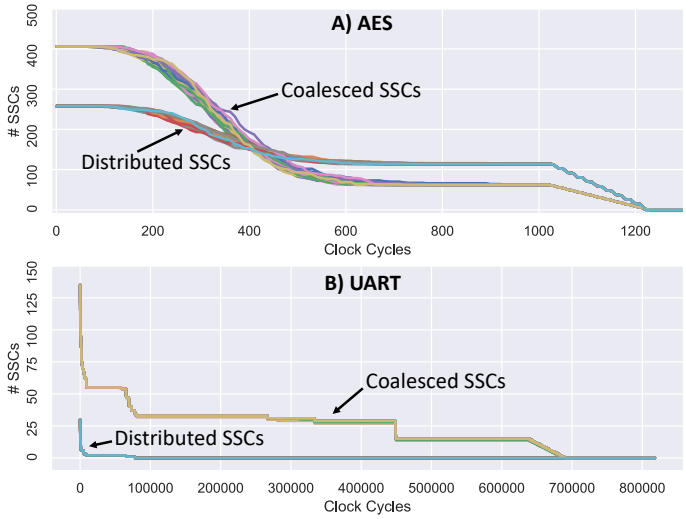

Fig. 12. **Randomized Testing.** Randomly generated verification test vectors do not affect Bomberman's performance. Rather, Bomberman's performance is dependent on verification test coverage.

D. Randomized Testing

Given the size and complexity of modern hardware designs, verification engineers typically use randomly-generated tests to maximize testing coverage. Similarly, we use LFSRs to generate random input vectors to exercise both the AES and UART designs. Thus, we ask the question: *does randomized testing 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. 7). 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-C), 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. 12, 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*.

E. Prior Work Analysis

Prior work, such as Li and Subramanyan *et al.*'s *WordRev* [32], [33] and Waksman *et al.*'s *Silencing Hardware Backdoors* [13], attempts to identify or thwart TTTs. While it is true these approaches work against known TTTs at the time of their respective publications, they fail to recognize the behavior of more complex TTTs presented in this work. Specifically, WordRev [32], [33] attempts to identify SSCs that behave like counters, via static analysis of gate-level netlists. WordRev leverages the notion that the carry bit propagates from the least-significant position to the most-significant position in counter registers. Thus the flip-flops that make up the counter register must be connected in a way to allow such propagation. However, this operating assumption causes WordRev to miss distributed TTTs, and TTTs with non-uniform increment values.

Alternatively, Waksman *et al.* [13] suggest intermittent power resets as a defense against TTTs. Intermittent power resets prevent untrusted logic from recognizing a specific amount of time has passed since the last system reset. Their approach requires formally verifying/validating the correct operation of the design for a set amount of time, denoted the validation epoch. Once they guarantee no TTT trigger activates within the validation epoch, the chip can safely operate as long as its power is cycled in time intervals less than the validation epoch. However, as Waksman *et al.* [13] 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 designs. Thus, power resets fail to defend against all TTT variants.

We compare the TTT defense capabilities of Bomberman with those of WordRev [32], [33] and power resets [13] in Table II. While WordRev and power resets are able to detect/defend against some TTTs, they miss complex TTTs that are detected by Bomberman.

TABLE II
PRIOR WORK ANALYSIS.

TTT Type	WordRev [32]	Power Resets [13]	Bomberman
CUP	✓	✓	✓
CUS	✓	✗	✓
CNP	✗	✗	✓
CNS	✗	✗	✓
D-Homogeneous	✗	✗*	✓
D-Heterogeneous	✗	✗	✓

XXX: **C**oalesced or **D**istributed SSC
X**X**X: **U**niform or **N**on-uniform Increment Value
XX**X**: **P**eriodic or **S**poradic Increment Event
* While power resets defend against homogeneous distributed TTTs compromised entirely of CUP triggers, they do not defend against any other form.

VII. DISCUSSION

A. Latches

The first step in identifying potential TTTs in a hardware design is locating all SSCs. Bomberman locates all SSCs by finding 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 induce unwanted timing errors. As a result, HDL compilers in synthesis CAD tools emit warnings when they infer latches in a design—highlighting the TTT. Nonetheless, to support such (bad) design practices, we can extend Bomberman's data-flow graph generation compiler back-end to recognize latches as potential SSCs.

B. TTT Identification in Physical Layouts

Bomberman is designed to integrate as an extension into existing front-end verification toolchains 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.* [24], there are several hardware reverse engineering tools for carrying out this task. In addition to a gate-level netlist, Bomberman also requires HDL device-level 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 data-flow graph. However, HDL device models are usually provided as a part of the process technology IP portfolio purchased by hardware designers.

C. SSC Volatility

As Waksman *et al.* point out in their power reset work [13], all on-chip SSCs may not be volatile, i.e., lose their values when power is reset—e.g., flash memory. Fortunately, our approach does not require any assumptions about the volatility of SSCs. Therefore, Bomberman can be extended to recognize non-volatile SSCs—that are not directly expressable in HDL—during its *SSC Identification* phase. Volatile or non-volatile,

Bomberman takes a conservative approach, assuming all SSCs are initially suspicious, only clearing SSCs who express behaviors that violate our formal definition rule set (§IV-A).

D. 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 data-flow graph 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, 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 data-flow graph filtering mechanism greatly reduces the overall set of potentially suspicious SSCs.

VIII. Related Work

The implantation, detection, and prevention of hardware backdoors across the hardware design phases are widely studied. Attacks range from design-time attacks [7], [23], [39], to layout-level modifications at fabrication time [24]–[26]. On the defensive side, most work focuses on post-fabrication Trojan detection [40]–[50], 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 also substantial prior work in both detective [8]–[10], [32], [33] and preventative [11], [13] measures against 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 design can be exploited to mount wide varieties of software-level attacks. Lin *et al.* [23] 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 power noise margin of the entire device. While both attacks demonstrate different payloads, they both require internal triggering mechanisms to remain undetectable during verification and post-fabrication testing. Rather, 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.

On the defensive side, both design- and run-time approaches are studied. At design-time, Hicks *et al.* [10] are the first to 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 inputs and outputs. Lastly, Li and Subramanyan *et al.* [32], [33] 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. Unfortunately, prior work has found gaps in UCI, VeriTrust, and FANCI [14]–[16], and, as we point out (§VI-E), WordRev is incapable of detecting all TTT variants.

At run-time, Waksman *et al.* [11] propose on-chip invariant monitors that dynamically verify correct system behavior. In another work, Waksman *et al.* [13] present architectural design principles and run-time mechanisms to make integrating any design-time Trojan intractable. Their approach is two-fold: First, they thwart data-based triggers by scrambling inputs and outputs between hardware modules at run-time. Second, they thwart time-based triggers, i.e., TTTs, using intermittent power resets. As we show in §VI-E, power-resets themselves are not capable of thwarting TTT variants with non-deterministic behavior.

IX. Conclusion

Bomberman is an effective example of a threat-specific detection-based defense against ticking timebomb Trojans. Unlike, prior work [8]–[10], [13], we do not attempt to provide a panacea against *all* design-time Trojans. Instead, we formally define the behavioral characteristics of a specific but important threat, TTTs, and develop a complete defense (Bomberman) capable of identifying *all* TTTs as we define them. Specifically, across 3 open-source hardware designs—an AES accelerator, a UART module, and an OR1200 CPU—we are able to detect all six TTT variants defined in §IV, with less then 0.3% false positives.

Bomberman demonstrates the power of threat-specific verification, and seeks to inspire future threat-specific defenses against hardware Trojans. 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 detection schemes, like UCI [10], FANCI [8], and VeriTrust [9], along with future threat-specific Trojan detection schemes, we aim to create an insurmountable barrier for design-time attackers.

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 *27th {USENIX} Security Symposium ({USENIX} Security)*, 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 *40th 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 *27th {USENIX} Security Symposium ({USENIX} Security)*, 2018.

[7] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the 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 *Proceedings of the ACM SIGSAC conference on Computer & communications security*, 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 *Proceedings of the 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 *Proceedings of the 20th 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 *Proceedings of the 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 *Proceedings of the ACM SIGSAC Conference on Computer and 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 *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, 1999, pp. 282–287.

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

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

[20] D. Takahashi, "Apple unveils 7-nanometer a12 bionic chip for new iphones," September 2018, https://venturebeat.com/2018/09/12/apple-unveils-7-nanometer-a12-bionic-chip-for-new-iphones/.

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

[22] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design & Test of Computers*, vol. 27, no. 1, 2010.

[23] 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.

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

[25] 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.

[26] 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.

[27] J. Robertson and M. Riley, "The big hack: How china used a tiny chip to infiltrate u.s. companies," Oct 2018, https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies.

[28] S. Gallagher, "Photos of an NSA "upgrade" factory show Cisco router getting implant," May 2014, https://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/.

[29] 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.

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

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

[32] 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.

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

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

[35] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin ii-an open-source verilog hdl synthesis tool for cad research," in *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2010.

[36] 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.

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

[38] ——, "Or1k-elf toolchain," https://openrisc.io/newlib/.

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

[40] S. Kelly, X. Zhang, M. Tehranipoor, and A. Ferraiuolo, "Detecting hardware trojans using on-chip sensors in an asic design," *Journal of Electronic Testing*, vol. 31, no. 1, pp. 11–26, 2015.

[41] 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.

[42] 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.

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

[44] 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.

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

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

[47] 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.

[48] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao, "Guided test generation for isolation and detection of embedded trojans in ics," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, 2008.

[49] T. Trippel, K. G. Shin, K. B. Bush, and M. Hicks, "Defensive routing: a preventive layout-level defense against untrusted foundries," *ArXiv*, 2019. [Online]. Available: http://arxiv.org/abs/1906.08842

[50] ——, "An extensible framework for quantifying the coverage of defenses against untrusted foundries," *ArXiv*, 2019. [Online]. Available: http://arxiv.org/abs/1906.08836