Silencing Hardware Backdoors

Adam Waksman
Computer Architecture and Security Technology Lab
Department of Computer Science
Columbia University
New York, USA
waksman@cs.columbia.edu

Simha Sethumadhavan
Computer Architecture and Security Technology Lab
Department of Computer Science
Columbia University
New York, USA
simha@cs.columbia.edu

***Abstract*—Hardware components can contain hidden backdoors, which can be enabled with catastrophic effects or for ill-gotten profit. These backdoors can be inserted by a malicious insider on the design team or a third-party IP provider. In this paper, we propose techniques that allow us to build trustworthy hardware systems from components designed by untrusted designers or procured from untrusted third-party IP providers.**

We present the first solution for disabling digital, design-level hardware backdoors. The principle is that rather than try to discover the malicious logic in the design – an extremely hard problem – we make the backdoor design problem itself intractable to the attacker. The key idea is to scramble inputs that are supplied to the hardware units at runtime, making it infeasible for malicious components to acquire the information they need to perform malicious actions.

We show that the proposed techniques cover the attack space of deterministic, digital HDL backdoors, provide probabilistic security guarantees, and can be applied to a wide variety of hardware components. Our evaluation with the SPEC 2006 benchmarks shows negligible performance loss (less than 1% on average) and that our techniques can be integrated into contemporary microprocessor designs.

***Index Terms*—hardware, security, performance, backdoors, triggers**

I. Introduction

Malicious modifications to hardware from insiders pose a significant threat today [1, 4, 6, 7, 11, 22, 25, 26, 27]. The complexity of hardware systems and the large number of engineers involved in the designing of them pose a security threat because it is easy for one malicious individual to alter one tiny piece of the system. Although this behavior is very risky, it can be very profitable for an attacker because a hardware backdoor provides a foothold into any sensitive or critical information in the system [13]. Such attacks can be especially devastating to security-critical domains, such as military and financial institutions. Hardware, as the root of the computing base, must be trustworthy, but this trust is becoming harder and harder to assume.

A malicious modification or a backdoor can find its way into a design in several ways. The modification could come from a core design component, *e.g.,* a few lines of Hardware Design Language (HDL) core code can be changed to cause malicious functionality. The use of third-party intellectual property (IP) provides another opportunity. Today's hardware designs use an extensive array of third party IP components, such as memory controllers, microcontrollers, display controllers, DSP and graphics cores, bus interfaces, network controllers, cryptographic units, and an assortment of building blocks, such as decoders, encoders, CAMs and memory blocks. Often these units are acquired from vendors as HDL implementations and integrated into designs only after passing validation tests without code review for malicious modifications. Even if complete code reviews are possible, they are extremely unlikely to find carefully hidden backdoors, as evidenced by the fact that non-malicious modern designs ship with many bugs today.

A key aspect of hardware backdoors that makes them so hard to detect during validation is that they can lie dormant during (random or directed) testing and can be triggered to wake up at a later time. Verification fails because designs are too large to formally verify, and there are exponentially many different ways to express a hardware backdoor.

However, even if we cannot find the malicious logic, we claim and show that it is still possible to disable backdoors.

Our key insight is that while validation testing is incomplete, it provides a strong foundation that can be leveraged to increase trustworthiness. Specifically, validation demonstrates that the hardware functions in a certain way for a subset of the possible inputs. We leverage the fact that since the hardware passes validation tests (which it must in order to make it to market), any malicious logic must be dormant for the entire testing input space, waiting for something to trigger it. If we can silence those triggers, we can prevent the backdoors from turning on without having to explicitly detect the backdoor logic.

Waksman and Sethumadhavan previously observed that there are finitely many types of deterministic, digital backdoor triggers that can be injected by an inside designer [26]. We leverage this observation and devise methods to disable all of these types of triggers by obfuscating or scrambling inputs supplied to the hardware units in order to prevent those units from recognizing triggers. These techniques must alter inputs in a benign way so that after validation testing, hardware can never receive inputs that appear distinct from what was already tested but can also produce correct outputs with minimal changes to the design. We describe three techniques (Figure 1) that, in concert, disable backdoor triggers.

• **Power Resets** The first technique prevents untrusted units from detecting or computing how long they have been active,

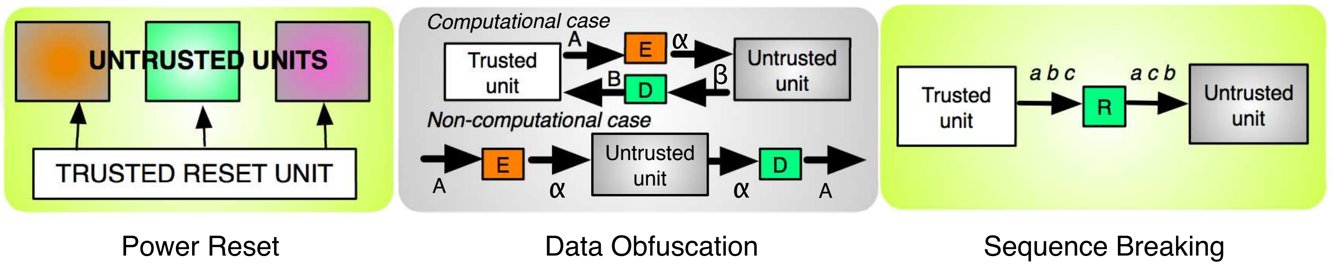


Fig. 1. Obfuscation techniques to disable backdoor triggers. The left picture shows power resets. The middle picture shows data obfuscation, both for computational and non-computational units. The right picture shows sequence breaking by reordering. Legend: **E:** Encryption Unit, **D:** Decryption Unit, **R:** Reordering Unit. These units are trusted and small enough to be formally verified.

thus preventing time-based attacks.

- **Data Obfuscation** The second technique encrypts input values to untrusted units to prevent them from receiving special codes, thus preventing them from recognizing data-based triggers.
- **Sequence Breaking** The final technique pseudo-randomly scrambles the order of events entering untrusted units to prevent them from recognizing sequences of events that can serve as data-based triggers.

Our solutions are broadly applicable to many types of digital hardware, but in this paper we study the feasibility of our techniques using the OpenSPARC T2 muticore chip from Oracle (formerly Sun Microsystems). Our feasibility study shows that the three techniques presented in the paper, taken together, provide coverage against all known types of digital hardware design backdoors for many on-chip hardware modules in the openSPARC design. This coverage can further be expanded with a small amount of duplication. Based on simulation of SPEC 2006 benchmarks, an industry standard benchmark suite for measuring performance of processors, we also show that these techniques incur negligible performance losses.

The rest of the paper is organized as follows: Section II discusses related work. Section III outlines our framework and model of hardware. Section IV outlines our threat model and assumptions. Section V describes our solutions and discusses applicability and implementation details. Section VI provides arguments for the security and coverage of our solutions. Section VII describes our experimental infrastructure, results and coverage. We summarize and conclude in Section VIII.

II. Related Work

Hardware backdoor protection is a relatively new area of research that protects against a serious threat. Recently, some attention has been given to protecting hardware designs from hardware backdoors implanted by malicious insiders, but there are currently only two known solutions that have been proposed. Hicks *et al*. designed a method for statically analyzing RTL code for potential backdoors, tagging suspicious circuits, and then detecting predicted malicious activity at runtime[11]. This hardware/software hybrid solution can work for some backdoors and even as a recovery mechanism. Its admitted weaknesses are that the software component is vulnerable to attack and additionally that the software emulator must itself run on some hardware, which can lead to infinite loops and DOS (denial of service).

Waksman and Sethumadhavan proposed a different method that detects unusual hardware behavior at runtime using a self-monitoring on-chip network[26]. This method, like the previous one, focusses on detection (as opposed to prevention). Unlike the previous solution, this is a purely hardware solution and thus not vulnerable to software deficiencies. However, it has admittedly incomplete coverage, as it applies to specific types of backdoor payloads and invariants.

A fundamental difference between this paper and previous work is that since we disable the backdoor at its origination point — the trigger — we provide a much more general solution than previous approaches. Both previous solutions use deterministic methods to protect against a *subset* of the attack space. Our methods, by contrast, provide probabilistic guarantees against *all* deterministic, digital backdoor triggers. Unlike other methods, our scheme can prevent DOS attacks.

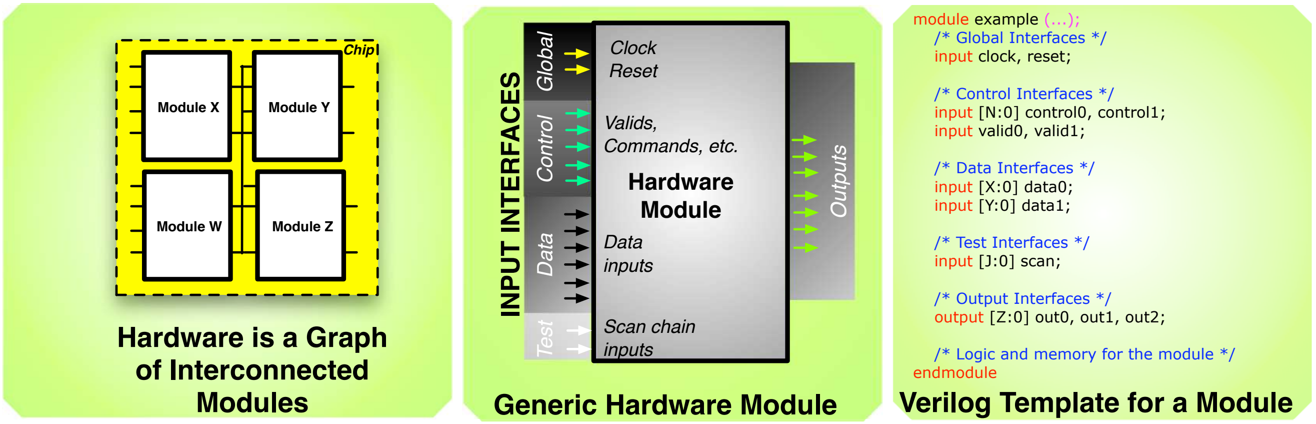
There has been prior work in tangentially related areas of hardware protection, usually leveraging a trusted piece of the design or design process. Significant work has been done (mainly in the fabrication phase) toward detecting active backdoors [5], analyzing side-channel effects [20], detecting suspicious path delays [12] and detecting backdoors added at the fabrication level[2, 3, 4, 7, 15, 18, 27]. However, all of this prior work assumes that the properties of the backdoors are limited and that there is a golden netlist (trusted RTL description). The reason for this common assumption of a trusted front end code base is that code is often written by insiders whereas the manufacturing process is often outsourced. However, increasing design team sizes and increasing use of third party IP on-chip are making this assumption about the front end less realistic.

III. Framework for Models and Solutions

Our model for digital hardware is an interconnected set of modules, which are connected via interfaces. Since hardware is usually composed of several small modules, and since communication happens via interfaces, we enforce security at the interface level. If we can ensure that trigger payloads cannot be delivered through any interface then we can be assured that backdoors cannot be triggered in hardware.

The interfaces to digital hardware modules can be broken down into five categories (Figure 2).

- *Global Interfaces:* A global interface is a set of signals



```
module example (...);
  /* Global Interfaces */
  input clock, reset;

  /* Control Interfaces */
  input [N:0] control0, control1;
  input valid0, valid1;

  /* Data Interfaces */
  input [X:0] data0;
  input [Y:0] data1;

  /* Test Interfaces */
  input [J:0] scan;

  /* Output Interfaces */
  output [Z:0] out0, out1, out2;

  /* Logic and memory for the module */
endmodule
```

Fig. 2. Any hardware module will have at most four types of input interfaces. A backdoor can only be triggered by malicious inputs on one of these input interfaces. The code on the right hand side shows the Verilog template for a module.

that is provided to all modules. This usually includes a clock signal, a reset signal, and power signals.

- *Control Interfaces:* An interface of this type is one or more wire groups that control how the unit operates. Examples include inputs that control transitions in a state machine and input bits that indicate validity of data supplied to the unit.
- *Data Interfaces:* An interface of this type represents a single value that is used as such in a module. For example, an integer being fed into an ALU or an address being passed into a memory controller are both data interfaces.
- *Test Interfaces:* A test interface is an interface that is only used for post-manufacture testing and serves no purpose after deployment. An example of this is a scan chain interface.
- *Output Interfaces:* These are the interfaces for the signals coming out of a module. They can potentially feed into any of the four types of input interfaces (data, control, global, test). In the common case, these will either feed into data or control interfaces.

For any given attack, one can pinpoint the interfaces that first violate specification, *i.e.* the first one to yield an incorrect result or cause an erroneous state transition. While an attack may be complex and involve coordination between several hardware modules, if each individual interface is forced to behave correctly, then the attack cannot be executed. Thus to prevent hardware backdoor triggers we examine hardware interfaces on a module by module basis to suggest security modifications. Further, there are only a limited number of ways in which attacks on these interfaces can be triggered (discussed in Section IV), which leads to few simple security methods (discussed in Section V).

IV. Threat Model

A. Attack Space and Vectors

Our threat model allows for any insider to modify the HDL specification of digital hardware. The attack space is the set of all input interfaces for all modules that constitute the hardware design. We focus only on the input interfaces (global, test, control, data) because if all input interfaces are secured and the unit's functionality has been validated, then the outputs can be trusted. Our attack vectors include two different types digital triggers — data and time. We build on the earlier taxonomy [26] by breaking data triggers into two further sub-types — sequence and single-shot. Next, we describe each of the three trigger types and explain how they are coupled with types of input interfaces.

- **Ticking Timebombs:** A malicious HDL designer can program a timebomb backdoor into HDL code so that a backdoor automatically triggers a fixed amount of time after the unit powers on. For example, a microcontroller can be programmed to fail after a pre-determined number of clock cycles. This type of attack poses a serious threat to many high security areas. Even if the hardware is used in a secure, tamper-free environment, running only trusted code, a timebomb can undermine the security of the system or function as a 'kill switch'. Additionally, this type of attack does not require the adversary to have any access to the machine under attack.

One aspect of ticking timebombs that makes them so dangerous is that they are completely undetectable by any validation techniques. Even a formal validation technique that verifies all possible input values cannot prove that a timebomb will never go off (since validation lasts only a finite amount of time, one can never know if validation has run for a long enough period of time). Thus a well-placed timebomb can be inserted by a designer, evade all validation techniques, and trigger at any time, without warning.

Ticking timebombs are associated with global interfaces. This is because the digital clock signal is the only way to monitor the passage of time in synchronous digital designs. Other information can serve as a way of keeping track of or estimating the passage of time, *e.g.,* turn on backdoor after a million cache misses. However, as we describe in Section V, these timebombs ultimately depend on the clock signal to record passage of time and thus can be protected by protecting the global interface.

- **Cheat Codes:** Backdoors that are triggered by data values

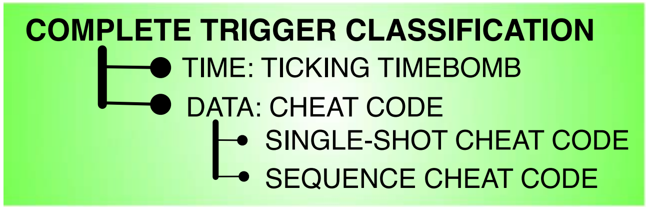


Fig. 3. Hardware backdoor trigger classification.

are called *cheat codes*. A cheat code is a special input (or sequence of inputs) that functions as a key to open up or 'turn on' malicious hardware. A cheat code can be thought of as secret information that the attacker uses to identify his or her self to the hardware backdoor logic. This identity must be unique to avoid being accidentally provided during validation tests. In contrast to timebombs this type of attack requires an additional attack vector: in addition to the malicious designer programming a backdoor into the HDL design, there must be a user who can execute code on the malicious hardware in order to provide the cheat code key.

There are two ways to communicate cheat codes. One way is to send a single data value containing the entire cheat code. We will call this a *single-shot cheat code*. A single-shot cheat code usually arrives at an interface as a large piece of data, such as an address. For example, the address `0xdecafbad` could be the secret trigger that turns on the backdoor. In theory, single-shot cheat codes can be passed to the backdoor through control or data interfaces.

The other way to communicate a large cheat code is in multiple pieces. We will call this a *sequence cheat code*. This type of cheat code arrives in small pieces over multiple cycles or multiple inputs. Just like the single-shot codes, these cheat codes can be supplied through the data or control interfaces. For example, if the secret trigger is `0xdecafbad`, and the malicious unit has a data interface big enough for a hex character, the attacker might pass the hex values `0xd`, `0xe`, `0xc`, `0xa`, `0xf`, `0xb`, `0xa`, `0xd` over eight different cycles (or inputs). Similarly, one could imagine an unusual series of loads and stores conveying a cheat code to a memory controller as a sequence through the control interface.

We note here that the inputs that compose a sequence cheat code do not necessarily have to arrive in consecutive cycles. They can arrive in a staggered fashion or over a long period of time. As long as the timing and the ordering is defined by the attacker and recognized in the backdoor trigger logic, the individual bits that together comprise the sequence cheat code can come in almost any arrangement, limited only by the creativity of the attacker.

To summarize the relationship between interfaces and triggers, data and control interfaces may be prone to cheat code attacks (either sequence or single-shot). Global interfaces are only open to timebomb attacks *i.e.* clock and reset can only take on two values and thus cannot serve as cheat codes. Output interfaces are not vulnerable so long as all input interfaces have been protected [1]. We do not handle test interfaces in this paper. One simple solution for test interfaces — if they are considered threatened — is to burn out those interfaces using programmable electronic fuses before deployment, since they are not needed post-deployment.

B. Attack Possibilities

We have two different attack settings that depend on how privileged the attacker(s) are. If the attacker has privileged access to the machine after it has been deployed (*e.g.,* the attacker is a user as well as designer) then we must defend against cheat codes that might be inserted by malicious programs. If not, then we only have to protect against ticking timebombs because these are the only triggers that can be used by a malicious designer without the aid of an user. An example of this latter setting might occur if one organization or nation-state procures hardware from another nation-state but allows the hardware to be used only by trusted operatives.

C. Assumptions

• *Assumption #1: Triggers* We assume that a hardware backdoor, by design, needs to escape validation testing. Therefore, it cannot be always active and must have some way of being triggered at a point in time after validation testing has been completed. We further assume that this trigger is a digital signal that can be designed into the HDL (as opposed to an internal analog circuit or any external factor, such as temperature). This is a reasonable assumption because at the HDL design level it is hard to program analog undriven circuits that pass validation. Nevertheless, one can imagine backdoors in analog circuitry or induced by external side channels. We leave these cases for future work.

• *Assumption #2: Trust in Validation* Our solutions leverage the fact that we can use validation to determine that a component or a third party IP unit functions correctly and does not exfiltrate information for some finite number N cycles (where N is a typical validation epoch, *e.g.,* a few million). This is typical practice when third party IP is procured. In the case that we are concerned about malicious insiders (as opposed to third party entities), validation engineers do not pose the same threat as a designer. This is because a single designer can insert a malicious backdoor that can circumvent the whole validation process, but validation teams tend to be large, and a single unit goes through multiple levels of validation tests (module, unit, core, chip, etc.), so it would take a conspiracy of almost the entire validation team to violate this trust.

• *Assumption #3: Unprotected units* We leverage trust in small, manually or formally verifiable units. This includes small circuits we include to implement our security measures. We do not externally protect these units.

[1] Results from a recent hardware backdoor programming competition [19] provide evidence that our taxonomy is reasonable. Not all competitors chose to implement digital HDL attacks. Of the ones that did, there were no attacks that did not fit neatly within our taxonomy. Three of the 19 digital attacks in that competition were timebombs. Five attacks used sequence cheat codes on small interfaces, such as one that caused the unit to break if the ASCII characters "new haven" were sent as inputs in that order. A majority of the attacks (eleven) used single-shot cheat codes directly against data interfaces by having one particular input turn on a malicious mode.

V. Solution

Our general approach is to introduce enough randomness into each hardware unit that a backdoor trigger cannot be reliably recognized by malicious circuitry. The objective of malicious circuitry is to detect unique or unusual inputs that are meant to trigger a backdoor, and if the inputs to the malicious logic are scrambled or encrypted, the act of detection becomes too difficult.

As described in Section IV, there are three different triggers we are concerned with — timebombs, single-shot cheat codes, and sequence cheat codes. A timebomb can be delivered only through the global interface (the clock signal), and the two types of cheat codes can be delivered through control or data interfaces. Each of these three triggers requires its own protection scheme. We discuss and present solutions for each of these three categories, as well as applicability, adaptation to modern microprocessors, and limitations.

A. Power Resets

The first category we consider is the time-based category — ticking timebombs. The power reset technique protects untrusted units from these timebomb triggers and is generally applicable to any digital hardware. The key to our strategy is to prevent untrusted logic from knowing that a large amount of time has passed since start-up. In other words, every untrusted hardware unit — regardless of whether it is in a core, memory system, off-chip, etc. — will at all times be in a state where it has only recently been turned on. We ensure this by frequently powering off and on each unit, causing data in local state (such as registers) to be lost.

The circuit for power resets is very simple. It is a counter that counts down from some preset value to zero. This value has to be smaller than the length of the validation epoch because the validation engineers need to validate that the hardware reaches a power reset without a timebomb going off. The validation epoch can vary, but it is a known value for any particular setting. The Verilog Hardware Description Language code that can issue this power reset is shown below (using as an example a validation epoch of $2^{20} = 1,048,576$ cycles). As can be seen from the implementation, it can be easily manually verified to be free of backdoors.

```
module reset(clk, rst, out);
  input clk;
  input rst;
  output out;
  reg [19:0] countdown;
  always @(posedge clk) begin
    if(rst) countdown <= 20'b0 - 1'b1;
    else countdown <= countdown - 1'b1;
  end
  assign out = (countdown == 0);
endmodule
```

Naturally, hardware will need to have some continuity across epochs. For example, in the case of microprocessors, users will want to run programs that take much longer than the validation epoch. We get around this problem by using a lightweight version of context saving and restoring so that program execution is not disrupted by power resets. Each time we approach the validation epoch, we write the current instruction pointer(s) to memory, flush the pipeline, and power off the hardware units for one or a few cycles. This wipes all internal, volatile state and resets all registers, including both helpful ones (such as branch history tables) and malicious ones (such as ticking timebombs). The program then picks up where it left off.

Several practical issues may arise when applying this method to various real-world components.

- *Main Memory Writes:* One security question that might arise is: *Since main memory stays on, and since we write the instruction pointer to memory, how come the timebomb counter cannot be written to main memory?*

Recall that by assumption, the microprocessor executes correctly during the validation epoch. This means that there cannot be any incorrect writes to main memory before the first power reset. Therefore, a trigger cannot be spread across multiple validation epochs.

- *Devices:* Resetting various devices may require fine-grained management in device drivers. The device drivers may need support to replay transactions when peripherals power-cycle in the middle of a transaction. Prior work on handling transient peripheral failures through intelligent device driver architectures can be used to provide this support [23, 24].

- *Non-Volatile Memory:* Another security issue that arises is non-volatile memory. Powering off wipes clean volatile memory and registers, but we may not be able to assume that all on-chip memory is volatile, as it may be possible to include a small amount of malicious on-chip flash or some other non-volatile memory.

This brings up the question: *Given a unit that we do not want to have hidden, non-volatile memory, how can we ensure that it has none?* One way to do this is to burn out the memory. Many non-volatiles memories, such as flash, have limited write endurance. If a unit may have been maliciously configured to write a value to an internal piece of flash every time it is about to be powered off, then we can hook the clock up to the power signal of the hardware unit that is suspected to contain flash, causing the unit to turn off and back on repeatedly until the burn-out threshold, thus destroying any flash that might be inside. This procedure could be done very easily post-tapeout. Another strategy would be to take a few copies of the manufactured unit and visually inspect them to confirm that there is no non-volatile memory [10].

- *Unmaskable Interrupts:* Even while powered off for a few cycles, it is possible that the microprocessor will receive an unmaskable interrupt from an external unit that is on. This signal should not be lost. In order to preserve correctness, a slight adjustment is required for off-chip components that can send unmaskable interrupts. These signals must go into a small FIFO and wait for acknowledgement. If power is off, this acknowledgement will not come until a few cycles after they are issued.

- *Performance Counters:* Some modern microprocessors include built-in performance counters that track certain performance statistics, such as clock cycles or cache misses. It is desirable for these counters to not be reset. However, this is a somewhat fundamental issue, because a performance counter is essentially a benign ticking timebomb trigger. Therefore, there is a trade-off between the ability to do easy performance tracking in hardware and the ability to be secure against ticking timebomb attacks. Our solution to this problem is to make use of a very small amount of trusted hardware (if logic is trivial enough it can be formally verified or checked by code review). This small hardware unit keeps track of the performance counters and keeps power during the resets. By keeping this unit trivial and allowing it only one output interface, we can make sure this unit is not sending information to other on-chip units or otherwise exfiltrating timing information.
- *Performance:* Another practical issue is performance. If we periodically flush the pipeline and wipe out volatile memory, this can cause a performance hit. We salvage most of this performance by keeping power on to large, standard RAMs (*e.g.,* caches, memory). We still lose various smaller pieces of state, such as branch history tables and information in prefetchers. In our experimental evaluation section, we study the effect on performance of power resets.
- *Applicability and Limitations:* The power reset method is universally applicable to any digital logic. It provides complete coverage against ticking timebombs, which is the more dangerous of the two general types of digital hardware backdoor triggers. More formal arguments as to why our solution is complete are provided in Section VI.

B. Data Obfuscation

The second category of attacks we consider are single-shot cheat codes. The insight behind our solution is that the attacker is expecting a particular input value to trigger the attack. If we obfuscate the inputs, then the attacker's unit can be deceived and fail to recognize the trigger.

The specific method for obfuscating the inputs depends on the type of hardware unit. We categorize hardware units into two general types — computational and non-computational — and discuss our solution for each type respectively.

- **Non-computational units:** These units do not operate on data values; they only move them around. Many common units in real microprocessors fit this category. For example, a memory controller accepts a large address and a large data write value, but it often does not perform any logical operations on these. Similarly, many buses, interconnects, routers, etc. move around data without performing computation on the data. Obfuscating inputs to non-computational units is simple. We use any encryption scheme to obfuscate the data before it enters the unit.

We can use very low overhead, simple encryption schemes to implement obfuscation. Since the value has to remain secret only for one or a few clock cycles, it does not have to be strong in the sense that software-based encryption schemes generally are. In the context of hardware backdoors, the attacker has very limited capabilities because of the restricted hardware budget and processing time to deploy an attack against the encryption scheme.

Some examples of simple encryption schemes include XOR or addition by a random value. For instance, a bit-wise XOR encryption scheme is provably secure when the ciphertext and plaintext cannot be simultaneously known or guessed. Using a hardware random number generator or a PUF, a random and secure key can be generated that only needs to be used and stored for a short time. This process can be orchestrated by encrypting the inputs to the unit with a small (manually or formally verifiable) circuit and decrypting the outputs from the unit with a similar circuit. From the perspective of the outside world, the hardware unit is unchanged. However, the hardware unit never sees any of the original data.

To take a simple and effective example of this hardware encryption or obfuscation, we can protect a black-box non-computational module called *BLACK_BOX* with the following short, manually verifiable wrapper:

```
module black_box_wrapper(clk, rst, data,
                         control, random, out);
  input clk, rst, data, control, random;
  wire untrusted_out;
  output out;
  BLACK_BOX untrusted(.clk(clk),
        .rst (rst),
        .data(data xor random),
        .control(control),
        .out(untrusted_out));
  assign out = untrusted_out xor random;
endmodule
```

- **Computational units:** Data encryption for computational units is more complex and must be done to some degree on a unit-by-unit basis. In a few cases, the complexity may be so great that duplication is more efficient, and duplication serves as a fall-back strategy.

Our method for obscuring these cheat codes is motivated by homomorphic encryption schemes from the realm of software. We call an operation f *homomorphic* with respect to another operation g if $f(g(x), g(y)) = g(f(x, y))$. One simple example of this is when f is multiplication and g is the squaring function. Explicitly,

$$x^2y^2 = (xy)^2$$

If the functionality required of a (toy example) untrusted unit is to compute the square of a value, we can obfuscate the input x to that unit by multiplying it by a random value y. The unit then computes the square $(xy)^2$, which is the same as x^2y^2. To decrypt, we only have to divide by the constant y^2 to get back x^2.

More generally, if our obfuscation function is homomorphic over the computational function, then the computation can be done on the data while it is encrypted, and thus the computational unit does not have to be trusted. Such schemes have been implemented at the software level, in theory by Gentry [9] and in practice by Osadchy *et al*. [17]. Any circuit can be obfuscated by a homomorphic function, but the cost can in theory be unacceptably large.

In the hardware context, we can place small encryption and decryption units (small enough to be manually or formally verified) between hardware components so that the component sees only encrypted values. In the non-computational case, since the internal function is the identity (*i.e.* nothing), we can use any invertible function. For units containing ALUs or other non-trivial logic, we require less trivial solutions. While a large portion of the units in real microprocessors are currently non-computational, especially units involved in memory subsystems and on-chip interconnects and routers, there is also an increasing use of accelerators and small functional units that use non-trivial logic.

• **Case Study:** As a case study of this method for protecting computational units against single-shot cheat codes on data interfaces, we discuss how one can apply a simple obfuscation function to the inputs of a cryptographic unit, such that the obfuscation function is homomorphic over the cryptographic function. Cryptographic units are an interesting case because they are the tool we normally use for encryption, but they see confidential data in unencrypted form and are thus profitable units to attack. Additionally, these units — and many other on-chip functional units — are often procured as third party IP. However, cryptographic units tend to use well known cryptographic schemes for which we can design homomorphic obfuscation functions. Take for example the RSA algorithm. For any data values x and y,

$$RSA(xy) = RSA(x)RSA(y)$$

If we want to encrypt a data value x using RSA on a special purpose RSA unit (as opposed to doing this with general purpose ISA instructions), we can perform the following algorithm.

1. Use hardware to generate a random value y.

2. Compute the product $z = xy$ using a regular, trusted ALU, where x is the value to be encrypted.

3. Ship z off to the cryptographic unit. That unit returns $RSA(z) = RSA(xy) = RSA(x)RSA(y)$.

4. Ship y off to the cryptographic unit to get $RSA(y)$.

5. Using the regular ALU, divide $RSA(z)$ by $RSA(y)$ to get $RSA(x)$.

We have used the untrusted cryptographic unit to encrypt the sensitive data x without allowing the unit to see the value of x. A potential weakness is that if this scheme is known and deterministic, the untrusted unit could divide alternating values by each other to derive x. Therefore, these values should be sent in a pseudo-random order. While this might leave a relatively small number of permutations (only a few million) to exhaust over, this scheme is perfectly good in this setting, because a hardware unit lacks the computational power to exhaust over millions of permutations each cycle.

• **Alternate Protection Schemes:** Although homomorphic encryption can be applied to any circuit, some circuits are more difficult than others[9]. Units that perform complex logical or arithmetic operations cannot usually be obfuscated using simple operations like XOR or multiplication because their custom logic is unlikely to have the right mathematical properties. For example, ALUs and decoders perform specific operations that cannot be trivially obfuscated. However, the code for this type of unit tends to be very small and can often be manually or formally verified. A final fall-back solution that can be applied if none of these techniques work is duplication, where n versions of the untrusted unit are designed by n different designers, and results are checked on a cycle by cycle basis. Duplication has a high area and power overhead, while the other techniques proposed are far more efficient and should be used whenever possible (if not 100% of the time).

• **Hardware Support:** Encryption schemes at the on-chip inter-unit level require the efficient generation of truly random bits. This can be done realistically due to recent innovations in the design of physical unclonable functions (PUFs), which can efficiently generate physically random bits[8, 14, 21, 28]. One simple way to obfuscate inputs once we have a PUF is to bitwise XOR the input value with the PUF going into an untrusted unit. Coming out of that unit, the data can be XOR'd again with the same PUF to get back the original value since $DATA \quad xor \quad PUF \quad xor \quad PUF = DATA$. Key storage for this mechanism should be handled by a few trusted bytes of data storage that should be invisible to the rest of the architecture.

• **Control Interfaces:** We do not apply obfuscation to inputs to control interfaces in our implementation. Generally control interfaces are very small (one or a few bits), and they cannot be scrambled without altering the operations performed within the unit. One of our assumptions is that control interfaces are small enough to be verified by validation engineers against single-shot cheat codes. For example, if a control interface is four bits wide, all 16 possibilities can be checked. This assumption worked fine for our analysis of OpenSPARC T2, discussed in Section VII. However, there are possible other settings where this would pose a problem. Given many small control interfaces, we are able to individually validate them. Doing so is sufficient to assure there is not a single-shot cheat code on the control interfaces, because a single-shot cheat code that combined bits from multiple separate control interfaces would be easily detectable automatically (by noticing that unrelated bits from separate controls are being fed into extraneous logic). However, this only works for designs where the source is viewable. A closed source third party IP unit with a large control interface or a very large number of control interfaces, for which we have no source visibility, could pose a problem. We discuss this as an item for future work in Section VIII.

C. *Sequence Breaking*

The last type of backdoor trigger in our complete taxonomy of triggers is the sequence cheat code. We protect against these cheat codes with a method called sequence breaking. The purpose of sequence breaking is to prevent cheat codes from being sent piecemeal. For example, if a unit receives T bits of information over a period of time from many packets, this is similar to receiving the T bits of information from one big packet as a single-shot cheat code. Therefore, we need to obfuscate the sequence of inputs to an untrusted unit, similarly to how we obfuscate the inputs themselves when we handle single-shot cheat codes.

Our solution is to benignly reorder sequences of inputs so as to preserve correctness but to prevent sequences from being deterministically supplied by a malicious user. If the attacker cannot determine the order in which events will occur, the attacker cannot with significant probability trigger a backdoor with a sequence cheat code. Even if the pieces of the trigger sequence are spread across multiple interfaces or over time, the attacker is unable to send the trigger, because the arrival times and order of arrival will always be mutated.

For a simple example, consider a memory controller with a backdoor that is triggered by a particular sequence of fifty loads and stores that must come in a pre-chosen order. We must make sure it is impossible (or extremely unlikely) for that particular sequence to be supplied by a malicious user.

For the example of the memory controller, we can change the order of those fifty loads and stores to prevent the sequence from looking like the cheat code. By adding physical randomness to the reordering scheme, we can provide strong likelihood (nearly 100%) that a specific malicious sequence will not occur.

It may not always be possible to reorder inputs to a unit. For example, a particular sequence of inputs to a memory controller may not be reorderable without violating sequential consistency or other consistency models. A smart, malicious user may be able to concoct such a sequence. Therefore, in addition to randomly reordering events, we need the ability to add dummy events in the case that reordering is impossible. For example, if we recognize a long stream of loads and stores that cannot be reordered, we can insert a few dummy loads (extra loads to pseudo-randomly chosen places in memory) into the stream to break up the sequence. As long as the unit never receives a sequence in the user-provided order (or in an order that could be deterministically predicted by the malicious user), the user cannot trigger the backdoor. A lot of the functionality for this sequence breaking already exists in modern microprocessors and can be integrated for this purpose with very minor changes.

VI. Probabilistic Security Guarantees

In this section, we describe the probabilistic security guarantees that our methods provide. Our three methods (power resets, data encryption, and data reordering/insertion) are able to provide probabilistic guarantees against the three types of attacks (timebombs, single-shot cheat codes, and sequence cheat codes). By adjusting the parameters in our methods, we can adjust the attacker's probability of success, at the cost of performance. Specific cost vs. security trade-offs are measured in Section VII-A.

The attacker's goal is to have a significant chance of triggering an attack while causing the validation engineers to have a significant chance of not triggering the attack during testing. If a validation engineer happens to trigger the backdoor, then the attacker's attempt to hide the backdoor is considered to have failed.

We first consider the case of a ticking timebomb. A ticking timebomb goes off after a fixed amount of time (or number of cycles) has passed. If power resets are implemented for every unit, then the attacker is forced to have the timebomb go off during the validation epoch, thus giving the validation engineer a 100% chance of catching the attack. Therefore, if the attacker wants a non-zero chance of success, he or she must allow the validation engineer a 100% chance of catching the attack. So the attacker cannot succeed.[2]

Second we consider a single-shot data trigger attack. If a unit has a large data interface and is covered by data obfuscation (*i.e.* the data is encrypted) then, assuming a reasonable encryption scheme, a correlation cannot be detected between the input data and the encrypted data received by the unit. This limitation results from the fact that the hardware unit must receive an input every cycle or every few cycles, and one or a few clock cycles is too little to break any reasonable encryption scheme. If the attacker wants to achieve a significant probability of the backdoor turning on when a piece of encrypted data is received, then the probability of the backdoor turning on for any random piece of data must be significant, meaning the probability of the validation engineer turning on the backdoor will be similarly high on each test instruction. Therefore, if the attacker wants a non-zero chance of success, he or she essentially guarantees that the validation engineer catches the attacker. For example, if the attackers wants a 1% chance of success, even if the validation epoch is only 10,000 cycles, the probability of the backdoor escaping detection is less than 10^{-43}.

Lastly we consider the sequence data trigger category. In this case the attacker wants to come up with a special sequence of inputs that the validation engineer is unlikely to supply in random testing. The sequence must be long or else the validation engineer can simply exhaust over all possible sequences of inputs. We will define 2^T to be the number of different sequences that a validation engineer can exhaustively search through. If a unit's interfaces are protected by reordering or insertion so that it never receives more than T input bits in the order specified by the user, then the attacker is out of luck because the validation engineer can exhaust through all 2^T possible combinations of inputs. If the attacker makes the secret sequence code less than or equal to T input bits long, then the validation engineer will trigger the backdoor while performing this search. Therefore, the attacker is forced to make the backdoor longer than T

[2] As discussed in Section VIII, it may be possible for analog circuitry or sources of true randomness to trigger randomized timebombs at uncontrolled times.

input bits long. This guarantees that the input bits will arrive at the hardware module scrambled and containing dummies. Each extra bit in the backdoor cheat code exponentially increases the number of possible permutations and dummies that must be recognized by the backdoor circuitry. This also exponentially increases the likelihood of the validation engineers tests triggering the backdoor.

For a tangible example, imagine the validation engineer can exhaustively test 2^{20} test cases but not more. The attacker, knowing this fact in advance, decides to use a length 21 sequence cheat code in the design and allows in the trigger detection logic that there can be one extra (wrong) input in the sequence (since 21 consecutive inputs cannot get through without obfuscation). There are 22 different places a dummy input can be inserted into the length 21 sequence, and the attacker must accept all 22 of these in order to have a chance of success. In that case, even though the validation engineer cannot exhaust over all 2^{21} test cases, he or she has less than a one in a billion chance of missing the backdoor when doing normal validation of 2^{20} random test cases. The intuitive reason for this is that the attacker has to allow for any possible reordering of his or her cheat code sequence, which is an exponential explosion of permutations (exponential in the number of dummies and reordering that occur). If we factor into account reordering in addition to dummy inserts, this probability gets even worse for the attacker. The amount of insertion and reordering can be tuned to achieve a desired probability, be it one in a billion, one in a trillion, etc.

The overall effect of our schemes is that an attacker cannot achieve a significant chance of triggering a backdoor without giving the validation engineers a chance of catching the backdoor, and this chance can be parametrized to be exponentially close to 100%. We believe this is the best guarantee that can be provided by a trigger prevention scheme of this nature.

Regardless of how we parameterize our methods, an attacker could use a source of true randomness to create a randomized timebomb that might or might not go off at anytime. The attacker and the validator would have equally small chances of setting it off at any time, and if the attacker gets lucky enough, it might go off at an opportune time. Since it is truly random, it doesn't matter if we reset power or scramble inputs. Future work may uncover alternative methods to prevent against non-digital, non-deterministic attacks, and we discuss this briefly in Section VIII.

VII. Evaluation

Our goals in evaluation are two-fold:

1) to determine what the performance impacts of our mechanisms are on typical programs.

2) to understand trade-offs between strength of security and cost/performance.

We implemented each of the three security mechanisms in the Zesto x86 simulator [16]. We describe our implementation and trade-offs we wish to study below:

- **Power Resets:** To emulate power resets, we added a mechanism that flushes the processor state at parameterizable time intervals. Flushing the processor state cancels all in-flight instructions, including pending memory requests, and wipes the local state of the core, such as branch prediction tables and other speculative structures, but leaves data in the caches and memory untouched, since we do not wipe out large, standardized structures.
- **Data Obfuscation:** To emulate our data obfuscation techniques, we added to the caches and memory controller the ability to stall for a parameterizable number of cycles on each access. This simulates the effect of adding an encryption step before each access.
- **Reordering and Insertion:** To emulate sequence breaking by reordering and insertion, we simulated a common memory controller and RAM that we found with publicly available specifications (a 4GB DDR3 SDRAM memory module from Kingston Hyper) and added the ability to stall pending memory accesses in the memory controller before sending them out to RAM. We additionally added the ability to psuedo-randomly reorder two memory accesses before the requests go out. We also added the ability to insert dummy loads to pseudo-randomly generated addresses into the memory controller. These loads are loads to random locations that look real to the memory controller (upper order bits make them look like loads to relevant parts of memory) but have no program relevance. Their results are thrown away rather than given back to the pipeline. The effects of hardware random number generation (*e.g.,* PUFs) were emulated with a pseudo-random number generator with fixed seed (for reproducibility).

Our baseline microarchitecture includes a 64KB, 8-way associative L1 instruction cache with 1 R/W port, a 2KB L-TAGE branch predictor, 6-way issue, out-of-order execution with speculation and prefetching support, 96-entry ROB, a 64KB, 8-way associative level 1 data cache with 1 R/W port, 256KB, 12-way associative unified L2 cache, and a detailed memory controller model. We simulated pinpoint representative regions of seven benchmarks from the SPEC CPU 2006 suite (representative regions generated with the ref input set).

Rather than make assumptions about how much delay to add for each of our modifications, we repeated these simulations with various delays, ranging from very optimistic to very conservative.

A. Experimental Results

Figure 4 shows the average slowdowns of each of our techniques. None of them caused more than a 1% performance hit on average. The highest bar (called 'Everything') is the result of a test with all of the techniques implemented together. The slowdown of all together was less than the sum of the parts, which we attribute to the fact that some of the slowdowns occur concurrently with each other. With all of these techniques together, our benchmarks slowed by an average of just under 0.9%. This figure also displays a breakdown of how each method affected each benchmark. The amount of effect of each method varied somewhat from benchmark to benchmark depending on program characteristics. The two benchmarks

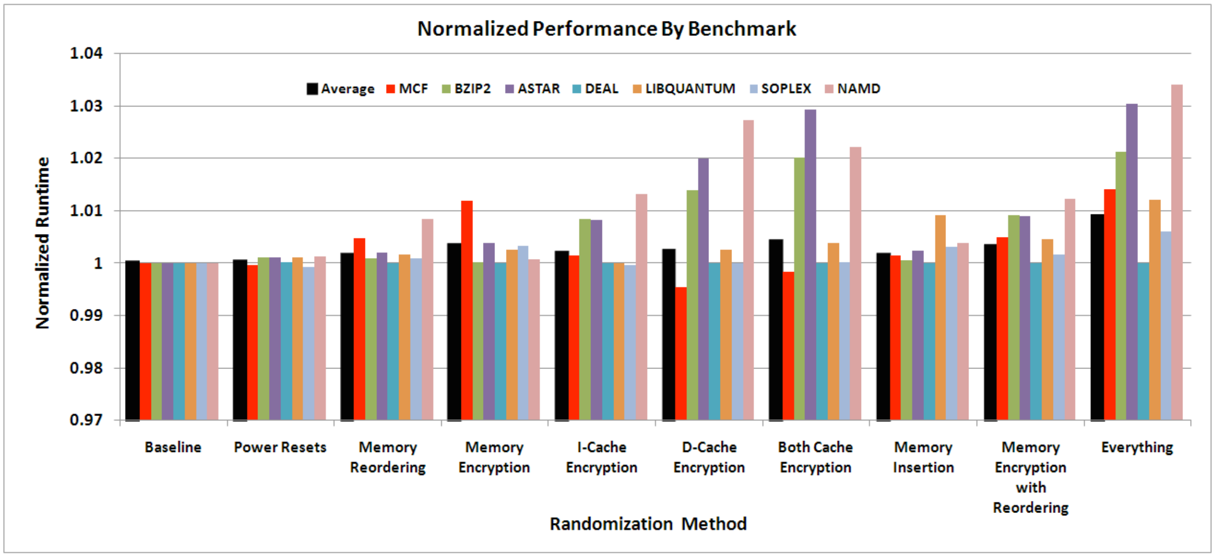


Fig. 4. This figure displays the average normalized runtimes (1 is the runtime on unmodified hardware) of some of our methods with default parameters, averaged over all of our 100 million instruction checkpoints, as well as breakdowns by benchmark. The reordering and insertion based schemes allowed a maximum of 20 bits of information before taking action. Our encryption schemes used one cycle of delay each. Our reset frequency was every 10 million instructions. The 'everything' test used all of these hardware modifications together.

that were affected the most were NAMD and ASTAR. We noticed that these two benchmarks had unusually high IPC, which we believe is why they were slightly more affected by our methods. The largest slowdown on any benchmark by any method was about 3.4%.

We can see that some of the techniques, such as power resets, memory reordering and memory insertion, had no significant effect on any of the benchmarks. These results fit our expectations. The power reset method is similar to causing a branch misprediction every 10 million cycles, which is fairly insignificant. The memory reordering in the memory controller, while it does alter the traffic patterns slightly, does not impact performance much because it does not increase overall bandwidth usage. The memory insertion method does increase bandwidth usage slightly, but we expected situations where this actually stalled the processor to be rare, and our results support this. For example, the checkpoint that experienced the highest impact from memory insertion only had about 23 misses per thousand instructions at the last level cache and thus was not too often bandwidth limited. Additionally, even for execution pieces that are bandwidth limited, these areas tend to come in bursts, thus allowing the overall performance hit of memory insertion to be amortized. For a hypothetical program that missed the last level cache on every single instruction, it would probably be best to use the memory reordering scheme, which does not increase overall bandwidth usage.

On the other hand, some techniques, especially data cache stalling, had larger effects. This was to be expected because adding a one cycle delay to every data cache access is significant and is likely to reduce pipeline throughput. This one cycle delay is our conservative measure of the impact of encryption. It is easy to implement the encryption as an extra step that takes one extra cycle before the operation reaches the cache. In reality, it is possible that this encryption, which can be as little as a one or two gate delay, can be squeezed into already existing steps and not cause this one cycle delay. Our results support that doing this may be desirable as the data cache stalling was the most significant performance impact of any of our methods.

Figure 5 shows the results from three experiments. The chart on the left elaborates on the trade-off between the power reset frequency and the performance loss that results. Recall that the power reset frequency needs to be less than the validation epoch. Our default frequency of 10 million cycles showed an insignificant performance loss. Pushing the frequency to 1 million cycles increased this performance loss to about 0.3%.

In the chart in the middle, we see the trade-off between memory insertion frequency and runtime performance. This frequency is determined by the maximum number of bits of information we allow to go through the memory controller before we insert a dummy load (dummy loads happen sooner depending on the random bits generated in the hardware. This value is the maximum that can possibly go through before a dummy must happen. The average time between dummies is about half this). Using a maximum of four inputs, we see just under a 1% performance hit on average. Using our default of 20 inputs, we get a little less than a 0.2% performance hit. Naturally, reducing the frequency of these insertions lessens the performance hit on average (with some degree of noise).

The trade-off between performance and the amount of memory reordering performed in the memory controller is displayed in the chart on the right. While we randomly reorder (benignly) memory accesses in the memory controller, we enforce that no more than a certain number of operations can go through without a reordering occuring. Using the low maximum of once every four, we see about a 0.6%

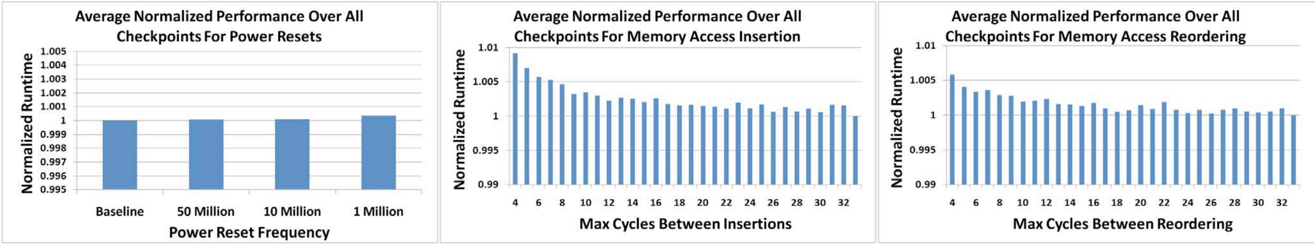


Fig. 5. These three charts show the trade-offs between performance and various parameters. The chart on the left shows the frequency of power resets. The chart in the middle shows the frequency of memory insertions. The one on the right shows the frequency of memory reordering. Higher frequencies tend to cause larger performance impacts.

performance hit. Using our default of once every twenty inputs, we got about a little under a 0.2% performance hit. This method appears to be a slight optimization over the method of simply inserting dummy operations, but not a very significant one.

B. *Coverage against Illinois Malicious Processors*

One of the earliest set of published HDL attacks came in the 'Illinois Malicious Processor' (IMP) [13]. This processor has components that can run in benign mode or malicious mode, and malicious mode can be triggered by certain deterministic events. Our defensive mechanisms in the core and memory system are enough to prevent all attacks that leverage the malicious circuitry in this processor.

There are two general mechanisms IMP can leverage to support an attack. The first - malicious memory access - allows any running program to access privileged portions of memory. The malicious user can trigger this backdoor by placing a sequence of (otherwise benign) bytes on the data bus. This attack cannot succeed against our defenses because we never allow a long enough series of bytes to reach the bus without reordering or insertion.

The second mechanism of IMP is called shadow mode. Shadow mode is a generic scheme for executing any software based attack as hidden or 'shadow' instructions. This shadow mode is turned on by a 'bootstrap trigger,' which is basically a special network packet that tells the hardware to initialize shadow mode. This is precisely a single-shot cheat code and would be prevented by our data obfuscation techniques since these packets can be trivially obfuscated (*i.e.* they are not needed for ALU computation in the unit that recognizes the trigger). Without changing much HDL code, IMP can be reconfigured to use a ticking timebomb to turn on shadow mode, but we would prevent this with our rapid reset technique. Therefore, we provide full guaranteed prevention against the IMP.

C. *Coverage and Practicality for OpenSPARC T2*

In order to assess our coverage for a real microprocessor, we analyzed the HDL source code for the OpenSPARC T2 microprocessor. The purpose of this analysis was to measure the practicality of implementing our mechanisms in a real microprocessor. Specifically, we were interested in (a) finding out if power resets can protect all on-chip modules, (b) sizes of the control interfaces to estimate the practicality of exhaustively testing control interfaces and (c) the number of data interfaces that need homomorphic encryption or duplication.

Timebombs are activated through the global interfaces, and we verified that in the OpenSPARC T2 core and memory subsystems, the modules are clocked and synchronous and hence can be protected with power resets. For the remaining two questions, we present first our analysis of the processor core code and then our analysis for the rest of the chip.

The results of manual analysis of an OpenSPARC T2 core are presented in Table I. We analyzed the code defining the interfaces for each module (roughly 5,840 inputs in total). Since the control interfaces are small, they can be exhaustively validated, thus negating the possibility of single-shot cheat codes against control interfaces. The number of cases required to protect control interfaces by exhaustion is less than 50,000 on average. The largest control interface requires exhausting through 262,144 cases during validation, which is very reasonable because it is common for validation to go for millions of cycles. Therefore, for the OpenSPARC T2 cores, all control interfaces can be validated individually, thus not requiring obfuscation.

Our analysis also reveals that most of the core would not be difficult to protect from single-shot data triggers. Of the eleven top level modules, only three of them perform non-trivial computations on their data interfaces. The rest can be protected by simple obfuscation schemes, such as XOR. The three difficult modules (Decoder, Execution, Floating Point/Graphics) can be protected with duplication. If design complexity is to be avoided, we can still protect the whole core while only duplicating a fraction of it.

We performed similar analysis on the top level interfaces for the modules in the memory system and the rest of the system-on-chip for the OpenSPARC T2. The results of this analysis are shown in Table II. Unsurprisingly, we found that nearly all of the data values moving through memory system and the rest of the system-on-chip are transported around but not operated upon arithmetically or logically. The only exception is the level 2 cache tag management unit, which would need to have one of its data interfaces duplicated or cleverly obfuscated (a routing packet that is fed into non-trivial logic for format parsing and ECC). For the rest of the modules, the only work done with data is queueing (mathematically the identity), equality checks (can be done with the encrypted data), and parity checks (can be done with the encrypted data). So nearly all of the system-on-chip can be protected without duplication or homomorphic functions. Additionally, the control interfaces are not vulnerable to single-shot cheat codes, as they average only 10,432 cases for exhaustion. So the control interfaces can be protected by only using sequence breaking. Therefore, the OpenSPARC T2 microprocessor can be practically and efficiently defended with our mechanisms.

A convenient feature of this methodology is that we were able to perform this analysis without having to inspect all the code by just focusing on interfaces. For the core, the analysis was possible by reading only a fraction of the HDL code (roughly 5000 lines of Verilog code out of the overall roughly 290,000 lines). Similarly for the full system-on-chip, the analysis was performed by reading only roughly 24,000 lines of Verilog code out of the total roughly one million lines.

VIII. Conclusions and Open Problems

In this paper we answer the question: *Given an untrusted hardware component, are there architectural solutions that prevent backdoor logic from turning on?*

Our solution is to obfuscate and randomize the inputs to hardware units to deceive the malicious logic and prevent it from recognizing triggers. We propose three methods of hardware randomization that correspond to the three types of digital backdoor triggers. Power resets obfuscate timing information to prevent units from detecting how long they have been powered on. Data obfuscation deceives malicious units by encrypting inputs. Sequence breaking reorders microarchitectural events, providing resilience against backdoors triggered by control information, *e.g.,* event types. These techniques, in concert, prevent malicious hardware logic from detecting trigger signals, thus preventing malicious designers from enabling ‘kill switches’ or other malicious modifications into hardware designs.

Our simulations show that our methods can be implemented with little performance impact (less than 1% on average). We also discuss how our methods can be parameterized to trade-off performance against probabilistic security.

We believe that our solutions are a significant step toward certifiably trustworthy hardware designs and set the stage for further exploration in this area. Next we outline some open problems in the area of hardware design backdoors.

• **Scalable Unintrusive Backdoor Protection** Two of the three input interfaces (global and data) can be protected without any insights into the inner workings of the design (except for the analysis required to identify these interfaces). However, protecting the control interface currently requires some insight into the design. To see why, consider a very large third party IP module (*e.g.,* a GPU) that is not open source. Such a unit may have a large number of small control interfaces which could be wired together to create a single-shot cheat code that would escape individual exhaustive validation of each interface. Without access to the source code to perform coverage testing or some other way to tell which control signals are combined together, it is currently not obvious how to distinguish the different control interfaces and decide which subset of them requires exhaustive validation. Techniques to relax this requirement would be highly valuable.

• **Attacks and Defenses against Non-deterministic Backdoors** As discussed previously, an analog circuit embedded inside a module at design time can be used to randomly trigger a backdoor. While the benefit of such a mechanism is unclear at this point, a deeper investigation on real hardware is required to understand the significance of this threat, and if necessary, devise mechanisms to catch such backdoors.

• **Attacks and Defenses for Side-channel Induced Backdoors** This work assumes that triggers can only be delivered through one or more of the valid input interfaces for a module. It may be possible to deliver triggers through side channels. Devising such a delivery mechanism for triggers and the receiving backdoor circuit, as well as protecting against such attacks, presents a significant challenge.

• **Design Guidelines for Trustworthiness Certification** As a result of our analysis, we uncovered a few properties that specifications should have in order to be easily protected against backdoors. Future security measures may either render these requirements unnecessary or add further to this list.

1) Untrusted modules should not be allowed to contain non volatile memory as they complicate the power reset process.
2) Untrusted modules should not be allowed internal analog components, as these may be used as a source of randomness to allow for randomized timebomb attacks.
3) If a source of true randomness is required in a design, that source should be contained within a small, trusted module.
4) Untrusted modules should not contain control interfaces that cannot be exhaustively validated.

• **Evaluating Applicability** We leave open for future work the endeavor to implement these solutions directly into real hardware. Based on our systematic classification of hardware interfaces and manual analysis of the OpenSPARC, we believe our solution to be practical. However, real world prototyping on a wide variety of hardware is essential to supplement this analysis.

The area of hardware backdoor protection poses many exciting and interesting challenges for security researchers. Solving these problems can lay the foundation for the future development of certifiably trustworthy hardware, a critical need for high security domains.

IX. Acknowledgements

We thank anonymous reviewers and members of the Computer Architecture and Security Technology Lab (CASTL) for their valuable feedback. Research conducted at CASTL is funded by grants from DARPA, AFRL (FA8750-10-2-0253, FA9950-09-1-0389), the NSF CAREER program, gifts from Microsoft Research and Columbia University, and infrastructure donations from Xilinx and Wind River Corp.

TABLE I
Analysis of OpenSPARC T2 Processor Core HDL

Unit Name	File Name	Control Interface Groups	Total Control Wires	Data Interface Groups	Total Data Wires	Largest Control Interface To Be Exhausted (in Bits)	Cases To Be Exhausted In Largest Conrol Interface	Purpose of Largest Control Signal	Fraction of Data Interfaces That Cannot Be Trivially Obfuscated	Explanation of Obfuscation Difficulties
Decoder	`dec.v`	13	369	8	264	8	256	Thread Activity Status	100%	Instruction Parsing Depends On Format
Execution	`exu.v`	6	165	5	271	10	1024	Memory Bus Interface Control	80%	Input Values Supplied To ALUs
FGU	`fgu.v`	8	135	7	449	8	256	Operation Control From Decoder	100%	Inputs Supplied To Non-Trivial Logic
Load/Store Gasket	`gkt.v`	7	64	8	452	11	2048	CPU ID Header In Crossbar Packet	0%	N/A
Fetch/Cache Unit	`ifu_cmu.v`	10	80	2	168	8	256	Thread-Based Fetch Trap Control	0%	N/A
Load/Store Unit	`lsu.v`	9	237	7	507	8	256	Immediate ASI Control	0%	N/A
Memory Management Unit	`mmu.v`	12	167	2	343	18	262,144	Cache To Processor Crossbar Control	0%	N/A
Thread Pick Unit	`pku.v`	7	684	8	354	8	256	Thread-Based Flush Control From TLU	0%	N/A
Power Management Unit	`pmu.v`	7	105	1	64	13	8192	Event Information From Decoder	0%	N/A
Stream Processing Unit	`spu.v`	1	18	1	48	3	8	Thread ID	0%	N/A
Trap Logic Unit	`tlu.v`	11	560	6	347	18	262,144	Cache To Processor Crossbar Control	0%	N/A
Average	N/A	8.27	234.91	5	297	10.27	48,804	N/A	25.45%	N/A

Fig. 6. Green indicates a module that can be easily protected. Yellow indicates some challenges. Red indicates a highly challenging module, possibly requiring duplication.

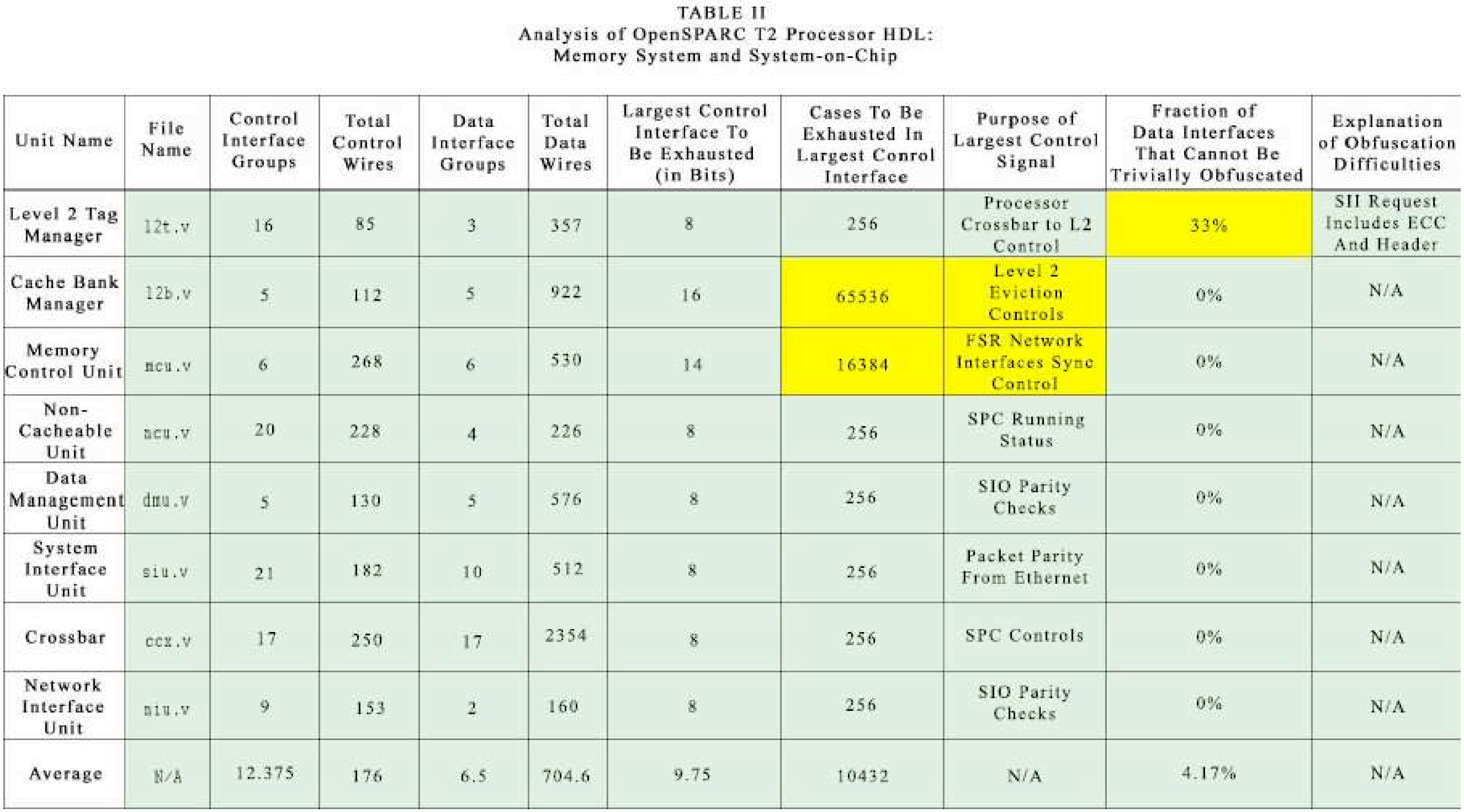


TABLE II
Analysis of OpenSPARC T2 Processor HDL:
Memory System and System-on-Chip

Unit Name	File Name	Control Interface Groups	Total Control Wires	Data Interface Groups	Total Data Wires	Largest Control Interface To Be Exhausted (in Bits)	Cases To Be Exhausted In Largest Conrol Interface	Purpose of Largest Control Signal	Fraction of Data Interfaces That Cannot Be Trivially Obfuscated	Explanation of Obfuscation Difficulties
Level 2 Tag Manager	l2t.v	16	85	3	357	8	256	Processor Crossbar to L2 Control	33%	SII Request Includes ECC And Header
Cache Bank Manager	l2b.v	5	112	5	922	16	65536	Level 2 Eviction Controls	0%	N/A
Memory Control Unit	mcu.v	6	268	6	530	14	16384	FSR Network Interfaces Sync Control	0%	N/A
Non-Cacheable Unit	ncu.v	20	228	4	226	8	256	SPC Running Status	0%	N/A
Data Management Unit	dmu.v	5	130	5	576	8	256	SIO Parity Checks	0%	N/A
System Interface Unit	siu.v	21	182	10	512	8	256	Packet Parity From Ethernet	0%	N/A
Crossbar	ccx.v	17	250	17	2354	8	256	SPC Controls	0%	N/A
Network Interface Unit	niu.v	9	153	2	160	8	256	SIO Parity Checks	0%	N/A
Average	N/A	12.375	176	6.5	704.6	9.75	10432	N/A	4.17%	N/A

Fig. 7. Green indicates a module that can be easily protected. Yellow indicates some challenges. Red indicates a highly challenging module, possibly requiring duplication.

References

[1] S. Adee. The hunt for the kill switch. *IEEE Spectrum Magazine*, 45(5):34–39, 2008.

[2] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using ic fingerprinting. In *Security and Privacy, 2007. SP '07. IEEE Symposium on*, pages 296–310, May 2007.

[3] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao. Guided test generation for isolation and detection of embedded trojans in ics. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 363–366, New York, NY, USA, 2008. ACM.

[4] M. Banga and M. Hsiao. A region based approach for the identification of hardware trojans. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 40–47, June 2008.

[5] M. Banga and M. S. Hsiao. A region based approach for the identification of hardware trojans. In *Hardware-Oriented Security and Trust, 2008. HOST '08. IEEE International Workshop on*, June 2008.

[6] E. Brown. New nist report advises: Securing critical computer systems begins at the beginning.

[7] R. Chakraborty, S. Paul, and S. Bhunia. On-demand transparency for improving hardware trojan detectability. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 48–50, June 2008.

[8] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas. Silicon physical random functions. In *ACM Conference on Computer and Communications Security*, pages 148–160, New York, NY, USA, 2002. ACM Press.

[9] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[10] J. Grand. Advanced hardware hacking techniques.

[11] M. Hicks, 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 31st IEEE Symposium on Security and Privacy*, 2010.

[12] Y. Jin and Y. Makris. Hardware trojan detection using path delay fingerpring. In *Hardware-Oriented Security and Trust, 2008. HOST '08. IEEE International Workshop on*, June 2008.

[13] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, pages 5:1–5:8, Berkeley, CA, USA, 2008. USENIX Association.

[14] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. A technique to build a secret key in integrated circuits for identification and authentication application. In *Proceedings of the Symposium on VLSI Circuits*, pages 176–159, 2004.

[15] J. Li and J. Lach. At-speed delay characterization for ic authentication and trojan horse detection. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 8–14, June 2008.

[16] G. Loh, S. Subramaniam, and Y. Xie. Zesto: A cycle-level simulator for highly detailed microarchitecture exploration. pages 53 –64, apr. 2009.

[17] M. Osadchy, B. Pinkas, A. Jarrous, and B. Moskovitch. Scifi - a system for secure computation of face identification. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[18] R. M. Rad, X. Wang, M. Tehranipoor, and J. Plusquellic. Power supply signal calibration techniques for improving detection resolution to hardware trojans. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 632–639, Piscataway, NJ, USA, 2008. IEEE Press.

[19] J. Rajendran. The alpha platform. *CSAW Conference. 2nd Embedded Systems Challenge.*, 2008.

[20] H. Salmani, M. Tehranipoor, and J. Plusquellic. New design strategy for improving hardware trojan detection and reducing trojan activation time. In *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pages 66 –73, july 2009.

[21] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference*, pages 9–14, New York, NY, USA, 2007. ACM Press.

[22] M. Swanson, N. Bartol, and R. Moorthy. Piloting supply chain risk management practices for federal information systems.

[23] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 207–222, New York, NY, USA, 2003. ACM.

[24] M. M. Swift, S. Martin, H. M. Levy, and S. J. Eggers. Nooks: an architecture for reliable device drivers. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 102–107, New York, NY, USA, 2002. ACM.

[25] United Stated Department of Defense. *High performance microchip supply*, February 2005.

[26] A. Waksman and S. Sethumadhavan. Tamper evident microprocessors. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[27] X. Wang, M. Tehranipoor, and J. Plusquellic. Detecting malicious inclusions in secure hardware: Challenges and solutions. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 15–19, June 2008.

[28] M.-D. M. Yu and S. Devadas. Secure and robust error correction for physical unclonable functions. 27(1):48–65, Jan.-Feb. 2010.